

FORM

v5.0.0-beta.1-319-gd776835

Generated by Doxygen 1.9.8

1 Data Structure Index	1
1.1 Data Structures	1
2 File Index	5
2.1 File List	5
3 Data Structure Documentation	7
3.1 AEdge Class Reference	7
3.1.1 Detailed Description	7
3.1.2 Constructor & Destructor Documentation	8
3.1.2.1 AEdge()	8
3.1.2.2 ~AEdge()	8
3.1.3 Field Documentation	8
3.1.3.1 cand	8
3.1.3.2 nodes	8
3.1.3.3 nlegs	8
3.1.3.4 ptcl	8
3.1.3.5 DUMMYPADDING	9
3.2 AllGlobals Struct Reference	9
3.2.1 Detailed Description	10
3.2.2 Field Documentation	10
3.2.2.1 M	10
3.2.2.2 Cc	10
3.2.2.3 S	10
3.2.2.4 R	10
3.2.2.5 N	11
3.2.2.6 O	11
3.2.2.7 P	11
3.2.2.8 T	11
3.2.2.9 X	11
3.3 ANode Class Reference	11
3.3.1 Detailed Description	12
3.3.2 Constructor & Destructor Documentation	12
3.3.2.1 ANode()	12
3.3.2.2 ~ANode()	12
3.3.3 Member Function Documentation	12
3.3.3.1 newleg()	12
3.3.4 Field Documentation	13
3.3.4.1 deg	13
3.3.4.2 nlegs	13
3.3.4.3 anodes	13
3.3.4.4 aedges	13
3.3.4.5 aelegs	13

3.3.4.6 cand	13
3.4 ARGBUFFER Struct Reference	14
3.4.1 Detailed Description	14
3.4.2 Field Documentation	14
3.4.2.1 buffer	14
3.4.2.2 dollar	14
3.4.2.3 size	15
3.4.2.4 type	15
3.4.2.5 dummy	15
3.5 Assign Class Reference	15
3.5.1 Detailed Description	17
3.5.2 Constructor & Destructor Documentation	17
3.5.2.1 Assign()	17
3.5.2.2 ~Assign()	17
3.5.3 Member Function Documentation	17
3.5.3.1 prCand()	17
3.5.3.2 checkAG()	17
3.5.3.3 assignAllVertices()	18
3.5.3.4 selectVertex()	18
3.5.3.5 selectVertexSimp()	18
3.5.3.6 selectLeg()	18
3.5.3.7 assignVertex()	18
3.5.3.8 allAssigned()	18
3.5.3.9 fromMGraph()	18
3.5.3.10 addEdge()	19
3.5.3.11 connect()	19
3.5.3.12 fillEGraph()	19
3.5.3.13 reordLeg()	19
3.5.3.14 getLegParticle()	19
3.5.3.15 legEdgeParticle()	20
3.5.3.16 legPart()	20
3.5.3.17 candPart()	20
3.5.3.18 selUnAssVertex()	20
3.5.3.19 selUnAssVertexSimp()	20
3.5.3.20 selUnAssLeg()	20
3.5.3.21 assignVertex()	21
3.5.3.22 assignPLeg()	21
3.5.3.23 isOrdPLeg()	21
3.5.3.24 detEdge()	21
3.5.3.25 candPartClassify()	21
3.5.3.26 updateCandNode()	21
3.5.3.27 checkOrderCpl()	22

3.5.3.28 isOrdLegs()	22
3.5.3.29 isIsomorphic()	22
3.5.3.30 cmpPermGraph()	22
3.5.3.31 cmpNodes()	22
3.5.3.32 edgeSym()	22
3.5.3.33 saveCouple()	23
3.5.3.34 restoreCouple()	23
3.5.3.35 subCouple()	23
3.5.3.36 checkCand()	23
3.5.3.37 checkNode()	23
3.5.4 Field Documentation	23
3.5.4.1 egraph	23
3.5.4.2 mgraph	24
3.5.4.3 sproc	24
3.5.4.4 proc	24
3.5.4.5 model	24
3.5.4.6 opt	24
3.5.4.7 astack	24
3.5.4.8 pnclass	24
3.5.4.9 orbits	24
3.5.4.10 nNodes	25
3.5.4.11 nEdges	25
3.5.4.12 nExtern	25
3.5.4.13 nETotal	25
3.5.4.14 nAGraphs	25
3.5.4.15 wAGraphs	25
3.5.4.16 nAOPI	25
3.5.4.17 wAOPI	25
3.5.4.18 nodes	26
3.5.4.19 edges	26
3.5.4.20 checkpoint0	26
3.5.4.21 cplleft	26
3.6 AStack Class Reference	26
3.6.1 Detailed Description	27
3.6.2 Constructor & Destructor Documentation	27
3.6.2.1 AStack()	27
3.6.2.2 ~AStack()	27
3.6.3 Member Function Documentation	27
3.6.3.1 setAGraph()	27
3.6.3.2 checkPoint()	28
3.6.3.3 restore()	28
3.6.3.4 restoreMsg()	28

3.6.3.5 prStack()	28
3.6.3.6 pushNode()	28
3.6.3.7 pushEdge()	28
3.6.4 Field Documentation	28
3.6.4.1 agraph	28
3.6.4.2 nStack	29
3.6.4.3 nStackP	29
3.6.4.4 nSize	29
3.6.4.5 eStack	29
3.6.4.6 eStackP	29
3.6.4.7 eSize	29
3.7 bit_field Struct Reference	29
3.7.1 Detailed Description	30
3.7.2 Field Documentation	30
3.7.2.1 bit_0	30
3.7.2.2 bit_1	30
3.7.2.3 bit_2	30
3.7.2.4 bit_3	30
3.7.2.5 bit_4	31
3.7.2.6 bit_5	31
3.7.2.7 bit_6	31
3.7.2.8 bit_7	31
3.8 BrAcKeTiNdEx Struct Reference	31
3.8.1 Detailed Description	32
3.8.2 Field Documentation	32
3.8.2.1 start	32
3.8.2.2 next	32
3.8.2.3 bracket	32
3.8.2.4 termsinbracket	32
3.9 BrAcKeTiNfO Struct Reference	33
3.9.1 Detailed Description	33
3.9.2 Field Documentation	33
3.9.2.1 indexbuffer	33
3.9.2.2 bracketbuffer	34
3.9.2.3 bracketbuffersize	34
3.9.2.4 indexbuffersize	34
3.9.2.5 bracketfill	34
3.9.2.6 indexfill	34
3.9.2.7 SortType	34
3.10 BracketInfo Struct Reference	35
3.10.1 Detailed Description	35
3.10.2 Constructor & Destructor Documentation	35

3.10.2.1 BracketInfo()	35
3.10.3 Member Function Documentation	36
3.10.3.1 operator<()	36
3.10.4 Field Documentation	36
3.10.4.1 pattern	36
3.10.4.2 num_terms	36
3.10.4.3 dummy	36
3.10.4.4 p	36
3.11 bufIPstruct Struct Reference	37
3.11.1 Detailed Description	37
3.11.2 Field Documentation	37
3.11.2.1 i	37
3.11.2.2 e	38
3.12 C_const Struct Reference	38
3.12.1 Detailed Description	42
3.12.2 Field Documentation	42
3.12.2.1 separators	42
3.12.2.2 StoreFileSize	42
3.12.2.3 dollarnames	43
3.12.2.4 exprnames	43
3.12.2.5 varnames	43
3.12.2.6 ChannelList	43
3.12.2.7 DubiousList	43
3.12.2.8 FunctionList	43
3.12.2.9 ExpressionList	44
3.12.2.10 IndexList	44
3.12.2.11 SetElementList	44
3.12.2.12 SetList	44
3.12.2.13 SymbolList	44
3.12.2.14 VectorList	44
3.12.2.15 PotModDolList	45
3.12.2.16 ModOptDolList	45
3.12.2.17 TableBaseList	45
3.12.2.18 cbufList	45
3.12.2.19 AutoSymbolList	45
3.12.2.20 AutoIndexList	45
3.12.2.21 AutoVectorList	45
3.12.2.22 AutoFunctionList	46
3.12.2.23 autonames	46
3.12.2.24 Symbols	46
3.12.2.25 Indices	46
3.12.2.26 Vectors	46

3.12.2.27 Functions	46
3.12.2.28 activenames	46
3.12.2.29 Streams	47
3.12.2.30 CurrentStream	47
3.12.2.31 SwitchArray	47
3.12.2.32 models	47
3.12.2.33 SwitchHeap	47
3.12.2.34 termstack	47
3.12.2.35 termsortstack	47
3.12.2.36 cmod	48
3.12.2.37 powmod	48
3.12.2.38 modpowers	48
3.12.2.39 halfmod	48
3.12.2.40 ProtoType	48
3.12.2.41 WildC	48
3.12.2.42 IfHeap	48
3.12.2.43 IfCount	49
3.12.2.44 IfStack	49
3.12.2.45 iBuffer	49
3.12.2.46 iPointer	49
3.12.2.47 iStop	49
3.12.2.48 LabelNames	49
3.12.2.49 FixIndices	50
3.12.2.50 termsumcheck	50
3.12.2.51 WildcardNames	50
3.12.2.52 Labels	50
3.12.2.53 tokens	50
3.12.2.54 toptokens	50
3.12.2.55 endoftokens	51
3.12.2.56 tokenarglevel	51
3.12.2.57 modinverses	51
3.12.2.58 Fortran90Kind	51
3.12.2.59 MultiBracketBuf	51
3.12.2.60 extrasym	51
3.12.2.61 doloopstack	51
3.12.2.62 doloopnest	52
3.12.2.63 CheckpointRunAfter	52
3.12.2.64 CheckpointRunBefore	52
3.12.2.65 IfSumCheck	52
3.12.2.66 CommuteInSet	52
3.12.2.67 TestValue	52
3.12.2.68 argstack	52

3.12.2.69 insidestack	53
3.12.2.70 inexprstack	53
3.12.2.71 iBufferSize	53
3.12.2.72 TransEname	53
3.12.2.73 ProcessBucketSize	53
3.12.2.74 mProcessBucketSize	53
3.12.2.75 CModule	53
3.12.2.76 ThreadBucketSize	53
3.12.2.77 CheckpointStamp	54
3.12.2.78 CheckpointInterval	54
3.12.2.79 cbufnum	54
3.12.2.80 AutoDeclareFlag	54
3.12.2.81 NoShowInput	54
3.12.2.82 ShortStats	54
3.12.2.83 compiletype	54
3.12.2.84 firstconstindex	55
3.12.2.85 insidefirst	55
3.12.2.86 minsidefirst	55
3.12.2.87 wildflag	55
3.12.2.88 NumLabels	55
3.12.2.89 MaxLabels	55
3.12.2.90 IDefDim	55
3.12.2.91 IDefDim4	55
3.12.2.92 NumWildcardNames	56
3.12.2.93 WildcardBufferSize	56
3.12.2.94 MaxIf	56
3.12.2.95 NumStreams	56
3.12.2.96 MaxNumStreams	56
3.12.2.97 firstctypemessage	56
3.12.2.98 tablecheck	56
3.12.2.99 idoption	56
3.12.2.100 BottomLevel	57
3.12.2.101 CompileLevel	57
3.12.2.102 TokensWriteFlag	57
3.12.2.103 UnsureDollarMode	57
3.12.2.104 outsidefun	57
3.12.2.105 funpowers	57
3.12.2.106 WarnFlag	57
3.12.2.107 StatsFlag	57
3.12.2.108 NamesFlag	58
3.12.2.109 CodesFlag	58
3.12.2.110 SetupFlag	58

3.12.2.111 SortType	58
3.12.2.112 ISortType	58
3.12.2.113 ThreadStats	58
3.12.2.114 FinalStats	58
3.12.2.115 OldParallelStats	58
3.12.2.116 ThreadsFlag	59
3.12.2.117 ThreadBalancing	59
3.12.2.118 ThreadSortFileSynch	59
3.12.2.119 ProcessStats	59
3.12.2.120 BracketNormalize	59
3.12.2.121 maxtermlevel	59
3.12.2.122 dumnumflag	59
3.12.2.123 bracketindexflag	59
3.12.2.124 parallelflag	60
3.12.2.125 mparallelflag	60
3.12.2.126 inparallelflag	60
3.12.2.127 partodoflag	60
3.12.2.128 properorderflag	60
3.12.2.129 vetofilling	60
3.12.2.130 tablefilling	60
3.12.2.131 vetotablebasefill	60
3.12.2.132 exprfillwarning	61
3.12.2.133 lhdollarflag	61
3.12.2.134 NoCompress	61
3.12.2.135 IsFortran90	61
3.12.2.136 MultiBracketLevels	61
3.12.2.137 topolynomialflag	61
3.12.2.138 ffbufnum	61
3.12.2.139 OldFactArgFlag	61
3.12.2.140 MemDebugFlag	62
3.12.2.141 OldGCDflag	62
3.12.2.142 WTimeStatsFlag	62
3.12.2.143 SortReallocateFlag	62
3.12.2.144 PrintBacktraceFlag	62
3.12.2.145 FlintPolyFlag	62
3.12.2.146 HumanStatsFlag	62
3.12.2.147 doloopstacksize	62
3.12.2.148 dolooplevel	63
3.12.2.149 CheckpointFlag	63
3.12.2.150 SizeCommutelnSet	63
3.12.2.151 origin	63
3.12.2.152 vectorlikeLHS	63

3.12.2.153 nummodels	63
3.12.2.154 modelspace	63
3.12.2.155 ModelLevel	64
3.12.2.156 ShortStatsMax	64
3.12.2.157 InnerTest	64
3.12.2.158 argsumcheck	64
3.12.2.159 insidesumcheck	64
3.12.2.160 inexprsumcheck	64
3.12.2.161 RepSumCheck	64
3.12.2.162 IUniTrace	64
3.12.2.163 RepLevel	65
3.12.2.164 arglevel	65
3.12.2.165 insidelevel	65
3.12.2.166 inexprlevel	65
3.12.2.167 termlevel	65
3.12.2.168 MustTestTable	65
3.12.2.169 DumNum	65
3.12.2.170 ncmmod	65
3.12.2.171 npowmod	66
3.12.2.172 modmode	66
3.12.2.173 nhalfmod	66
3.12.2.174 DirtPow	66
3.12.2.175 IUnitTrace	66
3.12.2.176 NwildC	66
3.12.2.177 ComDefer	66
3.12.2.178 CollectFun	66
3.12.2.179 AltCollectFun	67
3.12.2.180 OutputMode	67
3.12.2.181 Cnumpows	67
3.12.2.182 OutputSpaces	67
3.12.2.183 OutNumberType	67
3.12.2.184 DidClean	67
3.12.2.185 IfLevel	67
3.12.2.186 WhileLevel	67
3.12.2.187 SwitchLevel	68
3.12.2.188 SwitchInArray	68
3.12.2.189 MaxSwitch	68
3.12.2.190 LogHandle	68
3.12.2.191 LineLength	68
3.12.2.192 StoreHandle	68
3.12.2.193 HideLevel	68
3.12.2.194 IPolyFun	68

3.12.2.195 IPolyFunInv	69
3.12.2.196 IPolyFunType	69
3.12.2.197 IPolyFunExp	69
3.12.2.198 IPolyFunVar	69
3.12.2.199 IPolyFunPow	69
3.12.2.200 SymChangeFlag	69
3.12.2.201 CollectPercentage	69
3.12.2.202 extrasymbols	69
3.12.2.203 PolyRatFunChanged	70
3.12.2.204 ToBeInFactors	70
3.12.2.205 Commercial	70
3.12.2.206 debugFlags	70
3.13 CbUf Struct Reference	70
3.13.1 Detailed Description	71
3.13.2 Field Documentation	71
3.13.2.1 Buffer	71
3.13.2.2 Top	71
3.13.2.3 Pointer	72
3.13.2.4 lhs	72
3.13.2.5 rhs	72
3.13.2.6 CanCommu	72
3.13.2.7 NumTerms	72
3.13.2.8 numdum	73
3.13.2.9 dimension	73
3.13.2.10 boomlijst	73
3.13.2.11 BufferSize	73
3.13.2.12 numlhs	73
3.13.2.13 numrhs	73
3.13.2.14 maxlhs	74
3.13.2.15 maxrhs	74
3.13.2.16 mnumlhs	74
3.13.2.17 mnumrhs	74
3.13.2.18 numtree	74
3.13.2.19 rootnum	74
3.13.2.20 MaxTreeSize	74
3.14 ChAnNeL Struct Reference	75
3.14.1 Detailed Description	75
3.14.2 Field Documentation	75
3.14.2.1 name	75
3.14.2.2 handle	75
3.15 COST Struct Reference	75
3.15.1 Detailed Description	76

3.15.2 Field Documentation	76
3.15.2.1 add	76
3.15.2.2 mul	76
3.15.2.3 div	76
3.15.2.4 pow	76
3.16 CSEEq Struct Reference	76
3.16.1 Detailed Description	76
3.16.2 Member Function Documentation	77
3.16.2.1 operator()()	77
3.17 CSEHash Struct Reference	77
3.17.1 Detailed Description	77
3.17.2 Member Function Documentation	77
3.17.2.1 operator()()	77
3.18 dbase Struct Reference	78
3.18.1 Detailed Description	78
3.18.2 Field Documentation	78
3.18.2.1 info	78
3.18.2.2 mode	79
3.18.2.3 tablenameessize	79
3.18.2.4 topnumber	79
3.18.2.5 tablenameefill	79
3.18.2.6 rwmode	79
3.18.2.7 iblocks	79
3.18.2.8 nblocks	79
3.18.2.9 handle	79
3.18.2.10 name	80
3.18.2.11 fullname	80
3.18.2.12 tablenames	80
3.19 DGraph Struct Reference	80
3.19.1 Detailed Description	81
3.19.2 Field Documentation	81
3.19.2.1 nnodes	81
3.19.2.2 nedges	81
3.19.2.3 nodes	81
3.19.2.4 edges	81
3.20 DICTIONARY Struct Reference	81
3.20.1 Detailed Description	82
3.20.2 Field Documentation	82
3.20.2.1 elements	82
3.20.2.2 name	82
3.20.2.3 sizeelements	82
3.20.2.4 numelements	82

3.20.2.5 numbers	82
3.20.2.6 variables	83
3.20.2.7 characters	83
3.20.2.8 funwith	83
3.20.2.9 gnumelements	83
3.20.2.10 ranges	83
3.21 DICTIONARY_ELEMENT Struct Reference	83
3.21.1 Detailed Description	83
3.21.2 Field Documentation	84
3.21.2.1 lhs	84
3.21.2.2 rhs	84
3.21.2.3 type	84
3.21.2.4 size	84
3.22 DiStRiBuTe Struct Reference	84
3.22.1 Detailed Description	85
3.22.2 Field Documentation	85
3.22.2.1 obj1	85
3.22.2.2 obj2	85
3.22.2.3 out	85
3.22.2.4 sign	85
3.22.2.5 n1	85
3.22.2.6 n2	85
3.22.2.7 n	86
3.22.2.8 cycle	86
3.23 DNode Struct Reference	86
3.23.1 Detailed Description	86
3.23.2 Field Documentation	86
3.23.2.1 extloop	86
3.23.2.2 intrct	86
3.24 dollar_buf Struct Reference	87
3.24.1 Detailed Description	87
3.25 DoLIaRS Struct Reference	87
3.25.1 Detailed Description	87
3.25.2 Field Documentation	88
3.25.2.1 where	88
3.25.2.2 factors	88
3.25.2.3 size	88
3.25.2.4 name	88
3.25.2.5 type	88
3.25.2.6 node	88
3.25.2.7 index	88
3.25.2.8 zero	89

3.25.2.9 numdummies	89
3.25.2.10 nfactors	89
3.26 DoLoOp Struct Reference	89
3.26.1 Detailed Description	90
3.26.2 Field Documentation	90
3.26.2.1 p	90
3.26.2.2 name	90
3.26.2.3 vars	91
3.26.2.4 contents	91
3.26.2.5 dollarname	91
3.26.2.6 startlinenumber	91
3.26.2.7 firstnum	91
3.26.2.8 lastnum	91
3.26.2.9 incnum	91
3.26.2.10 type	92
3.26.2.11 NoShowInput	92
3.26.2.12 errorsinloop	92
3.26.2.13 firstloopcall	92
3.26.2.14 firstdollar	92
3.26.2.15 lastdollar	92
3.26.2.16 incdollar	92
3.26.2.17 NumPreTypes	92
3.26.2.18 PriefLevel	93
3.26.2.19 PreSwitchLevel	93
3.27 DuBiOuS Struct Reference	93
3.27.1 Detailed Description	93
3.27.2 Field Documentation	93
3.27.2.1 name	93
3.27.2.2 node	93
3.27.2.3 dummy	94
3.28 ECand Class Reference	94
3.28.1 Detailed Description	94
3.28.2 Constructor & Destructor Documentation	94
3.28.2.1 ECand()	94
3.28.2.2 ~ECand()	94
3.28.3 Member Function Documentation	95
3.28.3.1 prECand()	95
3.28.4 Field Documentation	95
3.28.4.1 det	95
3.28.4.2 nplist	95
3.28.4.3 plist	95
3.29 EEdge Class Reference	95

3.29.1 Detailed Description	96
3.29.2 Constructor & Destructor Documentation	96
3.29.2.1 EEdge() [1/2]	96
3.29.2.2 EEdge() [2/2]	96
3.29.2.3 ~EEdge()	97
3.29.3 Member Function Documentation	97
3.29.3.1 copy()	97
3.29.3.2 setId()	97
3.29.3.3 print()	97
3.29.3.4 setType()	97
3.29.3.5 setLMom()	97
3.29.3.6 setEMom()	98
3.29.4 Field Documentation	98
3.29.4.1 egraph	98
3.29.4.2 id	98
3.29.4.3 ext	98
3.29.4.4 ptcl	98
3.29.4.5 deleted	98
3.29.4.6 nodes	98
3.29.4.7 nlegs	99
3.29.4.8 emom	99
3.29.4.9 lmom	99
3.29.4.10 extMom	99
3.29.4.11 momset	99
3.29.4.12 momdir	99
3.29.4.13 cut	99
3.29.4.14 visited	99
3.29.4.15 conid	100
3.29.4.16 edtype	100
3.29.4.17 opicomp	100
3.29.4.18 dir	100
3.29.4.19 DUMMYPADDING	100
3.30 EFLine Class Reference	100
3.30.1 Detailed Description	101
3.30.2 Constructor & Destructor Documentation	101
3.30.2.1 EFLine()	101
3.30.3 Member Function Documentation	101
3.30.3.1 print()	101
3.30.4 Field Documentation	101
3.30.4.1 elist	101
3.30.4.2 ftype	101
3.30.4.3 fkind	101

3.30.4.4 nlist	102
3.31 EGraph Class Reference	102
3.31.1 Detailed Description	104
3.31.2 Constructor & Destructor Documentation	104
3.31.2.1 EGraph()	104
3.31.2.2 ~EGraph()	104
3.31.3 Member Function Documentation	104
3.31.3.1 copy()	104
3.31.3.2 print()	104
3.31.3.3 printPy()	105
3.31.3.4 fromDGraph()	105
3.31.3.5 fromMGraph()	105
3.31.3.6 optQGrafM()	105
3.31.3.7 optQGrafA()	105
3.31.3.8 isOptE()	105
3.31.3.9 setExtern()	105
3.31.3.10 isExternal()	106
3.31.3.11 isFermion()	106
3.31.3.12 setExtLoop()	106
3.31.3.13 endSetExtLoop()	106
3.31.3.14 connComp()	106
3.31.3.15 connVisit()	106
3.31.3.16 biconnE()	106
3.31.3.17 biinitE()	107
3.31.3.18 bisearchE()	107
3.31.3.19 findRoot()	107
3.31.3.20 dirEdge()	107
3.31.3.21 extMomConsv()	107
3.31.3.22 cmpMom()	107
3.31.3.23 groupLMom()	108
3.31.3.24 chkMomConsv()	108
3.31.3.25 prFLines()	108
3.31.3.26 getFLines()	108
3.31.3.27 fltrace()	108
3.31.3.28 addFLine()	108
3.31.3.29 legParticle()	109
3.31.4 Field Documentation	109
3.31.4.1 opt	109
3.31.4.2 model	109
3.31.4.3 proc	109
3.31.4.4 sproc	109
3.31.4.5 mgraph	109

3.31.4.6 econn	109
3.31.4.7 nodes	110
3.31.4.8 edges	110
3.31.4.9 mld	110
3.31.4.10 ald	110
3.31.4.11 sld	110
3.31.4.12 gSubld	110
3.31.4.13 assigned	110
3.31.4.14 fsign	110
3.31.4.15 nsym	111
3.31.4.16 esym	111
3.31.4.17 nsym1	111
3.31.4.18 extperm	111
3.31.4.19 multp	111
3.31.4.20 pld	111
3.31.4.21 sNodes	111
3.31.4.22 sEdges	111
3.31.4.23 sMaxdeg	112
3.31.4.24 sLoops	112
3.31.4.25 nNodes	112
3.31.4.26 nEdges	112
3.31.4.27 nExtern	112
3.31.4.28 maxdeg	112
3.31.4.29 nLoops	112
3.31.4.30 totalc	112
3.31.4.31 nopicomp	113
3.31.4.32 opi2plp	113
3.31.4.33 nopi2p	113
3.31.4.34 nadj2ptv	113
3.31.4.35 DUMMYPADDING	113
3.31.4.36 bidef	113
3.31.4.37 bilow	113
3.31.4.38 extMom	113
3.31.4.39 bconn	114
3.31.4.40 bicount	114
3.31.4.41 loopm	114
3.31.4.42 opiCount	114
3.31.4.43 flines	114
3.31.4.44 nFlines	114
3.31.4.45 DUMMYPADDING1	114
3.32 ENode Class Reference	115
3.32.1 Detailed Description	115

3.32.2 Constructor & Destructor Documentation	116
3.32.2.1 ENode() [1/2]	116
3.32.2.2 ENode() [2/2]	116
3.32.2.3 ~ENode()	116
3.32.3 Member Function Documentation	116
3.32.3.1 initAss()	116
3.32.3.2 setId()	116
3.32.3.3 copy()	116
3.32.3.4 setExtern()	117
3.32.3.5 setType()	117
3.32.3.6 print()	117
3.32.4 Field Documentation	117
3.32.4.1 egraph	117
3.32.4.2 edges	117
3.32.4.3 id	117
3.32.4.4 maxdeg	117
3.32.4.5 deg	118
3.32.4.6 extloop	118
3.32.4.7 ndtype	118
3.32.4.8 intrct	118
3.32.4.9 klow	118
3.32.4.10 visited	118
3.32.4.11 DUMMYPADDING	118
3.33 EStack Class Reference	119
3.33.1 Detailed Description	119
3.33.2 Constructor & Destructor Documentation	119
3.33.2.1 EStack()	119
3.33.2.2 ~EStack()	119
3.33.3 Member Function Documentation	119
3.33.3.1 print()	119
3.33.4 Field Documentation	119
3.33.4.1 edgen	119
3.33.4.2 det	120
3.33.4.3 nplist	120
3.33.4.4 plist	120
3.34 ExPrEsSiOn Struct Reference	120
3.34.1 Detailed Description	121
3.34.2 Field Documentation	121
3.34.2.1 onfile	121
3.34.2.2 prototype	121
3.34.2.3 size	121
3.34.2.4 renum	122

3.34.2.5 bracketinfo	122
3.34.2.6 newbracketinfo	122
3.34.2.7 renumlists	122
3.34.2.8 inmem	122
3.34.2.9 counter	122
3.34.2.10 name	122
3.34.2.11 hidelevel	123
3.34.2.12 vflags	123
3.34.2.13 printflag	123
3.34.2.14 status	123
3.34.2.15 replace	123
3.34.2.16 node	123
3.34.2.17 whichbuffer	123
3.34.2.18 namesize	123
3.34.2.19 compression	124
3.34.2.20 numdummies	124
3.34.2.21 numfactors	124
3.34.2.22 sizeprototype	124
3.34.2.23 uflags	124
3.35 FaCdOILaR Struct Reference	124
3.35.1 Detailed Description	124
3.35.2 Field Documentation	125
3.35.2.1 where	125
3.35.2.2 size	125
3.35.2.3 type	125
3.35.2.4 value	125
3.36 factorized_poly Class Reference	125
3.36.1 Detailed Description	126
3.36.2 Member Function Documentation	126
3.36.2.1 add_factor()	126
3.36.2.2 tostring()	126
3.36.3 Field Documentation	126
3.36.3.1 factor	126
3.36.3.2 power	126
3.37 FGInput Struct Reference	126
3.37.1 Detailed Description	127
3.37.2 Field Documentation	127
3.37.2.1 ninitl	127
3.37.2.2 initln	127
3.37.2.3 initlc	127
3.37.2.4 nfinal	127
3.37.2.5 finaln	127

3.37.2.6 finalc	127
3.37.2.7 coupl	128
3.38 FiLe Struct Reference	128
3.38.1 Detailed Description	129
3.38.2 Field Documentation	129
3.38.2.1 PPosition	129
3.38.2.2 filesize	129
3.38.2.3 PObuffer	129
3.38.2.4 PStop	129
3.38.2.5 POfill	129
3.38.2.6 POfull	129
3.38.2.7 name	130
3.38.2.8 numblocks	130
3.38.2.9 inbuffer	130
3.38.2.10 PSize	130
3.38.2.11 handle	130
3.38.2.12 active	130
3.39 FiLeDaTa Struct Reference	131
3.39.1 Detailed Description	131
3.39.2 Field Documentation	131
3.39.2.1 Index	131
3.39.2.2 Fill	132
3.39.2.3 Position	132
3.39.2.4 Handle	132
3.39.2.5 dirtyflag	132
3.40 FiLeInDeX Struct Reference	132
3.40.1 Detailed Description	133
3.40.2 Field Documentation	133
3.40.2.1 next	133
3.40.2.2 number	133
3.40.2.3 expression	133
3.40.2.4 empty	134
3.41 FixedGlobals Struct Reference	134
3.41.1 Detailed Description	134
3.41.2 Field Documentation	134
3.41.2.1 Operation	134
3.41.2.2 OperaFind	135
3.41.2.3 VarType	135
3.41.2.4 ExprStat	135
3.41.2.5 FunNam	135
3.41.2.6 swmes	135
3.41.2.7 fname	135

3.41.2.8 fname2	135
3.41.2.9 s_one	135
3.41.2.10 framebase	136
3.41.2.11 fname2base	136
3.41.2.12 framesize	136
3.41.2.13 fname2size	136
3.41.2.14 cTable	136
3.42 fixedset Struct Reference	136
3.42.1 Detailed Description	136
3.42.2 Field Documentation	137
3.42.2.1 name	137
3.42.2.2 description	137
3.42.2.3 type	137
3.42.2.4 dimension	137
3.43 flint::fmpz Class Reference	137
3.43.1 Detailed Description	137
3.43.2 Constructor & Destructor Documentation	138
3.43.2.1 fmpz()	138
3.43.2.2 ~fmpz()	138
3.43.3 Member Function Documentation	138
3.43.3.1 print()	138
3.43.4 Field Documentation	138
3.43.4.1 d	138
3.44 Fraction Class Reference	138
3.44.1 Detailed Description	139
3.44.2 Constructor & Destructor Documentation	139
3.44.2.1 Fraction() [1/2]	139
3.44.2.2 Fraction() [2/2]	139
3.44.3 Member Function Documentation	139
3.44.3.1 print()	139
3.44.3.2 setValue() [1/2]	139
3.44.3.3 setValue() [2/2]	140
3.44.3.4 add() [1/2]	140
3.44.3.5 add() [2/2]	140
3.44.3.6 sub()	140
3.44.3.7 gcd()	140
3.44.3.8 normal()	140
3.44.3.9 isEq()	140
3.44.4 Field Documentation	141
3.44.4.1 num	141
3.44.4.2 den	141
3.44.4.3 ratio	141

3.45 FUN_INFO Struct Reference	141
3.45.1 Detailed Description	141
3.45.2 Field Documentation	142
3.45.2.1 location	142
3.45.2.2 numargs	142
3.45.2.3 numfunnies	142
3.45.2.4 numwildcards	142
3.45.2.5 symmet	142
3.45.2.6 tensor	142
3.45.2.7 commute	142
3.46 FuNcTiOn Struct Reference	143
3.46.1 Detailed Description	143
3.46.2 Field Documentation	144
3.46.2.1 tabl	144
3.46.2.2 symminfo	144
3.46.2.3 name	144
3.46.2.4 commute	144
3.46.2.5 complex	144
3.46.2.6 number	145
3.46.2.7 flags	145
3.46.2.8 spec	145
3.46.2.9 symmetric	145
3.46.2.10 node	145
3.46.2.11 namesize	146
3.46.2.12 dimension	146
3.46.2.13 maxnumargs	146
3.46.2.14 minnumargs	146
3.47 gcd_heuristic_failed Class Reference	146
3.47.1 Detailed Description	146
3.48 HANDLERS Struct Reference	146
3.48.1 Detailed Description	147
3.48.2 Field Documentation	147
3.48.2.1 newlogonly	147
3.48.2.2 newhandle	147
3.48.2.3 oldhandle	147
3.48.2.4 oldlogonly	147
3.48.2.5 oldprinttype	147
3.48.2.6 oldsilent	148
3.49 IInput Struct Reference	148
3.49.1 Detailed Description	148
3.49.2 Field Documentation	148
3.49.2.1 name	148

3.49.2.2 icode	148
3.49.2.3 nplistn	148
3.49.2.4 plistn	149
3.49.2.5 plistc	149
3.49.2.6 cvallist	149
3.50 InDeX Struct Reference	149
3.50.1 Detailed Description	149
3.50.2 Field Documentation	149
3.50.2.1 name	149
3.50.2.2 type	150
3.50.2.3 dimension	150
3.50.2.4 number	150
3.50.2.5 flags	150
3.50.2.6 nmin4	150
3.50.2.7 node	150
3.50.2.8 namesize	150
3.51 indexblock Struct Reference	151
3.51.1 Detailed Description	151
3.51.2 Field Documentation	151
3.51.2.1 flags	151
3.51.2.2 previousblock	151
3.51.2.3 position	151
3.51.2.4 objects	152
3.52 InDeXeNtRy Struct Reference	152
3.52.1 Detailed Description	152
3.52.2 Field Documentation	153
3.52.2.1 position	153
3.52.2.2 length	153
3.52.2.3 variables	153
3.52.2.4 CompressSize	153
3.52.2.5 nsymbols	153
3.52.2.6 nindices	154
3.52.2.7 nvectors	154
3.52.2.8 nfunctions	154
3.52.2.9 size	154
3.52.2.10 name	154
3.53 iniinfo Struct Reference	155
3.53.1 Detailed Description	155
3.53.2 Field Documentation	155
3.53.2.1 entriesinindex	155
3.53.2.2 numberofindexblocks	155
3.53.2.3 firstindexblock	155

3.53.2.4 lastindexblock	155
3.53.2.5 numberoftables	156
3.53.2.6 numberofnamesblocks	156
3.53.2.7 firstnameblock	156
3.53.2.8 lastnameblock	156
3.54 INSIDEINFO Struct Reference	156
3.54.1 Detailed Description	157
3.54.2 Field Documentation	157
3.54.2.1 buffer	157
3.54.2.2 oldcompiletype	157
3.54.2.3 oldparallelflag	157
3.54.2.4 oldnumpotmoddollars	157
3.54.2.5 size	157
3.54.2.6 numdollars	157
3.54.2.7 oldcbuf	158
3.54.2.8 oldrbuf	158
3.54.2.9 inscbuf	158
3.54.2.10 oldcnumlhs	158
3.55 Interaction Class Reference	158
3.55.1 Detailed Description	159
3.55.2 Constructor & Destructor Documentation	159
3.55.2.1 Interaction()	159
3.55.2.2 ~Interaction()	159
3.55.3 Member Function Documentation	160
3.55.3.1 prInteraction()	160
3.55.4 Field Documentation	160
3.55.4.1 mdl	160
3.55.4.2 name	160
3.55.4.3 plist	160
3.55.4.4 clist	160
3.55.4.5 slist	160
3.55.4.6 id	160
3.55.4.7 csum	161
3.55.4.8 nlegs	161
3.55.4.9 loop	161
3.55.4.10 icode	161
3.55.4.11 nplist	161
3.55.4.12 nclist	161
3.55.4.13 nslist	161
3.56 KEYWORD Struct Reference	162
3.56.1 Detailed Description	162
3.56.2 Field Documentation	162

3.56.2.1 name	162
3.56.2.2 func	162
3.56.2.3 type	162
3.56.2.4 flags	162
3.57 KEYWORDV Struct Reference	163
3.57.1 Detailed Description	163
3.57.2 Field Documentation	163
3.57.2.1 name	163
3.57.2.2 var	163
3.57.2.3 type	163
3.57.2.4 flags	163
3.58 LIST Struct Reference	164
3.58.1 Detailed Description	164
3.58.2 Field Documentation	164
3.58.2.1 lijst	164
3.58.2.2 message	164
3.58.2.3 num	164
3.58.2.4 maxnum	165
3.58.2.5 size	165
3.58.2.6 numglobal	165
3.58.2.7 numtemp	165
3.58.2.8 numclear	165
3.59 longMultiStruct Struct Reference	166
3.59.1 Detailed Description	166
3.59.2 Field Documentation	166
3.59.2.1 buffer	166
3.59.2.2 bufpos	166
3.59.2.3 packpos	166
3.59.2.4 nPacks	167
3.59.2.5 lastLen	167
3.59.2.6 next	167
3.60 M_const Struct Reference	167
3.60.1 Detailed Description	171
3.60.2 Field Documentation	171
3.60.2.1 zeropos	171
3.60.2.2 S0	171
3.60.2.3 gcmmod	171
3.60.2.4 gpowmod	171
3.60.2.5 TempDir	171
3.60.2.6 TempSortDir	172
3.60.2.7 IncDir	172
3.60.2.8 InputFileName	172

3.60.2.9 LogFileName	172
3.60.2.10 OutBuffer	172
3.60.2.11 Path	172
3.60.2.12 SetupDir	172
3.60.2.13 SetupFile	172
3.60.2.14 gFortran90Kind	173
3.60.2.15 gextrasym	173
3.60.2.16 ggextrasym	173
3.60.2.17 oldnumextrasymbols	173
3.60.2.18 SpectatorFiles	173
3.60.2.19 MaxTer	173
3.60.2.20 CompressSize	173
3.60.2.21 ScratSize	173
3.60.2.22 HideSize	174
3.60.2.23 SizeStoreCache	174
3.60.2.24 MaxStreamSize	174
3.60.2.25 SIOsize	174
3.60.2.26 SLargeSize	174
3.60.2.27 SSmallEsize	174
3.60.2.28 SSmallSize	174
3.60.2.29 STermsInSmall	174
3.60.2.30 MaxBracketBufferSize	175
3.60.2.31 hProcessBucketSize	175
3.60.2.32 gProcessBucketSize	175
3.60.2.33 shmWinSize	175
3.60.2.34 OldChildTime	175
3.60.2.35 OldSecTime	175
3.60.2.36 OldMilliTime	175
3.60.2.37 WorkSize	175
3.60.2.38 gThreadBucketSize	176
3.60.2.39 ggThreadBucketSize	176
3.60.2.40 SumTime	176
3.60.2.41 SpectatorSize	176
3.60.2.42 TimeLimit	176
3.60.2.43 FileOnlyFlag	176
3.60.2.44 Interact	176
3.60.2.45 MaxParLevel	176
3.60.2.46 OutBufSize	177
3.60.2.47 SMaxFpatches	177
3.60.2.48 SMaxPatches	177
3.60.2.49 StdOut	177
3.60.2.50 ginsidefirst	177

3.60.2.51 gDefDim	177
3.60.2.52 gDefDim4	177
3.60.2.53 NumFixedSets	177
3.60.2.54 NumFixedFunctions	178
3.60.2.55 rbufnum	178
3.60.2.56 dbufnum	178
3.60.2.57 sbufnum	178
3.60.2.58 zbufnum	178
3.60.2.59 SkipClears	178
3.60.2.60 gTokensWriteFlag	178
3.60.2.61 gfunpowers	178
3.60.2.62 gStatsFlag	179
3.60.2.63 gNamesFlag	179
3.60.2.64 gCodesFlag	179
3.60.2.65 gSortType	179
3.60.2.66 gproperorderflag	179
3.60.2.67 hparallelflag	179
3.60.2.68 gparallelflag	179
3.60.2.69 totalnumberofthreads	179
3.60.2.70 gSizeCommuteInSet	180
3.60.2.71 gThreadStats	180
3.60.2.72 ggThreadStats	180
3.60.2.73 gFinalStats	180
3.60.2.74 ggFinalStats	180
3.60.2.75 gThreadsFlag	180
3.60.2.76 ggThreadsFlag	180
3.60.2.77 gThreadBalancing	180
3.60.2.78 ggThreadBalancing	181
3.60.2.79 gThreadSortFileSynch	181
3.60.2.80 ggThreadSortFileSynch	181
3.60.2.81 gProcessStats	181
3.60.2.82 ggProcessStats	181
3.60.2.83 gOldParallelStats	181
3.60.2.84 ggOldParallelStats	181
3.60.2.85 maxFlevels	181
3.60.2.86 resetTimeOnClear	182
3.60.2.87 gcNumDollars	182
3.60.2.88 MultiRun	182
3.60.2.89 gNoSpacesInNumbers	182
3.60.2.90 ggNoSpacesInNumbers	182
3.60.2.91 gIsFortran90	182
3.60.2.92 PrintTotalSize	182

3.60.2.93 fbufferSize	182
3.60.2.94 gOldFactArgFlag	183
3.60.2.95 ggOldFactArgFlag	183
3.60.2.96 gnumextrasym	183
3.60.2.97 ggnumextrasym	183
3.60.2.98 NumSpectatorFiles	183
3.60.2.99 SizeForSpectatorFiles	183
3.60.2.100 gOldGCDflag	183
3.60.2.101 ggOldGCDflag	183
3.60.2.102 gWTimeStatsFlag	184
3.60.2.103 ggWTimeStatsFlag	184
3.60.2.104 jumpratio	184
3.60.2.105 MaxTal	184
3.60.2.106 IndDum	184
3.60.2.107 DumInd	184
3.60.2.108 Willnd	184
3.60.2.109 gncmod	184
3.60.2.110 gnpowmod	185
3.60.2.111 gmodmode	185
3.60.2.112 gUnitTrace	185
3.60.2.113 gOutputMode	185
3.60.2.114 gOutputSpaces	185
3.60.2.115 gOutNumberType	185
3.60.2.116 gCnumpows	185
3.60.2.117 gUniTrace	185
3.60.2.118 MaxWildcards	186
3.60.2.119 mTraceDum	186
3.60.2.120 OffsetIndex	186
3.60.2.121 OffsetVector	186
3.60.2.122 RepMax	186
3.60.2.123 LogType	186
3.60.2.124 ggStatsFlag	186
3.60.2.125 gLineLength	186
3.60.2.126 qError	187
3.60.2.127 FortranCont	187
3.60.2.128 HoldFlag	187
3.60.2.129 Ordering	187
3.60.2.130 silent	187
3.60.2.131 tracebackflag	187
3.60.2.132 expnum	187
3.60.2.133 denomnum	187
3.60.2.134 facnum	188

3.60.2.135	invfacnum	188
3.60.2.136	sumnum	188
3.60.2.137	sumpnum	188
3.60.2.138	OldOrderFlag	188
3.60.2.139	termfunnum	188
3.60.2.140	matchfunnum	188
3.60.2.141	countfunnum	188
3.60.2.142	gPolyFun	189
3.60.2.143	gPolyFunInv	189
3.60.2.144	gPolyFunType	189
3.60.2.145	gPolyFunExp	189
3.60.2.146	gPolyFunVar	189
3.60.2.147	gPolyFunPow	189
3.60.2.148	dollarzero	189
3.60.2.149	atstartup	189
3.60.2.150	exitflag	190
3.60.2.151	NumStoreCaches	190
3.60.2.152	gIndentSpace	190
3.60.2.153	ggIndentSpace	190
3.60.2.154	gShortStatsMax	190
3.60.2.155	ggShortStatsMax	190
3.60.2.156	gextrasymbols	190
3.60.2.157	ggextrasymbols	191
3.60.2.158	zerorhs	191
3.60.2.159	onerhs	191
3.60.2.160	havesortdir	191
3.60.2.161	vectorzero	191
3.60.2.162	ClearStore	191
3.60.2.163	BracketFactors	191
3.60.2.164	FromStdin	191
3.60.2.165	IgnoreDeprecation	192
3.61	MCBlock Class Reference	192
3.61.1	Detailed Description	192
3.61.2	Constructor & Destructor Documentation	192
3.61.2.1	MCBlock()	192
3.61.2.2	~MCBlock()	192
3.61.3	Member Function Documentation	193
3.61.3.1	init()	193
3.61.4	Field Documentation	193
3.61.4.1	edges	193
3.61.4.2	nmedges	193
3.61.4.3	nartps	193

3.61.4.4 loop	193
3.61.4.5 DUMMY_PADDING	193
3.62 MCBridge Class Reference	194
3.62.1 Detailed Description	194
3.62.2 Constructor & Destructor Documentation	194
3.62.2.1 MCBridge()	194
3.62.2.2 ~MCBridge()	194
3.62.3 Field Documentation	194
3.62.3.1 nodes	194
3.62.3.2 next	194
3.63 MCEdge Class Reference	195
3.63.1 Detailed Description	195
3.63.2 Constructor & Destructor Documentation	195
3.63.2.1 MCEdge()	195
3.63.2.2 ~MCEdge()	195
3.63.3 Field Documentation	195
3.63.3.1 nodes	195
3.63.3.2 momset	195
3.63.3.3 momdir	196
3.63.3.4 DUMMY_PADDING	196
3.64 MConn Class Reference	196
3.64.1 Detailed Description	197
3.64.2 Constructor & Destructor Documentation	197
3.64.2.1 MConn()	197
3.64.2.2 ~MConn()	197
3.64.3 Member Function Documentation	198
3.64.3.1 init()	198
3.64.3.2 initCEdges()	198
3.64.3.3 pushNode()	198
3.64.3.4 pushEdge()	198
3.64.3.5 addCEdge()	198
3.64.3.6 addOPlc()	198
3.64.3.7 addBridge()	199
3.64.3.8 addArtic()	199
3.64.3.9 addBlock()	199
3.64.3.10 addBlockSelf()	199
3.64.3.11 print()	199
3.64.3.12 prEdges()	199
3.64.4 Field Documentation	200
3.64.4.1 cedges	200
3.64.4.2 opics	200
3.64.4.3 bridges	200

3.64.4.4 blocks	200
3.64.4.5 articuls	200
3.64.4.6 opisp	200
3.64.4.7 opistk	200
3.64.4.8 blksp	201
3.64.4.9 blkstk	201
3.64.4.10 snodes	201
3.64.4.11 sedges	201
3.64.4.12 nopic	201
3.64.4.13 nlpopic	201
3.64.4.14 nctopic	201
3.64.4.15 nbacked	201
3.64.4.16 nbridges	202
3.64.4.17 ne0bridges	202
3.64.4.18 ne1bridges	202
3.64.4.19 nselfloops	202
3.64.4.20 nmultiedges	202
3.64.4.21 nblocks	202
3.64.4.22 na1blocks	202
3.64.4.23 narticuls	202
3.64.4.24 neblocks	203
3.64.4.25 nopisp	203
3.64.4.26 opistkptr	203
3.64.4.27 nblksp	203
3.64.4.28 blkstkptr	203
3.64.4.29 DUMMYPADDING	203
3.65 MCOpI Class Reference	203
3.65.1 Detailed Description	204
3.65.2 Constructor & Destructor Documentation	204
3.65.2.1 MCOpI()	204
3.65.2.2 ~MCOpI()	204
3.65.3 Member Function Documentation	204
3.65.3.1 init()	204
3.65.4 Field Documentation	204
3.65.4.1 nodes	204
3.65.4.2 nnodes	205
3.65.4.3 nlegs	205
3.65.4.4 next	205
3.65.4.5 nedges	205
3.65.4.6 loop	205
3.65.4.7 ctloop	205
3.65.4.8 mom0lg	205

3.65.4.9 DUMMYPADDING	206
3.66 MGraph Class Reference	206
3.66.1 Detailed Description	207
3.66.2 Constructor & Destructor Documentation	208
3.66.2.1 MGraph() [1/2]	208
3.66.2.2 MGraph() [2/2]	208
3.66.2.3 ~MGraph()	208
3.66.3 Member Function Documentation	208
3.66.3.1 init()	208
3.66.3.2 generate()	208
3.66.3.3 isExternal()	209
3.66.3.4 printAdjMat()	209
3.66.3.5 print()	209
3.66.3.6 printPy()	209
3.66.3.7 isConnected()	209
3.66.3.8 visit()	209
3.66.3.9 isIsomorphic()	209
3.66.3.10 permMat()	210
3.66.3.11 compMat()	210
3.66.3.12 refineClass()	210
3.66.3.13 bisearchME()	210
3.66.3.14 biconnME()	210
3.66.3.15 isOptM()	211
3.66.3.16 connectClass()	211
3.66.3.17 connectNode()	211
3.66.3.18 connectLeg()	211
3.66.3.19 newGraph()	211
3.66.4 Field Documentation	211
3.66.4.1 opt	211
3.66.4.2 group	212
3.66.4.3 orbits	212
3.66.4.4 nodes	212
3.66.4.5 mld	212
3.66.4.6 clist	212
3.66.4.7 pld	212
3.66.4.8 nNodes	212
3.66.4.9 nEdges	212
3.66.4.10 nLoops	213
3.66.4.11 nExtern	213
3.66.4.12 nClasses	213
3.66.4.13 mindeg	213
3.66.4.14 maxdeg	213

3.66.4.15 adjMat	213
3.66.4.16 curcl	213
3.66.4.17 egraph	213
3.66.4.18 nsym	214
3.66.4.19 esym	214
3.66.4.20 cDiag	214
3.66.4.21 c1PI	214
3.66.4.22 cNoTadpole	214
3.66.4.23 cNoTadBlock	214
3.66.4.24 c1PINoTadBlock	214
3.66.4.25 wscon	214
3.66.4.26 wsopi	215
3.66.4.27 ngen	215
3.66.4.28 ngconn	215
3.66.4.29 nCallRefine	215
3.66.4.30 discardRefine	215
3.66.4.31 discardDisc	215
3.66.4.32 discardIso	215
3.66.4.33 opi	215
3.66.4.34 opiloop	216
3.66.4.35 extself	216
3.66.4.36 selfloop	216
3.66.4.37 multiedge	216
3.66.4.38 tadpole	216
3.66.4.39 tadblock	216
3.66.4.40 block	216
3.66.4.41 bipart	216
3.66.4.42 DUMMYPADDING1	217
3.66.4.43 mconn	217
3.66.4.44 modmat	217
3.66.4.45 bidef	217
3.66.4.46 bilow	217
3.66.4.47 bicol	217
3.66.4.48 bicount	217
3.66.4.49 DUMMYPADDING2	218
3.67 MiNmAx Struct Reference	218
3.67.1 Detailed Description	218
3.67.2 Field Documentation	218
3.67.2.1 mini	218
3.67.2.2 maxi	218
3.67.2.3 size	219
3.68 MInput Struct Reference	219

3.68.1 Detailed Description	219
3.68.2 Field Documentation	219
3.68.2.1 name	219
3.68.2.2 defpart	219
3.68.2.3 ncouple	219
3.68.2.4 cnamlist	220
3.69 MNode Class Reference	220
3.69.1 Detailed Description	220
3.69.2 Constructor & Destructor Documentation	220
3.69.2.1 MNode() [1/2]	220
3.69.2.2 MNode() [2/2]	221
3.69.3 Field Documentation	221
3.69.3.1 id	221
3.69.3.2 deg	221
3.69.3.3 clss	221
3.69.3.4 extloop	221
3.69.3.5 cmindeg	221
3.69.3.6 cmaxdeg	222
3.69.3.7 freelg	222
3.69.3.8 visited	222
3.70 MNodeClass Class Reference	222
3.70.1 Detailed Description	223
3.70.2 Constructor & Destructor Documentation	223
3.70.2.1 MNodeClass()	223
3.70.2.2 ~MNodeClass()	223
3.70.3 Member Function Documentation	223
3.70.3.1 init()	223
3.70.3.2 copy()	223
3.70.3.3 clCmp()	223
3.70.3.4 printMat()	224
3.70.3.5 mkFlist()	224
3.70.3.6 mkNdCI()	224
3.70.3.7 mkCI Mat()	224
3.70.3.8 incMat()	224
3.70.3.9 cmpMNCArry()	224
3.70.3.10 reorder()	225
3.70.4 Field Documentation	225
3.70.4.1 clmat	225
3.70.4.2 clist	225
3.70.4.3 ndcl	225
3.70.4.4 flist	225
3.70.4.5 clord	225

3.70.4.6 cmindeg	225
3.70.4.7 cmaxdeg	226
3.70.4.8 nNodes	226
3.70.4.9 nClasses	226
3.70.4.10 maxdeg	226
3.70.4.11 flg0	226
3.70.4.12 flg1	226
3.70.4.13 flg2	226
3.71 MoDeL Struct Reference	227
3.71.1 Detailed Description	227
3.71.2 Field Documentation	227
3.71.2.1 vertices	227
3.71.2.2 couplings	228
3.71.2.3 name	228
3.71.2.4 grccmodel	228
3.71.2.5 legcouple	228
3.71.2.6 nparticles	228
3.71.2.7 nvertices	228
3.71.2.8 invertices	228
3.71.2.9 sizevertices	228
3.71.2.10 sizecouplings	229
3.71.2.11 ncouplings	229
3.71.2.12 error	229
3.71.2.13 dummy	229
3.72 Model Class Reference	229
3.72.1 Detailed Description	231
3.72.2 Constructor & Destructor Documentation	231
3.72.2.1 Model()	231
3.72.2.2 ~Model()	231
3.72.3 Member Function Documentation	231
3.72.3.1 prModel()	231
3.72.3.2 addParticle()	231
3.72.3.3 addParticleEnd()	231
3.72.3.4 addInteraction()	232
3.72.3.5 addInteractionEnd()	232
3.72.3.6 findParticleName()	232
3.72.3.7 findParticleCode()	232
3.72.3.8 findInteractionName()	232
3.72.3.9 findInteractionCode()	232
3.72.3.10 particleName()	232
3.72.3.11 particleCode()	233
3.72.3.12 normalParticle()	233

3.72.3.13 antiParticle()	233
3.72.3.14 allParticles()	233
3.72.3.15 findMClass()	233
3.72.3.16 prParticleArray()	233
3.72.3.17 printMInput()	234
3.72.3.18 printPInput()	234
3.72.3.19 printIInput()	234
3.72.4 Field Documentation	234
3.72.4.1 name	234
3.72.4.2 cnlist	234
3.72.4.3 ncouple	234
3.72.4.4 nParticles	234
3.72.4.5 particles	235
3.72.4.6 pdef	235
3.72.4.7 nallPart	235
3.72.4.8 allPart	235
3.72.4.9 nintPart	235
3.72.4.10 intPart	235
3.72.4.11 nInteracts	235
3.72.4.12 interacts	235
3.72.4.13 vdef	236
3.72.4.14 maxnlegs	236
3.72.4.15 maxcpl	236
3.72.4.16 maxloop	236
3.72.4.17 cplgcp	236
3.72.4.18 cplglg	236
3.72.4.19 cplgnvl	236
3.72.4.20 cplgvl	236
3.72.4.21 ncplgcp	237
3.72.4.22 defpart	237
3.73 MODNUM Struct Reference	237
3.73.1 Detailed Description	237
3.73.2 Field Documentation	237
3.73.2.1 a	237
3.73.2.2 m	237
3.73.2.3 na	238
3.73.2.4 nm	238
3.74 MoDoPtDoLIaRS Struct Reference	238
3.74.1 Detailed Description	238
3.74.2 Field Documentation	238
3.74.2.1 number	238
3.74.2.2 type	238

3.75 monomial_larger Class Reference	239
3.75.1 Detailed Description	239
3.75.2 Constructor & Destructor Documentation	239
3.75.2.1 monomial_larger()	239
3.75.3 Member Function Documentation	239
3.75.3.1 operator()()	239
3.76 MOrbits Class Reference	239
3.76.1 Detailed Description	240
3.76.2 Constructor & Destructor Documentation	240
3.76.2.1 MOrbits()	240
3.76.2.2 ~MOrbits()	240
3.76.3 Member Function Documentation	240
3.76.3.1 print()	240
3.76.3.2 initPerm()	240
3.76.3.3 fromPerm()	240
3.76.3.4 toOrbits()	241
3.76.4 Field Documentation	241
3.76.4.1 nOrbits	241
3.76.4.2 nNodes	241
3.76.4.3 nd2or	241
3.76.4.4 or2nd	241
3.76.4.5 flist	241
3.77 flint::mpoly Class Reference	242
3.77.1 Detailed Description	242
3.77.2 Constructor & Destructor Documentation	242
3.77.2.1 mpoly()	242
3.77.2.2 ~mpoly()	242
3.77.3 Member Function Documentation	242
3.77.3.1 print()	242
3.77.4 Field Documentation	242
3.77.4.1 d	242
3.78 flint::mpoly_ctx Class Reference	243
3.78.1 Detailed Description	243
3.78.2 Constructor & Destructor Documentation	243
3.78.2.1 mpoly_ctx()	243
3.78.2.2 ~mpoly_ctx()	243
3.78.3 Field Documentation	243
3.78.3.1 d	243
3.79 flint::mpoly_factor Class Reference	243
3.79.1 Detailed Description	244
3.79.2 Constructor & Destructor Documentation	244
3.79.2.1 mpoly_factor()	244

3.79.2.2 ~mpoly_factor()	244
3.79.3 Field Documentation	244
3.79.3.1 d	244
3.80 N_const Struct Reference	245
3.80.1 Detailed Description	247
3.80.2 Field Documentation	247
3.80.2.1 OldPosIn	247
3.80.2.2 OldPosOut	248
3.80.2.3 theposition	248
3.80.2.4 EndNest	248
3.80.2.5 Frozen	248
3.80.2.6 FullProto	248
3.80.2.7 cTerm	248
3.80.2.8 RepPoint	248
3.80.2.9 WildValue	248
3.80.2.10 WildStop	249
3.80.2.11 argaddress	249
3.80.2.12 RepFunList	249
3.80.2.13 patstop	249
3.80.2.14 terstop	249
3.80.2.15 terstart	249
3.80.2.16 terfirstcomm	249
3.80.2.17 DumFound	249
3.80.2.18 DumPlace	250
3.80.2.19 DumFunPlace	250
3.80.2.20 UsedSymbol	250
3.80.2.21 UsedVector	250
3.80.2.22 UsedIndex	250
3.80.2.23 UsedFunction	250
3.80.2.24 MaskPointer	250
3.80.2.25 ForFindOnly	250
3.80.2.26 findTerm	251
3.80.2.27 findPattern	251
3.80.2.28 dummyrenumlist	251
3.80.2.29 funargs	251
3.80.2.30 funlocs	251
3.80.2.31 funinds	251
3.80.2.32 NoScrat2	251
3.80.2.33 ReplaceScrat	251
3.80.2.34 tracestack	252
3.80.2.35 selecttermundo	252
3.80.2.36 patternbuffer	252

3.80.2.37	termbuffer	252
3.80.2.38	PoinScratch	252
3.80.2.39	FunScratch	252
3.80.2.40	RenumScratch	252
3.80.2.41	FunInfo	252
3.80.2.42	SplitScratch	253
3.80.2.43	SplitScratch1	253
3.80.2.44	FunSorts	253
3.80.2.45	SoScratC	253
3.80.2.46	listinprint	253
3.80.2.47	currentTerm	253
3.80.2.48	arglist	253
3.80.2.49	tlistbuf	253
3.80.2.50	compressSpace	254
3.80.2.51	SHcombi	254
3.80.2.52	poly_vars	254
3.80.2.53	cmod	254
3.80.2.54	SHvar	254
3.80.2.55	deferskipped	254
3.80.2.56	InScratch	254
3.80.2.57	SplitScratchSize	254
3.80.2.58	InScratch1	255
3.80.2.59	SplitScratchSize1	255
3.80.2.60	ninterms	255
3.80.2.61	SHcombisize	255
3.80.2.62	NumTotWildArgs	255
3.80.2.63	UseFindOnly	255
3.80.2.64	UsedOtherFind	255
3.80.2.65	ErrorInDollar	255
3.80.2.66	numfargs	256
3.80.2.67	numflocs	256
3.80.2.68	nargs	256
3.80.2.69	tohunt	256
3.80.2.70	numoffuns	256
3.80.2.71	funisize	256
3.80.2.72	RSize	256
3.80.2.73	numtracesctack	256
3.80.2.74	intracestack	257
3.80.2.75	numfuninfo	257
3.80.2.76	NumFunSorts	257
3.80.2.77	MaxFunSorts	257
3.80.2.78	arglistsize	257

3.80.2.79 tlistsize	257
3.80.2.80 filenum	257
3.80.2.81 compressSize	257
3.80.2.82 polysortflag	258
3.80.2.83 nogroundlevel	258
3.80.2.84 subsubveto	258
3.80.2.85 MaxRenumScratch	258
3.80.2.86 oldtype	258
3.80.2.87 oldvalue	258
3.80.2.88 NumWild	258
3.80.2.89 RepFunNum	258
3.80.2.90 DisOrderFlag	259
3.80.2.91 WildDirt	259
3.80.2.92 NumFound	259
3.80.2.93 WildReserve	259
3.80.2.94 TelnFun	259
3.80.2.95 TeSuOut	259
3.80.2.96 WildArgs	259
3.80.2.97 WildEat	259
3.80.2.98 PolyNormFlag	260
3.80.2.99 PolyFunTodo	260
3.80.2.100 sizeselecttermundo	260
3.80.2.101 patternbuffersize	260
3.80.2.102 numlistinprint	260
3.80.2.103 ncmod	260
3.80.2.104 ExpectedSign	260
3.80.2.105 SignCheck	261
3.80.2.106 IndDum	261
3.80.2.107 poly_num_vars	261
3.80.2.108 idfunctionflag	261
3.80.2.109 poly_vars_type	261
3.80.2.110 tryterm	261
3.81 nameblock Struct Reference	262
3.81.1 Detailed Description	262
3.81.2 Field Documentation	262
3.81.2.1 previousblock	262
3.81.2.2 position	262
3.81.2.3 names	262
3.82 NaMeNode Struct Reference	262
3.82.1 Detailed Description	263
3.82.2 Field Documentation	263
3.82.2.1 name	263

3.82.2.2 parent	263
3.82.2.3 left	263
3.82.2.4 right	263
3.82.2.5 balance	264
3.82.2.6 type	264
3.82.2.7 number	264
3.83 NaMeSpAcE Struct Reference	264
3.83.1 Detailed Description	265
3.83.2 Field Documentation	265
3.83.2.1 previous	265
3.83.2.2 next	265
3.83.2.3 usernames	265
3.83.2.4 name	265
3.84 NaMeTree Struct Reference	266
3.84.1 Detailed Description	266
3.84.2 Field Documentation	266
3.84.2.1 namenode	266
3.84.2.2 namebuffer	267
3.84.2.3 nodesize	267
3.84.2.4 nodefill	267
3.84.2.5 namesize	267
3.84.2.6 namefill	267
3.84.2.7 oldnamefill	267
3.84.2.8 oldnodefill	268
3.84.2.9 globalnamefill	268
3.84.2.10 globalnodefill	268
3.84.2.11 clearnamefill	268
3.84.2.12 clearnodefill	268
3.84.2.13 headnode	268
3.85 NCand Class Reference	269
3.85.1 Detailed Description	269
3.85.2 Constructor & Destructor Documentation	269
3.85.2.1 NCand()	269
3.85.2.2 ~NCand()	269
3.85.3 Member Function Documentation	269
3.85.3.1 prNCand()	269
3.85.4 Field Documentation	270
3.85.4.1 deg	270
3.85.4.2 st	270
3.85.4.3 nilist	270
3.85.4.4 ilist	270
3.86 NCInput Struct Reference	270

3.86.1 Detailed Description	270
3.86.2 Field Documentation	271
3.86.2.1 cldeg	271
3.86.2.2 clnum	271
3.86.2.3 cltyp	271
3.86.2.4 cmind	271
3.86.2.5 cmaxd	271
3.86.2.6 ptcl	271
3.86.2.7 cple	271
3.87 NeStInG Struct Reference	272
3.87.1 Detailed Description	272
3.87.2 Field Documentation	272
3.87.2.1 termsize	272
3.87.2.2 funsize	272
3.87.2.3 argsize	272
3.88 NoDe Struct Reference	273
3.88.1 Detailed Description	273
3.88.2 Field Documentation	273
3.88.2.1 left	273
3.88.2.2 right	273
3.88.2.3 lloser	273
3.88.2.4 rloser	274
3.88.2.5 lsrc	274
3.88.2.6 rsrc	274
3.89 node Struct Reference	274
3.89.1 Detailed Description	275
3.89.2 Constructor & Destructor Documentation	275
3.89.2.1 node() [1/2]	275
3.89.2.2 node() [2/2]	275
3.89.3 Member Function Documentation	275
3.89.3.1 cmp()	275
3.89.3.2 operator()()	275
3.89.3.3 calcHash()	275
3.89.4 Field Documentation	276
3.89.4.1 data	276
3.89.4.2 l	276
3.89.4.3 r	276
3.89.4.4 sign	276
3.89.4.5 hash	276
3.90 NodeEq Struct Reference	276
3.90.1 Detailed Description	276
3.90.2 Member Function Documentation	277

3.90.2.1 operator()	277
3.91 NodeHash Struct Reference	277
3.91.1 Detailed Description	277
3.91.2 Member Function Documentation	277
3.91.2.1 operator()	277
3.92 NStack Class Reference	277
3.92.1 Detailed Description	278
3.92.2 Constructor & Destructor Documentation	278
3.92.2.1 NStack()	278
3.92.2.2 ~NStack()	278
3.92.3 Member Function Documentation	278
3.92.3.1 print()	278
3.92.4 Field Documentation	278
3.92.4.1 noden	278
3.92.4.2 deg	279
3.92.4.3 st	279
3.92.4.4 nilist	279
3.92.4.5 ilist	279
3.93 O_const Struct Reference	279
3.93.1 Detailed Description	281
3.93.2 Field Documentation	281
3.93.2.1 SaveData	281
3.93.2.2 SaveHeader	281
3.93.2.3 OptimizeResult	281
3.93.2.4 OutputLine	281
3.93.2.5 OutStop	282
3.93.2.6 OutFill	282
3.93.2.7 bracket	282
3.93.2.8 termbuf	282
3.93.2.9 tabstring	282
3.93.2.10 wpos	282
3.93.2.11 wpoin	282
3.93.2.12 DollarOutBuffer	282
3.93.2.13 CurBufWrt	283
3.93.2.14 FlipWORD	283
3.93.2.15 FlipLONG	283
3.93.2.16 FlipPOS	283
3.93.2.17 FlipPOINTER	283
3.93.2.18 ResizeData	283
3.93.2.19 ResizeWORD	283
3.93.2.20 ResizeNCWORD	283
3.93.2.21 ResizeLONG	284

3.93.2.22 ResizePOS	284
3.93.2.23 ResizePOINTER	284
3.93.2.24 CheckPower	284
3.93.2.25 RenumberVec	284
3.93.2.26 Dictionaries	284
3.93.2.27 tensorList	284
3.93.2.28 inscheme	284
3.93.2.29 NumInBrack	285
3.93.2.30 wlen	285
3.93.2.31 DollarOutSizeBuffer	285
3.93.2.32 DollarInOutBuffer	285
3.93.2.33 Optimize	285
3.93.2.34 OutInBuffer	285
3.93.2.35 NoSpacesInNumbers	285
3.93.2.36 BlockSpaces	285
3.93.2.37 CurrentDictionary	286
3.93.2.38 SizeDictionaries	286
3.93.2.39 NumDictionaries	286
3.93.2.40 CurDictNumbers	286
3.93.2.41 CurDictVariables	286
3.93.2.42 CurDictSpecials	286
3.93.2.43 CurDictFunWithArgs	286
3.93.2.44 CurDictNumberWarning	286
3.93.2.45 CurDictNotInFunctions	287
3.93.2.46 CurDictInDollars	287
3.93.2.47 gNumDictionaries	287
3.93.2.48 IndentSpace	287
3.93.2.49 schemenum	287
3.93.2.50 transFlag	287
3.93.2.51 powerFlag	287
3.93.2.52 mpower	287
3.93.2.53 resizeFlag	288
3.93.2.54 bufferedInd	288
3.93.2.55 OutSkip	288
3.93.2.56 IsBracket	288
3.93.2.57 InFbrack	288
3.93.2.58 PrintType	288
3.93.2.59 FortFirst	288
3.93.2.60 DoubleFlag	288
3.93.2.61 FactorMode	289
3.93.2.62 FactorNum	289
3.93.2.63 ErrorBlock	289

3.93.2.64 OptimizationLevel	289
3.93.2.65 FortDotChar	289
3.94 objects Struct Reference	289
3.94.1 Detailed Description	290
3.94.2 Field Documentation	290
3.94.2.1 position	290
3.94.2.2 size	290
3.94.2.3 date	290
3.94.2.4 tablenumber	290
3.94.2.5 uncompressed	290
3.94.2.6 spare1	290
3.94.2.7 spare2	291
3.94.2.8 spare3	291
3.94.2.9 element	291
3.95 OptDef Struct Reference	291
3.95.1 Detailed Description	291
3.95.2 Field Documentation	291
3.95.2.1 name	291
3.95.2.2 mean	292
3.95.2.3 defaultv	292
3.95.2.4 DUMMYPPADDING	292
3.96 optimization Class Reference	292
3.96.1 Detailed Description	293
3.96.2 Description	293
3.96.3 Member Function Documentation	293
3.96.3.1 operator<()	293
3.96.4 Field Documentation	293
3.96.4.1 type	293
3.96.4.2 arg1	294
3.96.4.3 arg2	294
3.96.4.4 improve	294
3.96.4.5 coeff	294
3.96.4.6 eqnidxs	294
3.97 OPTIMIZE Struct Reference	295
3.97.1 Detailed Description	295
3.97.2 Field Documentation	295
3.97.2.1 fval	295
3.97.2.2 ival	295
3.97.2.3 horner	296
3.97.2.4 hornerdirection	296
3.97.2.5 method	296
3.97.2.6 mctstimelimit	296

3.97.2.7 mctsnumexpand	296
3.97.2.8 mctsnumkeep	296
3.97.2.9 mctsnumrepeat	296
3.97.2.10 greedytimelimit	296
3.97.2.11 greedyminnum	297
3.97.2.12 greedymaxperc	297
3.97.2.13 printstats	297
3.97.2.14 debugflags	297
3.97.2.15 schemeflags	297
3.97.2.16 mctsdecaymode	297
3.97.2.17 salter	297
3.97.2.18 spare	298
3.98 OPTIMIZERESULT Struct Reference	298
3.98.1 Detailed Description	298
3.98.2 Field Documentation	298
3.98.2.1 code	298
3.98.2.2 nameofexpr	298
3.98.2.3 codesize	298
3.98.2.4 exprnr	299
3.98.2.5 minvar	299
3.98.2.6 maxvar	299
3.99 Options Class Reference	299
3.99.1 Detailed Description	300
3.99.2 Constructor & Destructor Documentation	301
3.99.2.1 Options()	301
3.99.2.2 ~Options()	301
3.99.3 Member Function Documentation	301
3.99.3.1 setDefaultValues()	301
3.99.3.2 setOldDefaultValues()	301
3.99.3.3 setValue()	301
3.99.3.4 getValue()	301
3.99.3.5 setQGrafOpt()	302
3.99.3.6 print()	302
3.99.3.7 setOutputF()	302
3.99.3.8 setOutputP()	302
3.99.3.9 printLevel()	302
3.99.3.10 printModel()	302
3.99.3.11 setOutMG()	303
3.99.3.12 setOutAG()	303
3.99.3.13 setEndMG()	303
3.99.3.14 setErExit()	303
3.99.3.15 getDef()	303

3.99.3.16 getOldDef()	303
3.99.3.17 getQGDef()	304
3.99.3.18 begin()	304
3.99.3.19 end()	304
3.99.3.20 beginProc()	304
3.99.3.21 endProc()	304
3.99.3.22 beginSubProc()	304
3.99.3.23 endSubProc()	304
3.99.3.24 newMGraph()	305
3.99.3.25 newAGraph()	305
3.99.3.26 outModel()	305
3.99.4 Field Documentation	305
3.99.4.1 model	305
3.99.4.2 proc	305
3.99.4.3 sproc	305
3.99.4.4 out	305
3.99.4.5 outmg	306
3.99.4.6 endmg	306
3.99.4.7 outag	306
3.99.4.8 argmg	306
3.99.4.9 argemg	306
3.99.4.10 argag	306
3.99.4.11 qqref	306
3.99.4.12 values	306
3.99.4.13 qqopt	307
3.99.4.14 nqqopt	307
3.99.4.15 DUMMYPADDING	307
3.99.4.16 time0	307
3.99.4.17 time1	307
3.100 OptQGDef Struct Reference	307
3.100.1 Detailed Description	307
3.100.2 Field Documentation	308
3.100.2.1 name	308
3.100.2.2 cname	308
3.100.2.3 mean	308
3.101 OptQGRef Struct Reference	308
3.101.1 Detailed Description	308
3.101.2 Field Documentation	308
3.101.2.1 name	308
3.101.2.2 index	309
3.101.2.3 sign	309
3.102 Output Class Reference	309

3.102.1 Detailed Description	310
3.102.2 Constructor & Destructor Documentation	310
3.102.2.1 Output()	310
3.102.2.2 ~Output()	310
3.102.3 Member Function Documentation	310
3.102.3.1 setOutgrf()	310
3.102.3.2 setOutgrp()	310
3.102.3.3 outBeginF()	311
3.102.3.4 outBeginP()	311
3.102.3.5 outEndF()	311
3.102.3.6 outEndP()	311
3.102.3.7 outProcBeginF()	311
3.102.3.8 outProcBeginP()	311
3.102.3.9 outProcBegin0()	312
3.102.3.10 outSProcBeginF()	312
3.102.3.11 outSProcBeginP()	312
3.102.3.12 outProcEndF()	312
3.102.3.13 outProcEndP()	312
3.102.3.14 outEGraphF()	312
3.102.3.15 outEGraphP()	312
3.102.3.16 outModelF()	313
3.102.3.17 outModelP()	313
3.102.4 Field Documentation	313
3.102.4.1 opt	313
3.102.4.2 model	313
3.102.4.3 proc	313
3.102.4.4 sproc	313
3.102.4.5 outgrf	313
3.102.4.6 outgrfp	314
3.102.4.7 outgrp	314
3.102.4.8 outgrpp	314
3.102.4.9 proclد	314
3.102.4.10 outproc	314
3.103 P_const Struct Reference	315
3.103.1 Detailed Description	316
3.103.2 Field Documentation	316
3.103.2.1 DollarList	316
3.103.2.2 PreVarList	316
3.103.2.3 LoopList	317
3.103.2.4 ProcList	317
3.103.2.5 inside	317
3.103.2.6 firstnamespace	317

3.103.2.7 lastnameSpace	317
3.103.2.8 fullname	317
3.103.2.9 PreSwitchStrings	317
3.103.2.10 preStart	317
3.103.2.11 preStop	318
3.103.2.12 preFill	318
3.103.2.13 procedureExtension	318
3.103.2.14 cprocedureExtension	318
3.103.2.15 PreAssignStack	318
3.103.2.16 PrefsStack	318
3.103.2.17 PreSwitchModes	318
3.103.2.18 PreTypes	318
3.103.2.19 StopWatchZero	319
3.103.2.20 InOutBuf	319
3.103.2.21 pSize	319
3.103.2.22 PreAssignFlag	319
3.103.2.23 PreContinuation	319
3.103.2.24 PreproFlag	319
3.103.2.25 iBufError	319
3.103.2.26 PreOut	319
3.103.2.27 PreSwitchLevel	320
3.103.2.28 NumPreSwitchStrings	320
3.103.2.29 MaxPreTypes	320
3.103.2.30 NumPreTypes	320
3.103.2.31 MaxPrefsLevel	320
3.103.2.32 PrefsLevel	320
3.103.2.33 PreInsideLevel	320
3.103.2.34 DelayPrevar	320
3.103.2.35 AllowDelay	321
3.103.2.36 lhdollarerror	321
3.103.2.37 eat	321
3.103.2.38 gNumPre	321
3.103.2.39 PreDebug	321
3.103.2.40 OpenDictionary	321
3.103.2.41 PreAssignLevel	321
3.103.2.42 MaxPreAssignLevel	321
3.103.2.43 fullnamesize	322
3.103.2.44 DebugFlag	322
3.103.2.45 preError	322
3.103.2.46 ComChar	322
3.103.2.47 cComChar	322
3.104 ParallelVars Struct Reference	323

3.104.1 Detailed Description	323
3.104.2 Field Documentation	323
3.104.2.1 slavebuf	323
3.104.2.2 sbuf	324
3.104.2.3 rbufs	324
3.104.2.4 me	324
3.104.2.5 numtasks	324
3.104.2.6 parallel	324
3.104.2.7 rhsInParallel	324
3.104.2.8 mkSlaveInfile	324
3.104.2.9 exprbufsize	324
3.104.2.10 exptodo	325
3.104.2.11 log	325
3.104.2.12 numsbufs	325
3.104.2.13 numrbufs	325
3.105 PaRtl Struct Reference	325
3.105.1 Detailed Description	325
3.105.2 Field Documentation	326
3.105.2.1 psize	326
3.105.2.2 args	326
3.105.2.3 nargs	326
3.105.2.4 nfun	326
3.105.2.5 numargs	326
3.105.2.6 numpart	326
3.105.2.7 where	326
3.106 PaRtlcLe Struct Reference	327
3.106.1 Detailed Description	327
3.106.2 Field Documentation	327
3.106.2.1 number	327
3.106.2.2 spin	327
3.106.2.3 mass	327
3.106.2.4 type	327
3.107 Particle Class Reference	328
3.107.1 Detailed Description	329
3.107.2 Constructor & Destructor Documentation	329
3.107.2.1 Particle()	329
3.107.2.2 ~Particle()	329
3.107.3 Member Function Documentation	329
3.107.3.1 particleName()	329
3.107.3.2 particleCode()	329
3.107.3.3 aparticle()	329
3.107.3.4 prParticle()	330

3.107.3.5 isNeutral()	330
3.107.3.6 typeName()	330
3.107.3.7 typeGName()	330
3.107.4 Field Documentation	330
3.107.4.1 mdl	330
3.107.4.2 name	330
3.107.4.3 aname	330
3.107.4.4 id	331
3.107.4.5 ptype	331
3.107.4.6 neutral	331
3.107.4.7 pcode	331
3.107.4.8 acode	331
3.107.4.9 cmindeg	331
3.107.4.10 cmaxdeg	331
3.107.4.11 extonly	332
3.108 PeRmUtE Struct Reference	332
3.108.1 Detailed Description	332
3.108.2 Field Documentation	332
3.108.2.1 objects	332
3.108.2.2 sign	332
3.108.2.3 n	333
3.108.2.4 cycle	333
3.109 PeRmUtEp Struct Reference	333
3.109.1 Detailed Description	333
3.109.2 Field Documentation	333
3.109.2.1 objects	333
3.109.2.2 sign	333
3.109.2.3 n	334
3.109.2.4 cycle	334
3.110 PF_BUFFER Struct Reference	334
3.110.1 Detailed Description	334
3.110.2 Field Documentation	334
3.110.2.1 buff	334
3.110.2.2 fill	335
3.110.2.3 full	335
3.110.2.4 stop	335
3.110.2.5 status	335
3.110.2.6 retstat	335
3.110.2.7 request	335
3.110.2.8 type	335
3.110.2.9 index	335
3.110.2.10 tag	336

3.110.2.11 from	336
3.110.2.12 numbufs	336
3.110.2.13 active	336
3.111 PGIInput Struct Reference	336
3.111.1 Detailed Description	336
3.111.2 Field Documentation	337
3.111.2.1 cdeg	337
3.111.2.2 ctyp	337
3.111.2.3 cnum	337
3.111.2.4 cple	337
3.111.2.5 pname	337
3.111.2.6 pcode	337
3.112 PInput Struct Reference	338
3.112.1 Detailed Description	338
3.112.2 Field Documentation	338
3.112.2.1 name	338
3.112.2.2 aname	338
3.112.2.3 pcode	338
3.112.2.4 acode	338
3.112.2.5 ptypen	338
3.112.2.6 ptypec	339
3.112.2.7 extonly	339
3.113 PNodeClass Class Reference	339
3.113.1 Detailed Description	340
3.113.2 Constructor & Destructor Documentation	340
3.113.2.1 PNodeClass() [1/2]	340
3.113.2.2 PNodeClass() [2/2]	340
3.113.2.3 ~PNodeClass()	340
3.113.3 Member Function Documentation	340
3.113.3.1 prPNodeClass()	340
3.113.3.2 prElem()	341
3.113.4 Field Documentation	341
3.113.4.1 sproc	341
3.113.4.2 deg	341
3.113.4.3 type	341
3.113.4.4 particle	341
3.113.4.5 couple	341
3.113.4.6 cmindeg	341
3.113.4.7 cmaxdeg	342
3.113.4.8 count	342
3.113.4.9 cl2nd	342
3.113.4.10 nd2cl	342

3.113.4.11 cl2mcl	342
3.113.4.12 nnodes	342
3.113.4.13 nclass	342
3.114 flint::poly Class Reference	343
3.114.1 Detailed Description	343
3.114.2 Constructor & Destructor Documentation	343
3.114.2.1 poly()	343
3.114.2.2 ~poly()	343
3.114.3 Member Function Documentation	343
3.114.3.1 print()	343
3.114.4 Field Documentation	343
3.114.4.1 d	343
3.115 poly Class Reference	344
3.115.1 Detailed Description	345
3.115.2 Constructor & Destructor Documentation	345
3.115.2.1 poly() [1/3]	345
3.115.2.2 poly() [2/3]	346
3.115.2.3 poly() [3/3]	346
3.115.2.4 ~poly()	346
3.115.3 Member Function Documentation	346
3.115.3.1 operator+=(())	346
3.115.3.2 operator-=(())	346
3.115.3.3 operator*=(())	346
3.115.3.4 operator/=(())	347
3.115.3.5 operator%=(())	347
3.115.3.6 operator+()	347
3.115.3.7 operator-()	347
3.115.3.8 operator*()	347
3.115.3.9 operator/()	347
3.115.3.10 operator%()	347
3.115.3.11 operator==(())	348
3.115.3.12 operator"!=(())	348
3.115.3.13 operator=()	348
3.115.3.14 operator[]() [1/2]	348
3.115.3.15 operator[]() [2/2]	348
3.115.3.16 termscopy()	348
3.115.3.17 check_memory()	348
3.115.3.18 expand_memory()	349
3.115.3.19 is_zero()	349
3.115.3.20 is_one()	349
3.115.3.21 is_integer()	349
3.115.3.22 is_monomial()	349

3.115.3.23 is_dense_univariate()	349
3.115.4 Description	349
3.115.5 Notes	350
3.115.5.1 sign()	350
3.115.5.2 degree()	350
3.115.5.3 total_degree()	350
3.115.5.4 first_variable()	350
3.115.5.5 number_of_terms()	350
3.115.5.6 all_variables()	350
3.115.5.7 integer_lcoeff()	351
3.115.5.8 lcoeff_univar()	351
3.115.5.9 lcoeff_multivar()	351
3.115.5.10 coefficient()	351
3.115.5.11 derivative()	351
3.115.5.12 setmod()	351
3.115.5.13 coefficients_modulo()	351
3.115.5.14 simple_poly() ^[1/2]	352
3.115.5.15 simple_poly() ^[2/2]	352
3.115.5.16 get_variables()	352
3.115.5.17 argument_to_poly()	352
3.115.5.18 poly_to_argument()	352
3.115.5.19 poly_to_argument_with_den()	353
3.115.5.20 size_of_form_notation()	353
3.115.5.21 size_of_form_notation_with_den()	353
3.115.5.22 normalize()	353
3.115.5.23 from_coefficient_list()	353
3.115.5.24 to_coefficient_list()	353
3.115.5.25 coefficient_list_divmod()	354
3.115.5.26 to_string()	354
3.115.5.27 monomial_compare()	354
3.115.5.28 last_monomial_index()	354
3.115.5.29 last_monomial()	354
3.115.5.30 compare_degree_vector()	354
3.115.5.31 degree_vector()	354
3.115.5.32 add()	355
3.115.5.33 sub()	355
3.115.5.34 mul()	355
3.115.6 Description	355
3.115.6.1 div()	356
3.115.6.2 mod()	356
3.115.6.3 divmod()	356
3.115.7 Description	356

3.115.7.1 divides()	357
3.115.7.2 mul_one_term()	357
3.115.7.3 mul_univar()	357
3.115.7.4 mul_heap()	357
3.115.8 Description	357
3.115.8.1 divmod_one_term()	358
3.115.8.2 divmod_univar()	358
3.115.9 Description	359
3.115.9.1 divmod_heap()	359
3.115.10 Description	359
3.115.10.1 push_heap()	360
3.115.10.2 pop_heap()	360
3.115.11 Field Documentation	361
3.115.11.1 terms	361
3.115.11.2 size_of_terms	361
3.115.11.3 modp	361
3.115.11.4 modn	361
3.116 flint::poly_factor Class Reference	361
3.116.1 Detailed Description	361
3.116.2 Constructor & Destructor Documentation	362
3.116.2.1 poly_factor()	362
3.116.2.2 ~poly_factor()	362
3.116.3 Field Documentation	362
3.116.3.1 d	362
3.117 POLYMOD Struct Reference	362
3.117.1 Detailed Description	362
3.117.2 Field Documentation	363
3.117.2.1 coefs	363
3.117.2.2 numsym	363
3.117.2.3 arraysize	363
3.117.2.4 polysize	363
3.117.2.5 modnum	363
3.118 PoSiTiOn Struct Reference	363
3.118.1 Detailed Description	363
3.118.2 Field Documentation	364
3.118.2.1 p1	364
3.119 PRELOAD Struct Reference	364
3.119.1 Detailed Description	364
3.119.2 Field Documentation	364
3.119.2.1 buffer	364
3.119.2.2 size	364
3.120 pReVaR Struct Reference	365

3.120.1 Detailed Description	365
3.120.2 Field Documentation	365
3.120.2.1 name	365
3.120.2.2 value	365
3.120.2.3 argnames	365
3.120.2.4 nargs	366
3.120.2.5 wildarg	366
3.121 PROCEDURE Struct Reference	366
3.121.1 Detailed Description	367
3.121.2 Field Documentation	367
3.121.2.1 p	367
3.121.2.2 name	367
3.121.2.3 loadmode	367
3.121.2.4 mustfree	367
3.122 Process Class Reference	367
3.122.1 Detailed Description	368
3.122.2 Constructor & Destructor Documentation	369
3.122.2.1 Process()	369
3.122.2.2 ~Process()	369
3.122.3 Member Function Documentation	369
3.122.3.1 prProcess()	369
3.122.3.2 outProcP()	369
3.122.3.3 prProcessP()	369
3.122.3.4 mkSProcess()	370
3.122.4 Field Documentation	370
3.122.4.1 model	370
3.122.4.2 opt	370
3.122.4.3 mgrcount	370
3.122.4.4 agrcount	370
3.122.4.5 initlPart	370
3.122.4.6 finalPart	370
3.122.4.7 id	371
3.122.4.8 ninitl	371
3.122.4.9 nfinal	371
3.122.4.10 ctotat	371
3.122.4.11 nExtern	371
3.122.4.12 loop	371
3.122.4.13 maxnlegs	371
3.122.4.14 clist	371
3.122.4.15 nSubproc	372
3.122.4.16 sptbl	372
3.122.4.17 sproc	372

3.122.4.18 astack	372
3.122.4.19 ngraphs	372
3.122.4.20 nopi	372
3.122.4.21 wgraphs	372
3.122.4.22 wopi	372
3.122.4.23 nMGraphs	373
3.122.4.24 nMOPI	373
3.122.4.25 wMGraphs	373
3.122.4.26 wMOPI	373
3.122.4.27 nAGraphs	373
3.122.4.28 nAOPI	373
3.122.4.29 wAGraphs	373
3.122.4.30 wAOPI	373
3.122.4.31 sec	374
3.123 R_const Struct Reference	374
3.123.1 Detailed Description	376
3.123.2 Field Documentation	376
3.123.2.1 StoreData	376
3.123.2.2 Fscr	376
3.123.2.3 FoStage4	376
3.123.2.4 DefPosition	376
3.123.2.5 infile	376
3.123.2.6 outfile	376
3.123.2.7 hidefile	377
3.123.2.8 CompressBuffer	377
3.123.2.9 ComprTop	377
3.123.2.10 CompressPointer	377
3.123.2.11 CompareRoutine	377
3.123.2.12 wranfia	377
3.123.2.13 moebiustable	377
3.123.2.14 OldTime	377
3.123.2.15 InInBuf	378
3.123.2.16 InHiBuf	378
3.123.2.17 pWorkSize	378
3.123.2.18 lWorkSize	378
3.123.2.19 posWorkSize	378
3.123.2.20 wranfseed	378
3.123.2.21 NoCompress	378
3.123.2.22 gzipCompress	378
3.123.2.23 Cnumlhs	379
3.123.2.24 outtohide	379
3.123.2.25 wranfcall	379

3.123.2.26 wranfnpair1	379
3.123.2.27 wranfnpair2	379
3.123.2.28 GetFile	379
3.123.2.29 KeptInHold	379
3.123.2.30 BracketOn	379
3.123.2.31 MaxBracket	380
3.123.2.32 CurDum	380
3.123.2.33 DeferFlag	380
3.123.2.34 TePos	380
3.123.2.35 sLevel	380
3.123.2.36 Stage4Name	380
3.123.2.37 GetOneFile	380
3.123.2.38 PolyFun	380
3.123.2.39 PolyFunInv	381
3.123.2.40 PolyFunType	381
3.123.2.41 PolyFunExp	381
3.123.2.42 PolyFunVar	381
3.123.2.43 PolyFunPow	381
3.123.2.44 Eside	381
3.123.2.45 MaxDum	381
3.123.2.46 level	381
3.123.2.47 expchanged	382
3.123.2.48 expflags	382
3.123.2.49 CurExpr	382
3.123.2.50 SortType	382
3.123.2.51 ShortSortCount	382
3.123.2.52 modeloptions	382
3.123.2.53 funoffset	382
3.123.2.54 moebiustablesizes	383
3.124 ReNuMbEr Struct Reference	383
3.124.1 Detailed Description	383
3.124.2 Field Documentation	384
3.124.2.1 startposition	384
3.124.2.2 symb	384
3.124.2.3 indi	384
3.124.2.4 vect	384
3.124.2.5 func	384
3.124.2.6 symnum	385
3.124.2.7 indnum	385
3.124.2.8 vecnum	385
3.124.2.9 funnum	385
3.125 S_const Struct Reference	386

3.125.1 Detailed Description	386
3.125.2 Field Documentation	386
3.125.2.1 MaxExprSize	386
3.125.2.2 OldOnFile	387
3.125.2.3 OldNumFactors	387
3.125.2.4 Oldvflags	387
3.125.2.5 Olduflags	387
3.125.2.6 NumOldOnFile	387
3.125.2.7 NumOldNumFactors	387
3.125.2.8 MultiThreaded	387
3.125.2.9 Balancing	387
3.125.2.10 ExecMode	388
3.125.2.11 CollectOverFlag	388
3.126 SeTs Struct Reference	388
3.126.1 Detailed Description	388
3.126.2 Field Documentation	388
3.126.2.1 name	388
3.126.2.2 type	388
3.126.2.3 first	389
3.126.2.4 last	389
3.126.2.5 node	389
3.126.2.6 namesize	389
3.126.2.7 dimension	389
3.126.2.8 flags	389
3.127 SETUPPARAMETERS Struct Reference	389
3.127.1 Detailed Description	390
3.127.2 Field Documentation	390
3.127.2.1 parameter	390
3.127.2.2 type	390
3.127.2.3 flags	390
3.127.2.4 value	390
3.128 SGroup Class Reference	391
3.128.1 Detailed Description	391
3.128.2 Constructor & Destructor Documentation	391
3.128.2.1 SGroup()	391
3.128.2.2 ~SGroup()	391
3.128.3 Member Function Documentation	392
3.128.3.1 print()	392
3.128.3.2 newGroup()	392
3.128.3.3 clearGroup()	392
3.128.3.4 delGroup()	392
3.128.3.5 genNext()	392

3.128.3.6 addGroup()	392
3.128.3.7 nElem()	393
3.128.3.8 nextElem()	393
3.128.4 Field Documentation	393
3.128.4.1 size	393
3.128.4.2 nelem	393
3.128.4.3 elem	393
3.128.4.4 nnodes	393
3.128.4.5 neclass	393
3.128.4.6 eclass	394
3.128.4.7 cgen	394
3.128.4.8 csav	394
3.128.4.9 permg	394
3.128.4.10 perms	394
3.128.4.11 pgr	394
3.128.4.12 pgq	394
3.128.4.13 psr	394
3.128.4.14 psq	395
3.129 SHvariables Struct Reference	395
3.129.1 Detailed Description	395
3.129.2 Field Documentation	395
3.129.2.1 outterm	395
3.129.2.2 outfun	395
3.129.2.3 incoef	396
3.129.2.4 stop1	396
3.129.2.5 stop2	396
3.129.2.6 ststop1	396
3.129.2.7 ststop2	396
3.129.2.8 finishuf	396
3.129.2.9 do_uffle	396
3.129.2.10 combilast	396
3.129.2.11 nincoef	397
3.129.2.12 level	397
3.129.2.13 thefunction	397
3.129.2.14 option	397
3.130 sOrT Struct Reference	397
3.130.1 Detailed Description	399
3.130.2 Field Documentation	399
3.130.2.1 file	399
3.130.2.2 SizeInFile	399
3.130.2.3 IBuffer	399
3.130.2.4 ITop	399

3.130.2.5 IFill	399
3.130.2.6 used	399
3.130.2.7 sBuffer	400
3.130.2.8 sTop	400
3.130.2.9 sTop2	400
3.130.2.10 sHalf	400
3.130.2.11 sFill	400
3.130.2.12 sPointer	400
3.130.2.13 PoinFill	400
3.130.2.14 cBuffer	400
3.130.2.15 Patches	401
3.130.2.16 pStop	401
3.130.2.17 poina	401
3.130.2.18 poin2a	401
3.130.2.19 ktoi	401
3.130.2.20 tree	401
3.130.2.21 fPatches	401
3.130.2.22 inPatches	401
3.130.2.23 fPatchesStop	402
3.130.2.24 iPatches	402
3.130.2.25 f	402
3.130.2.26 ff	402
3.130.2.27 sTerms	402
3.130.2.28 LargeSize	402
3.130.2.29 SmallSize	402
3.130.2.30 SmallEsize	402
3.130.2.31 TermsInSmall	403
3.130.2.32 Terms2InSmall	403
3.130.2.33 GenTerms	403
3.130.2.34 TermsLeft	403
3.130.2.35 GenSpace	403
3.130.2.36 SpaceLeft	403
3.130.2.37 putinsize	403
3.130.2.38 ninterms	403
3.130.2.39 MaxPatches	404
3.130.2.40 MaxFpatches	404
3.130.2.41 type	404
3.130.2.42 IPatch	404
3.130.2.43 fPatchN1	404
3.130.2.44 PolyWise	404
3.130.2.45 PolyFlag	404
3.130.2.46 cBufferSize	404

3.130.2.47 maxtermsize	405
3.130.2.48 newmaxtermsize	405
3.130.2.49 outputmode	405
3.130.2.50 stagelevel	405
3.130.2.51 fPatchN	405
3.130.2.52 inNum	405
3.130.2.53 stage4	405
3.131 SpecTatoR Struct Reference	406
3.131.1 Detailed Description	406
3.131.2 Field Documentation	406
3.131.2.1 position	406
3.131.2.2 readpos	406
3.131.2.3 fh	407
3.131.2.4 name	407
3.131.2.5 exprnumber	407
3.131.2.6 flags	407
3.132 SProcess Class Reference	407
3.132.1 Detailed Description	409
3.132.2 Constructor & Destructor Documentation	409
3.132.2.1 SProcess() [1/2]	409
3.132.2.2 SProcess() [2/2]	409
3.132.2.3 ~SProcess()	409
3.132.3 Member Function Documentation	409
3.132.3.1 prSProcess()	409
3.132.3.2 generate()	410
3.132.3.3 assign()	410
3.132.3.4 toMNodeClass()	410
3.132.3.5 match()	410
3.132.3.6 endMGraph()	410
3.132.3.7 endAGraph()	410
3.132.3.8 resultMGraph()	411
3.132.3.9 resultAGraph()	411
3.132.4 Field Documentation	411
3.132.4.1 model	411
3.132.4.2 proc	411
3.132.4.3 opt	411
3.132.4.4 pnclass	411
3.132.4.5 astack	412
3.132.4.6 mgraph	412
3.132.4.7 egraph	412
3.132.4.8 agraph	412
3.132.4.9 cl2nd	412

3.132.4.10 nd2cl	412
3.132.4.11 nclass	412
3.132.4.12 ninitl	412
3.132.4.13 nfinal	413
3.132.4.14 nvert	413
3.132.4.15 clist	413
3.132.4.16 id	413
3.132.4.17 loop	413
3.132.4.18 nNodes	413
3.132.4.19 nEdges	413
3.132.4.20 nExtern	413
3.132.4.21 ncouple	414
3.132.4.22 tCouple	414
3.132.4.23 DUMMY_PADDING	414
3.132.4.24 mgrcount	414
3.132.4.25 agrcount	414
3.132.4.26 extperm	414
3.132.4.27 nMGraphs	414
3.132.4.28 nMOPI	414
3.132.4.29 wMGraphs	415
3.132.4.30 wMOPI	415
3.132.4.31 nAGraphs	415
3.132.4.32 nAOPI	415
3.132.4.33 wAGraphs	415
3.132.4.34 wAOPI	415
3.133 StOrEcAcHe Struct Reference	416
3.133.1 Detailed Description	416
3.133.2 Field Documentation	416
3.133.2.1 position	416
3.133.2.2 toppos	416
3.133.2.3 next	417
3.133.2.4 buffer	417
3.134 STOREHEADER Struct Reference	417
3.134.1 Detailed Description	417
3.134.2 Field Documentation	418
3.134.2.1 headermark	418
3.134.2.2 lenWORD	418
3.134.2.3 lenLONG	418
3.134.2.4 lenPOS	418
3.134.2.5 lenPOINTER	418
3.134.2.6 endianness	419
3.134.2.7 sSym	419

3.134.2.8 sInd	419
3.134.2.9 sVec	419
3.134.2.10 sFun	419
3.134.2.11 maxpower	420
3.134.2.12 wildoffset	420
3.134.2.13 revision	420
3.134.2.14 reserved	420
3.135 Stream Struct Reference	420
3.135.1 Detailed Description	421
3.135.2 Field Documentation	421
3.135.2.1 fileposition	421
3.135.2.2 linenum	421
3.135.2.3 prevline	422
3.135.2.4 buffer	422
3.135.2.5 pointer	422
3.135.2.6 top	422
3.135.2.7 FoldName	422
3.135.2.8 name	422
3.135.2.9 pname	423
3.135.2.10 buffersize	423
3.135.2.11 bufferposition	423
3.135.2.12 inbuffer	423
3.135.2.13 previous	423
3.135.2.14 handle	423
3.135.2.15 type	423
3.135.2.16 prevars	424
3.135.2.17 previousNoShowInput	424
3.135.2.18 eqnum	424
3.135.2.19 afterwards	424
3.135.2.20 olddelay	424
3.135.2.21 oldnoshowninput	424
3.135.2.22 isnxtchar	424
3.135.2.23 nextchar	424
3.135.2.24 reserved	425
3.136 SuBbUf Struct Reference	425
3.136.1 Detailed Description	425
3.136.2 Field Documentation	425
3.136.2.1 subexpnum	425
3.136.2.2 buffernum	425
3.137 SWITCH Struct Reference	426
3.137.1 Detailed Description	426
3.137.2 Field Documentation	426

3.137.2.1 table	426
3.137.2.2 defaultcase	427
3.137.2.3 endswitch	427
3.137.2.4 typetable	427
3.137.2.5 maxcase	427
3.137.2.6 mincase	427
3.137.2.7 numcases	427
3.137.2.8 tablesizes	427
3.137.2.9 caseoffset	427
3.137.2.10 iflevel	428
3.137.2.11 whilelevel	428
3.137.2.12 nestingsum	428
3.137.2.13 padding	428
3.138 SWITCHTABLE Struct Reference	428
3.138.1 Detailed Description	428
3.138.2 Field Documentation	428
3.138.2.1 ncase	428
3.138.2.2 value	429
3.138.2.3 compbuffer	429
3.139 SyMbOl Struct Reference	429
3.139.1 Detailed Description	429
3.139.2 Field Documentation	429
3.139.2.1 name	429
3.139.2.2 minpower	429
3.139.2.3 maxpower	430
3.139.2.4 complex	430
3.139.2.5 number	430
3.139.2.6 flags	430
3.139.2.7 node	430
3.139.2.8 namesize	430
3.139.2.9 dimension	430
3.140 T_const Struct Reference	431
3.140.1 Detailed Description	433
3.140.2 Field Documentation	433
3.140.2.1 S0	433
3.140.2.2 SS	433
3.140.2.3 Nest	433
3.140.2.4 NestStop	433
3.140.2.5 NestPoin	434
3.140.2.6 BrackBuf	434
3.140.2.7 StoreCache	434
3.140.2.8 StoreCacheAlloc	434

3.140.2.9 pWorkSpace	434
3.140.2.10 IWorkSpace	434
3.140.2.11 posWorkSpace	434
3.140.2.12 WorkSpace	434
3.140.2.13 WorkTop	435
3.140.2.14 WorkPointer	435
3.140.2.15 RepCount	435
3.140.2.16 RepTop	435
3.140.2.17 WildArgTaken	435
3.140.2.18 factorials	435
3.140.2.19 small_power_n	435
3.140.2.20 small_power	435
3.140.2.21 bernoullis	436
3.140.2.22 primelist	436
3.140.2.23 pfac	436
3.140.2.24 pBer	436
3.140.2.25 TMaddr	436
3.140.2.26 WildMask	436
3.140.2.27 previousEfactor	436
3.140.2.28 TermMemHeap	436
3.140.2.29 NumberMemHeap	437
3.140.2.30 CacheNumberMemHeap	437
3.140.2.31 bracketinfo	437
3.140.2.32 ListPoly	437
3.140.2.33 ListSymbols	437
3.140.2.34 NumMem	437
3.140.2.35 TopologiesTerm	437
3.140.2.36 TopologiesStart	437
3.140.2.37 partitions	438
3.140.2.38 sBer	438
3.140.2.39 pWorkPointer	438
3.140.2.40 IWorkPointer	438
3.140.2.41 posWorkPointer	438
3.140.2.42 InNumMem	438
3.140.2.43 sfact	438
3.140.2.44 mfac	438
3.140.2.45 ebufnum	439
3.140.2.46 fbufnum	439
3.140.2.47 allbufnum	439
3.140.2.48 aebufnum	439
3.140.2.49 idallflag	439
3.140.2.50 idallnum	439

3.140.2.51 idallmaxnum	439
3.140.2.52 WildcardBufferSize	439
3.140.2.53 TermMemMax	440
3.140.2.54 TermMemTop	440
3.140.2.55 NumberMemMax	440
3.140.2.56 NumberMemTop	440
3.140.2.57 CacheNumberMemMax	440
3.140.2.58 CacheNumberMemTop	440
3.140.2.59 bracketindexflag	440
3.140.2.60 optentimes	440
3.140.2.61 ListSymbolsSize	441
3.140.2.62 NumListSymbols	441
3.140.2.63 numpoly	441
3.140.2.64 LeaveNegative	441
3.140.2.65 TrimPower	441
3.140.2.66 small_power_maxx	441
3.140.2.67 small_power_maxn	441
3.140.2.68 dummysubexp	441
3.140.2.69 comsym	442
3.140.2.70 comnum	442
3.140.2.71 comfun	442
3.140.2.72 comind	442
3.140.2.73 MinVecArg	442
3.140.2.74 FunArg	442
3.140.2.75 locwildvalue	442
3.140.2.76 mulpat	442
3.140.2.77 proexp	443
3.140.2.78 TMout	443
3.140.2.79 TMbuff	443
3.140.2.80 TMdolfac	443
3.140.2.81 nfac	443
3.140.2.82 nBer	443
3.140.2.83 mBer	443
3.140.2.84 PolyAct	443
3.140.2.85 RecFlag	444
3.140.2.86 inprimelist	444
3.140.2.87 sizeprimelist	444
3.140.2.88 fromindex	444
3.140.2.89 setinterntopo	444
3.140.2.90 setexterntopo	444
3.140.2.91 TopologiesLevel	444
3.140.2.92 TopologiesOptions	445

3.141 TaBIEbAsE Struct Reference	445
3.141.1 Detailed Description	445
3.141.2 Field Documentation	446
3.141.2.1 fillpoint	446
3.141.2.2 current	446
3.141.2.3 name	446
3.141.2.4 tablenumbers	446
3.141.2.5 subindex	446
3.141.2.6 numtables	446
3.142 TaBIEbAsEsUbInDeX Struct Reference	447
3.142.1 Detailed Description	447
3.142.2 Field Documentation	447
3.142.2.1 where	447
3.142.2.2 size	447
3.143 TaBIEs Struct Reference	448
3.143.1 Detailed Description	449
3.143.2 Field Documentation	449
3.143.2.1 tablepointers	449
3.143.2.2 prototype	449
3.143.2.3 pattern	449
3.143.2.4 mm	449
3.143.2.5 flags	450
3.143.2.6 boomlijst	450
3.143.2.7 argtail	450
3.143.2.8 spare	450
3.143.2.9 buffers	450
3.143.2.10 totind	451
3.143.2.11 reserved	451
3.143.2.12 defined	451
3.143.2.13 mdefined	451
3.143.2.14 prototypeSize	451
3.143.2.15 numind	452
3.143.2.16 bounds	452
3.143.2.17 strict	452
3.143.2.18 sparse	452
3.143.2.19 numtree	452
3.143.2.20 rootnum	453
3.143.2.21 MaxTreeSize	453
3.143.2.22 bufnum	453
3.143.2.23 bufferssize	453
3.143.2.24 buffersfill	453
3.143.2.25 tablenum	454

3.143.2.26 mode	454
3.143.2.27 numdummies	454
3.144 TERMINFO Struct Reference	454
3.144.1 Detailed Description	454
3.144.2 Field Documentation	455
3.144.2.1 term	455
3.144.2.2 currentModel	455
3.144.2.3 currentMODEL	455
3.144.2.4 legcouple	455
3.144.2.5 numtopo	455
3.144.2.6 numdia	455
3.144.2.7 diaoffset	455
3.144.2.8 level	456
3.144.2.9 externalset	456
3.144.2.10 internalset	456
3.144.2.11 numextern	456
3.144.2.12 flags	456
3.145 ToPoTyPe Struct Reference	456
3.145.1 Detailed Description	457
3.145.2 Field Documentation	457
3.145.2.1 vert	457
3.145.2.2 vertmax	457
3.145.2.3 opt	457
3.145.2.4 cldeg	457
3.145.2.5 cnum	457
3.145.2.6 clex	458
3.145.2.7 cmin	458
3.145.2.8 cmaxd	458
3.145.2.9 ncl	458
3.145.2.10 nvert	458
3.146 TrAcEn Struct Reference	458
3.146.1 Detailed Description	459
3.146.2 Field Documentation	459
3.146.2.1 accu	459
3.146.2.2 accup	459
3.146.2.3 temp	459
3.146.2.4 perm	459
3.146.2.5 inlist	459
3.146.2.6 sgn	459
3.146.2.7 num	460
3.146.2.8 level	460
3.146.2.9 factor	460

3.146.2.10 allsign	460
3.147 TrAcEs Struct Reference	460
3.147.1 Detailed Description	461
3.147.2 Field Documentation	461
3.147.2.1 accu	461
3.147.2.2 accup	462
3.147.2.3 temp	462
3.147.2.4 perm	462
3.147.2.5 inlist	462
3.147.2.6 nt3	462
3.147.2.7 nt4	462
3.147.2.8 j3	462
3.147.2.9 j4	462
3.147.2.10 e3	463
3.147.2.11 e4	463
3.147.2.12 eers	463
3.147.2.13 mepf	463
3.147.2.14 mdel	463
3.147.2.15 pepf	463
3.147.2.16 pdel	463
3.147.2.17 sgn	463
3.147.2.18 stap	464
3.147.2.19 step1	464
3.147.2.20 kstep	464
3.147.2.21 mdum	464
3.147.2.22 gamm	464
3.147.2.23 ad	464
3.147.2.24 a3	464
3.147.2.25 a4	464
3.147.2.26 lc3	465
3.147.2.27 lc4	465
3.147.2.28 sign1	465
3.147.2.29 sign2	465
3.147.2.30 gamma5	465
3.147.2.31 num	465
3.147.2.32 level	465
3.147.2.33 factor	465
3.147.2.34 allsign	466
3.147.2.35 finalstep	466
3.148 tree Struct Reference	466
3.148.1 Detailed Description	466
3.148.2 Field Documentation	466

3.148.2.1 parent	466
3.148.2.2 left	467
3.148.2.3 right	467
3.148.2.4 value	467
3.148.2.5 blnce	467
3.148.2.6 usage	467
3.149 tree_node Class Reference	468
3.149.1 Detailed Description	468
3.149.2 Constructor & Destructor Documentation	468
3.149.2.1 tree_node() [1/2]	468
3.149.2.2 tree_node() [2/2]	469
3.149.3 Member Function Documentation	469
3.149.3.1 operator=()	469
3.149.4 Field Documentation	469
3.149.4.1 childs	469
3.149.4.2 sum_results	469
3.149.4.3 num_visits	469
3.149.4.4 var	469
3.149.4.5 finished	470
3.150 VaRrEnUm Struct Reference	470
3.150.1 Detailed Description	470
3.150.2 Field Documentation	470
3.150.2.1 start	470
3.150.2.2 lo	470
3.150.2.3 hi	471
3.151 VeCtOr Struct Reference	471
3.151.1 Detailed Description	471
3.151.2 Field Documentation	471
3.151.2.1 name	471
3.151.2.2 complex	471
3.151.2.3 number	472
3.151.2.4 flags	472
3.151.2.5 node	472
3.151.2.6 namesize	472
3.151.2.7 dimension	472
3.152 VeRtEx Struct Reference	472
3.152.1 Detailed Description	473
3.152.2 Field Documentation	473
3.152.2.1 particles	473
3.152.2.2 couplings	473
3.152.2.3 nparticles	473
3.152.2.4 ncouplings	473

3.152.2.5 type	473
3.152.2.6 error	474
3.152.2.7 externonly	474
3.152.2.8 spare	474
3.153 X_const Struct Reference	474
3.153.1 Detailed Description	474
3.153.2 Field Documentation	475
3.153.2.1 currentPrompt	475
3.153.2.2 shellname	475
3.153.2.3 stderrname	475
3.153.2.4 timeout	475
3.153.2.5 killSignal	475
3.153.2.6 killWholeGroup	475
3.153.2.7 daemonize	475
3.153.2.8 currentExternalChannel	475
4 File Documentation	477
4.1 argument.c File Reference	477
4.1.1 Detailed Description	478
4.1.2 Macro Definition Documentation	478
4.1.2.1 NEWORDER	478
4.1.3 Function Documentation	478
4.1.3.1 execarg()	478
4.1.3.2 execterm()	478
4.1.3.3 ArgumentImplode()	478
4.1.3.4 ArgumentExplode()	479
4.1.3.5 ArgFactorize()	479
4.1.3.6 FindArg()	479
4.1.3.7 InsertArg()	479
4.1.3.8 CleanupArgCache()	480
4.1.3.9 ArgSymbolMerge()	480
4.1.3.10 ArgDotproductMerge()	480
4.1.3.11 TakeArgContent()	481
4.1.3.12 MakeInteger()	481
4.1.3.13 MakeMod()	482
4.1.3.14 SortWeights()	482
4.2 argument.c	483
4.3 bugtool.c File Reference	523
4.3.1 Detailed Description	523
4.3.2 Function Documentation	524
4.3.2.1 ExprStatus()	524
4.4 bugtool.c	524

4.5 checkpoint.c File Reference	525
4.5.1 Detailed Description	526
4.5.2 Macro Definition Documentation	526
4.5.2.1 CACHED_SNAPSHOT	526
4.5.2.2 CACHE_SIZE	526
4.5.2.3 R_FREE	527
4.5.2.4 R_FREE_NAMETREE	527
4.5.2.5 R_FREE_STREAM	527
4.5.2.6 R_SET	527
4.5.2.7 R_COPY_B	527
4.5.2.8 S_WRITE_B	528
4.5.2.9 S_FLUSH_B	528
4.5.2.10 R_COPY_S	528
4.5.2.11 S_WRITE_S	528
4.5.2.12 R_COPY_LIST	528
4.5.2.13 S_WRITE_LIST	529
4.5.2.14 R_COPY_NAMETREE	529
4.5.2.15 S_WRITE_NAMETREE	529
4.5.2.16 S_WRITE_DOLLAR	529
4.5.2.17 ANNOUNCE	530
4.5.3 Function Documentation	530
4.5.3.1 CheckRecoveryFile()	530
4.5.3.2 DeleteRecoveryFile()	530
4.5.3.3 RecoveryFilename()	530
4.5.3.4 InitRecovery()	531
4.5.3.5 fwrite_cached()	531
4.5.3.6 flush_cache()	531
4.5.3.7 DoRecovery()	531
4.5.3.8 DoCheckpoint()	532
4.5.4 Variable Documentation	533
4.5.4.1 cache_buffer	533
4.5.4.2 cache_fill	533
4.6 checkpoint.c	533
4.7 comexpr.c File Reference	569
4.7.1 Detailed Description	570
4.7.2 Function Documentation	570
4.7.2.1 CoLocal()	570
4.7.2.2 CoGlobal()	570
4.7.2.3 CoLocalFactorized()	570
4.7.2.4 CoGlobalFactorized()	570
4.7.2.5 DoExpr()	570
4.7.2.6 ColdOld()	571

4.7.2.7 Cold()	571
4.7.2.8 ColdNew()	571
4.7.2.9 CoDisorder()	571
4.7.2.10 CoMany()	571
4.7.2.11 CoMulti()	571
4.7.2.12 ColfMatch()	571
4.7.2.13 ColfNoMatch()	572
4.7.2.14 CoOnce()	572
4.7.2.15 CoOnly()	572
4.7.2.16 CoSelect()	572
4.7.2.17 ColdExpression()	572
4.7.2.18 CoMultiply()	572
4.7.2.19 CoFill()	572
4.7.2.20 CoFillExpression()	573
4.7.2.21 CoPrintTable()	573
4.7.2.22 CoAssign()	573
4.7.2.23 CoDeallocateTable()	573
4.8 comexpr.c	573
4.9 compcomm.c File Reference	597
4.9.1 Detailed Description	600
4.9.2 Function Documentation	600
4.9.2.1 CoFormat()	600
4.9.2.2 CoCollect()	600
4.9.2.3 setonoff()	600
4.9.2.4 CoCompress()	601
4.9.2.5 CoFlags()	601
4.9.2.6 CoOff()	601
4.9.2.7 CoOn()	601
4.9.2.8 ColInsideFirst()	601
4.9.2.9 CoProperCount()	601
4.9.2.10 CoDelete()	601
4.9.2.11 CoKeep()	602
4.9.2.12 CoFixIndex()	602
4.9.2.13 CoMetric()	602
4.9.2.14 DoPrint()	602
4.9.2.15 CoPrint()	602
4.9.2.16 CoPrintB()	602
4.9.2.17 CoNPrint()	602
4.9.2.18 CoPushHide()	603
4.9.2.19 CoPopHide()	603
4.9.2.20 SetExprCases()	603
4.9.2.21 SetExpr()	603

4.9.2.22 CoDrop()	603
4.9.2.23 CoNoDrop()	603
4.9.2.24 CoSkip()	604
4.9.2.25 CoNoSkip()	604
4.9.2.26 CoHide()	604
4.9.2.27 CoIntoHide()	604
4.9.2.28 CoNoIntoHide()	604
4.9.2.29 CoNoHide()	604
4.9.2.30 CoUnHide()	604
4.9.2.31 CoNoUnHide()	605
4.9.2.32 AddToCom()	605
4.9.2.33 AddComString()	605
4.9.2.34 Add2ComStrings()	605
4.9.2.35 CoDiscard()	605
4.9.2.36 CoContract()	605
4.9.2.37 CoGoTo()	606
4.9.2.38 CoLabel()	606
4.9.2.39 DoArgument()	606
4.9.2.40 CoArgument()	606
4.9.2.41 CoEndArgument()	606
4.9.2.42 CoInside()	606
4.9.2.43 CoEndInside()	606
4.9.2.44 CoNormalize()	607
4.9.2.45 CoMakeInteger()	607
4.9.2.46 CoSplitArg()	607
4.9.2.47 CoSplitFirstArg()	607
4.9.2.48 CoSplitLastArg()	607
4.9.2.49 CoFactArg()	607
4.9.2.50 DoSymmetrize()	607
4.9.2.51 CoSymmetrize()	608
4.9.2.52 CoAntiSymmetrize()	608
4.9.2.53 CoCycleSymmetrize()	608
4.9.2.54 CoRCycleSymmetrize()	608
4.9.2.55 CoWrite()	608
4.9.2.56 CoNWrite()	608
4.9.2.57 CoRatio()	608
4.9.2.58 CoRedefine()	609
4.9.2.59 CoRenumber()	609
4.9.2.60 CoSum()	609
4.9.2.61 CoToTensor()	609
4.9.2.62 CoToVector()	609
4.9.2.63 CoTrace4()	609

4.9.2.64 CoTraceN()	609
4.9.2.65 CoChisholm()	610
4.9.2.66 DoChain()	610
4.9.2.67 CoChainin()	610
4.9.2.68 CoChainout()	610
4.9.2.69 CoExit()	610
4.9.2.70 CoInParallel()	610
4.9.2.71 CoNotInParallel()	610
4.9.2.72 DoInParallel()	611
4.9.2.73 CoInExpression()	611
4.9.2.74 CoEndInExpression()	611
4.9.2.75 CoSetExitFlag()	611
4.9.2.76 CoTryReplace()	611
4.9.2.77 CoModulus()	611
4.9.2.78 CoRepeat()	611
4.9.2.79 CoEndRepeat()	612
4.9.2.80 DoBrackets()	612
4.9.2.81 CoBracket()	612
4.9.2.82 CoAntiBracket()	612
4.9.2.83 CoMultiBracket()	612
4.9.2.84 CountComp()	612
4.9.2.85 Colf()	612
4.9.2.86 CoElse()	613
4.9.2.87 CoElseif()	613
4.9.2.88 CoEndIf()	613
4.9.2.89 CoWhile()	613
4.9.2.90 CoEndWhile()	613
4.9.2.91 DoFindLoop()	613
4.9.2.92 CoFindLoop()	613
4.9.2.93 CoReplaceLoop()	614
4.9.2.94 CoFunPowers()	614
4.9.2.95 CoUnitTrace()	614
4.9.2.96 CoTerm()	614
4.9.2.97 CoEndTerm()	614
4.9.2.98 CoSort()	614
4.9.2.99 CoPolyFun()	614
4.9.2.100 CoPolyRatFun()	615
4.9.2.101 CoMerge()	615
4.9.2.102 CoStuffle()	615
4.9.2.103 CoProcessBucket()	615
4.9.2.104 CoThreadBucket()	615
4.9.2.105 DoArgPlode()	615

4.9.2.106 CoArgExplode()	615
4.9.2.107 CoArgImplode()	616
4.9.2.108 CoClearTable()	616
4.9.2.109 CoDenominators()	616
4.9.2.110 CoDropCoefficient()	616
4.9.2.111 CoDropSymbols()	616
4.9.2.112 CoToPolynomial()	616
4.9.2.113 CoFromPolynomial()	616
4.9.2.114 CoArgToExtraSymbol()	617
4.9.2.115 CoExtraSymbols()	617
4.9.2.116 GetIfDollarFactor()	617
4.9.2.117 GetDoParam()	617
4.9.2.118 CoDo()	617
4.9.2.119 CoEndDo()	617
4.9.2.120 CoFactDollar()	618
4.9.2.121 CoFactorize()	618
4.9.2.122 CoNFactorize()	618
4.9.2.123 CoUnFactorize()	618
4.9.2.124 CoNUnFactorize()	618
4.9.2.125 DoFactorize()	618
4.9.2.126 CoOptimizeOption()	618
4.9.2.127 CoPutInside()	619
4.9.2.128 CoAntiPutInside()	619
4.9.2.129 DoPutInside()	619
4.9.2.130 CoSwitch()	619
4.9.2.131 CoCase()	619
4.9.2.132 CoBreak()	619
4.9.2.133 CoDefault()	619
4.9.2.134 CoEndSwitch()	620
4.9.2.135 CoSetUserFlag()	620
4.9.2.136 CoClearUserFlag()	620
4.9.2.137 CoCreateAllLoops()	620
4.9.2.138 CoCreateAllPaths()	620
4.9.2.139 CoCreateAll()	620
4.10 compcomm.c	621
4.11 compiler.c File Reference	710
4.11.1 Detailed Description	711
4.11.2 Macro Definition Documentation	712
4.11.2.1 OPTION0	712
4.11.2.2 OPTION1	712
4.11.2.3 OPTION2	712
4.11.2.4 REDUCESUBEXPBUFFERS	712

4.11.3 Function Documentation	712
4.11.3.1 inictable()	712
4.11.3.2 findcommand()	712
4.11.3.3 ParenthesesTest()	713
4.11.3.4 SkipAName()	713
4.11.3.5 IsRHS()	713
4.11.3.6 IsIdStatement()	713
4.11.3.7 CompileAlgebra()	714
4.11.3.8 CompileStatement()	714
4.11.3.9 TestTables()	714
4.11.3.10 CompileSubExpressions()	714
4.11.3.11 CodeGenerator()	714
4.11.3.12 CompleteTerm()	714
4.11.3.13 CodeFactors()	715
4.11.3.14 GenerateFactors()	715
4.11.4 Variable Documentation	715
4.11.4.1 alfatable1	715
4.11.4.2 subexpbuffers	715
4.11.4.3 topsubexpbuffers	715
4.11.4.4 insubexpbuffers	715
4.12 compiler.c	716
4.13 compress.c File Reference	742
4.13.1 Detailed Description	742
4.14 compress.c	743
4.15 comtool.c File Reference	751
4.15.1 Detailed Description	752
4.15.2 Function Documentation	752
4.15.2.1 inicbufs()	752
4.15.2.2 finishcbuf()	752
4.15.2.3 clearcbuf()	753
4.15.2.4 DoubleCbuffer()	753
4.15.2.5 AddLHS()	753
4.15.2.6 AddRHS()	754
4.15.2.7 AddNtoL()	754
4.15.2.8 AddNtoC()	755
4.15.2.9 InsTree()	756
4.15.2.10 FindTree()	756
4.15.2.11 RedoTree()	756
4.15.2.12 ClearTree()	756
4.15.2.13 IniFbuffer()	757
4.15.2.14 numcommute()	757
4.16 comtool.c	757

4.17 comtool.h File Reference	764
4.17.1 Detailed Description	765
4.18 comtool.h	765
4.19 declare.h File Reference	766
4.19.1 Detailed Description	788
4.19.2 Macro Definition Documentation	788
4.19.2.1 MaX	788
4.19.2.2 MiN	788
4.19.2.3 ABS	789
4.19.2.4 SGN	789
4.19.2.5 REDLENG	789
4.19.2.6 INCLENG	789
4.19.2.7 GETCOEF	789
4.19.2.8 GETSTOP	789
4.19.2.9 StuffAdd	790
4.19.2.10 EXCHN	790
4.19.2.11 EXCH	790
4.19.2.12 TOKENTOLINE	790
4.19.2.13 UngetFromStream	790
4.19.2.14 AddLineFeed	790
4.19.2.15 TryRecover	791
4.19.2.16 UngetChar	791
4.19.2.17 ParseNumber	791
4.19.2.18 ParseSign	791
4.19.2.19 ParseSignedNumber	791
4.19.2.20 NCOPY	791
4.19.2.21 NCOPYI	792
4.19.2.22 NCOPYB	792
4.19.2.23 NCOPYI32	792
4.19.2.24 WCOPY	792
4.19.2.25 NeedNumber	792
4.19.2.26 SKIPBLANKS	793
4.19.2.27 FLUSHCONSOLE	793
4.19.2.28 SKIPBRA1	793
4.19.2.29 SKIPBRA2	793
4.19.2.30 SKIPBRA3	793
4.19.2.31 SKIPBRA4	794
4.19.2.32 SKIPBRA5	794
4.19.2.33 CYCLE1	794
4.19.2.34 AddToCB	794
4.19.2.35 EXCHINOUT	794
4.19.2.36 BACKINOUT	795

4.19.2.37 CopyArg	795
4.19.2.38 FILLARG	795
4.19.2.39 COPYARG	795
4.19.2.40 ZEROARG	795
4.19.2.41 FILLFUN	795
4.19.2.42 COPYFUN	796
4.19.2.43 COPYFUN3	796
4.19.2.44 FILLFUN3	796
4.19.2.45 FILLSUB	796
4.19.2.46 COPYSUB	796
4.19.2.47 FILLEXPR	796
4.19.2.48 NEXTARG	797
4.19.2.49 COPY1ARG	797
4.19.2.50 ZeroFillRange	797
4.19.2.51 TABLESIZE	797
4.19.2.52 WORDDIF	798
4.19.2.53 wsizeof	798
4.19.2.54 VARNAME	798
4.19.2.55 DOLLARNAME	798
4.19.2.56 EXPRNAME	798
4.19.2.57 PREV	798
4.19.2.58 Terminate	799
4.19.2.59 SETERROR	799
4.19.2.60 DUMMYUSE	799
4.19.2.61 ADDPOS	799
4.19.2.62 SETBASELENGTH	799
4.19.2.63 SETBASEPOSITION	799
4.19.2.64 ISEQUALPOSINC	800
4.19.2.65 ISGEPOSINC	800
4.19.2.66 DIVPOS	800
4.19.2.67 MULPOS	800
4.19.2.68 DIFPOS	800
4.19.2.69 DIFBASE	800
4.19.2.70 ADD2POS	801
4.19.2.71 PUTZERO	801
4.19.2.72 BASEPOSITION	801
4.19.2.73 SETSTARTPOS	801
4.19.2.74 NOTSTARTPOS	801
4.19.2.75 ISMINPOS	801
4.19.2.76 ISEQUALPOS	801
4.19.2.77 ISNOTEQUALPOS	802
4.19.2.78 ISLESSPOS	802

4.19.2.79 ISGEPOS	802
4.19.2.80 ISNOTZEROPOS	802
4.19.2.81 ISZEROPOS	802
4.19.2.82 ISPOSPOS	802
4.19.2.83 ISNEGPOS	803
4.19.2.84 TOLONG	803
4.19.2.85 Add2Com	803
4.19.2.86 Add3Com	803
4.19.2.87 Add4Com	803
4.19.2.88 Add5Com	803
4.19.2.89 WantAddPointers	804
4.19.2.90 WantAddLongs	804
4.19.2.91 WantAddPositions	804
4.19.2.92 FORM_INLINE	804
4.19.2.93 MEMORYMACROS	804
4.19.2.94 TermMalloc	804
4.19.2.95 NumberMalloc	805
4.19.2.96 CacheNumberMalloc	805
4.19.2.97 TermFree	805
4.19.2.98 NumberFree	805
4.19.2.99 CacheNumberFree	805
4.19.2.100 NestingChecksum	805
4.19.2.101 MesNesting	806
4.19.2.102 MarkPolyRatFunDirty	806
4.19.2.103 PUSHPREASSIGNLEVEL	806
4.19.2.104 POPPREASSIGNLEVEL	806
4.19.2.105 EXTERNLOCK	806
4.19.2.106 INILOCK	807
4.19.2.107 LOCK	807
4.19.2.108 UNLOCK	807
4.19.2.109 EXTERNRWLOCK	807
4.19.2.110 INIRWLOCK	807
4.19.2.111 RWLOCKR	807
4.19.2.112 RWLOCKW	807
4.19.2.113 UNRWLOCK	808
4.19.2.114 MLOCK	808
4.19.2.115 MUNLOCK	808
4.19.2.116 GETIDENTITY	808
4.19.2.117 GETBIDENTITY	808
4.19.2.118 M_alloc	808
4.19.2.119 CompareTerms	808
4.19.2.120 FiniShuffle	809

4.19.2.121 DoShuffle	809
4.19.3 Typedef Documentation	809
4.19.3.1 WRITEBUFTOEXTCHANNEL	809
4.19.3.2 GETCFROMEXTCHANNEL	809
4.19.3.3 SETTERMINATORFOREXTCHANNEL	809
4.19.3.4 SETKILLMODEFOREXTCHANNEL	809
4.19.3.5 WRITEFILE	809
4.19.3.6 GETTERM	810
4.19.4 Function Documentation	810
4.19.4.1 TELLFILE()	810
4.19.4.2 StartVariables()	810
4.19.4.3 CodeToLine()	811
4.19.4.4 AddArrayIndex()	811
4.19.4.5 FindInIndex()	811
4.19.4.6 NextFileIndex()	811
4.19.4.7 PasteTerm()	811
4.19.4.8 StrCopy()	812
4.19.4.9 WrtPower()	812
4.19.4.10 AccumGCD()	812
4.19.4.11 AddArgs()	812
4.19.4.12 AddCoef()	813
4.19.4.13 AddLong()	813
4.19.4.14 AddPLon()	814
4.19.4.15 AddPoly()	814
4.19.4.16 AddRat()	815
4.19.4.17 AddToLine()	815
4.19.4.18 AddWild()	815
4.19.4.19 BigLong()	816
4.19.4.20 BinomGen()	816
4.19.4.21 CheckWild()	816
4.19.4.22 Chisholm()	816
4.19.4.23 CleanExpr()	816
4.19.4.24 CleanUp()	817
4.19.4.25 ClearWild()	817
4.19.4.26 CompareFunctions()	817
4.19.4.27 Commute()	817
4.19.4.28 DetCommu()	817
4.19.4.29 DoesCommu()	817
4.19.4.30 CompArg()	818
4.19.4.31 CompCoef()	818
4.19.4.32 CompGroup()	818
4.19.4.33 Compare1()	818

4.19.4.34 CountDo()	819
4.19.4.35 CountFun()	819
4.19.4.36 DimensionSubterm()	819
4.19.4.37 DimensionTerm()	820
4.19.4.38 DimensionExpression()	820
4.19.4.39 Deferred()	820
4.19.4.40 DeleteStore()	821
4.19.4.41 DetCurDum()	821
4.19.4.42 DetVars()	821
4.19.4.43 Distribute()	822
4.19.4.44 DivLong()	822
4.19.4.45 DivRat()	822
4.19.4.46 Divvy()	822
4.19.4.47 DoDelta()	822
4.19.4.48 DoDelta3()	823
4.19.4.49 TestPartitions()	823
4.19.4.50 DoPartitions()	823
4.19.4.51 CoCanonicalize()	823
4.19.4.52 DoCanonicalize()	823
4.19.4.53 GenDiagrams()	823
4.19.4.54 DoTopologyCanonicalize()	824
4.19.4.55 DoShattering()	824
4.19.4.56 DoTableExpansion()	824
4.19.4.57 DoDistrib()	824
4.19.4.58 DoShuffle()	824
4.19.4.59 DoPermutations()	825
4.19.4.60 Shuffle()	825
4.19.4.61 FinishShuffle()	825
4.19.4.62 DoStuffle()	825
4.19.4.63 Stuffle()	825
4.19.4.64 FinishStuffle()	825
4.19.4.65 StuffRootAdd()	826
4.19.4.66 TestUse()	826
4.19.4.67 FindTB()	826
4.19.4.68 CheckTableDeclarations()	826
4.19.4.69 Apply()	826
4.19.4.70 ApplyExec()	826
4.19.4.71 ApplyReset()	827
4.19.4.72 TableReset()	827
4.19.4.73 ReWorkT()	827
4.19.4.74 GetIfDollarNum()	827
4.19.4.75 FindVar()	827

4.19.4.76 DolfStatement()	827
4.19.4.77 DoOnePow()	828
4.19.4.78 DoRevert()	829
4.19.4.79 DoSumF1()	829
4.19.4.80 DoSumF2()	829
4.19.4.81 DoTheta()	830
4.19.4.82 EndSort()	830
4.19.4.83 EntVar()	831
4.19.4.84 EpfCon()	831
4.19.4.85 EpfFind()	831
4.19.4.86 EpfGen()	831
4.19.4.87 EqualArg()	831
4.19.4.88 Factorial()	832
4.19.4.89 Bernoulli()	832
4.19.4.90 FactorIn()	832
4.19.4.91 FactorInExpr()	832
4.19.4.92 FindAll()	832
4.19.4.93 FindMulti()	832
4.19.4.94 FindOnce()	833
4.19.4.95 FindOnly()	833
4.19.4.96 FindRest()	833
4.19.4.97 FindrNumber()	833
4.19.4.98 FiniLine()	833
4.19.4.99 FiniTerm()	833
4.19.4.100 FlushOut()	834
4.19.4.101 FunLevel()	835
4.19.4.102 AdjustRenumScratch()	835
4.19.4.103 GarbHand()	835
4.19.4.104 GcdLong()	835
4.19.4.105 LcmLong()	835
4.19.4.106 Generator()	836
4.19.4.107 GetBinom()	838
4.19.4.108 GetFromStore()	838
4.19.4.109 GetLong()	838
4.19.4.110 GetMoreTerms()	838
4.19.4.111 GetMoreFromMem()	838
4.19.4.112 GetOneTerm()	839
4.19.4.113 GetTable()	839
4.19.4.114 GetTerm()	839
4.19.4.115 Glue()	839
4.19.4.116 InFunction()	839
4.19.4.117 IniLine()	840

4.19.4.118 IniVars()	840
4.19.4.119 InsertTerm()	841
4.19.4.120 LongToLine()	841
4.19.4.121 MakeDirty()	841
4.19.4.122 MarkDirty()	842
4.19.4.123 PolyFunDirty()	842
4.19.4.124 PolyFunClean()	842
4.19.4.125 MakeModTable()	842
4.19.4.126 MatchE()	842
4.19.4.127 MatchCy()	842
4.19.4.128 FunMatchCy()	843
4.19.4.129 FunMatchSy()	843
4.19.4.130 MatchArgument()	843
4.19.4.131 MatchFunction()	843
4.19.4.132 MergePatches()	843
4.19.4.133 MesCerr()	844
4.19.4.134 MesComp()	844
4.19.4.135 Modulus()	845
4.19.4.136 MoveDummies()	845
4.19.4.137 MulLong()	845
4.19.4.138 MulRat()	845
4.19.4.139 Mully()	845
4.19.4.140 MultDo()	846
4.19.4.141 NewSort()	846
4.19.4.142 ExtraSymbol()	846
4.19.4.143 Normalize()	846
4.19.4.144 BracketNormalize()	846
4.19.4.145 DropCoefficient()	847
4.19.4.146 DropSymbols()	847
4.19.4.147 PutInside()	847
4.19.4.148 OpenTemp()	847
4.19.4.149 Pack()	847
4.19.4.150 PasteFile()	847
4.19.4.151 Permute()	848
4.19.4.152 PermuteP()	848
4.19.4.153 PolyFunMul()	848
4.19.4.154 PopVariables()	849
4.19.4.155 PrepPoly()	849
4.19.4.156 Processor()	850
4.19.4.157 Product()	851
4.19.4.158 PrtLong()	851
4.19.4.159 PrtTerms()	852

4.19.4.160 PrintDeprecation()	852
4.19.4.161 PrintRunningTime()	852
4.19.4.162 GetRunningTime()	852
4.19.4.163 PutBracket()	852
4.19.4.164 PutIn()	852
4.19.4.165 PutInStore()	853
4.19.4.166 PutOut()	853
4.19.4.167 Quotient()	854
4.19.4.168 RaisPow()	854
4.19.4.169 RaisPowCached()	854
4.19.5 Description	854
4.19.6 Notes	855
4.19.6.1 RaisPowMod()	855
4.19.6.2 NormalModulus()	855
4.19.6.3 MakeInverses()	855
4.19.6.4 GetModInverses()	856
4.19.6.5 GetLongModInverses()	856
4.19.6.6 RatToLine()	856
4.19.6.7 RatioFind()	856
4.19.6.8 RatioGen()	857
4.19.6.9 ReNumber()	857
4.19.6.10 ReadSnum()	857
4.19.6.11 Remain10()	857
4.19.6.12 Remain4()	857
4.19.6.13 ResetScratch()	857
4.19.6.14 ResolveSet()	858
4.19.6.15 RevertScratch()	858
4.19.6.16 ScanFunctions()	858
4.19.6.17 SeekScratch()	858
4.19.6.18 SetEndScratch()	858
4.19.6.19 SetEndHScratch()	858
4.19.6.20 SetFileIndex()	859
4.19.6.21 Sflush()	859
4.19.6.22 Simplify()	859
4.19.6.23 SortWild()	860
4.19.6.24 LocateBase()	860
4.19.6.25 SplitMerge()	860
4.19.6.26 StoreTerm()	861
4.19.6.27 SubPLon()	862
4.19.6.28 Substitute()	862
4.19.6.29 SymFind()	862
4.19.6.30 SymGen()	863

4.19.6.31 Symmetrize()	863
4.19.6.32 FullSymmetrize()	863
4.19.6.33 TakeModulus()	863
4.19.6.34 TakeNormalModulus()	863
4.19.6.35 TalToLine()	864
4.19.6.36 TenVec()	864
4.19.6.37 TenVecFind()	864
4.19.6.38 TermRenumber()	864
4.19.6.39 TestDrop()	865
4.19.6.40 PutInVflags()	865
4.19.6.41 TestMatch()	865
4.19.6.42 TestSub()	866
4.19.6.43 TimeCPU()	866
4.19.6.44 TimeChildren()	867
4.19.6.45 TimeWallClock()	867
4.19.6.46 ToStorage()	867
4.19.6.47 TokenToLine()	867
4.19.6.48 Trace4()	867
4.19.6.49 Trace4Gen()	868
4.19.6.50 Trace4no()	868
4.19.6.51 TraceFind()	868
4.19.6.52 TraceN()	868
4.19.6.53 TraceNgen()	868
4.19.6.54 TraceNno()	868
4.19.6.55 Traces()	869
4.19.6.56 Trick()	869
4.19.6.57 TryDo()	869
4.19.6.58 UnPack()	869
4.19.6.59 VarStore()	869
4.19.6.60 WildFill()	870
4.19.6.61 WriteAll()	870
4.19.6.62 WriteOne()	870
4.19.6.63 WriteArgument()	870
4.19.6.64 WriteExpression()	870
4.19.6.65 WriteInnerTerm()	870
4.19.6.66 WriteLists()	871
4.19.6.67 WriteSetup()	871
4.19.6.68 WriteStats()	871
4.19.6.69 WriteSubTerm()	872
4.19.6.70 WriteTerm()	872
4.19.6.71 execarg()	872
4.19.6.72 execterm()	872

4.19.6.73 SpecialCleanup()	872
4.19.6.74 SetMods()	872
4.19.6.75 UnSetMods()	873
4.19.6.76 DoExecute()	873
4.19.6.77 SetScratch()	873
4.19.6.78 Warning()	873
4.19.6.79 HighWarning()	873
4.19.6.80 SpareTable()	873
4.19.6.81 strDup1()	874
4.19.6.82 Malloc1()	874
4.19.6.83 DoTail()	874
4.19.6.84 OpenInput()	874
4.19.6.85 PutPreVar()	874
4.19.6.86 Error0()	875
4.19.6.87 Error1()	875
4.19.6.88 Error2()	875
4.19.6.89 ReadFromStream()	875
4.19.6.90 GetFromStream()	875
4.19.6.91 LookInStream()	876
4.19.6.92 OpenStream()	876
4.19.6.93 LocateFile()	876
4.19.6.94 CloseStream()	876
4.19.6.95 PositionStream()	876
4.19.6.96 ReverseStatements()	876
4.19.6.97 ProcessOption()	877
4.19.6.98 DoSetups()	877
4.19.6.99 TerminateImpl()	877
4.19.6.100 GetNode()	877
4.19.6.101 AddName()	877
4.19.6.102 GetName()	878
4.19.6.103 GetFunction()	878
4.19.6.104 GetNumber()	878
4.19.6.105 GetLastExprName()	878
4.19.6.106 GetAutoName()	878
4.19.6.107 GetVar()	878
4.19.6.108 MakeDubious()	879
4.19.6.109 GetOName()	879
4.19.6.110 DumpTree()	879
4.19.6.111 DumpNode()	879
4.19.6.112 LinkTree()	879
4.19.6.113 CopyTree()	879
4.19.6.114 CompactifyTree()	880

4.19.6.115 MakeNameTree()	880
4.19.6.116 FreeNameTree()	880
4.19.6.117 AddExpression()	880
4.19.6.118 AddSymbol()	880
4.19.6.119 AddDollar()	880
4.19.6.120 ReplaceDollar()	881
4.19.6.121 DollarRaiseLow()	881
4.19.6.122 AddVector()	881
4.19.6.123 AddDubious()	881
4.19.6.124 AddIndex()	881
4.19.6.125 DoDimension()	881
4.19.6.126 AddFunction()	882
4.19.6.127 CoCommutelnSet()	882
4.19.6.128 CoFunction()	882
4.19.6.129 TestName()	882
4.19.6.130 AddSet()	882
4.19.6.131 DoElements()	882
4.19.6.132 DoTempSet()	883
4.19.6.133 NameConflict()	883
4.19.6.134 OpenFile()	883
4.19.6.135 OpenAddFile()	883
4.19.6.136 ReOpenFile()	883
4.19.6.137 CreateFile()	883
4.19.6.138 CreateLogFile()	883
4.19.6.139 CloseFile()	884
4.19.6.140 CopyFile()	884
4.19.6.141 CreateHandle()	884
4.19.6.142 ReadFile()	884
4.19.6.143 ReadPosFile()	884
4.19.6.144 WriteFileToFile()	885
4.19.6.145 SeekFile()	885
4.19.6.146 TellFile()	885
4.19.6.147 FlushFile()	885
4.19.6.148 GetPosFile()	885
4.19.6.149 SetPosFile()	885
4.19.6.150 SynchFile()	886
4.19.6.151 TruncateFile()	886
4.19.6.152 GetChannel()	886
4.19.6.153 GetAppendChannel()	886
4.19.6.154 CloseChannel()	886
4.19.6.155 inictable()	886
4.19.6.156 findcommand()	886

4.19.6.157 inicbufs()	887
4.19.6.158 StartFiles()	887
4.19.6.159 MakeDate()	887
4.19.6.160 PreProcessor()	887
4.19.6.161 FromList()	887
4.19.6.162 From0List()	888
4.19.6.163 FromVarList()	888
4.19.6.164 DoubleList()	888
4.19.6.165 DoubleLList()	888
4.19.6.166 DoubleBuffer()	888
4.19.6.167 ExpandBuffer()	888
4.19.6.168 iexp()	889
4.19.6.169 IsLikeVector()	889
4.19.6.170 AreArgsEqual()	889
4.19.6.171 CompareArgs()	889
4.19.6.172 SkipField()	889
4.19.6.173 StrCmp()	890
4.19.6.174 StrlCmp()	890
4.19.6.175 StrHlCmp()	890
4.19.6.176 StrlCont()	890
4.19.6.177 CmpArray()	890
4.19.6.178 ConWord()	890
4.19.6.179 StrLen()	891
4.19.6.180 GetPreVar()	891
4.19.6.181 ToGeneral()	891
4.19.6.182 ToPolyFunGeneral()	891
4.19.6.183 ToFast()	891
4.19.6.184 GetSetupPar()	891
4.19.6.185 RecalcSetups()	892
4.19.6.186 AllocSetups()	892
4.19.6.187 AllocSort()	892
4.19.6.188 AllocSortFileName()	892
4.19.6.189 LoadInputFile()	892
4.19.6.190 GetInput()	892
4.19.6.191 ClearPushback()	893
4.19.6.192 GetChar()	893
4.19.6.193 CharOut()	893
4.19.6.194 UnsetAllowDelay()	893
4.19.6.195 PopPreVars()	893
4.19.6.196 IniModule()	893
4.19.6.197 IniSpecialModule()	893
4.19.6.198 ModuleInstruction()	894

4.19.6.199 PreProInstruction()	894
4.19.6.200 LoadInstruction()	894
4.19.6.201 LoadStatement()	894
4.19.6.202 FindKeyWord()	894
4.19.6.203 FindInKeyWord()	894
4.19.6.204 DoDefine()	895
4.19.6.205 DoRedefine()	895
4.19.6.206 TheDefine()	895
4.19.6.207 TheUndefine()	896
4.19.6.208 ClearMacro()	896
4.19.6.209 DoUndefine()	896
4.19.6.210 DoInclude()	896
4.19.6.211 DoReverseInclude()	896
4.19.6.212 Include()	896
4.19.6.213 DoExternal()	896
4.19.6.214 DoToExternal()	897
4.19.6.215 DoFromExternal()	897
4.19.6.216 DoPrompt()	897
4.19.6.217 DoSetExternal()	897
4.19.6.218 DoSetExternalAttr()	897
4.19.6.219 DoRmExternal()	897
4.19.6.220 DoFactDollar()	897
4.19.6.221 GetDollarNumber()	898
4.19.6.222 DoSetRandom()	898
4.19.6.223 DoOptimize()	898
4.19.6.224 DoClearOptimize()	898
4.19.6.225 DoSkipExtraSymbols()	898
4.19.6.226 DoTimeOutAfter()	898
4.19.6.227 DoMessage()	898
4.19.6.228 DoPreOut()	899
4.19.6.229 DoPreAppend()	899
4.19.6.230 DoPreCreate()	899
4.19.6.231 DoPreAssign()	899
4.19.6.232 DoPreBreak()	899
4.19.6.233 DoPreDefault()	899
4.19.6.234 DoPreSwitch()	899
4.19.6.235 DoPreEndSwitch()	900
4.19.6.236 DoPreCase()	900
4.19.6.237 DoPreShow()	900
4.19.6.238 DoPreExchange()	900
4.19.6.239 DoSystem()	900
4.19.6.240 DoPipe()	900

4.19.6.241 StartPrepro()	900
4.19.6.242 Dolfdef()	901
4.19.6.243 Dolfydef()	901
4.19.6.244 Dolfndef()	901
4.19.6.245 DoElse()	901
4.19.6.246 DoElseif()	901
4.19.6.247 DoEndif()	901
4.19.6.248 DoTerminate()	901
4.19.6.249 Dolf()	902
4.19.6.250 DoCall()	902
4.19.6.251 DoDebug()	902
4.19.6.252 DoContinueDo()	902
4.19.6.253 DoDo()	902
4.19.6.254 DoBreakDo()	902
4.19.6.255 DoEnddo()	903
4.19.6.256 DoEndprocedure()	903
4.19.6.257 DoInside()	903
4.19.6.258 DoEndInside()	903
4.19.6.259 DoProcedure()	903
4.19.6.260 DoPrePrintTimes()	903
4.19.6.261 DoPreWrite()	903
4.19.6.262 DoPreClose()	904
4.19.6.263 DoPreRemove()	904
4.19.6.264 DoPreSortReallocate()	904
4.19.6.265 DoCommentChar()	904
4.19.6.266 DoProcExtension()	904
4.19.6.267 DoPreReset()	904
4.19.6.268 WriteString()	904
4.19.6.269 WriteUnfinString()	905
4.19.6.270 AddToString()	905
4.19.6.271 PreCalc()	905
4.19.6.272 PreEval()	905
4.19.6.273 NumToStr()	905
4.19.6.274 PreCmp()	905
4.19.6.275 PreEq()	906
4.19.6.276 pParseObject()	906
4.19.6.277 PriefEval()	906
4.19.6.278 EvalPrief()	906
4.19.6.279 PreLoad()	906
4.19.6.280 PreSkip()	907
4.19.6.281 EndOfToken()	907
4.19.6.282 SetSpecialMode()	907

4.19.6.283 MakeGlobal()	907
4.19.6.284 ExecModule()	908
4.19.6.285 ExecStore()	908
4.19.6.286 FullCleanUp()	908
4.19.6.287 DoExecStatement()	908
4.19.6.288 DoPipeStatement()	908
4.19.6.289 DoPolyfun()	908
4.19.6.290 DoPolyratfun()	908
4.19.6.291 CompileStatement()	909
4.19.6.292 ToToken()	909
4.19.6.293 GetDollar()	909
4.19.6.294 MesWork()	909
4.19.6.295 MesPrint()	909
4.19.6.296 MesCall()	910
4.19.6.297 NumCopy()	910
4.19.6.298 LongCopy()	910
4.19.6.299 LongLongCopy()	910
4.19.6.300 ReserveTempFiles()	910
4.19.6.301 PrintTerm()	910
4.19.6.302 PrintTermC()	911
4.19.6.303 PrintSubTerm()	911
4.19.6.304 PrintWords()	911
4.19.6.305 PrintSeq()	911
4.19.6.306 ExpandTripleDots()	911
4.19.6.307 ComPress()	911
4.19.6.308 StageSort()	912
4.19.6.309 M_free()	912
4.19.6.310 ClearWildcardNames()	912
4.19.6.311 AddWildcardName()	912
4.19.6.312 GetWildcardName()	913
4.19.6.313 Globalize()	913
4.19.6.314 ResetVariables()	913
4.19.6.315 AddToPreTypes()	913
4.19.6.316 MessPreNesting()	913
4.19.6.317 GetStreamPosition()	913
4.19.6.318 DoubleCbuffer()	913
4.19.6.319 AddLHS()	914
4.19.6.320 AddRHS()	914
4.19.6.321 AddNtoL()	915
4.19.6.322 AddNtoC()	916
4.19.6.323 DoubleIfBuffers()	916
4.19.6.324 CreateStream()	916

4.19.6.325 setonoff()	917
4.19.6.326 DoPrint()	917
4.19.6.327 SetExpr()	917
4.19.6.328 AddToCom()	917
4.19.6.329 Add2ComStrings()	917
4.19.6.330 DoSymmetrize()	917
4.19.6.331 DoArgument()	918
4.19.6.332 ArgFactorize()	918
4.19.6.333 TakeArgContent()	918
4.19.6.334 MakeInteger()	919
4.19.6.335 MakeMod()	919
4.19.6.336 FindArg()	919
4.19.6.337 InsertArg()	920
4.19.6.338 CleanupArgCache()	920
4.19.6.339 ArgSymbolMerge()	921
4.19.6.340 ArgDotproductMerge()	921
4.19.6.341 SortWeights()	921
4.19.6.342 DoBrackets()	921
4.19.6.343 DoPutInside()	922
4.19.6.344 CountComp()	922
4.19.6.345 CoAntiBracket()	922
4.19.6.346 CoAntiSymmetrize()	922
4.19.6.347 DoArgPlode()	922
4.19.6.348 CoArgExplode()	922
4.19.6.349 CoArgImplode()	923
4.19.6.350 CoArgument()	923
4.19.6.351 CoInside()	923
4.19.6.352 ExecInside()	923
4.19.6.353 CoInExpression()	923
4.19.6.354 CoInParallel()	923
4.19.6.355 CoNotInParallel()	923
4.19.6.356 DoInParallel()	924
4.19.6.357 CoEndInExpression()	924
4.19.6.358 CoBracket()	924
4.19.6.359 CoPutInside()	924
4.19.6.360 CoAntiPutInside()	924
4.19.6.361 CoMultiBracket()	924
4.19.6.362 CoCFunction()	924
4.19.6.363 CoCTensor()	925
4.19.6.364 CoCollect()	925
4.19.6.365 CoCompress()	925
4.19.6.366 CoContract()	925

4.19.6.367 CoCycleSymmetrize()	925
4.19.6.368 CoDelete()	925
4.19.6.369 CoTableBase()	925
4.19.6.370 CoApply()	926
4.19.6.371 CoDenominators()	926
4.19.6.372 CoDimension()	926
4.19.6.373 CoDiscard()	926
4.19.6.374 CoDisorder()	926
4.19.6.375 CoDrop()	926
4.19.6.376 CoDropCoefficient()	926
4.19.6.377 CoDropSymbols()	927
4.19.6.378 CoElse()	927
4.19.6.379 CoElseif()	927
4.19.6.380 CoEndArgument()	927
4.19.6.381 CoEndInside()	927
4.19.6.382 CoEndIf()	927
4.19.6.383 CoEndRepeat()	927
4.19.6.384 CoEndTerm()	928
4.19.6.385 CoEndWhile()	928
4.19.6.386 CoExit()	928
4.19.6.387 CoFactArg()	928
4.19.6.388 CoFactDollar()	928
4.19.6.389 CoFactorize()	928
4.19.6.390 CoNFactorize()	928
4.19.6.391 CoUnFactorize()	929
4.19.6.392 CoNUnFactorize()	929
4.19.6.393 DoFactorize()	929
4.19.6.394 CoFill()	929
4.19.6.395 CoFillExpression()	929
4.19.6.396 CoFixIndex()	929
4.19.6.397 CoFormat()	929
4.19.6.398 CoGlobal()	930
4.19.6.399 CoGlobalFactorized()	930
4.19.6.400 CoGoTo()	930
4.19.6.401 Cold()	930
4.19.6.402 ColdNew()	930
4.19.6.403 ColdOld()	930
4.19.6.404 Colf()	930
4.19.6.405 ColfMatch()	931
4.19.6.406 ColfNoMatch()	931
4.19.6.407 ColIndex()	931
4.19.6.408 ColInsideFirst()	931

4.19.6.409 CoKeep()	931
4.19.6.410 CoLabel()	931
4.19.6.411 CoLoad()	931
4.19.6.412 CoLocal()	932
4.19.6.413 CoLocalFactorized()	932
4.19.6.414 CoMany()	932
4.19.6.415 CoMerge()	932
4.19.6.416 CoShuffle()	932
4.19.6.417 CoMetric()	932
4.19.6.418 CoModOption()	932
4.19.6.419 CoModuleOption()	933
4.19.6.420 CoModulus()	933
4.19.6.421 CoMulti()	933
4.19.6.422 CoMultiply()	933
4.19.6.423 CoNFunction()	933
4.19.6.424 CoNPrint()	933
4.19.6.425 CoNTensor()	933
4.19.6.426 CoNWrite()	934
4.19.6.427 CoNoDrop()	934
4.19.6.428 CoNoSkip()	934
4.19.6.429 CoNormalize()	934
4.19.6.430 CoMakeInteger()	934
4.19.6.431 CoFlags()	934
4.19.6.432 CoOff()	934
4.19.6.433 CoOn()	935
4.19.6.434 CoOnce()	935
4.19.6.435 CoOnly()	935
4.19.6.436 CoOptimizeOption()	935
4.19.6.437 CoPolyFun()	935
4.19.6.438 CoPolyRatFun()	935
4.19.6.439 CoPrint()	935
4.19.6.440 CoPrintB()	936
4.19.6.441 CoProperCount()	936
4.19.6.442 CoUnitTrace()	936
4.19.6.443 CoRCycleSymmetrize()	936
4.19.6.444 CoRatio()	936
4.19.6.445 CoRedefine()	936
4.19.6.446 CoRenumber()	936
4.19.6.447 CoRepeat()	937
4.19.6.448 CoSave()	937
4.19.6.449 CoSelect()	937
4.19.6.450 CoSet()	937

4.19.6.451 CoSetExitFlag()	937
4.19.6.452 CoSkip()	937
4.19.6.453 CoProcessBucket()	937
4.19.6.454 CoPushHide()	938
4.19.6.455 CoPopHide()	938
4.19.6.456 CoHide()	938
4.19.6.457 CoIntoHide()	938
4.19.6.458 CoNoIntoHide()	938
4.19.6.459 CoNoHide()	938
4.19.6.460 CoUnHide()	938
4.19.6.461 CoNoUnHide()	939
4.19.6.462 CoSort()	939
4.19.6.463 CoSplitArg()	939
4.19.6.464 CoSplitFirstArg()	939
4.19.6.465 CoSplitLastArg()	939
4.19.6.466 CoSum()	939
4.19.6.467 CoSymbol()	939
4.19.6.468 CoSymmetrize()	940
4.19.6.469 DoTable()	940
4.19.6.470 CoTable()	940
4.19.6.471 CoTerm()	940
4.19.6.472 CoNTable()	940
4.19.6.473 CoCTable()	940
4.19.6.474 EmptyTable()	940
4.19.6.475 CoToTensor()	941
4.19.6.476 CoToVector()	941
4.19.6.477 CoTrace4()	941
4.19.6.478 CoTraceN()	941
4.19.6.479 CoChisholm()	941
4.19.6.480 CoTransform()	941
4.19.6.481 CoClearTable()	941
4.19.6.482 DoChain()	942
4.19.6.483 CoChainin()	942
4.19.6.484 CoChainout()	942
4.19.6.485 CoTryReplace()	942
4.19.6.486 CoVector()	942
4.19.6.487 CoWhile()	942
4.19.6.488 CoWrite()	942
4.19.6.489 CoAuto()	943
4.19.6.490 CoSwitch()	943
4.19.6.491 CoCase()	943
4.19.6.492 CoBreak()	943

4.19.6.493 CoDefault()	943
4.19.6.494 CoEndSwitch()	943
4.19.6.495 CoTBaddto()	943
4.19.6.496 CoTBaudit()	944
4.19.6.497 CoTbcleanup()	944
4.19.6.498 CoTBcreate()	944
4.19.6.499 CoTBenter()	944
4.19.6.500 CoTBhelp()	944
4.19.6.501 CoTBload()	944
4.19.6.502 CoTBoff()	944
4.19.6.503 CoTBon()	945
4.19.6.504 CoTBopen()	945
4.19.6.505 CoTBreplace()	945
4.19.6.506 CoTBuse()	945
4.19.6.507 CoTestUse()	945
4.19.6.508 CoThreadBucket()	945
4.19.6.509 AddComString()	945
4.19.6.510 CompileAlgebra()	946
4.19.6.511 IsIdStatement()	946
4.19.6.512 IsRHS()	946
4.19.6.513 ParenthesesTest()	946
4.19.6.514 tokenize()	946
4.19.6.515 WriteTokens()	946
4.19.6.516 simp1token()	947
4.19.6.517 simpwtoken()	947
4.19.6.518 simp2token()	947
4.19.6.519 simp3atoken()	947
4.19.6.520 simp3btoken()	947
4.19.6.521 simp4token()	947
4.19.6.522 simp5token()	948
4.19.6.523 simp6token()	948
4.19.6.524 SkipAName()	948
4.19.6.525 TestTables()	948
4.19.6.526 GetLabel()	949
4.19.6.527 ColdExpression()	949
4.19.6.528 CoAssign()	949
4.19.6.529 DoExpr()	949
4.19.6.530 CompileSubExpressions()	949
4.19.6.531 CodeGenerator()	949
4.19.6.532 CompleteTerm()	950
4.19.6.533 CodeFactors()	950
4.19.6.534 GenerateFactors()	950

4.19.6.535 InsTree()	950
4.19.6.536 FindTree()	950
4.19.6.537 RedoTree()	950
4.19.6.538 ClearTree()	951
4.19.6.539 CatchDollar()	951
4.19.6.540 AssignDollar()	951
4.19.6.541 WriteDollarToBuffer()	951
4.19.6.542 WriteDollarFactorToBuffer()	951
4.19.6.543 AddToDollarBuffer()	951
4.19.6.544 PutTermInDollar()	952
4.19.6.545 TermAssign()	952
4.19.6.546 WildDollars()	952
4.19.6.547 numcommute()	952
4.19.6.548 FullRenummer()	952
4.19.6.549 Lus()	952
4.19.6.550 FindLus()	953
4.19.6.551 CoReplaceLoop()	953
4.19.6.552 CoFindLoop()	953
4.19.6.553 DoFindLoop()	953
4.19.6.554 CoFunPowers()	953
4.19.6.555 SortTheList()	953
4.19.6.556 MatchIsPossible()	954
4.19.6.557 StudyPattern()	954
4.19.6.558 DolToTensor()	954
4.19.6.559 DolToFunction()	954
4.19.6.560 DolToVector()	954
4.19.6.561 DolToNumber()	954
4.19.6.562 DolToSymbol()	954
4.19.6.563 DolToIndex()	955
4.19.6.564 DolToLong()	955
4.19.6.565 DollarFactorize()	955
4.19.6.566 CoPrintTable()	955
4.19.6.567 CoDeallocateTable()	955
4.19.6.568 CleanDollarFactors()	955
4.19.6.569 TakeDollarContent()	955
4.19.6.570 MakeDollarInteger()	956
4.19.6.571 MakeDollarMod()	956
4.19.6.572 GetDolNum()	957
4.19.6.573 AddPotModdollar()	957
4.19.6.574 Optimize()	958
4.19.7 Description	958
4.19.7.1 ClearOptimize()	959

4.19.8 Description	959
4.19.8.1 TakeLongRoot()	960
4.19.8.2 TakeRatRoot()	960
4.19.8.3 MakeRational()	960
4.19.8.4 MakeLongRational()	960
4.19.8.5 ClearTableTree()	961
4.19.8.6 InsTableTree()	961
4.19.8.7 RedoTableTree()	961
4.19.8.8 FindTableTree()	961
4.19.8.9 finishcbuf()	961
4.19.8.10 clearcbuf()	962
4.19.8.11 CleanUpSort()	962
4.19.8.12 AllocFileHandle()	962
4.19.8.13 DeAllocFileHandle()	962
4.19.8.14 LowerSortLevel()	962
4.19.8.15 PolyRatFunSpecial()	963
4.19.8.16 SimpleSplitMergeRec()	963
4.19.8.17 SimpleSplitMerge()	963
4.19.8.18 BinarySearch()	963
4.19.8.19 InsideDollar()	963
4.19.8.20 DoItoTerms()	963
4.19.8.21 EvalDoLoopArg()	964
4.19.8.22 SetExprCases()	964
4.19.8.23 TestSelect()	964
4.19.8.24 SubstInAll()	965
4.19.8.25 TransferBuffer()	965
4.19.8.26 TakeIDfunction()	965
4.19.8.27 MakeSetupAllocs()	965
4.19.8.28 TryFileSetups()	965
4.19.8.29 ExchangeExpressions()	965
4.19.8.30 ExchangeDollars()	966
4.19.8.31 GetFirstBracket()	966
4.19.8.32 GetFirstTerm()	966
4.19.8.33 GetContent()	966
4.19.8.34 CleanupTerm()	967
4.19.8.35 ContentMerge()	967
4.19.8.36 PrelfDollarEval()	967
4.19.8.37 TermsInDollar()	967
4.19.8.38 SizeOfDollar()	967
4.19.8.39 TermsInExpression()	967
4.19.8.40 SizeOfExpression()	967
4.19.8.41 TranslateExpression()	968

4.19.8.42 IsSetMember()	968
4.19.8.43 IsMultipleOf()	968
4.19.8.44 TwoExprCompare()	968
4.19.8.45 UpdatePositions()	968
4.19.8.46 M_print()	968
4.19.8.47 M_check1()	969
4.19.8.48 PrintTime()	969
4.19.8.49 FindBracket()	969
4.19.8.50 PutBracketInIndex()	969
4.19.8.51 ClearBracketIndex()	969
4.19.8.52 OpenBracketIndex()	969
4.19.8.53 DoNoParallel()	969
4.19.8.54 DoParallel()	970
4.19.8.55 DoModSum()	970
4.19.8.56 DoModMax()	970
4.19.8.57 DoModMin()	970
4.19.8.58 DoModLocal()	970
4.19.8.59 DoModDollar()	970
4.19.8.60 DoProcessBucket()	970
4.19.8.61 DoinParallel()	971
4.19.8.62 DonotinParallel()	971
4.19.8.63 FlipTable()	971
4.19.8.64 ChainIn()	971
4.19.8.65 ChainOut()	971
4.19.8.66 ArgumentImplode()	971
4.19.8.67 ArgumentExplode()	972
4.19.8.68 DenToFunction()	972
4.19.8.69 HowMany()	972
4.19.8.70 RemoveDollars()	972
4.19.8.71 CountTerms1()	972
4.19.8.72 TermsInBracket()	972
4.19.8.73 Crash()	973
4.19.8.74 str_dup()	973
4.19.8.75 convertblock()	973
4.19.8.76 convertnamesblock()	973
4.19.8.77 convertiniinfo()	973
4.19.8.78 ReadIndex()	973
4.19.8.79 WriteIndexBlock()	974
4.19.8.80 WriteNamesBlock()	974
4.19.8.81 WriteIndex()	974
4.19.8.82 WriteIniInfo()	974
4.19.8.83 ReadIniInfo()	974

4.19.8.84 AddToIndex()	974
4.19.8.85 GetDbase()	975
4.19.8.86 OpenDbase()	975
4.19.8.87 ReadObject()	975
4.19.8.88 ReadijObject()	975
4.19.8.89 ExistsObject()	975
4.19.8.90 DeleteObject()	975
4.19.8.91 WriteObject()	976
4.19.8.92 AddObject()	976
4.19.8.93 NewDbase()	976
4.19.8.94 FreeTableBase()	976
4.19.8.95 ComposeTableNames()	976
4.19.8.96 PutTableNames()	976
4.19.8.97 AddTableName()	977
4.19.8.98 GetTableName()	977
4.19.8.99 FindTableNumber()	977
4.19.8.100 TryEnvironment()	977
4.19.8.101 CopyExpression()	977
4.19.8.102 set_in()	977
4.19.8.103 set_set()	978
4.19.8.104 set_del()	978
4.19.8.105 set_sub()	978
4.19.8.106 DoPreAddSeparator()	978
4.19.8.107 DoPreRmSeparator()	978
4.19.8.108 openExternalChannel()	978
4.19.8.109 initPresetExternalChannels()	979
4.19.8.110 closeExternalChannel()	979
4.19.8.111 selectExternalChannel()	979
4.19.8.112 getCurrentExternalChannel()	979
4.19.8.113 closeAllExternalChannels()	979
4.19.8.114 defineChannel()	979
4.19.8.115 writeToChannel()	980
4.19.8.116 ReleaseTB()	980
4.19.8.117 SymbolNormalize()	980
4.19.8.118 TestFunFlag()	980
4.19.8.119 CompareSymbols()	980
4.19.8.120 CompareHSymbols()	981
4.19.8.121 NextPrime()	981
4.19.8.122 Moebius()	981
4.19.8.123 wranf()	981
4.19.8.124 iranf()	981
4.19.8.125 iniwranf()	982

4.19.8.126 PreRandom()	982
4.19.8.127 TreatPolyRatFun()	982
4.19.8.128 ReadSaveHeader()	982
4.19.8.129 ReadSaveIndex()	982
4.19.8.130 ReadSaveExpression()	983
4.19.8.131 ReadSaveTerm32()	984
4.19.8.132 ReadSaveVariables()	985
4.19.8.133 WriteStoreHeader()	986
4.19.8.134 InitRecovery()	986
4.19.8.135 CheckRecoveryFile()	986
4.19.8.136 DeleteRecoveryFile()	987
4.19.8.137 RecoveryFilename()	987
4.19.8.138 DoRecovery()	987
4.19.8.139 DoCheckpoint()	988
4.19.8.140 NumberMallocAddMemory()	989
4.19.8.141 CacheNumberMallocAddMemory()	989
4.19.8.142 TermMallocAddMemory()	989
4.19.8.143 ExprStatus()	989
4.19.8.144 iniTools()	989
4.19.8.145 TestTerm()	989
4.19.8.146 RunTransform()	990
4.19.8.147 RunEncode()	990
4.19.8.148 RunDecode()	990
4.19.8.149 RunReplace()	990
4.19.8.150 RunImplode()	991
4.19.8.151 RunExplode()	991
4.19.8.152 TestArgNum()	991
4.19.8.153 PutArgInScratch()	991
4.19.8.154 ReadRange()	991
4.19.8.155 FindRange()	991
4.19.8.156 RunPermute()	992
4.19.8.157 RunReverse()	992
4.19.8.158 RunCycle()	992
4.19.8.159 RunAddArg()	992
4.19.8.160 RunMulArg()	992
4.19.8.161 RunIsLyndon()	992
4.19.8.162 RunToLyndon()	993
4.19.8.163 RunDropArg()	993
4.19.8.164 RunSelectArg()	993
4.19.8.165 RunDedup()	993
4.19.8.166 RunZtoHArg()	993
4.19.8.167 RunHtoZArg()	993

4.19.8.168 NormPolyTerm()	994
4.19.8.169 ConvertToPoly()	994
4.19.8.170 LocalConvertToPoly()	994
4.19.8.171 ConvertFromPoly()	994
4.19.8.172 FindSubterm()	994
4.19.8.173 FindLocalSubterm()	995
4.19.8.174 PrintSubtermList()	995
4.19.8.175 PrintExtraSymbol()	995
4.19.8.176 FindSubexpression()	995
4.19.8.177 UpdateMaxSize()	995
4.19.8.178 CoToPolynomial()	995
4.19.8.179 CoFromPolynomial()	996
4.19.8.180 CoArgToExtraSymbol()	996
4.19.8.181 CoExtraSymbols()	996
4.19.8.182 GetDoParam()	996
4.19.8.183 GetIfDollarFactor()	996
4.19.8.184 CoDo()	996
4.19.8.185 CoEndDo()	997
4.19.8.186 ExtraSymFun()	997
4.19.8.187 PruneExtraSymbols()	997
4.19.8.188 IniFbuffer()	997
4.19.8.189 GCDfunction()	997
4.19.8.190 GCDfunction3()	997
4.19.8.191 ReadPolyRatFun()	998
4.19.8.192 FromPolyRatFun()	998
4.19.8.193 PolyDiv()	998
4.19.8.194 GCDclean()	998
4.19.8.195 TakeSymbolContent()	998
4.19.8.196 GCDterms()	999
4.19.8.197 PutExtraSymbols()	999
4.19.8.198 TakeExtraSymbols()	999
4.19.8.199 MultiplyWithTerm()	999
4.19.8.200 TakeContent()	999
4.19.8.201 MergeSymbolLists()	1000
4.19.8.202 MergeDotproductLists()	1000
4.19.8.203 CreateExpression()	1000
4.19.8.204 DIVfunction()	1000
4.19.8.205 MULfunc()	1001
4.19.8.206 ConvertArgument()	1001
4.19.8.207 ExpandRat()	1001
4.19.8.208 InvPoly()	1001
4.19.8.209 TestDoLoop()	1001

4.19.8.210 TestEndDoLoop()	1001
4.19.8.211 poly_gcd()	1002
4.19.9 Description	1002
4.19.10 Notes	1002
4.19.10.1 poly_div()	1002
4.19.10.2 poly_rem()	1003
4.19.10.3 poly_inverse()	1003
4.19.10.4 poly_mul()	1003
4.19.10.5 poly_ratfun_add()	1003
4.19.11 Description	1003
4.19.12 Notes	1004
4.19.12.1 poly_ratfun_normalize()	1004
4.19.13 Description	1004
4.19.14 Notes	1005
4.19.14.1 poly_factorize_argument()	1005
4.19.15 Description	1005
4.19.16 Notes	1006
4.19.16.1 poly_factorize_dollar()	1006
4.19.17 Description	1006
4.19.18 Notes	1006
4.19.18.1 poly_factorize_expression()	1007
4.19.19 Description	1007
4.19.20 Notes	1007
4.19.20.1 poly_unfactorize_expression()	1008
4.19.21 Description	1008
4.19.22 Notes	1008
4.19.22.1 poly_free_poly_vars()	1009
4.19.22.2 optimize_print_code()	1009
4.19.23 Description	1009
4.19.23.1 DoPreAdd()	1009
4.19.23.2 DoPreUseDictionary()	1009
4.19.23.3 DoPreCloseDictionary()	1009
4.19.23.4 DoPreOpenDictionary()	1010
4.19.23.5 RemoveDictionary()	1010
4.19.23.6 UnSetDictionary()	1010
4.19.23.7 SetDictionaryOptions()	1010
4.19.23.8 AddToDictionary()	1010
4.19.23.9 AddDictionary()	1010
4.19.23.10 FindDictionary()	1010
4.19.23.11 IsExponentSign()	1011
4.19.23.12 IsMultiplySign()	1011
4.19.23.13 TransformRational()	1011

4.19.23.14 WriteDictionary()	1011
4.19.23.15 ShrinkDictionary()	1011
4.19.23.16 MultiplyToLine()	1011
4.19.23.17 FindSymbol()	1011
4.19.23.18 FindVector()	1012
4.19.23.19 FindIndex()	1012
4.19.23.20 FindFunction()	1012
4.19.23.21 FindFunWithArgs()	1012
4.19.23.22 FindExtraSymbol()	1012
4.19.23.23 DictToBytes()	1012
4.19.23.24 DictFromBytes()	1012
4.19.23.25 CoCreateSpectator()	1013
4.19.23.26 CoToSpectator()	1013
4.19.23.27 CoRemoveSpectator()	1013
4.19.23.28 CoEmptySpectator()	1013
4.19.23.29 CoCopySpectator()	1013
4.19.23.30 PutInSpectator()	1013
4.19.23.31 ClearSpectators()	1013
4.19.23.32 GetFromSpectator()	1014
4.19.23.33 FlushSpectators()	1014
4.19.23.34 ReadFromScratch()	1014
4.19.23.35 AddToScratch()	1014
4.19.23.36 DoPreAppendPath()	1014
4.19.23.37 DoPrePrependPath()	1015
4.19.23.38 DoSwitch()	1015
4.19.23.39 DoEndSwitch()	1015
4.19.23.40 FindCase()	1015
4.19.23.41 SwitchSplitMergeRec()	1015
4.19.23.42 SwitchSplitMerge()	1015
4.19.23.43 DoubleSwitchBuffers()	1016
4.19.23.44 DistrN()	1016
4.19.23.45 DoNamespace()	1016
4.19.23.46 DoEndNamespace()	1016
4.19.23.47 DoUse()	1016
4.19.23.48 SkipName()	1016
4.19.23.49 ConstructName()	1017
4.19.23.50 DoSetUserFlag()	1017
4.19.23.51 DoClearUserFlag()	1017
4.19.23.52 CreateVertex()	1017
4.19.23.53 ReadParticle()	1017
4.19.23.54 CoModel()	1017
4.19.23.55 CoParticle()	1018

4.19.23.56 CoVertex()	1018
4.19.23.57 CoEndModel()	1018
4.19.23.58 LoadModel()	1018
4.19.23.59 CoSetUserFlag()	1018
4.19.23.60 CoClearUserFlag()	1018
4.19.23.61 CoCreateAllLoops()	1018
4.19.23.62 CoCreateAllPaths()	1019
4.19.23.63 CoCreateAll()	1019
4.19.23.64 AllLoops()	1019
4.19.23.65 StartLoops()	1019
4.19.23.66 GenLoops()	1019
4.19.23.67 LoopOutput()	1020
4.19.23.68 AllPaths()	1020
4.19.23.69 GenPaths()	1020
4.19.23.70 PathOutput()	1020
4.20 declare.h	1021
4.21 diagrams.c File Reference	1041
4.21.1 Detailed Description	1041
4.21.2 Function Documentation	1041
4.21.2.1 CoCanonicalize()	1041
4.21.2.2 DoCanonicalize()	1041
4.21.2.3 DoTopologyCanonicalize()	1042
4.21.2.4 DoShattering()	1042
4.22 diagrams.c	1042
4.23 diawrap.cc File Reference	1050
4.23.1 Detailed Description	1051
4.23.2 Macro Definition Documentation	1051
4.23.2.1 MAXPOINTS	1051
4.23.3 Function Documentation	1051
4.23.3.1 LoadModel()	1051
4.23.3.2 ConvertParticle()	1051
4.23.3.3 ReConvertParticle()	1051
4.23.3.4 numParticle()	1051
4.23.3.5 fendMG()	1052
4.23.3.6 ProcessTopology()	1052
4.23.3.7 SetDualOpts()	1052
4.23.3.8 GenDiagrams()	1052
4.24 diawrap.cc	1052
4.25 dict.c File Reference	1065
4.25.1 Detailed Description	1065
4.25.2 Function Documentation	1066
4.25.2.1 TransformRational()	1066

4.25.2.2 IsMultiplySign()	1066
4.25.2.3 IsExponentSign()	1066
4.25.2.4 FindSymbol()	1066
4.25.2.5 FindVector()	1066
4.25.2.6 FindIndex()	1066
4.25.2.7 FindFunction()	1067
4.25.2.8 FindFunWithArgs()	1067
4.25.2.9 FindExtraSymbol()	1067
4.25.2.10 FindDictionary()	1067
4.25.2.11 AddDictionary()	1067
4.25.2.12 AddToDictionary()	1067
4.25.2.13 UseDictionary()	1068
4.25.2.14 SetDictionaryOptions()	1068
4.25.2.15 UnSetDictionary()	1068
4.25.2.16 RemoveDictionary()	1068
4.25.2.17 ShrinkDictionary()	1068
4.25.2.18 DoPreOpenDictionary()	1068
4.25.2.19 DoPreCloseDictionary()	1068
4.25.2.20 DoPreUseDictionary()	1069
4.25.2.21 DoPreAdd()	1069
4.25.2.22 DictToBytes()	1069
4.25.2.23 DictFromBytes()	1069
4.26 dict.c	1069
4.27 dollar.c File Reference	1083
4.27.1 Detailed Description	1084
4.27.2 Macro Definition Documentation	1084
4.27.2.1 STEP2	1084
4.27.3 Function Documentation	1084
4.27.3.1 CatchDollar()	1084
4.27.3.2 AssignDollar()	1084
4.27.3.3 WriteDollarToBuffer()	1085
4.27.3.4 WriteDollarFactorToBuffer()	1085
4.27.3.5 AddToDollarBuffer()	1085
4.27.3.6 TermAssign()	1085
4.27.3.7 PutTermInDollar()	1085
4.27.3.8 WildDollars()	1085
4.27.3.9 DolToTensor()	1086
4.27.3.10 DolToFunction()	1086
4.27.3.11 DolToVector()	1086
4.27.3.12 DolToNumber()	1086
4.27.3.13 DolToSymbol()	1086
4.27.3.14 DolToIndex()	1086

4.27.3.15 DolToTerms()	1086
4.27.3.16 DolToLong()	1087
4.27.3.17 ExecInside()	1087
4.27.3.18 InsideDollar()	1087
4.27.3.19 ExchangeDollars()	1087
4.27.3.20 TermsInDollar()	1087
4.27.3.21 SizeOfDollar()	1087
4.27.3.22 PriefDollarEval()	1088
4.27.3.23 TranslateExpression()	1088
4.27.3.24 IsSetMember()	1088
4.27.3.25 IsMultipleOf()	1088
4.27.3.26 TwoExprCompare()	1088
4.27.3.27 DollarRaiseLow()	1088
4.27.3.28 EvalDoLoopArg()	1089
4.27.3.29 TestDoLoop()	1089
4.27.3.30 TestEndDoLoop()	1089
4.27.3.31 DollarFactorize()	1090
4.27.3.32 CleanDollarFactors()	1090
4.27.3.33 TakeDollarContent()	1090
4.27.3.34 MakeDollarInteger()	1090
4.27.3.35 MakeDollarMod()	1091
4.27.3.36 GetDolNum()	1092
4.27.3.37 AddPotModdollar()	1092
4.28 dollar.c	1093
4.29 evaluate.c File Reference	1138
4.29.1 Detailed Description	1139
4.29.2 Macro Definition Documentation	1139
4.29.2.1 EXTRAPRECISION	1139
4.29.2.2 RND	1139
4.29.2.3 auxr1	1139
4.29.2.4 auxr2	1139
4.29.2.5 auxr3	1139
4.29.2.6 auxr4	1139
4.29.2.7 auxr5	1140
4.29.3 Function Documentation	1140
4.29.3.1 PackFloat()	1140
4.29.3.2 UnpackFloat()	1140
4.29.3.3 RatToFloat()	1140
4.29.3.4 FormtoZ()	1140
4.29.3.5 IntegerToFloat()	1140
4.29.3.6 RatToFloat()	1141
4.29.3.7 SetFloatPrecision()	1141

4.29.3.8 ClearFloat()	1141
4.29.3.9 GetFloatArgument()	1141
4.29.3.10 GetPiArgument()	1141
4.29.3.11 EvaluateFun()	1141
4.29.4 Variable Documentation	1142
4.29.4.1 ln2	1142
4.30 evaluate.c	1142
4.31 execute.c File Reference	1151
4.31.1 Detailed Description	1151
4.31.2 Macro Definition Documentation	1152
4.31.2.1 CURRENTBRACKET	1152
4.31.2.2 BRACKETCURRENTEXPR	1152
4.31.2.3 BRACKETOTHEREXPR	1152
4.31.2.4 NOBRACKETACTIVE	1152
4.31.3 Function Documentation	1152
4.31.3.1 CleanExpr()	1152
4.31.3.2 PopVariables()	1152
4.31.3.3 MakeGlobal()	1152
4.31.3.4 TestDrop()	1153
4.31.3.5 PutInVflags()	1153
4.31.3.6 DoExecute()	1153
4.31.3.7 PutBracket()	1153
4.31.3.8 SpecialCleanup()	1153
4.31.3.9 SetMods()	1153
4.31.3.10 UnSetMods()	1153
4.31.3.11 ExchangeExpressions()	1154
4.31.3.12 GetFirstBracket()	1154
4.31.3.13 GetFirstTerm()	1154
4.31.3.14 GetContent()	1154
4.31.3.15 CleanupTerm()	1155
4.31.3.16 ContentMerge()	1155
4.31.3.17 TermsInExpression()	1155
4.31.3.18 SizeOfExpression()	1155
4.31.3.19 UpdatePositions()	1155
4.31.3.20 CountTerms1()	1155
4.31.3.21 TermsInBracket()	1155
4.32 execute.c	1156
4.33 extcmd.c File Reference	1187
4.33.1 Detailed Description	1188
4.33.2 Function Documentation	1188
4.33.2.1 openExternalChannel()	1188
4.33.2.2 initPresetExternalChannels()	1188

4.33.2.3	closeExternalChannel()	1188
4.33.2.4	selectExternalChannel()	1188
4.33.2.5	getCurrentExternalChannel()	1189
4.33.2.6	closeAllExternalChannels()	1189
4.34	extcmd.c	1189
4.35	factor.c File Reference	1208
4.35.1	Detailed Description	1208
4.35.2	Function Documentation	1208
4.35.2.1	FactorIn()	1208
4.35.2.2	FactorInExpr()	1209
4.36	factor.c	1209
4.37	findpat.c File Reference	1219
4.37.1	Detailed Description	1219
4.37.2	Function Documentation	1220
4.37.2.1	FindOnly()	1220
4.37.2.2	FindOnce()	1220
4.37.2.3	FindMulti()	1220
4.37.2.4	FindRest()	1220
4.38	findpat.c	1220
4.39	flintinterface.cc File Reference	1237
4.39.1	Detailed Description	1237
4.39.2	Macro Definition Documentation	1237
4.39.2.1	UNIVARIATE_DENSITY_THR	1237
4.39.2.2	IFW	1237
4.40	flintinterface.cc	1238
4.41	flintinterface.h File Reference	1263
4.41.1	Detailed Description	1265
4.41.2	Typedef Documentation	1265
4.41.2.1	var_map_t	1265
4.41.3	Function Documentation	1266
4.41.3.1	cleanup()	1266
4.41.3.2	cleanup_master()	1266
4.41.3.3	divmod_mpoly()	1266
4.41.3.4	divmod_poly()	1266
4.41.3.5	factorize_mpoly()	1266
4.41.3.6	factorize_poly()	1267
4.41.3.7	form_sort()	1267
4.41.3.8	from_argument_mpoly()	1267
4.41.3.9	from_argument_poly()	1267
4.41.3.10	fmpz_get_form()	1267
4.41.3.11	fmpz_set_form()	1268
4.41.3.12	gcd_mpoly()	1268

4.41.3.13	gcd_poly()	1268
4.41.3.14	get_variables()	1268
4.41.3.15	inverse_poly()	1268
4.41.3.16	mul_mpoly()	1269
4.41.3.17	mul_poly()	1269
4.41.3.18	ratfun_add_mpoly()	1269
4.41.3.19	ratfun_add_poly()	1269
4.41.3.20	ratfun_normalize_mpoly()	1269
4.41.3.21	ratfun_normalize_poly()	1270
4.41.3.22	ratfun_read_mpoly()	1270
4.41.3.23	ratfun_read_poly()	1270
4.41.3.24	to_argument_mpoly() [1/2]	1270
4.41.3.25	to_argument_mpoly() [2/2]	1270
4.41.3.26	to_argument_poly() [1/2]	1271
4.41.3.27	to_argument_poly() [2/2]	1271
4.41.3.28	simplify_fmpz()	1271
4.41.3.29	simplify_fmpz_poly()	1271
4.41.3.30	fix_sign_fmpz_mpoly_ratfun()	1271
4.41.3.31	fix_sign_fmpz_poly_ratfun()	1272
4.42	flintinterface.h	1272
4.43	flintwrap.cc File Reference	1274
4.43.1	Detailed Description	1275
4.43.2	Function Documentation	1275
4.43.2.1	flint_final_cleanup_thread()	1275
4.43.2.2	flint_final_cleanup_master()	1275
4.43.2.3	flint_div()	1275
4.43.2.4	flint_factorize_argument()	1275
4.43.2.5	flint_factorize_dollar()	1275
4.43.2.6	flint_gcd()	1276
4.43.2.7	flint_inverse()	1276
4.43.2.8	flint_mul()	1276
4.43.2.9	flint_ratfun_add()	1276
4.43.2.10	flint_ratfun_normalize()	1276
4.43.2.11	flint_rem()	1276
4.43.2.12	flint_check_version()	1277
4.44	flintwrap.cc	1277
4.45	float.c File Reference	1281
4.45.1	Detailed Description	1282
4.45.2	Macro Definition Documentation	1282
4.45.2.1	WITHCUTOFF	1282
4.45.2.2	GMPSREAD	1283
4.45.3	Function Documentation	1283

4.45.3.1 Form_mpf_init()	1283
4.45.3.2 Form_mpf_clear()	1283
4.45.3.3 Form_mpf_set_prec_raw()	1283
4.45.3.4 FormtoZ()	1283
4.45.3.5 ZtoForm()	1283
4.45.3.6 FloatToInteger()	1284
4.45.3.7 IntegerToFloat()	1284
4.45.3.8 FloatToRat()	1284
4.45.3.9 AddFloats()	1284
4.45.3.10 MulFloats()	1284
4.45.3.11 DivFloats()	1284
4.45.3.12 AddRatToFloat()	1285
4.45.3.13 MulRatToFloat()	1285
4.45.3.14 SimpleDelta()	1285
4.45.3.15 SimpleDeltaC()	1285
4.45.3.16 SingleTable()	1285
4.45.3.17 DoubleTable()	1286
4.45.3.18 EndTable()	1286
4.45.3.19 deltaMZV()	1286
4.45.3.20 deltaEuler()	1286
4.45.3.21 deltaEulerC()	1286
4.45.3.22 CalculateMZVhalf()	1287
4.45.3.23 CalculateMZV()	1287
4.45.3.24 CalculateEuler()	1287
4.45.3.25 ExpandMZV()	1287
4.45.3.26 ExpandEuler()	1287
4.45.3.27 PackFloat()	1287
4.45.3.28 UnpackFloat()	1288
4.45.3.29 RatToFloat()	1288
4.45.3.30 Form_mpf_empty()	1288
4.45.3.31 TestFloat()	1288
4.45.3.32 FloatFunToRat()	1288
4.45.3.33 ZeroTable()	1288
4.45.3.34 ReadFloat()	1289
4.45.3.35 CheckFloat()	1289
4.45.3.36 SetFloatPrecision()	1289
4.45.3.37 PrintFloat()	1289
4.45.3.38 SetupMZVTables()	1289
4.45.3.39 SetupMPFTables()	1289
4.45.3.40 ClearMZVTables()	1289
4.45.3.41 CoToFloat()	1290
4.45.3.42 CoToRat()	1290

4.45.3.43 CoStrictRounding()	1290
4.45.3.44 CoChop()	1290
4.45.3.45 ToFloat()	1290
4.45.3.46 ToRat()	1290
4.45.3.47 StrictRounding()	1291
4.45.3.48 Chop()	1291
4.45.3.49 AddWithFloat()	1291
4.45.3.50 MergeWithFloat()	1291
4.45.3.51 EvaluateEuler()	1291
4.45.3.52 CoEvaluate()	1291
4.46 float.c	1292
4.47 form3.h File Reference	1323
4.47.1 Detailed Description	1325
4.47.2 Macro Definition Documentation	1326
4.47.2.1 MAJORVERSION	1326
4.47.2.2 MINORVERSION	1326
4.47.2.3 PRODUCTIONDATE	1326
4.47.2.4 BETAVERSION	1326
4.47.2.5 inline	1326
4.47.2.6 NDEBUG	1326
4.47.2.7 STATIC_ASSERT	1326
4.47.2.8 STATIC_ASSERT__1	1327
4.47.2.9 STATIC_ASSERT__2	1327
4.47.2.10 STATIC_ASSERT__3	1327
4.47.2.11 BITSINWORD	1327
4.47.2.12 BITSINLONG	1327
4.47.2.13 WORD_MIN_VALUE	1327
4.47.2.14 WORD_MAX_VALUE	1327
4.47.2.15 LONG_MIN_VALUE	1328
4.47.2.16 LONG_MAX_VALUE	1328
4.47.2.17 TOPBITONLY	1328
4.47.2.18 TOPLONGBITONLY	1328
4.47.2.19 SPECMASK	1328
4.47.2.20 WILDMASK	1328
4.47.2.21 WORDMASK	1328
4.47.2.22 AWORDMASK	1328
4.47.2.23 FULLMAX	1329
4.47.2.24 MAXPOSITIVE	1329
4.47.2.25 MAXLONG	1329
4.47.2.26 MAXPOSITIVE2	1329
4.47.2.27 MAXPOSITIVE4	1329
4.47.2.28 form_alignof	1329

4.47.2.29 WITHSORTBOTS	1329
4.47.2.30 FILES	1330
4.47.2.31 Uopen	1330
4.47.2.32 Uflush	1330
4.47.2.33 Uclose	1330
4.47.2.34 Uread	1330
4.47.2.35 Uwrite	1330
4.47.2.36 Usetbuf	1331
4.47.2.37 Useek	1331
4.47.2.38 Utell	1331
4.47.2.39 Ugetpos	1331
4.47.2.40 Usetpos	1331
4.47.2.41 Usync	1331
4.47.2.42 Utruncate	1332
4.47.2.43 Ustdout	1332
4.47.2.44 MAX_OPEN_FILES	1332
4.47.2.45 bzero	1332
4.47.2.46 GetPID	1332
4.47.3 Typedef Documentation	1332
4.47.3.1 WORD	1332
4.47.3.2 LONG	1332
4.47.3.3 UWORD	1333
4.47.3.4 ULONG	1333
4.47.3.5 SBYTE	1333
4.47.3.6 UBYTE	1333
4.47.3.7 UINT	1333
4.47.3.8 RLONG	1333
4.47.3.9 MLONG	1333
4.47.3.10 BOOL	1333
4.48 form3.h	1334
4.49 fsizes.h File Reference	1338
4.49.1 Detailed Description	1340
4.49.2 Macro Definition Documentation	1340
4.49.2.1 MAXPRENAMESIZE	1340
4.49.2.2 MAXENAME	1340
4.49.2.3 MAXSAVEFUNCTION	1340
4.49.2.4 MAXPARLEVEL	1340
4.49.2.5 MAXNUMBERSIZE	1340
4.49.2.6 MAXREPEAT	1341
4.49.2.7 NORMSIZE	1341
4.49.2.8 INITNODESIZE	1341
4.49.2.9 INITNAMESIZE	1341

4.49.2.10 NUMFIXED	1341
4.49.2.11 MAXNEST	1341
4.49.2.12 MAXMATCH	1341
4.49.2.13 MAXIF	1341
4.49.2.14 SIZEFACS	1342
4.49.2.15 NUMFACS	1342
4.49.2.16 MAXLOOPS	1342
4.49.2.17 MAXLABELS	1342
4.49.2.18 COMMERCIALSIZE	1342
4.49.2.19 MAXFLAGS	1342
4.49.2.20 COMPRESSBUFFER	1342
4.49.2.21 FORTRANCONTINUATIONLINES	1342
4.49.2.22 MAXLEVELS	1343
4.49.2.23 MAXLHS	1343
4.49.2.24 MAXWILDC	1343
4.49.2.25 NUMTABLEENTRIES	1343
4.49.2.26 COMPILERBUFFER	1343
4.49.2.27 MAXPATCHES	1343
4.49.2.28 MAXFPATCHES	1343
4.49.2.29 SORTIOSIZE	1343
4.49.2.30 SSMALLBUFFER	1344
4.49.2.31 SSMALLOVERFLOW	1344
4.49.2.32 STERMSSMALL	1344
4.49.2.33 SLARGEBUFFER	1344
4.49.2.34 SMAXPATCHES	1344
4.49.2.35 SMAXFPATCHES	1344
4.49.2.36 SSORTIOSIZE	1344
4.49.2.37 SPECTATORSIZE	1344
4.49.2.38 MAXFLEVELS	1345
4.49.2.39 COMPINC	1345
4.49.2.40 MAXNUMSIZE	1345
4.49.2.41 MAXBRACKETBUFFERSIZE	1345
4.49.2.42 SFHSIZE	1345
4.49.2.43 DEFAULTPROCESSBUCKETSIZE	1345
4.49.2.44 SHMWINSIZE	1345
4.49.2.45 TABLEEXTENSION	1345
4.49.2.46 GZIPDEFAULT	1346
4.49.2.47 DEFAULTTHREADS	1346
4.49.2.48 DEFAULTTHREADBUCKETSIZE	1346
4.49.2.49 DEFAULTTHREADLOADBALANCING	1346
4.49.2.50 THREADSCRATCHSIZE	1346
4.49.2.51 THREADSCRATCHOUTSIZE	1346

4.49.2.52 NUMSTORECACHES	1346
4.49.2.53 SIZESTORECACHE	1346
4.49.2.54 INDENTSPACE	1347
4.49.2.55 MULTIINDENTSPACE	1347
4.49.2.56 MAXMULTIBRACKETLEVELS	1347
4.49.2.57 FBUFFERSIZE	1347
4.49.2.58 NPAIR1	1347
4.49.2.59 NPAIR2	1347
4.49.2.60 MAXLINELENGTH	1347
4.49.2.61 MINALLOC	1347
4.49.2.62 JUMPRATIO	1348
4.49.2.63 MAXPARTICLES	1348
4.49.2.64 MAXCOUPLINGS	1348
4.49.2.65 NUMOPTIONS	1348
4.49.2.66 MAXLEGS	1348
4.50 fsizes.h	1348
4.51 ftypes.h File Reference	1351
4.51.1 Detailed Description	1363
4.51.2 Macro Definition Documentation	1363
4.51.2.1 WITHOUTERROR	1363
4.51.2.2 WITHERROR	1364
4.51.2.3 FILESTREAM	1364
4.51.2.4 PREVARSTREAM	1364
4.51.2.5 PREREADSTREAM	1364
4.51.2.6 PIPESTREAM	1364
4.51.2.7 PRECALCSTREAM	1364
4.51.2.8 DOLLARSTREAM	1364
4.51.2.9 PREREADSTREAM2	1364
4.51.2.10 EXTERNALCHANNELSTREAM	1365
4.51.2.11 PREREADSTREAM3	1365
4.51.2.12 REVERSEFILESTREAM	1365
4.51.2.13 INPUTSTREAM	1365
4.51.2.14 ENDOFSTREAM	1365
4.51.2.15 ENDOFINPUT	1365
4.51.2.16 SUBROUTINEFILE	1365
4.51.2.17 PROCEDUREFILE	1365
4.51.2.18 HEADERFILE	1366
4.51.2.19 SETUPFILE	1366
4.51.2.20 TABLEBASEFILE	1366
4.51.2.21 FIRSTMODULE	1366
4.51.2.22 GLOBALMODULE	1366
4.51.2.23 SORTMODULE	1366

4.51.2.24 STOREMODULE	1366
4.51.2.25 CLEARMODULE	1366
4.51.2.26 ENDMODULE	1367
4.51.2.27 POLYFUN	1367
4.51.2.28 NOPARALLEL_DOLLAR	1367
4.51.2.29 NOPARALLEL_RHS	1367
4.51.2.30 NOPARALLEL_CONVPOLY	1367
4.51.2.31 NOPARALLEL_SPECTATOR	1367
4.51.2.32 NOPARALLEL_USER	1367
4.51.2.33 NOPARALLEL_TBLDOLLAR	1367
4.51.2.34 NOPARALLEL_NPROC	1368
4.51.2.35 PARALLELFLAG	1368
4.51.2.36 PRENOACTION	1368
4.51.2.37 PRERAISEAFTER	1368
4.51.2.38 PRELOWERAFTER	1368
4.51.2.39 WITHSEMICOLON	1368
4.51.2.40 WITHOUTSEMICOLON	1368
4.51.2.41 MODULEINSTACK	1368
4.51.2.42 EXECUTINGIF	1369
4.51.2.43 LOOKINGFORELSE	1369
4.51.2.44 LOOKINGFORENDIF	1369
4.51.2.45 NEWSTATEMENT	1369
4.51.2.46 OLDSTATEMENT	1369
4.51.2.47 EXECUTINGPRESWITCH	1369
4.51.2.48 SEARCHINGPRECASE	1369
4.51.2.49 SEARCHINGPREENDSWITCH	1369
4.51.2.50 PREPROONLY	1370
4.51.2.51 DUMPTOCOMPILER	1370
4.51.2.52 DUMPOUTTERMS	1370
4.51.2.53 DUMPINTERMS	1370
4.51.2.54 DUMPTOSORT	1370
4.51.2.55 DUMPTOPARALLEL	1370
4.51.2.56 THREADSDEBUG	1370
4.51.2.57 ERROROUT	1370
4.51.2.58 INPUTOUT	1371
4.51.2.59 STATSOUT	1371
4.51.2.60 EXPRSOUT	1371
4.51.2.61 WRITEOUT	1371
4.51.2.62 EXTERNALCHANNELOUT	1371
4.51.2.63 NUMERICALLOOP	1371
4.51.2.64 LISTEDLOOP	1371
4.51.2.65 ONEEXPRESSION	1371

4.51.2.66 PRETYPENONE	1372
4.51.2.67 PRETYPEIF	1372
4.51.2.68 PRETYPEDO	1372
4.51.2.69 PRETYPEPROCEDURE	1372
4.51.2.70 PRETYPESWITCH	1372
4.51.2.71 PRETYPEINSIDE	1372
4.51.2.72 DECLARATION	1372
4.51.2.73 SPECIFICATION	1372
4.51.2.74 DEFINITION	1373
4.51.2.75 STATEMENT	1373
4.51.2.76 TOOOUTPUT	1373
4.51.2.77 ATENDOFMODULE	1373
4.51.2.78 MIXED2	1373
4.51.2.79 MIXED	1373
4.51.2.80 NOAUTO	1373
4.51.2.81 PARTEST	1373
4.51.2.82 WITHAUTO	1374
4.51.2.83 ALLVARIABLES	1374
4.51.2.84 SYMBOLONLY	1374
4.51.2.85 INDEXONLY	1374
4.51.2.86 VECTORONLY	1374
4.51.2.87 FUNCTIONONLY	1374
4.51.2.88 SETONLY	1374
4.51.2.89 EXPRESSIONONLY	1374
4.51.2.90 CDELETE	1375
4.51.2.91 ANYTYPE	1375
4.51.2.92 CSYMBOL	1375
4.51.2.93 CINDEX	1375
4.51.2.94 CVECTOR	1375
4.51.2.95 CFUNCTION	1375
4.51.2.96 CSET	1375
4.51.2.97 CEXPRESSION	1375
4.51.2.98 CDOTPRODUCT	1376
4.51.2.99 CNUMBER	1376
4.51.2.100 CSUBEXP	1376
4.51.2.101 CDELTA	1376
4.51.2.102 CDOLLAR	1376
4.51.2.103 CDUBIOUS	1376
4.51.2.104 CRANGE	1376
4.51.2.105 CMODEL	1376
4.51.2.106 CVECTOR1	1377
4.51.2.107 CDOUBLEDOT	1377

4.51.2.108 TSYMBOL	1377
4.51.2.109 TINDEX	1377
4.51.2.110 TVECTOR	1377
4.51.2.111 TFUNCTION	1377
4.51.2.112 TSET	1377
4.51.2.113 TEXPRESSON	1377
4.51.2.114 TDOTPRODUCT	1378
4.51.2.115 TNUMBER	1378
4.51.2.116 TSUBEXP	1378
4.51.2.117 TDELTA	1378
4.51.2.118 TDOLLAR	1378
4.51.2.119 TDUBIOUS	1378
4.51.2.120 LPARENTHESIS	1378
4.51.2.121 RPARENTHESIS	1378
4.51.2.122 TWILDCARD	1379
4.51.2.123 TWILDARG	1379
4.51.2.124 TDOT	1379
4.51.2.125 LBRACE	1379
4.51.2.126 RBRACE	1379
4.51.2.127 TCOMMA	1379
4.51.2.128 TFUNOPEN	1379
4.51.2.129 TFUNCLOSE	1379
4.51.2.130 TMULTIPLY	1380
4.51.2.131 TDIVIDE	1380
4.51.2.132 TPOWER	1380
4.51.2.133 TPLUS	1380
4.51.2.134 TMINUS	1380
4.51.2.135 TNOT	1380
4.51.2.136 TENDOFIT	1380
4.51.2.137 TSETOPEN	1380
4.51.2.138 TSETCLOSE	1381
4.51.2.139 TGENINDEX	1381
4.51.2.140 TCONJUGATE	1381
4.51.2.141 LRPARENTHESSES	1381
4.51.2.142 TNUMBER1	1381
4.51.2.143 TPOWER1	1381
4.51.2.144 TEMPTY	1381
4.51.2.145 TSETNUM	1381
4.51.2.146 TSGAMMA	1382
4.51.2.147 TSETDOL	1382
4.51.2.148 TFLOAT	1382
4.51.2.149 TYPEISFUN	1382

4.51.2.150 TYPEISSUB	1382
4.51.2.151 TYPEISMYSTERY	1382
4.51.2.152 LHSIDEX	1382
4.51.2.153 LHSIDE	1382
4.51.2.154 RHSIDE	1383
4.51.2.155 FORTRANMODE	1383
4.51.2.156 REDUCEMODE	1383
4.51.2.157 MAPLEMODE	1383
4.51.2.158 MATHEMATICAMODE	1383
4.51.2.159 CMODE	1383
4.51.2.160 VORTRANMODE	1383
4.51.2.161 PFORTRANMODE	1383
4.51.2.162 DOUBLEFORTRANMODE	1384
4.51.2.163 DOUBLEPRECISIONFLAG	1384
4.51.2.164 NODOUBLEMASK	1384
4.51.2.165 QUADRUPLEFORTRANMODE	1384
4.51.2.166 QUADRUPLEPRECISIONFLAG	1384
4.51.2.167 ALLINTEGERDOUBLE	1384
4.51.2.168 NOQUADMASK	1384
4.51.2.169 NORMALFORMAT	1384
4.51.2.170 NOSPACEFORMAT	1385
4.51.2.171 ISNOTFORTRAN90	1385
4.51.2.172 ISFORTRAN90	1385
4.51.2.173 ALSOREVERSE	1385
4.51.2.174 CHISHOLM	1385
4.51.2.175 NOTRICK	1385
4.51.2.176 SORTLOWFIRST	1385
4.51.2.177 SORTHIGHFIRST	1385
4.51.2.178 SORTPOWERFIRST	1386
4.51.2.179 SORTANTIPOWER	1386
4.51.2.180 STATSSPLITMERGE	1386
4.51.2.181 STATSMERGETOFILE	1386
4.51.2.182 STATSPOSTSORT	1386
4.51.2.183 NOCHECKLOGTYPE	1386
4.51.2.184 CHECKLOGTYPE	1386
4.51.2.185 NMIN4SHIFT	1386
4.51.2.186 SYMBOL	1387
4.51.2.187 DOTPRODUCT	1387
4.51.2.188 VECTOR	1387
4.51.2.189 INDEX	1387
4.51.2.190 EXPRESSION	1387
4.51.2.191 SUBEXPRESSION	1387

4.51.2.192 DOLLAREXPRESSION	1387
4.51.2.193 SETSET	1387
4.51.2.194 ARGWILD	1388
4.51.2.195 MINVECTOR	1388
4.51.2.196 SETEXP	1388
4.51.2.197 DOLLAREXPR2	1388
4.51.2.198 HAAKJE0	1388
4.51.2.199 FUNCTION	1388
4.51.2.200 TMPPOLYFUN	1388
4.51.2.201 ARGFIELD	1388
4.51.2.202 SNUMBER	1389
4.51.2.203 LNUMBER	1389
4.51.2.204 HAAKJE	1389
4.51.2.205 DELTA	1389
4.51.2.206 EXPONENT	1389
4.51.2.207 DENOMINATOR	1389
4.51.2.208 SETFUNCTION	1389
4.51.2.209 GAMMA	1389
4.51.2.210 GAMMAI	1390
4.51.2.211 GAMMAFIVE	1390
4.51.2.212 GAMMASIX	1390
4.51.2.213 GAMMASEVEN	1390
4.51.2.214 SUMF1	1390
4.51.2.215 SUMF2	1390
4.51.2.216 DUMFUN	1390
4.51.2.217 REPLACEMENT	1390
4.51.2.218 REVERSEFUNCTION	1391
4.51.2.219 DISTRIBUTION	1391
4.51.2.220 DELTA3	1391
4.51.2.221 DUMMYFUN	1391
4.51.2.222 DUMMYTEN	1391
4.51.2.223 LEVICIVITA	1391
4.51.2.224 FACTORIAL	1391
4.51.2.225 INVERSEFACTORIAL	1391
4.51.2.226 BINOMIAL	1392
4.51.2.227 NUMARGSFUN	1392
4.51.2.228 SIGNFUN	1392
4.51.2.229 MODFUNCTION	1392
4.51.2.230 MOD2FUNCTION	1392
4.51.2.231 MINFUNCTION	1392
4.51.2.232 MAXFUNCTION	1392
4.51.2.233 ABSFUNCTION	1392

4.51.2.234 SIGFUNCTION	1393
4.51.2.235 INTFUNCTION	1393
4.51.2.236 THETA	1393
4.51.2.237 THETA2	1393
4.51.2.238 DELTA2	1393
4.51.2.239 DELTAP	1393
4.51.2.240 BERNOULLIFUNCTION	1393
4.51.2.241 COUNTFUNCTION	1393
4.51.2.242 MATCHFUNCTION	1394
4.51.2.243 PATTERNFUNCTION	1394
4.51.2.244 TERMFUNCTION	1394
4.51.2.245 CONJUGATION	1394
4.51.2.246 ROOTFUNCTION	1394
4.51.2.247 TABLEFUNCTION	1394
4.51.2.248 FIRSTBRACKET	1394
4.51.2.249 TERMSINEXPR	1394
4.51.2.250 NUMTERMSFUN	1395
4.51.2.251 GCDFUNCTION	1395
4.51.2.252 DIVFUNCTION	1395
4.51.2.253 REMFUNCTION	1395
4.51.2.254 MAXPOWEROF	1395
4.51.2.255 MINPOWEROF	1395
4.51.2.256 TABLESTUB	1395
4.51.2.257 FACTORIN	1395
4.51.2.258 TERMSINBRACKET	1396
4.51.2.259 WILDARGFUN	1396
4.51.2.260 SQRTFUNCTION	1396
4.51.2.261 LNFUNCTION	1396
4.51.2.262 SINFUNCTION	1396
4.51.2.263 COSFUNCTION	1396
4.51.2.264 TANFUNCTION	1396
4.51.2.265 ASINFUNCTION	1396
4.51.2.266 ACOSFUNCTION	1397
4.51.2.267 ATANFUNCTION	1397
4.51.2.268 ATAN2FUNCTION	1397
4.51.2.269 SINHFUNCTION	1397
4.51.2.270 COSHFUNCTION	1397
4.51.2.271 TANHFUNCTION	1397
4.51.2.272 ASINHFUNCTION	1397
4.51.2.273 ACOSHFUNCTION	1397
4.51.2.274 ATANHFUNCTION	1398
4.51.2.275 LI2FUNCTION	1398

4.51.2.276 LINFUNCTION	1398
4.51.2.277 EXTRASYMFUN	1398
4.51.2.278 RANDOMFUNCTION	1398
4.51.2.279 RANPERM	1398
4.51.2.280 NUMFACTORS	1398
4.51.2.281 FIRSTTERM	1398
4.51.2.282 CONTENTTERM	1399
4.51.2.283 PRIMENUMBER	1399
4.51.2.284 EXTEUCLIDEAN	1399
4.51.2.285 MAKERATIONAL	1399
4.51.2.286 INVERSEFUNCTION	1399
4.51.2.287 IDFUNCTION	1399
4.51.2.288 PUTFIRST	1399
4.51.2.289 PERMUTATIONS	1399
4.51.2.290 PARTITIONS	1400
4.51.2.291 MULFUNCTION	1400
4.51.2.292 MOEBIUS	1400
4.51.2.293 TOPOLOGIES	1400
4.51.2.294 DIAGRAMS	1400
4.51.2.295 TOPO	1400
4.51.2.296 NODEFUNCTION	1400
4.51.2.297 EDGE	1400
4.51.2.298 SIZEOFFUNCTION	1401
4.51.2.299 BLOCK	1401
4.51.2.300 ONEPI	1401
4.51.2.301 PHI	1401
4.51.2.302 FLOATFUN	1401
4.51.2.303 TOFLOAT	1401
4.51.2.304 TORAT	1401
4.51.2.305 MZV	1401
4.51.2.306 EULER	1402
4.51.2.307 MZVHALF	1402
4.51.2.308 AGMFUNCTION	1402
4.51.2.309 GAMMAFUN	1402
4.51.2.310 EXPFUNCTION	1402
4.51.2.311 HPLFUNCTION	1402
4.51.2.312 MPLFUNCTION	1402
4.51.2.313 MAXBUILTINFUNCTION	1402
4.51.2.314 FIRSTUSERFUNCTION	1403
4.51.2.315 ALLFUNCTIONS	1403
4.51.2.316 ALLMZVFUNCTIONS	1403
4.51.2.317 ISYMBOL	1403

4.51.2.318 PISYMBOL	1403
4.51.2.319 COEFFSYMBOL	1403
4.51.2.320 NUMERATORSYMBOL	1403
4.51.2.321 DENOMINATORSYMBOL	1403
4.51.2.322 WILDARGSYMBOL	1404
4.51.2.323 DIMENSIONSYMBOL	1404
4.51.2.324 FACTORSYMBOL	1404
4.51.2.325 SEPARATESYMBOL	1404
4.51.2.326 EESYMBOL	1404
4.51.2.327 EMSYMBOL	1404
4.51.2.328 BUILTINSYMBOLS	1404
4.51.2.329 FIRSTUSERSYMBOL	1404
4.51.2.330 BUILTININDICES	1405
4.51.2.331 BUILTINVECTORS	1405
4.51.2.332 BUILTINDOLLARS	1405
4.51.2.333 WILDARGVECTOR	1405
4.51.2.334 WILDARGINDEX	1405
4.51.2.335 TYPEEXPRESSION	1405
4.51.2.336 TYPEIDNEW	1405
4.51.2.337 TYPEIDOLD	1405
4.51.2.338 TYPEOPERATION	1406
4.51.2.339 TYPEEREPEAT	1406
4.51.2.340 TYPEENDREPEAT	1406
4.51.2.341 TYPECOUNT	1406
4.51.2.342 TYPEMULT	1406
4.51.2.343 TYPEGOTO	1406
4.51.2.344 TYPEDISCARD	1406
4.51.2.345 TYPEIF	1406
4.51.2.346 TYPEELSE	1407
4.51.2.347 TYPEELIF	1407
4.51.2.348 TYPEENDIF	1407
4.51.2.349 TYPESUM	1407
4.51.2.350 TYPECHISHOLM	1407
4.51.2.351 TYPEREVERSE	1407
4.51.2.352 TYPEARG	1407
4.51.2.353 TYPENORM	1407
4.51.2.354 TYPENORM2	1408
4.51.2.355 TYPENORM3	1408
4.51.2.356 TYPEEXIT	1408
4.51.2.357 TYPESETEXIT	1408
4.51.2.358 TYPEPRINT	1408
4.51.2.359 TYPEFPRINT	1408

4.51.2.360 TYPEREDEFPRE	1408
4.51.2.361 TYPESPLITARG	1408
4.51.2.362 TYPESPLITARG2	1409
4.51.2.363 TYPEFACTARG	1409
4.51.2.364 TYPEFACTARG2	1409
4.51.2.365 TYPETRY	1409
4.51.2.366 TYPEASSIGN	1409
4.51.2.367 TYPERENUMBER	1409
4.51.2.368 TYPESUMFIX	1409
4.51.2.369 TYPEFINDLOOP	1409
4.51.2.370 TYPEUNRAVEL	1410
4.51.2.371 TYPEADJUSTBOUNDS	1410
4.51.2.372 TYPEINSIDE	1410
4.51.2.373 TYPETERM	1410
4.51.2.374 TYPESORT	1410
4.51.2.375 TYPEDETCURDUM	1410
4.51.2.376 TYPEINEXPRESSION	1410
4.51.2.377 TYPESPLITFIRSTARG	1410
4.51.2.378 TYPESPLITLASTARG	1411
4.51.2.379 TYPEMERGE	1411
4.51.2.380 TYPETESTUSE	1411
4.51.2.381 TYPEAPPLY	1411
4.51.2.382 TYPEAPPLYRESET	1411
4.51.2.383 TYPECHAININ	1411
4.51.2.384 TYPECHAINOUT	1411
4.51.2.385 TYPENORM4	1411
4.51.2.386 TYPEFACTOR	1412
4.51.2.387 TYPEARGIMPLODE	1412
4.51.2.388 TYPEARGEXPLODE	1412
4.51.2.389 TYPEDENOMINATORS	1412
4.51.2.390 TYPESTUFFLE	1412
4.51.2.391 TYPEDROPCOEFFICIENT	1412
4.51.2.392 TYPETRANSFORM	1412
4.51.2.393 TYPETOPOLYNOMIAL	1412
4.51.2.394 TYPEFROMPOLYNOMIAL	1413
4.51.2.395 TYPEDOLOOP	1413
4.51.2.396 TYPEENDDOLOOP	1413
4.51.2.397 TYPEDROPSYMBOLS	1413
4.51.2.398 TYPEPUTINSIDE	1413
4.51.2.399 TYPETOSPECTATOR	1413
4.51.2.400 TYPEARGTOEXTRASymbol	1413
4.51.2.401 TYPECANONICALIZE	1413

4.51.2.402 TYPESWITCH	1414
4.51.2.403 TYPEENDSWITCH	1414
4.51.2.404 TYPESETUSERFLAG	1414
4.51.2.405 TYPECLEARUSERFLAG	1414
4.51.2.406 TYPEALLLOOPS	1414
4.51.2.407 TYPEALLPATHS	1414
4.51.2.408 TAKETRACE	1414
4.51.2.409 CONTRACT	1414
4.51.2.410 RATIO	1415
4.51.2.411 SYMMETRIZE	1415
4.51.2.412 TENVEC	1415
4.51.2.413 SUMNUM1	1415
4.51.2.414 SUMNUM2	1415
4.51.2.415 WILDDUMMY	1415
4.51.2.416 SYMTONUM	1415
4.51.2.417 SYMTOSYM	1415
4.51.2.418 SYMTOSUB	1416
4.51.2.419 VECTOMIN	1416
4.51.2.420 VECTOVEC	1416
4.51.2.421 VECTOSUB	1416
4.51.2.422 INDTOIND	1416
4.51.2.423 INDTOSUB	1416
4.51.2.424 FUNTOFUN	1416
4.51.2.425 ARGTOARG	1416
4.51.2.426 ARLTOARL	1417
4.51.2.427 EXPTOEXP	1417
4.51.2.428 FROMBRAC	1417
4.51.2.429 FROMSET	1417
4.51.2.430 SETTONUM	1417
4.51.2.431 WILDCARDS	1417
4.51.2.432 SETNUMBER	1417
4.51.2.433 LOADDOLLAR	1417
4.51.2.434 NUMTONUM	1418
4.51.2.435 NUMTOSYM	1418
4.51.2.436 NUMTOIND	1418
4.51.2.437 NUMTOSUB	1418
4.51.2.438 EATTENSOR	1418
4.51.2.439 CLEANFLAG	1418
4.51.2.440 DIRTYFLAG	1418
4.51.2.441 DIRTSYMSYFLAG	1418
4.51.2.442 MUSTCLEANPRF	1419
4.51.2.443 SUBTERMUSED1	1419

4.51.2.444 SUBTERMUSED2	1419
4.51.2.445 ALLDIRTY	1419
4.51.2.446 ARGHEAD	1419
4.51.2.447 FUNHEAD	1419
4.51.2.448 SUBEXPSIZE	1419
4.51.2.449 EXPRHEAD	1419
4.51.2.450 TYPEARGHEADSIZE	1420
4.51.2.451 SKIP	1420
4.51.2.452 DROP	1420
4.51.2.453 HIDE	1420
4.51.2.454 UNHIDE	1420
4.51.2.455 INTOHIDE	1420
4.51.2.456 LOCALEXPRESSION	1420
4.51.2.457 SKIPLEXPRESSION	1420
4.51.2.458 DROPLEXPRESSION	1421
4.51.2.459 DROPPEDEXPRESSION	1421
4.51.2.460 GLOBALEXPRESSION	1421
4.51.2.461 SKIPGEXPRESSION	1421
4.51.2.462 DROPGEXPRESSION	1421
4.51.2.463 STOREDEXPRESSION	1421
4.51.2.464 HIDDENLEXPRESSION	1421
4.51.2.465 HIDDENGEXPRESSION	1421
4.51.2.466 INCEXPRESSION	1422
4.51.2.467 HIDELEXPRESSION	1422
4.51.2.468 HIDEGEXPRESSION	1422
4.51.2.469 DROPHLEXPRESSION	1422
4.51.2.470 DROPHGEXPRESSION	1422
4.51.2.471 UNHIDELEXPRESSION	1422
4.51.2.472 UNHIDEGEXPRESSION	1422
4.51.2.473 INTOHIDELEXPRESSION	1422
4.51.2.474 INTOHIDEGEXPRESSION	1423
4.51.2.475 SPECTATOREXPRESSION	1423
4.51.2.476 DROPSPECTATOREXPRESSION	1423
4.51.2.477 SKIPUNHIDELEXPRESSION	1423
4.51.2.478 SKIPUNHIDEGEXPRESSION	1423
4.51.2.479 PRINTOFF	1423
4.51.2.480 PRINTON	1423
4.51.2.481 PRINTCONTENTS	1423
4.51.2.482 PRINTCONTENT	1424
4.51.2.483 PRINTLFILE	1424
4.51.2.484 PRINTONETERM	1424
4.51.2.485 PRINTONEFUNCTION	1424

4.51.2.486 PRINTALL	1424
4.51.2.487 REGularexpression	1424
4.51.2.488 REDEFINEDexpression	1424
4.51.2.489 NEWLYDEFINEDexpression	1424
4.51.2.490 VERTEXFUNCTION	1425
4.51.2.491 GENERALFUNCTION	1425
4.51.2.492 TENSORFUNCTION	1425
4.51.2.493 GAMMAFUNCTION	1425
4.51.2.494 POS_	1425
4.51.2.495 POS0_	1425
4.51.2.496 NEG_	1425
4.51.2.497 NEG0_	1425
4.51.2.498 EVEN_	1426
4.51.2.499 ODD_	1426
4.51.2.500 Z_	1426
4.51.2.501 SYMBOL_	1426
4.51.2.502 FIXED_	1426
4.51.2.503 INDEX_	1426
4.51.2.504 Q_	1426
4.51.2.505 DUMMYINDEX_	1426
4.51.2.506 VECTOR_	1427
4.51.2.507 GAMMA1	1427
4.51.2.508 GAMMA5	1427
4.51.2.509 GAMMA6	1427
4.51.2.510 GAMMA7	1427
4.51.2.511 FUNNYVEC	1427
4.51.2.512 FUNNYWILD	1427
4.51.2.513 SUMMEDIND	1427
4.51.2.514 NOINDEX	1428
4.51.2.515 FUNNYDOLLAR	1428
4.51.2.516 EMPTYINDEX	1428
4.51.2.517 MINSPEC	1428
4.51.2.518 USEDFLAG	1428
4.51.2.519 DUMMYFLAG	1428
4.51.2.520 MAINSORT	1428
4.51.2.521 FUNCTIONSORT	1428
4.51.2.522 SUBSORT	1429
4.51.2.523 FLOATMODE	1429
4.51.2.524 RATIONALMODE	1429
4.51.2.525 NUMSPECSETS	1429
4.51.2.526 ISZERO	1429
4.51.2.527 ISUNMODIFIED	1429

4.51.2.528 ISCOMPRESSED	1429
4.51.2.529 ISINRHS	1429
4.51.2.530 ISFACTORIZED	1430
4.51.2.531 TOBEFACTORED	1430
4.51.2.532 TOBEUNFACTORED	1430
4.51.2.533 KEEPZERO	1430
4.51.2.534 VARTYPENONE	1430
4.51.2.535 VARTYPECOMPLEX	1430
4.51.2.536 VARTYPEIMAGINARY	1430
4.51.2.537 VARTYPEROOTOFUNITY	1430
4.51.2.538 VARTYPEMINUS	1431
4.51.2.539 CYCLESYMMETRIC	1431
4.51.2.540 RCYCLESYMMETRIC	1431
4.51.2.541 SYMMETRIC	1431
4.51.2.542 ANTISYMMETRIC	1431
4.51.2.543 REVERSEORDER	1431
4.51.2.544 SUBMULTI	1431
4.51.2.545 SUBONCE	1431
4.51.2.546 SUBONLY	1432
4.51.2.547 SUBMANY	1432
4.51.2.548 SUBVECTOR	1432
4.51.2.549 SUBSELECT	1432
4.51.2.550 SUBALL	1432
4.51.2.551 SUBMASK	1432
4.51.2.552 SUBDISORDER	1432
4.51.2.553 SUBAFTER	1432
4.51.2.554 SUBAFTERNOT	1433
4.51.2.555 IDHEAD	1433
4.51.2.556 DOLLARFLAG	1433
4.51.2.557 NORMALIZEFLAG	1433
4.51.2.558 GIDENT	1433
4.51.2.559 GFIVE	1433
4.51.2.560 GPLUS	1433
4.51.2.561 GMINUS	1433
4.51.2.562 LONGNUMBER	1434
4.51.2.563 MATCH	1434
4.51.2.564 COEFFI	1434
4.51.2.565 SUBEXPR	1434
4.51.2.566 MULTIPLEOF	1434
4.51.2.567 IFDOLLAR	1434
4.51.2.568 IFEXPRESSION	1434
4.51.2.569 IFDOLLAREXTRA	1434

4.51.2.570 IFISFACTORIZED	1435
4.51.2.571 IFOCCURS	1435
4.51.2.572 IFUSERFLAG	1435
4.51.2.573 IFFLOATNUMBER	1435
4.51.2.574 GREATER	1435
4.51.2.575 GREATEREQUAL	1435
4.51.2.576 LESS	1435
4.51.2.577 LESSEQUAL	1435
4.51.2.578 EQUAL	1436
4.51.2.579 NOTEQUAL	1436
4.51.2.580 ORCOND	1436
4.51.2.581 ANDCOND	1436
4.51.2.582 DUMMY	1436
4.51.2.583 SORT	1436
4.51.2.584 STORE	1436
4.51.2.585 END	1436
4.51.2.586 GLOBAL	1437
4.51.2.587 CLEAR	1437
4.51.2.588 VECTBIT	1437
4.51.2.589 DOTPBIT	1437
4.51.2.590 FUNBIT	1437
4.51.2.591 SETBIT	1437
4.51.2.592 EXTRAPARAMETER	1437
4.51.2.593 GENCOMMUTE	1437
4.51.2.594 GENNONCOMMUTE	1438
4.51.2.595 NAMENOTFOUND	1438
4.51.2.596 DOLUNDEFINED	1438
4.51.2.597 DOLNUMBER	1438
4.51.2.598 DOLARGUMENT	1438
4.51.2.599 DOLSUBTERM	1438
4.51.2.600 DOLTERMS	1438
4.51.2.601 DOLWILDARGS	1438
4.51.2.602 DOLINDEX	1439
4.51.2.603 DOLZERO	1439
4.51.2.604 FINDLOOP	1439
4.51.2.605 REPLACELOOP	1439
4.51.2.606 NOFUNPOWERS	1439
4.51.2.607 COMFUNPOWERS	1439
4.51.2.608 ALLFUNPOWERS	1439
4.51.2.609 PROPERORDERFLAG	1439
4.51.2.610 REGULAR	1440
4.51.2.611 FINISH	1440

4.51.2.612 POLYADD	1440
4.51.2.613 POLYSUB	1440
4.51.2.614 POLYMUL	1440
4.51.2.615 POLYDIV	1440
4.51.2.616 POLYREM	1440
4.51.2.617 POLYGCD	1440
4.51.2.618 POLYINTFAC	1441
4.51.2.619 POLYNORM	1441
4.51.2.620 MODNONE	1441
4.51.2.621 MODSUM	1441
4.51.2.622 MODMAX	1441
4.51.2.623 MODMIN	1441
4.51.2.624 MODLOCAL	1441
4.51.2.625 ELEMENTUSED	1441
4.51.2.626 ELEMENTLOADED	1442
4.51.2.627 POSNEG	1442
4.51.2.628 INVERSETABLE	1442
4.51.2.629 COEFFICIENTSONLY	1442
4.51.2.630 ALSOPOWERS	1442
4.51.2.631 ALSOFUNARGS	1442
4.51.2.632 ALSODOLLARS	1442
4.51.2.633 NOINVERSES	1442
4.51.2.634 POSITIVEONLY	1443
4.51.2.635 UNPACK	1443
4.51.2.636 NOUNPACK	1443
4.51.2.637 FROMFUNCTION	1443
4.51.2.638 VARNAMEs	1443
4.51.2.639 AUTONAMES	1443
4.51.2.640 EXPRNAMEs	1443
4.51.2.641 DOLLARNAMEs	1443
4.51.2.642 DUMMYBUFFER	1444
4.51.2.643 ALLARGS	1444
4.51.2.644 NUMARG	1444
4.51.2.645 ARG RANGE	1444
4.51.2.646 MAKEARGS	1444
4.51.2.647 MAXRANGEINDICATOR	1444
4.51.2.648 REPLACEARG	1444
4.51.2.649 ENCODEARG	1444
4.51.2.650 DECODEARG	1445
4.51.2.651 IMplodeARG	1445
4.51.2.652 EXPLODEARG	1445
4.51.2.653 PERMUTEARG	1445

4.51.2.654 REVERSEARG	1445
4.51.2.655 CYCLEARG	1445
4.51.2.656 ISLYNDON	1445
4.51.2.657 ISLYNDONR	1445
4.51.2.658 TOLYNDON	1446
4.51.2.659 TOLYNDONR	1446
4.51.2.660 ADDARG	1446
4.51.2.661 MULTIPLYARG	1446
4.51.2.662 DROPARG	1446
4.51.2.663 SELECTARG	1446
4.51.2.664 DEDUPARG	1446
4.51.2.665 ZTOHARG	1446
4.51.2.666 HTOZARG	1447
4.51.2.667 BASECODE	1447
4.51.2.668 YESLYNDON	1447
4.51.2.669 NOLYNDON	1447
4.51.2.670 TOPOLYNOMIALFLAG	1447
4.51.2.671 FACTARGFLAG	1447
4.51.2.672 OLDFACTARG	1447
4.51.2.673 NEWFACTARG	1447
4.51.2.674 FROMMODULEOPTION	1448
4.51.2.675 FROMPOINTINSTRUCTION	1448
4.51.2.676 EXTRASYMBOL	1448
4.51.2.677 REGULARSYMBOL	1448
4.51.2.678 EXPRESSIONNUMBER	1448
4.51.2.679 O_NONE	1448
4.51.2.680 O_CSE	1448
4.51.2.681 O_CSEGREEDY	1448
4.51.2.682 O_GREEDY	1449
4.51.2.683 O_OCCURRENCE	1449
4.51.2.684 O_MCTS	1449
4.51.2.685 O_SIMULATED_ANNEALING	1449
4.51.2.686 O_FORWARD	1449
4.51.2.687 O_BACKWARD	1449
4.51.2.688 O_FORWARDORBACKWARD	1449
4.51.2.689 O_FORWARDANDBACKWARD	1449
4.51.2.690 OPTHEAD	1450
4.51.2.691 DOALL	1450
4.51.2.692 ONLYFUNCTIONS	1450
4.51.2.693 INUSE	1450
4.51.2.694 COULDCOMMUTE	1450
4.51.2.695 DOESNOTCOMMUTE	1450

4.51.2.696	DICT_NONNUMBERS	1450
4.51.2.697	DICT_INTEGERONLY	1450
4.51.2.698	DICT_RATIONALONLY	1451
4.51.2.699	DICT_ALLNUMBERS	1451
4.51.2.700	DICT_NOVARIABLES	1451
4.51.2.701	DICT_DOVARIABLES	1451
4.51.2.702	DICT_NOSPECIALS	1451
4.51.2.703	DICT_DOSPECIALS	1451
4.51.2.704	DICT_NOFUNWITHARGS	1451
4.51.2.705	DICT_DOFUNWITHARGS	1451
4.51.2.706	DICT_NOTINDOLLARS	1452
4.51.2.707	DICT_INDOLLARS	1452
4.51.2.708	DICT_INTEGERNUMBER	1452
4.51.2.709	DICT_RATIONALNUMBER	1452
4.51.2.710	DICT_SYMBOL	1452
4.51.2.711	DICT_VECTOR	1452
4.51.2.712	DICT_INDEX	1452
4.51.2.713	DICT_FUNCTION	1452
4.51.2.714	DICT_FUNCTION_WITH_ARGUMENTS	1453
4.51.2.715	DICT_SPECIALCHARACTER	1453
4.51.2.716	DICT_RANGE	1453
4.51.2.717	READSPECTATORFLAG	1453
4.51.2.718	GLOBALSPECTATORFLAG	1453
4.51.2.719	ORDEREDSET	1453
4.51.2.720	DENSETABLE	1453
4.51.2.721	SPARSETABLE	1453
4.51.2.722	TOPOLOGIESONLY	1454
4.51.2.723	WITHOUTNODES	1454
4.51.2.724	WITHEDGES	1454
4.51.2.725	WITHBLOCKS	1454
4.51.2.726	WITHONEPISETS	1454
4.51.2.727	WITHSYMMETRIZEI	1454
4.51.2.728	WITHSYMMETRIZEF	1454
4.51.2.729	CHECKEXTERN	1454
4.51.2.730	ONEPARTI	1455
4.51.2.731	ONEPARTR	1455
4.51.2.732	ONHELL	1455
4.51.2.733	OFFSHELL	1455
4.51.2.734	NOSIGMA	1455
4.51.2.735	SIGMA	1455
4.51.2.736	NOSNAIL	1455
4.51.2.737	SNAIL	1455

4.51.2.738 NOTADPOLE	1456
4.51.2.739 TADPOLE	1456
4.51.2.740 SIMPLE	1456
4.51.2.741 NOTSIMPLE	1456
4.51.2.742 BIPART	1456
4.51.2.743 NONBIPART	1456
4.51.2.744 CYCLI	1456
4.51.2.745 CYCLR	1456
4.51.2.746 FLOOP	1457
4.51.2.747 NOTFLOOP	1457
4.51.3 Typedef Documentation	1457
4.51.3.1 PVFUNWP	1457
4.51.3.2 PVFUNV	1457
4.51.3.3 CFUN	1457
4.51.3.4 TFUN	1457
4.51.3.5 TFUN1	1457
4.52 ftypes.h	1458
4.53 function.c File Reference	1470
4.53.1 Detailed Description	1471
4.53.2 Function Documentation	1471
4.53.2.1 MakeDirty()	1471
4.53.2.2 MarkDirty()	1471
4.53.2.3 PolyFunDirty()	1471
4.53.2.4 PolyFunClean()	1471
4.53.2.5 Symmetrize()	1472
4.53.2.6 CompGroup()	1472
4.53.2.7 FullSymmetrize()	1472
4.53.2.8 SymGen()	1472
4.53.2.9 SymFind()	1472
4.53.2.10 ChainIn()	1473
4.53.2.11 ChainOut()	1473
4.53.2.12 MatchFunction()	1473
4.53.2.13 ScanFunctions()	1473
4.54 function.c	1473
4.55 fwin.h File Reference	1496
4.55.1 Detailed Description	1496
4.55.2 Macro Definition Documentation	1496
4.55.2.1 LINEFEED	1496
4.55.2.2 CARRIAGERETURN	1497
4.55.2.3 WITHRETURN	1497
4.55.2.4 WITHSYSTEM	1497
4.55.2.5 P_term	1497

4.55.2.6 SEPARATOR	1497
4.55.2.7 ALTSEPARATOR	1497
4.55.2.8 PATHSEPARATOR	1497
4.55.2.9 WITH_ENV	1498
4.56 fwin.h	1498
4.57 grcc.cc	1498
4.58 grcc.h	1626
4.59 grccparam.h	1643
4.60 if.c File Reference	1646
4.60.1 Detailed Description	1646
4.60.2 Function Documentation	1647
4.60.2.1 GetIfDollarNum()	1647
4.60.2.2 FindVar()	1647
4.60.2.3 DoIfStatement()	1647
4.60.2.4 HowMany()	1647
4.60.2.5 DoubleIfBuffers()	1647
4.60.2.6 DoSwitch()	1647
4.60.2.7 DoEndSwitch()	1648
4.60.2.8 FindCase()	1648
4.60.2.9 DoubleSwitchBuffers()	1648
4.60.2.10 SwitchSplitMergeRec()	1648
4.60.2.11 SwitchSplitMerge()	1648
4.61 if.c	1649
4.62 index.c File Reference	1663
4.62.1 Detailed Description	1663
4.62.2 Function Documentation	1663
4.62.2.1 FindBracket()	1663
4.62.2.2 PutBracketInIndex()	1664
4.62.2.3 ClearBracketIndex()	1664
4.62.2.4 OpenBracketIndex()	1664
4.62.2.5 PutInside()	1664
4.63 index.c	1664
4.64 inivar.h File Reference	1673
4.64.1 Detailed Description	1673
4.64.2 Variable Documentation	1673
4.64.2.1 FG	1673
4.64.2.2 A	1673
4.64.2.3 fixedsets	1674
4.64.2.4 BufferForOutput	1674
4.64.2.5 setupfilename	1674
4.65 inivar.h	1674
4.66 lus.c File Reference	1678

4.66.1 Detailed Description	1678
4.66.2 Function Documentation	1678
4.66.2.1 Lus()	1678
4.66.2.2 FindLus()	1679
4.66.2.3 SortTheList()	1679
4.66.2.4 AllLoops()	1679
4.66.2.5 StartLoops()	1679
4.66.2.6 GenLoops()	1679
4.66.2.7 LoopOutput()	1680
4.66.2.8 AllPaths()	1680
4.66.2.9 GenPaths()	1680
4.66.2.10 PathOutput()	1680
4.66.2.11 AllOnePI()	1680
4.66.2.12 RemoveBridges()	1681
4.66.2.13 TakeOneLine()	1681
4.66.2.14 OutputOnePI()	1681
4.67 lus.c	1681
4.68 mallocprotect.h	1700
4.69 message.c File Reference	1704
4.69.1 Detailed Description	1705
4.69.2 Function Documentation	1705
4.69.2.1 Error0()	1705
4.69.2.2 Error1()	1705
4.69.2.3 Error2()	1705
4.69.2.4 MesWork()	1706
4.69.2.5 MesPrint()	1706
4.69.2.6 Warning()	1706
4.69.2.7 HighWarning()	1706
4.69.2.8 MesCall()	1706
4.69.2.9 MesCerr()	1706
4.69.2.10 MesComp()	1707
4.69.2.11 PrintTerm()	1707
4.69.2.12 PrintTermC()	1707
4.69.2.13 PrintSubTerm()	1707
4.69.2.14 PrintWords()	1707
4.69.2.15 PrintSeq()	1707
4.70 message.c	1708
4.71 minos.c File Reference	1719
4.71.1 Detailed Description	1720
4.71.2 Macro Definition Documentation	1720
4.71.2.1 CFD	1720
4.71.2.2 CTD	1720

4.71.3 Function Documentation	1720
4.71.3.1 minosread()	1720
4.71.3.2 minoswrite()	1721
4.71.3.3 str_dup()	1721
4.71.3.4 convertblock()	1721
4.71.3.5 convertnamesblock()	1721
4.71.3.6 convertiniinfo()	1721
4.71.3.7 LocateBase()	1721
4.71.3.8 ReadIndex()	1722
4.71.3.9 WriteIndexBlock()	1722
4.71.3.10 WriteNamesBlock()	1722
4.71.3.11 WriteIndex()	1722
4.71.3.12 WriteIniInfo()	1722
4.71.3.13 ReadIniInfo()	1722
4.71.3.14 GetDbase()	1723
4.71.3.15 NewDbase()	1723
4.71.3.16 FreeTableBase()	1723
4.71.3.17 ComposeTableNames()	1723
4.71.3.18 OpenDbase()	1723
4.71.3.19 AddTableName()	1723
4.71.3.20 GetTableName()	1724
4.71.3.21 PutTableNames()	1724
4.71.3.22 AddToIndex()	1724
4.71.3.23 AddObject()	1724
4.71.3.24 FindTableNumber()	1724
4.71.3.25 WriteObject()	1724
4.71.3.26 ReadObject()	1725
4.71.3.27 ReadijObject()	1725
4.71.3.28 ExistsObject()	1725
4.71.3.29 DeleteObject()	1725
4.71.4 Variable Documentation	1725
4.71.4.1 withoutflush	1725
4.72 minos.c	1726
4.73 minos.h File Reference	1743
4.73.1 Detailed Description	1744
4.73.2 Macro Definition Documentation	1744
4.73.2.1 TODISK	1744
4.73.2.2 FROMDISK	1744
4.73.2.3 MDIRTYFLAG	1744
4.73.2.4 MCLEANFLAG	1745
4.73.2.5 INANDOUT	1745
4.73.2.6 INPUTONLY	1745

4.73.2.7 OUTPUTONLY	1745
4.73.2.8 NOCOMPRESS	1745
4.73.3 Function Documentation	1745
4.73.3.1 minosread()	1745
4.73.3.2 minoswrite()	1745
4.73.4 Variable Documentation	1746
4.73.4.1 withoutflush	1746
4.74 minos.h	1746
4.75 model.c File Reference	1748
4.75.1 Detailed Description	1748
4.75.2 Function Documentation	1748
4.75.2.1 CreateVertex()	1748
4.75.2.2 ReadParticle()	1748
4.75.2.3 CoModel()	1749
4.75.2.4 CoParticle()	1749
4.75.2.5 CoVertex()	1749
4.75.2.6 CoEndModel()	1749
4.76 model.c	1749
4.77 module.c File Reference	1755
4.77.1 Detailed Description	1755
4.77.2 Function Documentation	1756
4.77.2.1 ModuleInstruction()	1756
4.77.2.2 CoModuleOption()	1756
4.77.2.3 CoModOption()	1756
4.77.2.4 SetSpecialMode()	1756
4.77.2.5 ExecModule()	1756
4.77.2.6 ExecStore()	1756
4.77.2.7 FullCleanUp()	1757
4.77.2.8 DoPolyfun()	1757
4.77.2.9 DoPolyratfun()	1757
4.77.2.10 DoNoParallel()	1757
4.77.2.11 DoParallel()	1757
4.77.2.12 DoModSum()	1757
4.77.2.13 DoModMax()	1757
4.77.2.14 DoModMin()	1758
4.77.2.15 DoModLocal()	1758
4.77.2.16 DoProcessBucket()	1758
4.77.2.17 DoModDollar()	1758
4.77.2.18 DoinParallel()	1758
4.77.2.19 DonotinParallel()	1758
4.77.2.20 DoExecStatement()	1758
4.77.2.21 DoPipeStatement()	1759

4.78 module.c	1759
4.79 mpi.c File Reference	1768
4.79.1 Detailed Description	1769
4.79.2 Macro Definition Documentation	1770
4.79.2.1 PF_PACKSIZE	1770
4.79.2.2 MPI_ERRCODE_CHECK	1770
4.79.3 Function Documentation	1770
4.79.3.1 PF_RealTime()	1770
4.79.3.2 PF_LibInit()	1771
4.79.3.3 PF_LibTerminate()	1771
4.79.3.4 PF_Probe()	1771
4.79.3.5 PF_ISendSbuf()	1772
4.79.3.6 PF_RecvWbuf()	1772
4.79.3.7 PF_IRecvRbuf()	1773
4.79.3.8 PF_WaitRbuf()	1773
4.79.3.9 PF_Bcast()	1774
4.79.3.10 PF_RawSend()	1774
4.79.3.11 PF_RawRecv()	1774
4.79.3.12 PF_RawProbe()	1775
4.79.3.13 PF_PrintPackBuf()	1775
4.79.3.14 PF_PreparePack()	1776
4.79.3.15 PF_Pack()	1776
4.79.3.16 PF_Unpack()	1777
4.79.3.17 PF_PackString()	1777
4.79.3.18 PF_UnpackString()	1778
4.79.3.19 PF_Send()	1778
4.79.3.20 PF_Receive()	1779
4.79.3.21 PF_Broadcast()	1779
4.79.3.22 PF_PrepareLongSinglePack()	1780
4.79.3.23 PF_LongSinglePack()	1780
4.79.3.24 PF_LongSingleUnpack()	1780
4.79.3.25 PF_LongSingleSend()	1781
4.79.3.26 PF_LongSingleReceive()	1781
4.79.3.27 PF_PrepareLongMultiPack()	1782
4.79.3.28 PF_LongMultiPackImpl()	1782
4.79.3.29 PF_LongMultiUnpackImpl()	1783
4.79.3.30 PF_LongMultiBroadcast()	1784
4.79.4 Variable Documentation	1784
4.79.4.1 PF_maxDollarChunkSize	1784
4.80 mpi.c	1785
4.81 mpidbg.h File Reference	1802
4.81.1 Detailed Description	1803

4.81.2 Macro Definition Documentation	1804
4.81.2.1 MPIDBG_RANK	1804
4.81.2.2 MPIDBG_Out	1804
4.81.2.3 MPIDBG_EXTARG	1804
4.81.2.4 MPI_Send	1804
4.81.2.5 MPI_Recv	1804
4.81.2.6 MPI_Bsend	1804
4.81.2.7 MPI_Ssend	1804
4.81.2.8 MPI_Rsend	1805
4.81.2.9 MPI_Isend	1805
4.81.2.10 MPI_lbsend	1805
4.81.2.11 MPI_Issend	1805
4.81.2.12 MPI_Irsend	1805
4.81.2.13 MPI_Irecv	1805
4.81.2.14 MPI_Wait	1805
4.81.2.15 MPI_Test	1806
4.81.2.16 MPI_Waitany	1806
4.81.2.17 MPI_Testany	1806
4.81.2.18 MPI_Waitall	1806
4.81.2.19 MPI_Testall	1806
4.81.2.20 MPI_Waitsome	1806
4.81.2.21 MPI_Testsome	1806
4.81.2.22 MPI_lprobe	1807
4.81.2.23 MPI_Probe	1807
4.81.2.24 MPI_Cancel	1807
4.81.2.25 MPI_Test_cancelled	1807
4.81.2.26 MPI_Barrier	1807
4.81.2.27 MPI_Bcast	1807
4.82 mpidbg.h	1808
4.83 mytime.cc	1816
4.84 mytime.h	1816
4.85 names.c File Reference	1817
4.85.1 Detailed Description	1818
4.85.2 Function Documentation	1819
4.85.2.1 GetNode()	1819
4.85.2.2 AddName()	1819
4.85.2.3 GetName()	1819
4.85.2.4 GetFunction()	1819
4.85.2.5 GetNumber()	1819
4.85.2.6 GetLastExprName()	1820
4.85.2.7 GetOName()	1820
4.85.2.8 GetAutoName()	1820

4.85.2.9 GetVar()	1820
4.85.2.10 EntVar()	1820
4.85.2.11 GetDollar()	1821
4.85.2.12 DumpTree()	1821
4.85.2.13 DumpNode()	1821
4.85.2.14 CompactifyTree()	1821
4.85.2.15 CopyTree()	1821
4.85.2.16 LinkTree()	1821
4.85.2.17 MakeNameTree()	1822
4.85.2.18 FreeNameTree()	1822
4.85.2.19 ClearWildcardNames()	1822
4.85.2.20 AddWildcardName()	1822
4.85.2.21 GetWildcardName()	1822
4.85.2.22 AddSymbol()	1822
4.85.2.23 CoSymbol()	1823
4.85.2.24 AddIndex()	1823
4.85.2.25 CoIndex()	1823
4.85.2.26 DoDimension()	1823
4.85.2.27 CoDimension()	1823
4.85.2.28 AddVector()	1823
4.85.2.29 CoVector()	1824
4.85.2.30 AddFunction()	1824
4.85.2.31 CoCommutInSet()	1824
4.85.2.32 CoFunction()	1824
4.85.2.33 CoNFunction()	1824
4.85.2.34 CoCFunction()	1824
4.85.2.35 CoNTensor()	1825
4.85.2.36 CoCTensor()	1825
4.85.2.37 DoTable()	1825
4.85.2.38 CoTable()	1825
4.85.2.39 CoNTable()	1825
4.85.2.40 CoCTable()	1825
4.85.2.41 EmptyTable()	1825
4.85.2.42 AddSet()	1826
4.85.2.43 DoElements()	1826
4.85.2.44 CoSet()	1826
4.85.2.45 DoTempSet()	1826
4.85.2.46 CoAuto()	1826
4.85.2.47 AddDollar()	1826
4.85.2.48 ReplaceDollar()	1827
4.85.2.49 AddDubious()	1827
4.85.2.50 MakeDubious()	1827

4.85.2.51 NameConflict()	1827
4.85.2.52 AddExpression()	1827
4.85.2.53 GetLabel()	1827
4.85.2.54 ResetVariables()	1828
4.85.2.55 RemoveDollars()	1828
4.85.2.56 Globalize()	1828
4.85.2.57 TestName()	1828
4.86 names.c	1828
4.87 normal.c File Reference	1866
4.87.1 Detailed Description	1867
4.87.2 Macro Definition Documentation	1867
4.87.2.1 MAXNUMBEROFNONCOMTERMS	1867
4.87.3 Function Documentation	1867
4.87.3.1 CompareFunctions()	1867
4.87.3.2 Commute()	1867
4.87.3.3 Normalize()	1867
4.87.3.4 ExtraSymbol()	1868
4.87.3.5 DoTheta()	1868
4.87.3.6 DoDelta()	1868
4.87.3.7 DoRevert()	1868
4.87.3.8 DetCommu()	1868
4.87.3.9 DoesCommu()	1868
4.87.3.10 TreatPolyRatFun()	1869
4.87.3.11 DropCoefficient()	1869
4.87.3.12 DropSymbols()	1869
4.87.3.13 SymbolNormalize()	1869
4.87.3.14 TestFunFlag()	1869
4.87.3.15 BracketNormalize()	1869
4.88 normal.c	1870
4.89 notation.c File Reference	1932
4.89.1 Detailed Description	1933
4.89.2 Function Documentation	1933
4.89.2.1 NormPolyTerm()	1933
4.89.2.2 ConvertToPoly()	1933
4.89.2.3 LocalConvertToPoly()	1933
4.89.2.4 ConvertFromPoly()	1934
4.89.2.5 FindSubterm()	1934
4.89.2.6 FindLocalSubterm()	1934
4.89.2.7 PrintSubtermList()	1934
4.89.2.8 PrintExtraSymbol()	1934
4.89.2.9 FindSubexpression()	1934
4.89.2.10 ExtraSymFun()	1935

4.89.2.11 PruneExtraSymbols()	1935
4.90 notation.c	1935
4.91 opera.c File Reference	1949
4.91.1 Detailed Description	1949
4.91.2 Function Documentation	1949
4.91.2.1 EpfFind()	1949
4.91.2.2 EpfCon()	1950
4.91.2.3 EpfGen()	1950
4.91.2.4 Trick()	1950
4.91.2.5 Trace4no()	1950
4.91.2.6 Trace4()	1950
4.91.2.7 Trace4Gen()	1951
4.91.2.8 TraceNno()	1951
4.91.2.9 TraceN()	1951
4.91.2.10 TraceNgen()	1951
4.91.2.11 Traces()	1951
4.91.2.12 TraceFind()	1951
4.91.2.13 Chisholm()	1952
4.91.2.14 TenVecFind()	1952
4.91.2.15 TenVec()	1952
4.92 opera.c	1952
4.93 optimize.cc File Reference	1981
4.93.1 Detailed Description	1983
4.93.2 Function Documentation	1983
4.93.2.1 print_instr()	1983
4.93.2.2 my_random_shuffle()	1983
4.93.3 Description	1983
4.93.3.1 get_expression()	1984
4.93.4 Description	1984
4.93.4.1 get_brackets()	1984
4.93.5 Description	1984
4.93.5.1 count_operators() [1/2]	1985
4.93.6 Description	1985
4.93.6.1 count_operators() [2/2]	1985
4.93.7 Description	1985
4.93.7.1 occurrence_order()	1985
4.93.8 Description	1985
4.93.8.1 getpower()	1986
4.93.9 Description	1986
4.93.10 Note	1986
4.93.11 Description	1986
4.93.11.1 fixarg()	1986

4.93.12 Description	1987
4.93.12.1 GcdLong_fix_args()	1987
4.93.12.2 DivLong_fix_args()	1987
4.93.12.3 build_Horner_tree()	1987
4.93.13 Description	1988
4.93.13.1 term_compare()	1988
4.93.14 Description	1988
4.93.14.1 Horner_tree()	1989
4.93.15 Description	1989
4.93.15.1 print_tree()	1989
4.93.15.2 hash_range()	1989
4.93.15.3 generate_instructions()	1990
4.93.16 Description	1990
4.93.17 Implementation details	1990
4.93.17.1 count_operators_cse()	1991
4.93.17.2 buildTree()	1991
4.93.17.3 count_operators_cse_topdown()	1991
4.93.17.4 simulated_annealing()	1991
4.93.17.5 find_Horner_MCTS_expand_tree()	1991
4.93.17.6 find_Horner_MCTS()	1991
4.93.18 Description	1992
4.93.18.1 merge_operators()	1992
4.93.19 Description	1992
4.93.20 Implementation details	1993
4.93.20.1 find_optimizations()	1993
4.93.21 Description	1993
4.93.21.1 do_optimization()	1993
4.93.22 Description	1994
4.93.22.1 partial_factorize()	1994
4.93.23 Description	1994
4.93.23.1 optimize_greedy()	1994
4.93.24 Description	1995
4.93.24.1 recycle_variables()	1995
4.93.25 Description	1995
4.93.26 Implementation details	1996
4.93.26.1 optimize_expression_given_Horner()	1996
4.93.27 Description	1996
4.93.27.1 generate_output()	1997
4.93.28 Description	1997
4.93.28.1 generate_expression()	1997
4.93.29 Description	1997
4.93.29.1 optimize_print_code()	1998

4.93.30 Description	1998
4.93.30.1 Optimize()	1998
4.93.31 Description	1999
4.93.31.1 ClearOptimize()	2000
4.93.32 Description	2000
4.93.33 Variable Documentation	2001
4.93.33.1 OPER_ADD	2001
4.93.33.2 OPER_MUL	2001
4.93.33.3 OPER_COMMA	2001
4.93.33.4 optimize_expr	2001
4.93.33.5 optimize_best_Horner_schemes	2001
4.93.33.6 optimize_num_vars	2002
4.93.33.7 optimize_best_num_oper	2002
4.93.33.8 optimize_best_instr	2002
4.93.33.9 optimize_best_vars	2002
4.93.33.10 mcts_factorized	2002
4.93.33.11 mcts_separated	2002
4.93.33.12 mcts_vars	2002
4.93.33.13 mcts_root	2002
4.93.33.14 mcts_expr_score	2003
4.93.33.15 mcts_best_schemes	2003
4.94 optimize.cc	2003
4.95 parallel.c File Reference	2055
4.95.1 Detailed Description	2057
4.95.2 Macro Definition Documentation	2057
4.95.2.1 PRINTFBUF	2057
4.95.2.2 SWAP	2058
4.95.2.3 PACK_LONG	2058
4.95.2.4 UNPACK_LONG	2058
4.95.2.5 CHECK	2058
4.95.2.6 _CHECK	2059
4.95.2.7 __CHECK	2059
4.95.2.8 DBGOUT	2059
4.95.2.9 DBGOUT_NINTERMS	2059
4.95.2.10 PF_STATS_SIZE	2059
4.95.2.11 PF_SNDFILEBUFSIZE	2060
4.95.2.12 recvBuffer	2060
4.95.3 Typedef Documentation	2060
4.95.3.1 NODE	2060
4.95.4 Function Documentation	2060
4.95.4.1 PF_RealTime()	2060
4.95.4.2 PF_LibInit()	2060

4.95.4.3 PF_LibTerminate()	2061
4.95.4.4 PF_Probe()	2061
4.95.4.5 PF_RecvWbuf()	2062
4.95.4.6 PF_IRecvRbuf()	2062
4.95.4.7 PF_WaitRbuf()	2062
4.95.4.8 PF_RawSend()	2063
4.95.4.9 PF_RawRecv()	2063
4.95.4.10 PF_RawProbe()	2064
4.95.4.11 PF_EndSort()	2064
4.95.4.12 PF_Deferred()	2065
4.95.4.13 PF_Processor()	2066
4.95.4.14 PF_Init()	2067
4.95.4.15 PF_Terminate()	2068
4.95.4.16 PF_GetSlaveTimes()	2069
4.95.4.17 PF_BroadcastNumber()	2069
4.95.4.18 PF_BroadcastBuffer()	2070
4.95.4.19 PF_BroadcastString()	2071
4.95.4.20 PF_BroadcastPreDollar()	2071
4.95.4.21 PF_CollectModifiedDollars()	2072
4.95.4.22 PF_BroadcastModifiedDollars()	2073
4.95.4.23 PF_BroadcastRedefinedPreVars()	2074
4.95.4.24 PF_StoreInsideInfo()	2075
4.95.4.25 PF_RestoreInsideInfo()	2075
4.95.4.26 PF_BroadcastCBuf()	2075
4.95.4.27 PF_BroadcastExpFlags()	2076
4.95.4.28 PF_BroadcastExpr()	2076
4.95.4.29 PF_BroadcastRHS()	2077
4.95.4.30 PF_InParallelProcessor()	2077
4.95.4.31 PF_SendFile()	2077
4.95.4.32 PF_RecvFile()	2078
4.95.4.33 PF_MLock()	2079
4.95.4.34 PF_MUnlock()	2079
4.95.4.35 PF_WriteFileToFile()	2079
4.95.4.36 PF_FlushStdOutBuffer()	2080
4.95.4.37 PF_FreeErrorMessageBuffers()	2080
4.95.5 Variable Documentation	2080
4.95.5.1 PF	2080
4.96 parallel.c	2081
4.97 parallel.h File Reference	2129
4.97.1 Detailed Description	2131
4.97.2 Macro Definition Documentation	2131
4.97.2.1 MASTER	2131

4.97.2.2 PF_RESET	2131
4.97.2.3 PF_TIME	2132
4.97.2.4 PF_TERM_MSGTAG	2132
4.97.2.5 PF_ENDSORT_MSGTAG	2132
4.97.2.6 PF_DOLLAR_MSGTAG	2132
4.97.2.7 PF_BUFFER_MSGTAG	2132
4.97.2.8 PF_ENDBUFFER_MSGTAG	2132
4.97.2.9 PF_READY_MSGTAG	2132
4.97.2.10 PF_DATA_MSGTAG	2133
4.97.2.11 PF_EMPTY_MSGTAG	2133
4.97.2.12 PF_STDOUT_MSGTAG	2133
4.97.2.13 PF_LOG_MSGTAG	2133
4.97.2.14 PF_OPT_MCTS_MSGTAG	2133
4.97.2.15 PF_OPT_HORNER_MSGTAG	2133
4.97.2.16 PF_OPT_COLLECT_MSGTAG	2133
4.97.2.17 PF_MISC_MSGTAG	2133
4.97.2.18 GNUC_PREREQ	2134
4.97.2.19 OMPI_SKIP_MPICXX	2134
4.97.2.20 indices	2134
4.97.2.21 PF_ANY_SOURCE	2134
4.97.2.22 PF_ANY_MSGTAG	2134
4.97.2.23 PF_COMM	2134
4.97.2.24 PF_BYTE	2134
4.97.2.25 PF_INT	2135
4.97.2.26 PF_LongMultiPack	2135
4.97.2.27 PF_LongMultiUnpack	2135
4.97.3 Function Documentation	2135
4.97.3.1 PF_IsendSbuf()	2135
4.97.3.2 PF_Bcast()	2136
4.97.3.3 PF_RawSend()	2136
4.97.3.4 PF_RawRecv()	2136
4.97.3.5 PF_PreparePack()	2137
4.97.3.6 PF_Pack()	2137
4.97.3.7 PF_Unpack()	2138
4.97.3.8 PF_PackString()	2138
4.97.3.9 PF_UnpackString()	2139
4.97.3.10 PF_Send()	2139
4.97.3.11 PF_Receive()	2140
4.97.3.12 PF_Broadcast()	2140
4.97.3.13 PF_PrepareLongSinglePack()	2141
4.97.3.14 PF_LongSinglePack()	2141
4.97.3.15 PF_LongSingleUnpack()	2141

4.97.3.16 PF_LongSingleSend()	2142
4.97.3.17 PF_LongSingleReceive()	2142
4.97.3.18 PF_PrepareLongMultiPack()	2143
4.97.3.19 PF_LongMultiPackImpl()	2143
4.97.3.20 PF_LongMultiUnpackImpl()	2144
4.97.3.21 PF_LongMultiBroadcast()	2145
4.97.3.22 PF_EndSort()	2145
4.97.3.23 PF_Deferred()	2146
4.97.3.24 PF_Processor()	2147
4.97.3.25 PF_Init()	2148
4.97.3.26 PF_Terminate()	2149
4.97.3.27 PF_GetSlaveTimes()	2150
4.97.3.28 PF_BroadcastNumber()	2150
4.97.3.29 PF_BroadcastBuffer()	2151
4.97.3.30 PF_BroadcastString()	2152
4.97.3.31 PF_BroadcastPreDollar()	2152
4.97.3.32 PF_CollectModifiedDollars()	2153
4.97.3.33 PF_BroadcastModifiedDollars()	2154
4.97.3.34 PF_BroadcastRedefinedPreVars()	2155
4.97.3.35 PF_BroadcastCBuf()	2156
4.97.3.36 PF_BroadcastExpFlags()	2156
4.97.3.37 PF_StoreInsideInfo()	2157
4.97.3.38 PF_RestoreInsideInfo()	2157
4.97.3.39 PF_BroadcastExpr()	2157
4.97.3.40 PF_BroadcastRHS()	2158
4.97.3.41 PF_InParallelProcessor()	2158
4.97.3.42 PF_SendFile()	2158
4.97.3.43 PF_RecvFile()	2159
4.97.3.44 PF_MLock()	2160
4.97.3.45 PF_MUnlock()	2160
4.97.3.46 PF_WriteFileToFile()	2160
4.97.3.47 PF_FlushStdOutBuffer()	2161
4.97.4 Variable Documentation	2161
4.97.4.1 PF	2161
4.97.4.2 PF_maxDollarChunkSize	2161
4.98 parallel.h	2162
4.99 pattern.c File Reference	2165
4.99.1 Detailed Description	2165
4.99.2 Macro Definition Documentation	2166
4.99.2.1 PutInBuffers	2166
4.99.3 Function Documentation	2166
4.99.3.1 TestMatch()	2166

4.99.3.2 Substitute()	2167
4.99.3.3 FindAll()	2167
4.99.3.4 TestSelect()	2167
4.99.3.5 SubsInAll()	2168
4.99.3.6 TransferBuffer()	2168
4.99.3.7 TakeIDfunction()	2168
4.100 pattern.c	2168
4.101 poly.cc	2195
4.102 poly.h	2228
4.103 polyfact.cc File Reference	2231
4.103.1 Detailed Description	2232
4.103.2 Function Documentation	2232
4.103.2.1 operator<<() [1/2]	2232
4.103.2.2 operator<<() [2/2]	2232
4.104 polyfact.cc	2232
4.105 polyfact.h	2248
4.106 polygcd.cc File Reference	2249
4.106.1 Detailed Description	2250
4.106.2 Function Documentation	2250
4.106.2.1 gcd_heuristic_possible()	2250
4.106.3 Description	2250
4.106.4 Notes	2251
4.106.4.1 gcd_linear_helper()	2251
4.107 polygcd.cc	2251
4.108 polygcd.h	2267
4.109 polywrap.cc File Reference	2268
4.109.1 Detailed Description	2269
4.109.2 Function Documentation	2270
4.109.2.1 poly_determine_modulus()	2270
4.109.3 Description	2270
4.109.4 Notes	2270
4.109.4.1 poly_gcd()	2270
4.109.5 Description	2270
4.109.6 Notes	2271
4.109.6.1 poly_divmod()	2271
4.109.6.2 poly_div()	2271
4.109.6.3 poly_rem()	2271
4.109.6.4 poly_ratfun_read()	2272
4.109.7 Description	2272
4.109.8 Notes	2272
4.109.8.1 poly_sort()	2272
4.109.9 Description	2272

4.109.10 Notes	2273
4.109.10.1 poly_ratfun_add()	2273
4.109.11 Description	2273
4.109.12 Notes	2274
4.109.12.1 poly_ratfun_normalize()	2274
4.109.13 Description	2274
4.109.14 Notes	2275
4.109.14.1 poly_fix_minus_signs()	2275
4.109.14.2 poly_factorize()	2275
4.109.15 Description	2276
4.109.16 Notes	2276
4.109.16.1 poly_factorize_argument()	2276
4.109.17 Description	2276
4.109.18 Notes	2277
4.109.18.1 poly_factorize_dollar()	2277
4.109.19 Description	2277
4.109.20 Notes	2277
4.109.20.1 poly_factorize_expression()	2278
4.109.21 Description	2278
4.109.22 Notes	2278
4.109.22.1 poly_unfactorize_expression()	2279
4.109.23 Description	2279
4.109.24 Notes	2279
4.109.24.1 poly_inverse()	2280
4.109.24.2 poly_mul()	2280
4.109.24.3 poly_free_poly_vars()	2280
4.109.25 Variable Documentation	2280
4.109.25.1 POLYWRAP_DENOMPOWER_INCREASE_FACTOR	2280
4.110 polywrap.cc	2280
4.111 portsignals.h File Reference	2302
4.111.1 Detailed Description	2303
4.111.2 Macro Definition Documentation	2303
4.111.2.1 FATAL_SIG_ERROR	2303
4.111.2.2 NSIG	2303
4.111.2.3 SIGSEGV	2303
4.111.2.4 SIGFPE	2304
4.111.2.5 SIGILL	2304
4.111.2.6 SIGEMT	2304
4.111.2.7 SIGSYS	2304
4.111.2.8 SIGPIPE	2304
4.111.2.9 SIGLOST	2304
4.111.2.10 SIGXCPU	2304

4.111.2.11 SIGXFSZ	2304
4.111.2.12 SIGTERM	2305
4.111.2.13 SIGINT	2305
4.111.2.14 SIGQUIT	2305
4.111.2.15 SIGHUP	2305
4.111.2.16 SIGALRM	2305
4.111.2.17 SIGVTALRM	2305
4.111.2.18 SIGPROF	2305
4.112 portsignals.h	2306
4.113 pre.c File Reference	2307
4.113.1 Detailed Description	2309
4.113.2 Macro Definition Documentation	2310
4.113.2.1 STRINGIFY	2310
4.113.2.2 STRINGIFY__	2310
4.113.2.3 SKIPBUFSIZE	2310
4.113.2.4 KILL	2310
4.113.2.5 KILLALL	2310
4.113.2.6 DAEMON	2310
4.113.2.7 SHELL	2310
4.113.2.8 STDERR	2311
4.113.2.9 TRUE_EXPR	2311
4.113.2.10 FALSE_EXPR	2311
4.113.2.11 NOSHELL	2311
4.113.2.12 TERMINAL	2311
4.113.3 Function Documentation	2311
4.113.3.1 GetInput()	2311
4.113.3.2 ClearPushback()	2311
4.113.3.3 GetChar()	2312
4.113.3.4 CharOut()	2312
4.113.3.5 UnsetAllowDelay()	2312
4.113.3.6 GetPreVar()	2312
4.113.3.7 PutPreVar()	2312
4.113.3.8 PopPreVars()	2313
4.113.3.9 IniModule()	2313
4.113.3.10 IniSpecialModule()	2313
4.113.3.11 PreProcessor()	2313
4.113.3.12 PreProInstruction()	2313
4.113.3.13 LoadInstruction()	2313
4.113.3.14 LoadStatement()	2314
4.113.3.15 ExpandTripleDots()	2314
4.113.3.16 FindKeyWord()	2314
4.113.3.17 FindInKeyWord()	2314

4.113.3.18 TheDefine()	2314
4.113.3.19 DoCommentChar()	2315
4.113.3.20 DoPreAssign()	2315
4.113.3.21 DoDefine()	2315
4.113.3.22 DoRedefine()	2315
4.113.3.23 ClearMacro()	2316
4.113.3.24 TheUndefine()	2316
4.113.3.25 DoUndefine()	2316
4.113.3.26 DoInclude()	2316
4.113.3.27 DoReverseInclude()	2316
4.113.3.28 Include()	2316
4.113.3.29 DoPreExchange()	2316
4.113.3.30 DoCall()	2317
4.113.3.31 DoDebug()	2317
4.113.3.32 DoTerminate()	2317
4.113.3.33 DoContinueDo()	2317
4.113.3.34 DoDo()	2317
4.113.3.35 DoBreakDo()	2317
4.113.3.36 DoElse()	2318
4.113.3.37 DoElseif()	2318
4.113.3.38 DoEnddo()	2318
4.113.3.39 DoEndif()	2318
4.113.3.40 DoEndprocedure()	2318
4.113.3.41 Dolf()	2318
4.113.3.42 Dolfdef()	2318
4.113.3.43 Dolfydef()	2319
4.113.3.44 Dolfndef()	2319
4.113.3.45 DoInside()	2319
4.113.3.46 DoEndInside()	2319
4.113.3.47 DoMessage()	2319
4.113.3.48 DoPipe()	2319
4.113.3.49 DoPrcExtension()	2319
4.113.3.50 DoPreOut()	2320
4.113.3.51 DoPrePrintTimes()	2320
4.113.3.52 DoPreSortReallocate()	2320
4.113.3.53 DoPreAppend()	2320
4.113.3.54 DoPreCreate()	2320
4.113.3.55 DoPreRemove()	2320
4.113.3.56 DoPreClose()	2320
4.113.3.57 DoPreWrite()	2321
4.113.3.58 DoProcedure()	2321
4.113.3.59 DoPreBreak()	2321

4.113.3.60 DoPreCase()	2321
4.113.3.61 DoPreDefault()	2321
4.113.3.62 DoPreEndSwitch()	2321
4.113.3.63 DoPreSwitch()	2321
4.113.3.64 DoPreShow()	2322
4.113.3.65 DoSystem()	2322
4.113.3.66 PreLoad()	2322
4.113.3.67 PreSkip()	2322
4.113.3.68 StartPrepro()	2322
4.113.3.69 EvalPrelf()	2322
4.113.3.70 PrelfEval()	2323
4.113.3.71 PreCmp()	2323
4.113.3.72 PreEq()	2323
4.113.3.73 pParseObject()	2323
4.113.3.74 PreCalc()	2323
4.113.3.75 PreEval()	2324
4.113.3.76 AddToPreTypes()	2324
4.113.3.77 MessPreNesting()	2324
4.113.3.78 DoPreAddSeparator()	2324
4.113.3.79 DoPreRmSeparator()	2324
4.113.3.80 DoExternal()	2324
4.113.3.81 DoPrompt()	2324
4.113.3.82 DoSetExternal()	2325
4.113.3.83 DoSetExternalAttr()	2325
4.113.3.84 DoRmExternal()	2325
4.113.3.85 DoFromExternal()	2325
4.113.3.86 DoToExternal()	2325
4.113.3.87 defineChannel()	2325
4.113.3.88 writeToChannel()	2325
4.113.3.89 DoFactDollar()	2326
4.113.3.90 GetDollarNumber()	2326
4.113.3.91 DoSetRandom()	2326
4.113.3.92 DoOptimize()	2326
4.113.3.93 DoClearOptimize()	2326
4.113.3.94 DoSkipExtraSymbols()	2326
4.113.3.95 DoPreReset()	2326
4.113.3.96 DoPreAppendPath()	2327
4.113.3.97 DoPrePrependPath()	2327
4.113.3.98 DoTimeOutAfter()	2327
4.113.3.99 DoNamespace()	2327
4.113.3.100 DoEndNamespace()	2327
4.113.3.101 SkipName()	2327

4.113.3.102 ConstructName()	2328
4.113.3.103 DoUse()	2328
4.113.3.104 UserFlags()	2328
4.113.3.105 DoClearUserFlag()	2328
4.113.3.106 DoSetUserFlag()	2328
4.114 pre.c	2329
4.115 proces.c File Reference	2419
4.115.1 Detailed Description	2419
4.115.2 Macro Definition Documentation	2420
4.115.2.1 DONE	2420
4.115.3 Function Documentation	2420
4.115.3.1 Processor()	2420
4.115.3.2 TestSub()	2421
4.115.3.3 InFunction()	2421
4.115.3.4 InsertTerm()	2422
4.115.3.5 PasteFile()	2424
4.115.3.6 PasteTerm()	2425
4.115.3.7 FiniTerm()	2425
4.115.3.8 Generator()	2426
4.115.3.9 DoOnePow()	2427
4.115.3.10 Deferred()	2428
4.115.3.11 PrepPoly()	2429
4.115.3.12 PolyFunMul()	2430
4.115.4 Variable Documentation	2431
4.115.4.1 printscratch	2431
4.116 proces.c	2431
4.117 ratio.c File Reference	2496
4.117.1 Detailed Description	2497
4.117.2 Function Documentation	2497
4.117.2.1 RatioFind()	2497
4.117.2.2 RatioGen()	2497
4.117.2.3 BinomGen()	2498
4.117.2.4 DoSumF1()	2498
4.117.2.5 Glue()	2498
4.117.2.6 DoSumF2()	2498
4.117.2.7 GCDfunction()	2498
4.117.2.8 GCDfunction3()	2499
4.117.2.9 PutExtraSymbols()	2499
4.117.2.10 TakeExtraSymbols()	2499
4.117.2.11 MultiplyWithTerm()	2499
4.117.2.12 TakeContent()	2499
4.117.2.13 MergeSymbolLists()	2500

4.117.2.14 MergeDotproductLists()	2500
4.117.2.15 CreateExpression()	2500
4.117.2.16 GCDterms()	2500
4.117.2.17 ReadPolyRatFun()	2501
4.117.2.18 FromPolyRatFun()	2501
4.117.2.19 TakeSymbolContent()	2501
4.117.2.20 GCDclean()	2501
4.117.2.21 PolyDiv()	2502
4.117.2.22 DIVfunction()	2502
4.117.2.23 MULfunc()	2502
4.117.2.24 ConvertArgument()	2502
4.117.2.25 ExpandRat()	2502
4.117.2.26 InvPoly()	2502
4.117.3 Variable Documentation	2503
4.117.3.1 divrem	2503
4.117.3.2 TheErrorMessage	2503
4.118 ratio.c	2503
4.119 reken.c File Reference	2545
4.119.1 Detailed Description	2547
4.119.2 Macro Definition Documentation	2547
4.119.2.1 GCDMAX	2547
4.119.2.2 NEWTRICK	2547
4.119.2.3 COPYLONG	2547
4.119.2.4 WARMUP	2548
4.119.3 Function Documentation	2548
4.119.3.1 Pack()	2548
4.119.3.2 UnPack()	2548
4.119.3.3 Mully()	2548
4.119.3.4 Divvy()	2548
4.119.3.5 AddRat()	2549
4.119.3.6 MulRat()	2549
4.119.3.7 DivRat()	2549
4.119.3.8 Simplify()	2549
4.119.3.9 AccumGCD()	2549
4.119.3.10 TakeRatRoot()	2550
4.119.3.11 AddLong()	2550
4.119.3.12 AddPLon()	2550
4.119.3.13 SubPLon()	2550
4.119.3.14 MulLong()	2550
4.119.3.15 BigLong()	2551
4.119.3.16 DivLong()	2551
4.119.3.17 RaisPow()	2551

4.119.3.18 RaisPowCached()	2551
4.119.4 Description	2551
4.119.5 Notes	2552
4.119.5.1 RaisPowMod()	2552
4.119.5.2 NormalModulus()	2552
4.119.5.3 MakeInverses()	2552
4.119.5.4 GetModInverses()	2553
4.119.5.5 GetLongModInverses()	2553
4.119.5.6 Product()	2553
4.119.5.7 Quotient()	2553
4.119.5.8 Remain10()	2554
4.119.5.9 Remain4()	2554
4.119.5.10 PrtLong()	2554
4.119.5.11 GetLong()	2554
4.119.5.12 GcdLong()	2554
4.119.5.13 GetBinom()	2555
4.119.5.14 LcmLong()	2555
4.119.5.15 TakeLongRoot()	2555
4.119.5.16 MakeRational()	2555
4.119.5.17 MakeLongRational()	2555
4.119.5.18 CompCoef()	2556
4.119.5.19 Modulus()	2556
4.119.5.20 TakeModulus()	2556
4.119.5.21 TakeNormalModulus()	2556
4.119.5.22 MakeModTable()	2556
4.119.5.23 Factorial()	2557
4.119.5.24 Bernoulli()	2557
4.119.5.25 NextPrime()	2557
4.119.5.26 Moebius()	2557
4.119.5.27 iniwranf()	2557
4.119.5.28 wranf()	2558
4.119.5.29 iranf()	2558
4.119.5.30 PreRandom()	2558
4.120 reken.c	2558
4.121 reshuf.c File Reference	2604
4.121.1 Detailed Description	2604
4.121.2 Macro Definition Documentation	2605
4.121.2.1 NEWCODE	2605
4.121.3 Function Documentation	2605
4.121.3.1 ReNumber()	2605
4.121.3.2 FunLevel()	2605
4.121.3.3 DetCurDum()	2605

4.121.3.4 FullRenumber()	2605
4.121.3.5 MoveDummies()	2605
4.121.3.6 AdjustRenumScratch()	2606
4.121.3.7 CountDo()	2606
4.121.3.8 CountFun()	2606
4.121.3.9 DimensionSubterm()	2606
4.121.3.10 DimensionTerm()	2606
4.121.3.11 DimensionExpression()	2606
4.121.3.12 MultDo()	2607
4.121.3.13 TryDo()	2607
4.121.3.14 DoDistrib()	2607
4.121.3.15 EqualArg()	2607
4.121.3.16 DoDelta3()	2607
4.121.3.17 TestPartitions()	2607
4.121.3.18 DoPartitions()	2608
4.121.3.19 DoPermutations()	2608
4.121.3.20 DoShuffle()	2608
4.121.3.21 Shuffle()	2608
4.121.3.22 FinishShuffle()	2608
4.121.3.23 DoStuffle()	2608
4.121.3.24 Stuffle()	2609
4.121.3.25 FinishStuffle()	2609
4.121.3.26 StuffRootAdd()	2609
4.122 reshuf.c	2609
4.123 sch.c File Reference	2645
4.123.1 Detailed Description	2646
4.123.2 Function Documentation	2646
4.123.2.1 StrCopy()	2646
4.123.2.2 AddToLine()	2646
4.123.2.3 FiniLine()	2646
4.123.2.4 IniLine()	2647
4.123.2.5 LongToLine()	2647
4.123.2.6 RatToLine()	2647
4.123.2.7 TalToLine()	2647
4.123.2.8 TokenToLine()	2647
4.123.2.9 CodeToLine()	2647
4.123.2.10 MultiplyToLine()	2648
4.123.2.11 AddArrayIndex()	2648
4.123.2.12 PrtTerms()	2648
4.123.2.13 WrtPower()	2648
4.123.2.14 PrintTime()	2648
4.123.2.15 WriteLists()	2648

4.123.2.16 WriteDictionary()	2648
4.123.2.17 WriteArgument()	2649
4.123.2.18 WriteSubTerm()	2649
4.123.2.19 WriteInnerTerm()	2649
4.123.2.20 WriteTerm()	2649
4.123.2.21 WriteExpression()	2649
4.123.2.22 WriteAll()	2649
4.123.2.23 WriteOne()	2650
4.124 sch.c	2650
4.125 setfile.c File Reference	2683
4.125.1 Detailed Description	2684
4.125.2 Macro Definition Documentation	2684
4.125.2.1 NUMERICALVALUE	2684
4.125.2.2 STRINGVALUE	2684
4.125.2.3 PATHVALUE	2684
4.125.2.4 ONOFFVALUE	2684
4.125.2.5 DEFINEVALUE	2685
4.125.2.6 SETBUFSIZE	2685
4.125.3 Function Documentation	2685
4.125.3.1 DoSetups()	2685
4.125.3.2 ProcessOption()	2685
4.125.3.3 GetSetupPar()	2685
4.125.3.4 RecalcSetups()	2685
4.125.3.5 AllocSetups()	2686
4.125.3.6 WriteSetup()	2686
4.125.3.7 AllocSort()	2686
4.125.3.8 AllocSortFileName()	2686
4.125.3.9 AllocFileHandle()	2686
4.125.3.10 DeAllocFileHandle()	2686
4.125.3.11 MakeSetupAllocs()	2687
4.125.3.12 TryFileSetups()	2687
4.125.3.13 TryEnvironment()	2687
4.125.4 Variable Documentation	2687
4.125.4.1 curdirp	2687
4.125.4.2 cursortdirp	2687
4.125.4.3 commentchar	2687
4.125.4.4 dotchar	2687
4.125.4.5 highfirst	2688
4.125.4.6 lowfirst	2688
4.125.4.7 procedureextension	2688
4.125.4.8 setupparameters	2688
4.126 setfile.c	2688

4.127 smart.c File Reference	2703
4.127.1 Detailed Description	2703
4.127.2 Function Documentation	2703
4.127.2.1 StudyPattern()	2703
4.127.2.2 MatchIsPossible()	2703
4.128 smart.c	2704
4.129 sort.c File Reference	2709
4.129.1 Detailed Description	2710
4.129.2 Macro Definition Documentation	2710
4.129.2.1 NEWSPLITMERGE	2710
4.129.2.2 NEWSPLITMERGETIMSORT	2710
4.129.2.3 HUMANSTRLEN	2710
4.129.2.4 HUMANSUFFLEN	2710
4.129.2.5 INSLENGTH	2711
4.129.3 Function Documentation	2711
4.129.3.1 HumanString()	2711
4.129.3.2 WriteStats()	2711
4.129.3.3 NewSort()	2712
4.129.3.4 EndSort()	2712
4.129.3.5 PutIn()	2713
4.129.3.6 Sflush()	2714
4.129.3.7 PutOut()	2714
4.129.3.8 FlushOut()	2715
4.129.3.9 AddCoef()	2716
4.129.3.10 AddPoly()	2716
4.129.3.11 AddArgs()	2717
4.129.3.12 Compare1()	2718
4.129.3.13 CompareSymbols()	2719
4.129.3.14 CompareHSymbols()	2719
4.129.3.15 ComPress()	2719
4.129.3.16 SplitMerge()	2720
4.129.3.17 GarbHand()	2721
4.129.3.18 MergePatches()	2721
4.129.3.19 StoreTerm()	2722
4.129.3.20 StageSort()	2723
4.129.3.21 SortWild()	2723
4.129.3.22 CleanUpSort()	2724
4.129.3.23 LowerSortLevel()	2724
4.129.3.24 PolyRatFunSpecial()	2724
4.129.3.25 SimpleSplitMergeRec()	2724
4.129.3.26 SimpleSplitMerge()	2724
4.129.3.27 BinarySearch()	2725

4.129.4 Variable Documentation	2725
4.129.4.1 toterms	2725
4.129.4.2 humanTermsSuffix	2725
4.129.4.3 humanBytesSuffix	2725
4.130 sort.c	2725
4.131 spectator.c File Reference	2781
4.131.1 Detailed Description	2781
4.131.2 Function Documentation	2781
4.131.2.1 CoCreateSpectator()	2781
4.131.2.2 CoToSpectator()	2782
4.131.2.3 CoRemoveSpectator()	2782
4.131.2.4 CoEmptySpectator()	2782
4.131.2.5 PutInSpectator()	2782
4.131.2.6 FlushSpectators()	2782
4.131.2.7 CoCopySpectator()	2782
4.131.2.8 GetFromSpectator()	2782
4.131.2.9 ClearSpectators()	2783
4.132 spectator.c	2783
4.133 startup.c File Reference	2791
4.133.1 Detailed Description	2792
4.133.2 Macro Definition Documentation	2792
4.133.2.1 STRINGIFY	2792
4.133.2.2 STRINGIFY__	2792
4.133.2.3 FORMNAME	2792
4.133.2.4 VERSIONSTR__	2793
4.133.2.5 VERSIONSTR	2793
4.133.2.6 TAKEPATH	2793
4.133.2.7 DEFAULTFNAMELENGTH	2793
4.133.3 Function Documentation	2793
4.133.3.1 DoTail()	2793
4.133.3.2 OpenInput()	2793
4.133.3.3 ReserveTempFiles()	2793
4.133.3.4 StartVariables()	2794
4.133.3.5 StartMore()	2794
4.133.3.6 IniVars()	2794
4.133.3.7 main()	2794
4.133.3.8 CleanUp()	2795
4.133.3.9 TerminateImpl()	2795
4.133.3.10 PrintDeprecation()	2795
4.133.3.11 PrintRunningTime()	2795
4.133.3.12 GetRunningTime()	2795
4.133.4 Variable Documentation	2796

4.133.4.1 emptystring	2796
4.133.4.2 defaulttempfilename	2796
4.134 startup.c	2796
4.135 store.c File Reference	2821
4.135.1 Detailed Description	2822
4.135.2 Macro Definition Documentation	2822
4.135.2.1 SAVEREVISION	2822
4.135.3 Function Documentation	2822
4.135.3.1 OpenTemp()	2822
4.135.3.2 SeekScratch()	2823
4.135.3.3 SetEndScratch()	2823
4.135.3.4 SetEndHScratch()	2823
4.135.3.5 SetScratch()	2823
4.135.3.6 RevertScratch()	2823
4.135.3.7 ResetScratch()	2823
4.135.3.8 ReadFromScratch()	2824
4.135.3.9 AddToScratch()	2824
4.135.3.10 CoSave()	2824
4.135.3.11 CoLoad()	2824
4.135.3.12 DeleteStore()	2824
4.135.3.13 PutInStore()	2824
4.135.3.14 GetTerm()	2825
4.135.3.15 GetOneTerm()	2825
4.135.3.16 GetMoreTerms()	2825
4.135.3.17 GetMoreFromMem()	2825
4.135.3.18 GetFromStore()	2825
4.135.3.19 DetVars()	2825
4.135.3.20 ToStorage()	2826
4.135.3.21 NextFileIndex()	2826
4.135.3.22 SetFileIndex()	2826
4.135.3.23 VarStore()	2826
4.135.3.24 TermRenummer()	2827
4.135.3.25 FindrNumber()	2827
4.135.3.26 FindInIndex()	2827
4.135.3.27 GetTable()	2828
4.135.3.28 CopyExpression()	2828
4.135.3.29 WriteStoreHeader()	2828
4.135.3.30 ReadSaveHeader()	2828
4.135.3.31 ReadSaveIndex()	2829
4.135.3.32 ReadSaveVariables()	2829
4.135.3.33 ReadSaveTerm32()	2830
4.135.3.34 ReadSaveExpression()	2831

4.136 store.c	2832
4.137 structs.h File Reference	2890
4.137.1 Detailed Description	2893
4.137.2 Macro Definition Documentation	2893
4.137.2.1 INFILEINDEX	2893
4.137.2.2 EMPTYINDEX	2893
4.137.2.3 PHEAD	2893
4.137.2.4 PHEAD0	2893
4.137.2.5 BHEAD	2894
4.137.2.6 BHEAD0	2894
4.137.3 Typedef Documentation	2894
4.137.3.1 INDEXENTRY	2894
4.137.3.2 FILEINDEX	2894
4.137.3.3 VARRENUM	2894
4.137.3.4 RENUMBER	2894
4.137.3.5 NAMENODE	2895
4.137.3.6 NAMETREE	2895
4.137.3.7 COMPTREE	2895
4.137.3.8 TABLES	2895
4.137.3.9 FUNCTIONS	2895
4.137.3.10 FILEHANDLE	2895
4.137.3.11 STREAM	2896
4.137.3.12 TRACES	2896
4.137.3.13 TRACEN	2896
4.137.3.14 PREVAR	2896
4.137.3.15 DOLOOP	2896
4.137.3.16 set_of_char	2896
4.137.3.17 one_byte	2897
4.137.3.18 FINISHUFFLE	2897
4.137.3.19 DO_UFFLE	2897
4.137.3.20 COMPAREDUMMY	2897
4.137.3.21 CBUF	2897
4.137.3.22 CHANNEL	2897
4.137.3.23 NESTING	2897
4.137.3.24 STORECACHE	2898
4.137.3.25 PERM	2898
4.137.3.26 PERMP	2898
4.137.3.27 DISTRIBUTE	2898
4.137.3.28 PARTI	2898
4.137.3.29 SORTING	2898
4.137.3.30 ALLGLOBALS	2898
4.137.3.31 WCN	2899

4.137.3.32 WCN2	2899
4.137.3.33 COMPARE	2899
4.137.3.34 FIXEDGLOBALS	2899
4.138 structs.h	2899
4.139 symmetr.c File Reference	2923
4.139.1 Detailed Description	2924
4.139.2 Function Documentation	2924
4.139.2.1 MatchE()	2924
4.139.2.2 Permute()	2924
4.139.2.3 PermuteP()	2924
4.139.2.4 Distribute()	2925
4.139.2.5 MatchCy()	2925
4.139.2.6 FunMatchCy()	2925
4.139.2.7 FunMatchSy()	2925
4.139.2.8 MatchArgument()	2925
4.140 symmetr.c	2926
4.141 tables.c File Reference	2953
4.141.1 Detailed Description	2954
4.141.2 Function Documentation	2954
4.141.2.1 ClearTableTree()	2954
4.141.2.2 InsTableTree()	2954
4.141.2.3 RedoTableTree()	2954
4.141.2.4 FindTableTree()	2954
4.141.2.5 DoTableExpansion()	2955
4.141.2.6 CoTableBase()	2955
4.141.2.7 FlipTable()	2955
4.141.2.8 SpareTable()	2955
4.141.2.9 FindTB()	2955
4.141.2.10 CoTBcreate()	2955
4.141.2.11 CoTBopen()	2955
4.141.2.12 CoTBaddto()	2956
4.141.2.13 CoTBenter()	2956
4.141.2.14 CoTestUse()	2956
4.141.2.15 CheckTableDeclarations()	2956
4.141.2.16 CoTBload()	2956
4.141.2.17 TestUse()	2956
4.141.2.18 CoTBaudit()	2956
4.141.2.19 CoTBon()	2957
4.141.2.20 CoTBoff()	2957
4.141.2.21 CoTBCleanup()	2957
4.141.2.22 CoTBreplace()	2957
4.141.2.23 CoTBuse()	2957

4.141.2.24 CoApply()	2957
4.141.2.25 CoTBhelp()	2957
4.141.2.26 ReWorkT()	2958
4.141.2.27 Apply()	2958
4.141.2.28 ApplyExec()	2958
4.141.2.29 ApplyReset()	2958
4.141.2.30 TableReset()	2958
4.141.2.31 ReleaseTB()	2958
4.141.3 Variable Documentation	2959
4.141.3.1 helptb	2959
4.142 tables.c	2959
4.143 threads.c File Reference	2986
4.143.1 Detailed Description	2986
4.144 threads.c	2986
4.145 token.c File Reference	3040
4.145.1 Detailed Description	3041
4.145.2 Macro Definition Documentation	3041
4.145.2.1 CHECKPOLY	3041
4.145.3 Function Documentation	3041
4.145.3.1 tokenize()	3041
4.145.3.2 WriteTokens()	3041
4.145.3.3 simp1token()	3041
4.145.3.4 simpwtoken()	3042
4.145.3.5 simp2token()	3042
4.145.3.6 simp3atoken()	3042
4.145.3.7 simp3btoken()	3042
4.145.3.8 simp4token()	3042
4.145.3.9 simp5token()	3042
4.145.3.10 simp6token()	3043
4.145.4 Variable Documentation	3043
4.145.4.1 ttypes	3043
4.146 token.c	3043
4.147 tools.c File Reference	3068
4.147.1 Detailed Description	3070
4.147.2 Macro Definition Documentation	3070
4.147.2.1 FILLVALUE	3070
4.147.2.2 TERMMEMSTARTNUM	3071
4.147.2.3 TERMEXTRAWORDS	3071
4.147.2.4 NUMBERMEMSTARTNUM	3071
4.147.2.5 NUMBEREXTRAWORDS	3071
4.147.2.6 DODOUBLE	3071
4.147.2.7 DOEXPAND	3072

4.147.3 Function Documentation	3072
4.147.3.1 LoadInputFile()	3072
4.147.3.2 ReadFromStream()	3072
4.147.3.3 GetFromStream()	3072
4.147.3.4 LookInStream()	3072
4.147.3.5 OpenStream()	3072
4.147.3.6 LocateFile()	3073
4.147.3.7 CloseStream()	3073
4.147.3.8 CreateStream()	3073
4.147.3.9 GetStreamPosition()	3073
4.147.3.10 PositionStream()	3073
4.147.3.11 ReverseStatements()	3073
4.147.3.12 StartFiles()	3073
4.147.3.13 OpenFile()	3074
4.147.3.14 OpenAddFile()	3074
4.147.3.15 ReOpenFile()	3074
4.147.3.16 CreateFile()	3074
4.147.3.17 CreateLogFile()	3074
4.147.3.18 CloseFile()	3074
4.147.3.19 CopyFile()	3075
4.147.3.20 CreateHandle()	3075
4.147.3.21 ReadFile()	3075
4.147.3.22 ReadPosFile()	3075
4.147.3.23 WriteFileToFile()	3075
4.147.3.24 SeekFile()	3076
4.147.3.25 TellFile()	3076
4.147.3.26 TELLFILE()	3076
4.147.3.27 FlushFile()	3076
4.147.3.28 GetPosFile()	3076
4.147.3.29 SetPosFile()	3076
4.147.3.30 SynchFile()	3077
4.147.3.31 TruncateFile()	3077
4.147.3.32 GetChannel()	3077
4.147.3.33 GetAppendChannel()	3077
4.147.3.34 CloseChannel()	3077
4.147.3.35 UpdateMaxSize()	3077
4.147.3.36 StrCmp()	3077
4.147.3.37 StrlCmp()	3078
4.147.3.38 StrHlCmp()	3078
4.147.3.39 StrlCont()	3078
4.147.3.40 CmpArray()	3078
4.147.3.41 ConWord()	3078

4.147.3.42 StrLen()	3078
4.147.3.43 NumToStr()	3079
4.147.3.44 WriteString()	3079
4.147.3.45 WriteUnfinString()	3079
4.147.3.46 AddToString()	3079
4.147.3.47 strDup1()	3079
4.147.3.48 EndOfToken()	3079
4.147.3.49 ToToken()	3080
4.147.3.50 SkipField()	3080
4.147.3.51 ReadSnum()	3081
4.147.3.52 NumCopy()	3081
4.147.3.53 LongCopy()	3081
4.147.3.54 LongLongCopy()	3081
4.147.3.55 MakeDate()	3081
4.147.3.56 set_in()	3081
4.147.3.57 set_set()	3082
4.147.3.58 set_del()	3082
4.147.3.59 set_sub()	3082
4.147.3.60 iniTools()	3082
4.147.3.61 Malloc1()	3082
4.147.3.62 M_free()	3082
4.147.3.63 M_check1()	3083
4.147.3.64 M_print()	3083
4.147.3.65 TermMallocAddMemory()	3083
4.147.3.66 NumberMallocAddMemory()	3083
4.147.3.67 CacheNumberMallocAddMemory()	3083
4.147.3.68 FromList()	3083
4.147.3.69 From0List()	3083
4.147.3.70 FromVarList()	3084
4.147.3.71 DoubleList()	3084
4.147.3.72 DoubleLList()	3084
4.147.3.73 DoubleBuffer()	3084
4.147.3.74 ExpandBuffer()	3084
4.147.3.75 iexp()	3085
4.147.3.76 ToGeneral()	3085
4.147.3.77 ToFast()	3085
4.147.3.78 ToPolyFunGeneral()	3085
4.147.3.79 IsLikeVector()	3085
4.147.3.80 AreArgsEqual()	3085
4.147.3.81 CompareArgs()	3086
4.147.3.82 CompArg()	3086
4.147.3.83 TimeWallClock()	3086

4.147.3.84 TimeChildren()	3086
4.147.3.85 TimeCPU()	3086
4.147.3.86 Crash()	3087
4.147.3.87 TestTerm()	3087
4.147.3.88 DistrN()	3088
4.147.4 Variable Documentation	3088
4.147.4.1 filelist	3088
4.147.4.2 numinfilelist	3088
4.147.4.3 filelistsize	3088
4.147.4.4 WriteFile	3088
4.148 tools.c	3089
4.149 transform.c File Reference	3134
4.149.1 Detailed Description	3135
4.149.2 Function Documentation	3135
4.149.2.1 CoTransform()	3135
4.149.2.2 RunTransform()	3135
4.149.2.3 RunEncode()	3135
4.149.2.4 RunDecode()	3135
4.149.2.5 RunReplace()	3135
4.149.2.6 RunImplode()	3136
4.149.2.7 RunExplode()	3136
4.149.2.8 RunPermute()	3136
4.149.2.9 RunReverse()	3136
4.149.2.10 RunDedup()	3136
4.149.2.11 RunCycle()	3136
4.149.2.12 RunAddArg()	3137
4.149.2.13 RunMulArg()	3137
4.149.2.14 RunIsLyndon()	3137
4.149.2.15 RunToLyndon()	3137
4.149.2.16 RunDropArg()	3137
4.149.2.17 RunSelectArg()	3137
4.149.2.18 RunZtoHArg()	3138
4.149.2.19 RunHtoZArg()	3138
4.149.2.20 TestArgNum()	3138
4.149.2.21 PutArgInScratch()	3138
4.149.2.22 ReadRange()	3138
4.149.2.23 FindRange()	3138
4.150 transform.c	3139
4.151 unix.h File Reference	3180
4.151.1 Detailed Description	3181
4.151.2 Macro Definition Documentation	3181
4.151.2.1 LINEFEED	3181

4.151.2.2 CARRIAGEReturn	3181
4.151.2.3 WITHPIPE	3181
4.151.2.4 WITHSYSTEM	3181
4.151.2.5 WITHEXTERNALCHANNEL	3181
4.151.2.6 TRAPsignals	3181
4.151.2.7 P_term	3182
4.151.2.8 SEPARATOR	3182
4.151.2.9 ALTSEPARATOR	3182
4.151.2.10 PATHSEPARATOR	3182
4.151.2.11 WITH_ENV	3182
4.151.2.12 WITH_ALARM	3182
4.152 unix.h	3183
4.153 unixfile.c File Reference	3183
4.153.1 Detailed Description	3184
4.154 unixfile.c	3184
4.155 variable.h File Reference	3187
4.155.1 Detailed Description	3188
4.155.2 Macro Definition Documentation	3188
4.155.2.1 chartype	3188
4.155.2.2 Procedures	3188
4.155.2.3 NumProcedures	3189
4.155.2.4 MaxProcedures	3189
4.155.2.5 DoLoops	3189
4.155.2.6 NumDoLoops	3189
4.155.2.7 MaxDoLoops	3189
4.155.2.8 PreVar	3189
4.155.2.9 NumPre	3189
4.155.2.10 MaxNumPre	3189
4.155.2.11 SetElements	3190
4.155.2.12 Sets	3190
4.155.2.13 functions	3190
4.155.2.14 indices	3190
4.155.2.15 symbols	3190
4.155.2.16 vectors	3190
4.155.2.17 tablebases	3190
4.155.2.18 NumFunctions	3190
4.155.2.19 NumIndices	3191
4.155.2.20 NumSymbols	3191
4.155.2.21 NumVectors	3191
4.155.2.22 NumSets	3191
4.155.2.23 NumSetElements	3191
4.155.2.24 NumTableBases	3191

4.155.2.25 GlobalFunctions	3191
4.155.2.26 GlobalIndices	3191
4.155.2.27 GlobalSymbols	3192
4.155.2.28 GlobalVectors	3192
4.155.2.29 GlobalSets	3192
4.155.2.30 GlobalSetElements	3192
4.155.2.31 cbuf	3192
4.155.2.32 channels	3192
4.155.2.33 NumOutputChannels	3192
4.155.2.34 Dollars	3192
4.155.2.35 NumDollars	3193
4.155.2.36 Dubious	3193
4.155.2.37 NumDubious	3193
4.155.2.38 Expressions	3193
4.155.2.39 NumExpressions	3193
4.155.2.40 autofunctions	3193
4.155.2.41 autoindices	3193
4.155.2.42 autosymbols	3193
4.155.2.43 autovectors	3194
4.155.2.44 xsymbol	3194
4.155.2.45 numxsymbol	3194
4.155.2.46 PotModdollars	3194
4.155.2.47 NumPotModdollars	3194
4.155.2.48 ModOptdollars	3194
4.155.2.49 NumModOptdollars	3194
4.155.2.50 BUG	3194
4.155.2.51 AC	3195
4.155.2.52 AM	3195
4.155.2.53 AN	3195
4.155.2.54 AO	3195
4.155.2.55 AP	3195
4.155.2.56 AR	3195
4.155.2.57 AS	3195
4.155.2.58 AT	3195
4.155.2.59 AX	3196
4.155.3 Variable Documentation	3196
4.155.3.1 WriteFile	3196
4.155.3.2 A	3196
4.155.3.3 FG	3196
4.155.3.4 fixedsets	3196
4.155.3.5 setupfilename	3196
4.156 variable.h	3197

4.157 vector.h File Reference	3198
4.157.1 Detailed Description	3200
4.157.2 Macro Definition Documentation	3200
4.157.2.1 VectorStruct	3200
4.157.2.2 Vector	3200
4.157.2.3 DeclareVector	3201
4.157.2.4 VectorInit	3201
4.157.2.5 VectorFree	3201
4.157.2.6 VectorPtr	3202
4.157.2.7 VectorFront	3202
4.157.2.8 VectorBack	3202
4.157.2.9 VectorSize	3203
4.157.2.10 VectorCapacity	3203
4.157.2.11 VectorEmpty	3203
4.157.2.12 VectorClear	3204
4.157.2.13 VectorReserve	3204
4.157.2.14 VectorPushBack	3205
4.157.2.15 VectorPushBacks	3205
4.157.2.16 VectorPopBack	3205
4.157.2.17 VectorInsert	3206
4.157.2.18 VectorInserts	3206
4.157.2.19 VectorErase	3207
4.157.2.20 VectorErases	3207
4.158 vector.h	3207
4.159 wildcard.c File Reference	3211
4.159.1 Detailed Description	3211
4.159.2 Macro Definition Documentation	3211
4.159.2.1 DEBUG	3211
4.159.3 Function Documentation	3212
4.159.3.1 WildFill()	3212
4.159.3.2 ResolveSet()	3212
4.159.3.3 ClearWild()	3212
4.159.3.4 AddWild()	3212
4.159.3.5 CheckWild()	3212
4.159.3.6 DenToFunction()	3213
4.160 wildcard.c	3213

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

AEdge	7
AllGlobals	9
ANode	11
ARGBUFFER	14
Assign	15
AStack	26
bit_field	29
BrAcKeTiNdEx	31
BrAcKeTiNfO	33
BracketInfo	35
bufIPstruct	37
C_const	38
CbUf	70
ChAnNeL	75
COST	75
CSEEq	76
CSEHash	77
dbase	78
DGraph	80
DICTIONARY	81
DICTIONARY_ELEMENT	83
DiStRiBuTe	84
DNode	86
dollar_buf	87
DoLIArS	87
DoLoOp	89
DuBiOuS	93
ECand	94
EEdge	95
EFLine	100
EGraph	102
ENode	115
EStack	119
ExPrEsSiOn	120
FaCdOILaR	124

factorized_poly	125
FGInput	126
FiLe	128
FiLeDaTa	131
FiLeInDeX	132
FixedGlobals	134
fixedset	136
flint::fmpz	137
Fraction	138
FUN_INFO	141
FuNcTiOn	143
gcd_heuristic_failed	146
HANDLERS	146
IInput	148
InDeX	149
indexblock	151
InDeXeNtRy	152
iniinfo	155
INSIDEINFO	156
Interaction	158
KEYWORD	162
KEYWORDV	163
LIST	164
longMultiStruct	166
M_const	167
MCBlock	192
MCBridge	194
MCEdge	195
MConn	196
MCOpi	203
MGraph	206
MiNmAx	218
MInput	219
MNode	220
MNodeClass	222
MoDeL	227
Model	229
MODNUM	237
MoDoPtDoLIaRS	238
monomial_larger	239
MOrbits	239
flint::mpoly	242
flint::mpoly_ctx	243
flint::mpoly_factor	243
N_const	245
nameblock	262
NaMeNode	262
NaMeSpAcE	264
NaMeTree	266
NCand	269
NCInput	270
NeStInG	272
NoDe	273
node	274
NodeEq	276
NodeHash	277
NStack	277
O_const	279

objects	289
OptDef	291
optimization	292
OPTIMIZE	295
OPTIMIZERESULT	298
Options	299
OptQGDef	307
OptQGRef	308
Output	309
P_const	315
ParallelVars	323
PaRtl	325
PaRtlcLe	327
Particle	328
PeRmUtE	332
PeRmUtEp	333
PF_BUFFER	334
PGInput	336
PInput	338
PNodeClass	339
flint::poly	343
poly	344
flint::poly_factor	361
POLYMOD	362
PoSiTiOn	363
PRELOAD	364
pReVaR	365
PROCEDURE	366
Process	367
R_const	374
ReNuMbEr	383
S_const	386
SeTs	388
SETUPPARAMETERS	389
SGroup	391
SHvariables	395
sOrT	397
SpecTatoR	406
SProcess	407
StOrEcAcHe	416
STOREHEADER	417
StreaM	420
SuBbUf	425
SWITCH	426
SWITCHTABLE	428
SyMbOl	429
T_const	431
TaBIEbAsE	445
TaBIEbAsEsUbInDeX	447
TaBIes	448
TERMINFO	454
ToPoTyPe	456
TrAcEn	458
TrAcEs	460
tree	466
tree_node	468
VaRrEnUm	470
VeCtOr	471

VeRtEx	472
X_const	474

Chapter 2

File Index

2.1 File List

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

argument.c	477
bugtool.c	523
checkpoint.c	525
comexpr.c	569
compcomm.c	597
compiler.c	710
compress.c	742
comtool.c	751
comtool.h	764
declare.h	766
diagrams.c	1041
diawrap.cc	1050
dict.c	1065
dollar.c	1083
evaluate.c	1138
execute.c	1151
extcmd.c	1187
factor.c	1208
findpat.c	1219
flintinterface.cc	1237
flintinterface.h	1263
flintwrap.cc	1274
float.c	1281
form3.h	1323
fsizes.h	1338
ftypes.h	1351
function.c	1470
fwin.h	1496
grcc.cc	1498
grcc.h	1626
grccparam.h	1643
if.c	1646
index.c	1663
inivar.h	1673
lus.c	1678

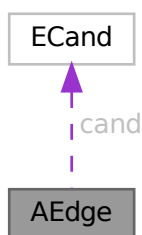
mallocprotect.h	1700
message.c	1704
minos.c	1719
minos.h	1743
model.c	1748
module.c	1755
mpi.c	1768
mpidbg.h	1802
mytime.cc	1816
mytime.h	1816
names.c	1817
normal.c	1866
notation.c	1932
opera.c	1949
optimize.cc	1981
parallel.c	2055
parallel.h	2129
pattern.c	2165
poly.cc	2195
poly.h	2228
polyfact.cc	2231
polyfact.h	2248
polygcd.cc	2249
polygcd.h	2267
polywrap.cc	2268
portsignals.h	2302
pre.c	2307
proces.c	2419
ratio.c	2496
reken.c	2545
reshuf.c	2604
sch.c	2645
setfile.c	2683
smart.c	2703
sort.c	2709
spectator.c	2781
startup.c	2791
store.c	2821
structs.h	2890
symmetr.c	2923
tables.c	2953
threads.c	2986
token.c	3040
tools.c	3068
transform.c	3134
unix.h	3180
unixfile.c	3183
variable.h	3187
vector.h	3198
wildcard.c	3211

Chapter 3

Data Structure Documentation

3.1 AEdge Class Reference

Collaboration diagram for AEdge:



Public Member Functions

- [AEdge](#) (int n0, int l0, int n1, int l1)

Data Fields

- [ECand](#) * [cand](#)
- int [nodes](#) [2]
- int [nlegs](#) [2]
- int [ptcl](#)
- int [DUMMY_PADDING](#)

3.1.1 Detailed Description

Definition at line [1164](#) of file [grcc.h](#).

3.1.2 Constructor & Destructor Documentation

3.1.2.1 AEdge()

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

Definition at line [8274](#) of file [grcc.cc](#).

3.1.2.2 ~AEdge()

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

Definition at line [8285](#) of file [grcc.cc](#).

3.1.3 Field Documentation

3.1.3.1 cand

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

Definition at line [1171](#) of file [grcc.h](#).

3.1.3.2 nodes

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

Definition at line [1172](#) of file [grcc.h](#).

3.1.3.3 nlegs

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

Definition at line [1173](#) of file [grcc.h](#).

3.1.3.4 ptcl

```
int AEdge::ptcl
```

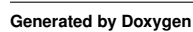
Definition at line [1174](#) of file [grcc.h](#).


```
int AEdge::DUMMYPADDING
```

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

- ## 3.2 AllGlobals Struct Reference

Collaboration diagram for AllGlobals:



Data Fields

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

3.2.1 Detailed Description

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

Definition at line [2468](#) of file [structs.h](#).

3.2.2 Field Documentation

3.2.2.1 M

```
struct M\_const AllGlobals::M
```

Definition at line [2469](#) of file [structs.h](#).

3.2.2.2 Cc

```
struct C\_const AllGlobals::Cc
```

Definition at line [2470](#) of file [structs.h](#).

3.2.2.3 S

```
struct S\_const AllGlobals::S
```

Definition at line [2471](#) of file [structs.h](#).

3.2.2.4 R

```
struct R\_const AllGlobals::R
```

Definition at line [2472](#) of file [structs.h](#).

3.2.2.5 N

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

Definition at line 2473 of file [structs.h](#).

3.2.2.6 O

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

Definition at line 2474 of file [structs.h](#).

3.2.2.7 P

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

Definition at line 2475 of file [structs.h](#).

3.2.2.8 T

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

Definition at line 2476 of file [structs.h](#).

3.2.2.9 X

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

Definition at line 2477 of file [structs.h](#).

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

- [structs.h](#)

3.3 ANode Class Reference

Collaboration diagram for ANode:



Public Member Functions

- [ANode](#) (int dg)
- int [newleg](#) (void)

Data Fields

- int [deg](#)
- int [nlegs](#)
- int * [anodes](#)
- int * [aedges](#)
- int * [aelegs](#)
- [NCand](#) * [cand](#)

3.3.1 Detailed Description

Definition at line [1140](#) of file [grcc.h](#).

3.3.2 Constructor & Destructor Documentation

3.3.2.1 ANode()

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

Definition at line [8219](#) of file [grcc.cc](#).

3.3.2.2 ~ANode()

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

Definition at line [8237](#) of file [grcc.cc](#).

3.3.3 Member Function Documentation

3.3.3.1 newleg()

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

Definition at line [8254](#) of file [grcc.cc](#).

3.3.4 Field Documentation

3.3.4.1 deg

```
int ANode::deg
```

Definition at line 1150 of file [grcc.h](#).

3.3.4.2 nlegs

```
int ANode::nlegs
```

Definition at line 1151 of file [grcc.h](#).

3.3.4.3 anodes

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

Definition at line 1152 of file [grcc.h](#).

3.3.4.4 aedges

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

Definition at line 1153 of file [grcc.h](#).

3.3.4.5 aelegs

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

Definition at line 1154 of file [grcc.h](#).

3.3.4.6 cand

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

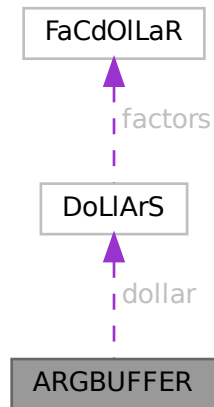
Definition at line 1155 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.4 ARGBUFFER Struct Reference

Collaboration diagram for ARGBUFFER:



Data Fields

- WORD * [buffer](#)
- DOLLARS [dollar](#)
- LONG [size](#)
- int [type](#)
- int [dummy](#)

3.4.1 Detailed Description

Definition at line [601](#) of file [ratio.c](#).

3.4.2 Field Documentation

3.4.2.1 buffer

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

Definition at line [602](#) of file [ratio.c](#).

3.4.2.2 dollar

```
DOLLARS ARGBUFFER::dollar
```

Definition at line [603](#) of file [ratio.c](#).

3.4.2.3 size

```
LONG ARGBUFFER::size
```

Definition at line 604 of file [ratio.c](#).

3.4.2.4 type

```
int ARGBUFFER::type
```

Definition at line 605 of file [ratio.c](#).

3.4.2.5 dummy

```
int ARGBUFFER::dummy
```

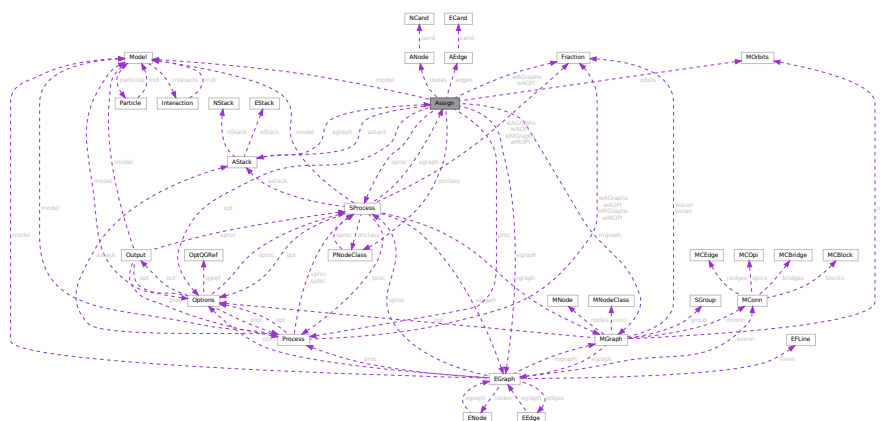
Definition at line 606 of file [ratio.c](#).

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

- [ratio.c](#)

3.5 Assign Class Reference

Collaboration diagram for Assign:



Public Member Functions

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

Data Fields

- [EGraph](#) * egraph
- [MGraph](#) * mgraph
- [SProcess](#) * sprc
- [Process](#) * proc
- [Model](#) * model
- [Options](#) * opt
- [AStack](#) * astack
- [PNodeClass](#) * pnclass
- [MOrbits](#) * orbits
- int [nNodes](#)
- int [nEdges](#)

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

3.5.1 Detailed Description

Definition at line [1183](#) of file [grcc.h](#).

3.5.2 Constructor & Destructor Documentation

3.5.2.1 Assign()

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

Definition at line [8294](#) of file [grcc.cc](#).

3.5.2.2 ~Assign()

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

Definition at line [8370](#) of file [grcc.cc](#).

3.5.3 Member Function Documentation

3.5.3.1 prCand()

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

Definition at line [8390](#) of file [grcc.cc](#).

3.5.3.2 checkAG()

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

Definition at line [8426](#) of file [grcc.cc](#).

3.5.3.3 assignAllVertices()

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

Definition at line [8450](#) of file [grcc.cc](#).

3.5.3.4 selectVertex()

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

Definition at line [8513](#) of file [grcc.cc](#).

3.5.3.5 selectVertexSimp()

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

Definition at line [8474](#) of file [grcc.cc](#).

3.5.3.6 selectLeg()

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

Definition at line [8552](#) of file [grcc.cc](#).

3.5.3.7 assignVertex()

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

Definition at line [8654](#) of file [grcc.cc](#).

3.5.3.8 allAssigned()

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

Definition at line [8777](#) of file [grcc.cc](#).

3.5.3.9 fromMGraph()

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

Definition at line [8857](#) of file [grcc.cc](#).

3.5.3.10 addEdge()

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

Definition at line 9003 of file [grcc.cc](#).

3.5.3.11 connect()

```
void Assign::connect (
    int n0,
    int l0,
    int eg,
    int e1,
    int n1,
    int l1 )
```

Definition at line 9048 of file [grcc.cc](#).

3.5.3.12 fillEGraph()

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

Definition at line 9077 of file [grcc.cc](#).

3.5.3.13 reordLeg()

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

Definition at line 9217 of file [grcc.cc](#).

3.5.3.14 getLegParticle()

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

Definition at line 9292 of file [grcc.cc](#).

3.5.3.15 legEdgeParticle()

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

Definition at line 9319 of file [grcc.cc](#).

3.5.3.16 legPart()

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

Definition at line 9339 of file [grcc.cc](#).

3.5.3.17 candPart()

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

Definition at line 9355 of file [grcc.cc](#).

3.5.3.18 selUnAssVertex()

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

Definition at line 9408 of file [grcc.cc](#).

3.5.3.19 selUnAssVertexSimp()

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

Definition at line 9388 of file [grcc.cc](#).

3.5.3.20 selUnAssLeg()

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

Definition at line 9437 of file [grcc.cc](#).

3.5.3.21 assignVertex()

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

Definition at line 9476 of file [grcc.cc](#).

3.5.3.22 assignPLeg()

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

Definition at line 9522 of file [grcc.cc](#).

3.5.3.23 isOrdPLeg()

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

Definition at line 9595 of file [grcc.cc](#).

3.5.3.24 detEdge()

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

Definition at line 9574 of file [grcc.cc](#).

3.5.3.25 candPartClassify()

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

Definition at line 9639 of file [grcc.cc](#).

3.5.3.26 updateCandNode()

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

Definition at line 9676 of file [grcc.cc](#).

3.5.3.27 checkOrderCpl()

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

Definition at line 9815 of file [grcc.cc](#).

3.5.3.28 isOrdLegs()

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

Definition at line 9853 of file [grcc.cc](#).

3.5.3.29 isIsomorphic()

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

Definition at line 9891 of file [grcc.cc](#).

3.5.3.30 cmpPermGraph()

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

Definition at line 9967 of file [grcc.cc](#).

3.5.3.31 cmpNodes()

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

Definition at line 10049 of file [grcc.cc](#).

3.5.3.32 edgeSym()

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

Definition at line 10069 of file [grcc.cc](#).

3.5.3.33 saveCouple()

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

Definition at line 8614 of file [grcc.cc](#).

3.5.3.34 restoreCouple()

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

Definition at line 8626 of file [grcc.cc](#).

3.5.3.35 subCouple()

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

Definition at line 8638 of file [grcc.cc](#).

3.5.3.36 checkCand()

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

Definition at line 10128 of file [grcc.cc](#).

3.5.3.37 checkNode()

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

Definition at line 10200 of file [grcc.cc](#).

3.5.4 Field Documentation

3.5.4.1 egraph

[EGraph*](#) Assign::egraph

Definition at line 1188 of file [grcc.h](#).

3.5.4.2 mgraph

`MGraph*` `Assign::mgraph`

Definition at line 1189 of file `grcc.h`.

3.5.4.3 sproc

`SProcess*` `Assign::sproc`

Definition at line 1190 of file `grcc.h`.

3.5.4.4 proc

`Process*` `Assign::proc`

Definition at line 1191 of file `grcc.h`.

3.5.4.5 model

`Model*` `Assign::model`

Definition at line 1192 of file `grcc.h`.

3.5.4.6 opt

`Options*` `Assign::opt`

Definition at line 1193 of file `grcc.h`.

3.5.4.7 astack

`AStack*` `Assign::astack`

Definition at line 1194 of file `grcc.h`.

3.5.4.8 pnclass

`PNodeClass*` `Assign::pnclass`

Definition at line 1195 of file `grcc.h`.

3.5.4.9 orbits

`MOrbits*` `Assign::orbits`

Definition at line 1196 of file `grcc.h`.

3.5.4.10 nNodes

```
int Assign::nNodes
```

Definition at line 1198 of file [grcc.h](#).

3.5.4.11 nEdges

```
int Assign::nEdges
```

Definition at line 1199 of file [grcc.h](#).

3.5.4.12 nExtern

```
int Assign::nExtern
```

Definition at line 1200 of file [grcc.h](#).

3.5.4.13 nETotal

```
int Assign::nETotal
```

Definition at line 1201 of file [grcc.h](#).

3.5.4.14 nAGraphs

```
BigInt Assign::nAGraphs
```

Definition at line 1203 of file [grcc.h](#).

3.5.4.15 wAGraphs

```
Fraction Assign::wAGraphs
```

Definition at line 1204 of file [grcc.h](#).

3.5.4.16 nAOPI

```
BigInt Assign::nAOPI
```

Definition at line 1205 of file [grcc.h](#).

3.5.4.17 wAOPI

```
Fraction Assign::wAOPI
```

Definition at line 1206 of file [grcc.h](#).

Public Member Functions

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

Data Fields

- [Assign](#) * [agraph](#)
- [NStack](#) ** [nStack](#)
- int [nStackP](#)
- int [nSize](#)
- [EStack](#) ** [eStack](#)
- int [eStackP](#)
- int [eSize](#)

3.6.1 Detailed Description

Definition at line 1367 of file [grcc.h](#).

3.6.2 Constructor & Destructor Documentation

3.6.2.1 AStack()

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

Definition at line 10239 of file [grcc.cc](#).

3.6.2.2 ~AStack()

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

Definition at line 10271 of file [grcc.cc](#).

3.6.3 Member Function Documentation

3.6.3.1 setAGraph()

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

Definition at line 10299 of file [grcc.cc](#).

3.6.3.2 checkPoint()

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

Definition at line 10364 of file [grcc.cc](#).

3.6.3.3 restore()

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

Definition at line 10408 of file [grcc.cc](#).

3.6.3.4 restoreMsg()

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

Definition at line 10416 of file [grcc.cc](#).

3.6.3.5 prStack()

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

Definition at line 10431 of file [grcc.cc](#).

3.6.3.6 pushNode()

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

Definition at line 10305 of file [grcc.cc](#).

3.6.3.7 pushEdge()

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

Definition at line 10335 of file [grcc.cc](#).

3.6.4 Field Documentation

3.6.4.1 agraph

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

Definition at line 1371 of file [grcc.h](#).

3.6.4.2 nStack

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

Definition at line 1373 of file [grcc.h](#).

3.6.4.3 nStackP

```
int AStack::nStackP
```

Definition at line 1374 of file [grcc.h](#).

3.6.4.4 nSize

```
int AStack::nSize
```

Definition at line 1375 of file [grcc.h](#).

3.6.4.5 eStack

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

Definition at line 1377 of file [grcc.h](#).

3.6.4.6 eStackP

```
int AStack::eStackP
```

Definition at line 1378 of file [grcc.h](#).

3.6.4.7 eSize

```
int AStack::eSize
```

Definition at line 1379 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.7 bit_field Struct Reference

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

Data Fields

- UBYTE [bit_0](#): 1
- UBYTE [bit_1](#): 1
- UBYTE [bit_2](#): 1
- UBYTE [bit_3](#): 1
- UBYTE [bit_4](#): 1
- UBYTE [bit_5](#): 1
- UBYTE [bit_6](#): 1
- UBYTE [bit_7](#): 1

3.7.1 Detailed Description

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

Definition at line [906](#) of file [structs.h](#).

3.7.2 Field Documentation

3.7.2.1 `bit_0`

```
UBYTE bit_field::bit_0
```

Definition at line [907](#) of file [structs.h](#).

3.7.2.2 `bit_1`

```
UBYTE bit_field::bit_1
```

Definition at line [908](#) of file [structs.h](#).

3.7.2.3 `bit_2`

```
UBYTE bit_field::bit_2
```

Definition at line [909](#) of file [structs.h](#).

3.7.2.4 `bit_3`

```
UBYTE bit_field::bit_3
```

Definition at line [910](#) of file [structs.h](#).

3.7.2.5 bit_4

```
UBYTE bit_field::bit_4
```

Definition at line 911 of file [structs.h](#).

3.7.2.6 bit_5

```
UBYTE bit_field::bit_5
```

Definition at line 912 of file [structs.h](#).

3.7.2.7 bit_6

```
UBYTE bit_field::bit_6
```

Definition at line 913 of file [structs.h](#).

3.7.2.8 bit_7

```
UBYTE bit_field::bit_7
```

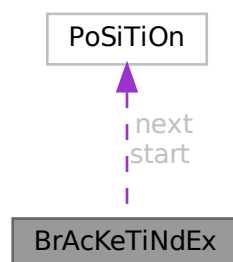
Definition at line 914 of file [structs.h](#).

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

- [structs.h](#)

3.8 BrAcKeTiNdEx Struct Reference

Collaboration diagram for BrAcKeTiNdEx:



Data Fields

- [POSITION start](#)
- [POSITION next](#)
- LONG [bracket](#)
- LONG [termsinbracket](#)

3.8.1 Detailed Description

Definition at line [311](#) of file [structs.h](#).

3.8.2 Field Documentation

3.8.2.1 start

```
POSITION BrAcKeTiNdEx::start
```

Definition at line [312](#) of file [structs.h](#).

3.8.2.2 next

```
POSITION BrAcKeTiNdEx::next
```

Definition at line [313](#) of file [structs.h](#).

3.8.2.3 bracket

```
LONG BrAcKeTiNdEx::bracket
```

Definition at line [314](#) of file [structs.h](#).

3.8.2.4 termsinbracket

```
LONG BrAcKeTiNdEx::termsinbracket
```

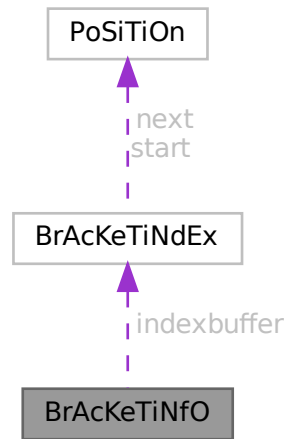
Definition at line [315](#) of file [structs.h](#).

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

- [structs.h](#)

3.9 BrAcKeTiNfO Struct Reference

Collaboration diagram for BrAcKeTiNfO:



Data Fields

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

3.9.1 Detailed Description

Definition at line 322 of file [structs.h](#).

3.9.2 Field Documentation

3.9.2.1 indexbuffer

`BRACKETINDEX* BrAcKeTiNfO::indexbuffer`

[D]

Definition at line 323 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#), and [FlushOut\(\)](#).

3.9.2.2 bracketbuffer

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

[D]

Definition at line 324 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#).

3.9.2.3 bracketbuffersize

```
LONG BrAcKeTiNfO::bracketbuffersize
```

Definition at line 325 of file [structs.h](#).

3.9.2.4 indexbuffersize

```
LONG BrAcKeTiNfO::indexbuffersize
```

Definition at line 326 of file [structs.h](#).

3.9.2.5 bracketfill

```
LONG BrAcKeTiNfO::bracketfill
```

Definition at line 327 of file [structs.h](#).

3.9.2.6 indexfill

```
LONG BrAcKeTiNfO::indexfill
```

Definition at line 328 of file [structs.h](#).

3.9.2.7 SortType

```
WORD BrAcKeTiNfO::SortType
```

The sorting criterium used (like POWERFIRST etc)

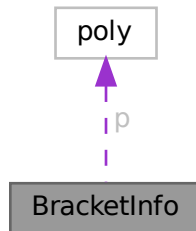
Definition at line 329 of file [structs.h](#).

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

- [structs.h](#)

3.10 BracketInfo Struct Reference

Collaboration diagram for BracketInfo:



Public Member Functions

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

Data Fields

- std::vector< int > [pattern](#)
- int [num_terms](#)
- int [dummy](#) = 0
- const [poly](#) * [p](#)

3.10.1 Detailed Description

Definition at line [1389](#) of file [polygcd.cc](#).

3.10.2 Constructor & Destructor Documentation

3.10.2.1 BracketInfo()

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

Definition at line [1394](#) of file [polygcd.cc](#).

3.10.3 Member Function Documentation

3.10.3.1 `operator<()`

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

Definition at line [1395](#) of file [polygcd.cc](#).

3.10.4 Field Documentation

3.10.4.1 `pattern`

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

Definition at line [1390](#) of file [polygcd.cc](#).

3.10.4.2 `num_terms`

```
int BracketInfo::num_terms
```

Definition at line [1391](#) of file [polygcd.cc](#).

3.10.4.3 `dummy`

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

Definition at line [1391](#) of file [polygcd.cc](#).

3.10.4.4 `p`

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

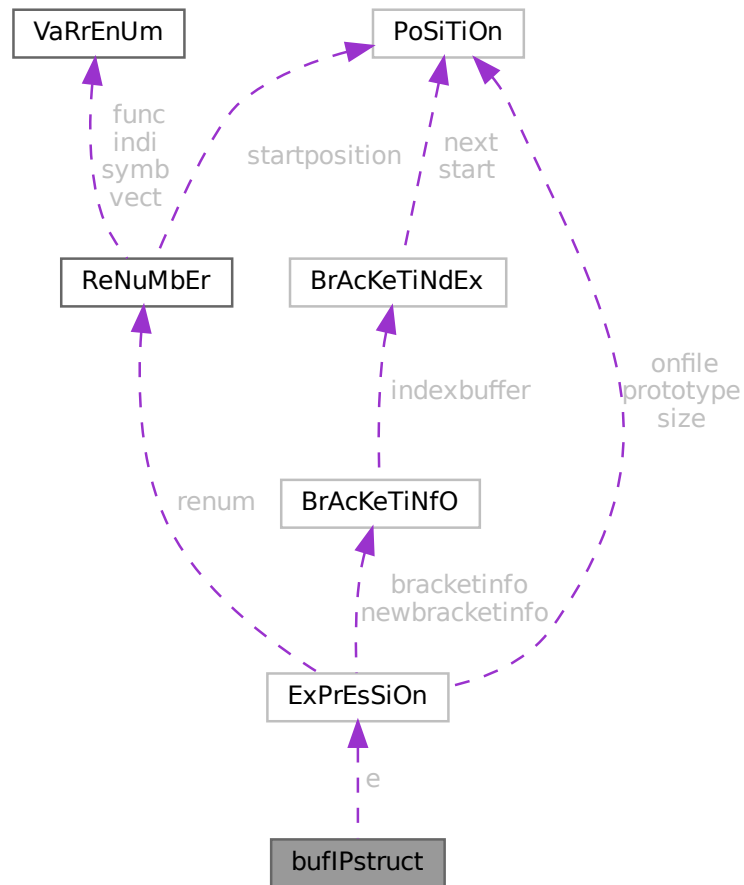
Definition at line [1392](#) of file [polygcd.cc](#).

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

- [polygcd.cc](#)

3.11 bufIPstruct Struct Reference

Collaboration diagram for bufIPstruct:



Data Fields

- LONG `i`
- struct `ExPrEsSiOn e`

3.11.1 Detailed Description

Definition at line 3922 of file [parallel.c](#).

3.11.2 Field Documentation

3.11.2.1 `i`

LONG `bufIPstruct::i`

Definition at line 3923 of file [parallel.c](#).

3.11.2.2 e

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

Definition at line 3924 of file [parallel.c](#).

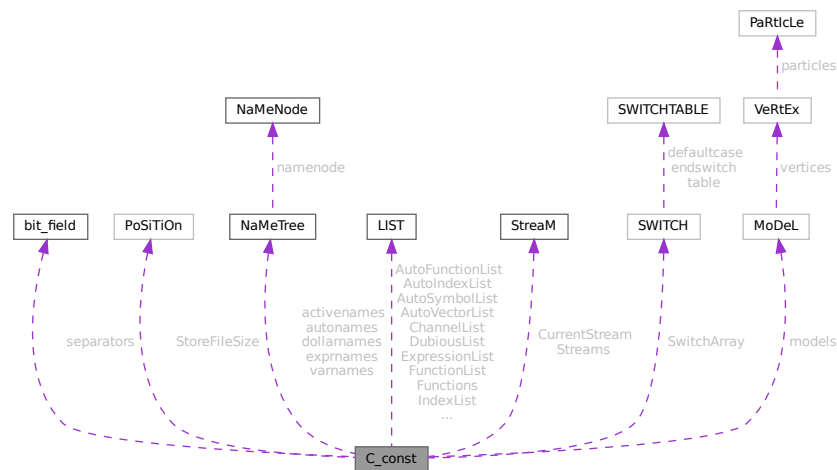
The documentation for this struct was generated from the following file:

- [parallel.c](#)

3.12 C_const Struct Reference

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

Collaboration diagram for C_const:



Data Fields

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

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

- LONG [iBufferSize](#)
- LONG [TransEname](#)
- LONG [ProcessBucketSize](#)
- LONG [mProcessBucketSize](#)
- LONG [CModule](#)
- LONG [ThreadBucketSize](#)
- LONG [CheckpointStamp](#)
- LONG [CheckpointInterval](#)
- int [cbufnum](#)
- int [AutoDeclareFlag](#)
- int [NoShowInput](#)
- int [ShortStats](#)
- int [compiletype](#)
- int [firstconstindex](#)
- int [insidefirst](#)
- int [minsidefirst](#)
- int [wildflag](#)
- int [NumLabels](#)
- int [MaxLabels](#)
- int [IDefDim](#)
- int [IDefDim4](#)
- int [NumWildcardNames](#)
- int [WildcardBufferSize](#)
- int [MaxIf](#)
- int [NumStreams](#)
- int [MaxNumStreams](#)
- int [firstctypemessage](#)
- int [tablecheck](#)
- int [idoption](#)
- int [BottomLevel](#)
- int [CompileLevel](#)
- int [TokensWriteFlag](#)
- int [UnsureDollarMode](#)
- int [outsidefun](#)
- int [funpowers](#)
- int [WarnFlag](#)
- int [StatsFlag](#)
- int [NamesFlag](#)
- int [CodesFlag](#)
- int [SetupFlag](#)
- int [SortType](#)
- int [ISortType](#)
- int [ThreadStats](#)
- int [FinalStats](#)
- int [OldParallelStats](#)
- int [ThreadsFlag](#)
- int [ThreadBalancing](#)
- int [ThreadSortFileSynch](#)
- int [ProcessStats](#)
- int [BracketNormalize](#)
- int [maxtermlevel](#)
- int [dumnumflag](#)
- int [bracketindexflag](#)
- int [parallelflag](#)
- int [mparallelflag](#)

- int [inparallelflag](#)
- int [partodoflag](#)
- int [properorderflag](#)
- int [vetofilling](#)
- int [tablefilling](#)
- int [vetotablebasefill](#)
- int [exprfillwarning](#)
- int [lhdollarflag](#)
- int [NoCompress](#)
- int [IsFortran90](#)
- int [MultiBracketLevels](#)
- int [topolynomialflag](#)
- int [ffbufnum](#)
- int [OldFactArgFlag](#)
- int [MemDebugFlag](#)
- int [OldGCDflag](#)
- int [WTimeStatsFlag](#)
- int [SortReallocateFlag](#)
- int [PrintBacktraceFlag](#)
- int [FlintPolyFlag](#)
- int [HumanStatsFlag](#)
- int [doloopstacksize](#)
- int [dolooplevel](#)
- int [CheckpointFlag](#)
- int [SizeCommutelnSet](#)
- int [origin](#)
- int [vectorlikeLHS](#)
- int [nummodels](#)
- int [modelspace](#)
- int [ModelLevel](#)
- int [ShortStatsMax](#)
- int [InnerTest](#)
- WORD [argsumcheck](#) [MAXNEST]
- WORD [insidesumcheck](#) [MAXNEST]
- WORD [inexprsumcheck](#) [MAXNEST]
- WORD [RepSumCheck](#) [MAXREPEAT]
- WORD [IUniTrace](#) [4]
- WORD [RepLevel](#)
- WORD [arglevel](#)
- WORD [insidelevel](#)
- WORD [inexprlevel](#)
- WORD [termlevel](#)
- WORD [MustTestTable](#)
- WORD [DumNum](#)
- WORD [ncmod](#)
- WORD [npowmod](#)
- WORD [modmode](#)
- WORD [nhalfmod](#)
- WORD [DirtPow](#)
- WORD [IUnitTrace](#)
- WORD [NwildC](#)
- WORD [ComDefer](#)
- WORD [CollectFun](#)
- WORD [AltCollectFun](#)
- WORD [OutputMode](#)

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

3.12.1 Detailed Description

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

Definition at line [1648](#) of file [structs.h](#).

3.12.2 Field Documentation

3.12.2.1 separators

[set_of_char](#) C_const::separators

Separators in #call and #do

Definition at line [1649](#) of file [structs.h](#).

3.12.2.2 StoreFileSize

[POSITION](#) C_const::StoreFileSize

Definition at line [1650](#) of file [structs.h](#).

3.12.2.3 dollarnames

`NAMETREE* C_const::dollarnames`

[D] Names of dollar variables

Definition at line 1651 of file [structs.h](#).

3.12.2.4 exprnames

`NAMETREE* C_const::exprnames`

[D] Names of expressions

Definition at line 1652 of file [structs.h](#).

3.12.2.5 varnames

`NAMETREE* C_const::varnames`

[D] Names of regular variables

Definition at line 1653 of file [structs.h](#).

3.12.2.6 ChannelList

`LIST C_const::ChannelList`

Used for the #write statement. Contains [CHANNEL](#)

Definition at line 1654 of file [structs.h](#).

3.12.2.7 DubiousList

`LIST C_const::DubiousList`

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

Definition at line 1656 of file [structs.h](#).

3.12.2.8 FunctionList

`LIST C_const::FunctionList`

List of functions and properties. Contains [FUNCTIONS](#)

Definition at line 1658 of file [structs.h](#).

3.12.2.9 ExpressionList

`LIST C_const::ExpressionList`

List of expressions, locations etc.

Definition at line 1659 of file [structs.h](#).

3.12.2.10 IndexList

`LIST C_const::IndexList`

List of indices

Definition at line 1660 of file [structs.h](#).

3.12.2.11 SetElementList

`LIST C_const::SetElementList`

List of all elements of all sets

Definition at line 1661 of file [structs.h](#).

3.12.2.12 SetList

`LIST C_const::SetList`

List of the sets

Definition at line 1662 of file [structs.h](#).

3.12.2.13 SymbolList

`LIST C_const::SymbolList`

List of the symbols and their properties

Definition at line 1663 of file [structs.h](#).

3.12.2.14 VectorList

`LIST C_const::VectorList`

List of the vectors

Definition at line 1664 of file [structs.h](#).

3.12.2.15 PotModDolList

`LIST C_const::PotModDolList`

Potentially changed dollars

Definition at line 1665 of file [structs.h](#).

3.12.2.16 ModOptDolList

`LIST C_const::ModOptDolList`

Module Option Dollars list

Definition at line 1666 of file [structs.h](#).

3.12.2.17 TableBaseList

`LIST C_const::TableBaseList`

TableBase list

Definition at line 1667 of file [structs.h](#).

3.12.2.18 cbufList

`LIST C_const::cbufList`

List of compiler buffers

Definition at line 1671 of file [structs.h](#).

3.12.2.19 AutoSymbolList

`LIST C_const::AutoSymbolList`

Definition at line 1675 of file [structs.h](#).

3.12.2.20 AutoIndexList

`LIST C_const::AutoIndexList`

Definition at line 1676 of file [structs.h](#).

3.12.2.21 AutoVectorList

`LIST C_const::AutoVectorList`

Definition at line 1677 of file [structs.h](#).

3.12.2.22 AutoFunctionList

`LIST C_const::AutoFunctionList`

Definition at line 1678 of file [structs.h](#).

3.12.2.23 autonames

`NAMETREE* C_const::autonames`

[D] Names in autodeclare

Definition at line 1679 of file [structs.h](#).

3.12.2.24 Symbols

`LIST* C_const::Symbols`

Definition at line 1681 of file [structs.h](#).

3.12.2.25 Indices

`LIST* C_const::Indices`

Definition at line 1683 of file [structs.h](#).

3.12.2.26 Vectors

`LIST* C_const::Vectors`

Definition at line 1684 of file [structs.h](#).

3.12.2.27 Functions

`LIST* C_const::Functions`

Definition at line 1685 of file [structs.h](#).

3.12.2.28 activenames

`NAMETREE** C_const::activenames`

Definition at line 1686 of file [structs.h](#).

3.12.2.29 Streams

`STREAM* C_const::Streams`

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

Definition at line 1688 of file [structs.h](#).

3.12.2.30 CurrentStream

`STREAM* C_const::CurrentStream`

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

Definition at line 1689 of file [structs.h](#).

3.12.2.31 SwitchArray

`SWITCH* C_const::SwitchArray`

Definition at line 1691 of file [structs.h](#).

3.12.2.32 models

`MODEL** C_const::models`

Definition at line 1692 of file [structs.h](#).

3.12.2.33 SwitchHeap

`WORD* C_const::SwitchHeap`

Definition at line 1693 of file [structs.h](#).

3.12.2.34 termstack

`LONG* C_const::termstack`

[D] Last term statement {offset}

Definition at line 1694 of file [structs.h](#).

3.12.2.35 termsortstack

`LONG* C_const::termsortstack`

[D] Last sort statement {offset}

Definition at line 1695 of file [structs.h](#).

3.12.2.36 cmod

UWORD* C_const::cmod

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

Definition at line 1696 of file [structs.h](#).

3.12.2.37 powmod

UWORD* C_const::powmod

Local setting printing as powers. Points into cmod memory

Definition at line 1697 of file [structs.h](#).

3.12.2.38 modpowers

UWORD* C_const::modpowers

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

Definition at line 1698 of file [structs.h](#).

3.12.2.39 halfmod

UWORD* C_const::halfmod

Definition at line 1699 of file [structs.h](#).

3.12.2.40 ProtoType

WORD* C_const::ProtoType

Definition at line 1700 of file [structs.h](#).

3.12.2.41 WildC

WORD* C_const::WildC

Definition at line 1701 of file [structs.h](#).

3.12.2.42 IfHeap

LONG* C_const::IfHeap

[D] Keeps track of where to go in if

Definition at line 1702 of file [structs.h](#).

3.12.2.43 IfCount

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

[D] Keeps track of where to go in if

Definition at line 1703 of file [structs.h](#).

3.12.2.44 IfStack

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

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

Definition at line 1704 of file [structs.h](#).

3.12.2.45 iBuffer

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

[D] Compiler input buffer

Definition at line 1705 of file [structs.h](#).

3.12.2.46 iPointer

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

Running pointer in the compiler input buffer

Definition at line 1706 of file [structs.h](#).

3.12.2.47 iStop

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

Top of iBuffer

Definition at line 1707 of file [structs.h](#).

3.12.2.48 LabelNames

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

[D] List of names in label statements

Definition at line 1708 of file [structs.h](#).

3.12.2.49 FixIndices

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

[D] Buffer of fixed indices

Definition at line 1709 of file [structs.h](#).

3.12.2.50 termsumcheck

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

[D] Checking of nesting

Definition at line 1710 of file [structs.h](#).

3.12.2.51 WildcardNames

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

[D] Names of ?a variables

Definition at line 1711 of file [structs.h](#).

3.12.2.52 Labels

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

Label information for during run. Pointer into LabelNames memory.

Definition at line 1712 of file [structs.h](#).

3.12.2.53 tokens

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

[D] Array with tokens for tokenizer

Definition at line 1713 of file [structs.h](#).

3.12.2.54 toptokens

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

Top of tokens

Definition at line 1714 of file [structs.h](#).

3.12.2.55 endoftokens

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

End of the actual tokens

Definition at line 1715 of file [structs.h](#).

3.12.2.56 tokenarglevel

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

[D] Keeps track of function arguments

Definition at line 1716 of file [structs.h](#).

3.12.2.57 modinverses

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

Definition at line 1717 of file [structs.h](#).

3.12.2.58 Fortran90Kind

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

Definition at line 1718 of file [structs.h](#).

3.12.2.59 MultiBracketBuf

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

Definition at line 1719 of file [structs.h](#).

3.12.2.60 extrasym

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

Definition at line 1720 of file [structs.h](#).

3.12.2.61 doloopstack

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

Definition at line 1721 of file [structs.h](#).

3.12.2.62 doloopnest

WORD* C_const::doloopnest

Definition at line 1722 of file [structs.h](#).

3.12.2.63 CheckpointRunAfter

char* C_const::CheckpointRunAfter

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

Definition at line 1723 of file [structs.h](#).

3.12.2.64 CheckpointRunBefore

char* C_const::CheckpointRunBefore

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

Definition at line 1725 of file [structs.h](#).

3.12.2.65 IfSumCheck

WORD* C_const::IfSumCheck

[D] Keeps track of if-nesting

Definition at line 1727 of file [structs.h](#).

3.12.2.66 CommuteInSet

WORD* C_const::CommuteInSet

Definition at line 1728 of file [structs.h](#).

3.12.2.67 TestValue

UBYTE* C_const::TestValue

Definition at line 1729 of file [structs.h](#).

3.12.2.68 argstack

LONG C_const::argstack[MAXNEST]

Definition at line 1737 of file [structs.h](#).

3.12.2.69 insidestack

LONG C_const::insidestack[MAXNEST]

Definition at line 1738 of file [structs.h](#).

3.12.2.70 inexprstack

LONG C_const::inexprstack[MAXNEST]

Definition at line 1739 of file [structs.h](#).

3.12.2.71 iBufferSize

LONG C_const::iBufferSize

Definition at line 1740 of file [structs.h](#).

3.12.2.72 TransName

LONG C_const::TransName

Definition at line 1741 of file [structs.h](#).

3.12.2.73 ProcessBucketSize

LONG C_const::ProcessBucketSize

Definition at line 1743 of file [structs.h](#).

3.12.2.74 mProcessBucketSize

LONG C_const::mProcessBucketSize

Definition at line 1744 of file [structs.h](#).

3.12.2.75 CModule

LONG C_const::CModule

Definition at line 1745 of file [structs.h](#).

3.12.2.76 ThreadBucketSize

LONG C_const::ThreadBucketSize

Definition at line 1746 of file [structs.h](#).

3.12.2.77 CheckpointStamp

```
LONG C_const::CheckpointStamp
```

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

Definition at line 1747 of file [structs.h](#).

3.12.2.78 CheckpointInterval

```
LONG C_const::CheckpointInterval
```

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

Definition at line 1748 of file [structs.h](#).

3.12.2.79 cbufnum

```
int C_const::cbufnum
```

Current compiler buffer

Definition at line 1756 of file [structs.h](#).

3.12.2.80 AutoDeclareFlag

```
int C_const::AutoDeclareFlag
```

Definition at line 1757 of file [structs.h](#).

3.12.2.81 NoShowInput

```
int C_const::NoShowInput
```

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

Definition at line 1759 of file [structs.h](#).

3.12.2.82 ShortStats

```
int C_const::ShortStats
```

Definition at line 1760 of file [structs.h](#).

3.12.2.83 compiletype

```
int C_const::compiletype
```

Definition at line 1761 of file [structs.h](#).

3.12.2.84 firstconstindex

```
int C_const::firstconstindex
```

Definition at line 1762 of file [structs.h](#).

3.12.2.85 insidefirst

```
int C_const::insidefirst
```

Definition at line 1763 of file [structs.h](#).

3.12.2.86 minsidefirst

```
int C_const::minsidefirst
```

Definition at line 1764 of file [structs.h](#).

3.12.2.87 wildflag

```
int C_const::wildflag
```

Definition at line 1765 of file [structs.h](#).

3.12.2.88 NumLabels

```
int C_const::NumLabels
```

Definition at line 1766 of file [structs.h](#).

3.12.2.89 MaxLabels

```
int C_const::MaxLabels
```

Definition at line 1767 of file [structs.h](#).

3.12.2.90 IDefDim

```
int C_const::lDefDim
```

Definition at line 1768 of file [structs.h](#).

3.12.2.91 IDefDim4

```
int C_const::lDefDim4
```

Definition at line 1769 of file [structs.h](#).

3.12.2.92 NumWildcardNames

```
int C_const::NumWildcardNames
```

Definition at line 1770 of file [structs.h](#).

3.12.2.93 WildcardBufferSize

```
int C_const::WildcardBufferSize
```

Definition at line 1771 of file [structs.h](#).

3.12.2.94 MaxIf

```
int C_const::MaxIf
```

Definition at line 1772 of file [structs.h](#).

3.12.2.95 NumStreams

```
int C_const::NumStreams
```

Definition at line 1773 of file [structs.h](#).

3.12.2.96 MaxNumStreams

```
int C_const::MaxNumStreams
```

Definition at line 1774 of file [structs.h](#).

3.12.2.97 firstctypemessage

```
int C_const::firstctypemessage
```

Definition at line 1775 of file [structs.h](#).

3.12.2.98 tablecheck

```
int C_const::tablecheck
```

Definition at line 1776 of file [structs.h](#).

3.12.2.99 idoption

```
int C_const::idoption
```

Definition at line 1777 of file [structs.h](#).

3.12.2.100 BottomLevel

```
int C_const::BottomLevel
```

Definition at line 1778 of file [structs.h](#).

3.12.2.101 CompileLevel

```
int C_const::CompileLevel
```

Definition at line 1779 of file [structs.h](#).

3.12.2.102 TokensWriteFlag

```
int C_const::TokensWriteFlag
```

Definition at line 1780 of file [structs.h](#).

3.12.2.103 UnsureDollarMode

```
int C_const::UnsureDollarMode
```

Definition at line 1781 of file [structs.h](#).

3.12.2.104 outsidefun

```
int C_const::outsidefun
```

Definition at line 1782 of file [structs.h](#).

3.12.2.105 funpowers

```
int C_const::funpowers
```

Definition at line 1783 of file [structs.h](#).

3.12.2.106 WarnFlag

```
int C_const::WarnFlag
```

Definition at line 1784 of file [structs.h](#).

3.12.2.107 StatsFlag

```
int C_const::StatsFlag
```

Definition at line 1785 of file [structs.h](#).

3.12.2.108 NamesFlag

```
int C_const::NamesFlag
```

Definition at line 1786 of file [structs.h](#).

3.12.2.109 CodesFlag

```
int C_const::CodesFlag
```

Definition at line 1787 of file [structs.h](#).

3.12.2.110 SetupFlag

```
int C_const::SetupFlag
```

Definition at line 1788 of file [structs.h](#).

3.12.2.111 SortType

```
int C_const::SortType
```

Definition at line 1789 of file [structs.h](#).

3.12.2.112 ISortType

```
int C_const::ISortType
```

Definition at line 1790 of file [structs.h](#).

3.12.2.113 ThreadStats

```
int C_const::ThreadStats
```

Definition at line 1791 of file [structs.h](#).

3.12.2.114 FinalStats

```
int C_const::FinalStats
```

Definition at line 1792 of file [structs.h](#).

3.12.2.115 OldParallelStats

```
int C_const::OldParallelStats
```

Definition at line 1793 of file [structs.h](#).

3.12.2.116 ThreadsFlag

```
int C_const::ThreadsFlag
```

Definition at line 1794 of file [structs.h](#).

3.12.2.117 ThreadBalancing

```
int C_const::ThreadBalancing
```

Definition at line 1795 of file [structs.h](#).

3.12.2.118 ThreadSortFileSynch

```
int C_const::ThreadSortFileSynch
```

Definition at line 1796 of file [structs.h](#).

3.12.2.119 ProcessStats

```
int C_const::ProcessStats
```

Definition at line 1797 of file [structs.h](#).

3.12.2.120 BracketNormalize

```
int C_const::BracketNormalize
```

Definition at line 1798 of file [structs.h](#).

3.12.2.121 maxtermlevel

```
int C_const::maxtermlevel
```

Definition at line 1799 of file [structs.h](#).

3.12.2.122 dumnumflag

```
int C_const::dumnumflag
```

Definition at line 1800 of file [structs.h](#).

3.12.2.123 bracketindexflag

```
int C_const::bracketindexflag
```

Definition at line 1801 of file [structs.h](#).

3.12.2.124 parallelflag

```
int C_const::parallelflag
```

Definition at line [1802](#) of file [structs.h](#).

3.12.2.125 mparallelflag

```
int C_const::mparallelflag
```

Definition at line [1803](#) of file [structs.h](#).

3.12.2.126 inparallelflag

```
int C_const::inparallelflag
```

Definition at line [1804](#) of file [structs.h](#).

3.12.2.127 partodoflag

```
int C_const::partodoflag
```

Definition at line [1805](#) of file [structs.h](#).

3.12.2.128 properorderflag

```
int C_const::properorderflag
```

Definition at line [1806](#) of file [structs.h](#).

3.12.2.129 vetofilling

```
int C_const::vetofilling
```

Definition at line [1807](#) of file [structs.h](#).

3.12.2.130 tablefilling

```
int C_const::tablefilling
```

Definition at line [1808](#) of file [structs.h](#).

3.12.2.131 vetotablebasefill

```
int C_const::vetotablebasefill
```

Definition at line [1809](#) of file [structs.h](#).

3.12.2.132 exprfillwarning

```
int C_const::exprfillwarning
```

Definition at line 1810 of file [structs.h](#).

3.12.2.133 lhdollarflag

```
int C_const::lhdollarflag
```

Definition at line 1811 of file [structs.h](#).

3.12.2.134 NoCompress

```
int C_const::NoCompress
```

Definition at line 1812 of file [structs.h](#).

3.12.2.135 IsFortran90

```
int C_const::IsFortran90
```

Definition at line 1813 of file [structs.h](#).

3.12.2.136 MultiBracketLevels

```
int C_const::MultiBracketLevels
```

Definition at line 1814 of file [structs.h](#).

3.12.2.137 topolynomialflag

```
int C_const::topolynomialflag
```

Definition at line 1815 of file [structs.h](#).

3.12.2.138 ffbufnum

```
int C_const::ffbufnum
```

Definition at line 1816 of file [structs.h](#).

3.12.2.139 OldFactArgFlag

```
int C_const::OldFactArgFlag
```

Definition at line 1817 of file [structs.h](#).

3.12.2.140 MemDebugFlag

```
int C_const::MemDebugFlag
```

Definition at line 1818 of file [structs.h](#).

3.12.2.141 OldGCDflag

```
int C_const::OldGCDflag
```

Definition at line 1819 of file [structs.h](#).

3.12.2.142 WTimeStatsFlag

```
int C_const::WTimeStatsFlag
```

Definition at line 1820 of file [structs.h](#).

3.12.2.143 SortReallocateFlag

```
int C_const::SortReallocateFlag
```

Definition at line 1821 of file [structs.h](#).

3.12.2.144 PrintBacktraceFlag

```
int C_const::PrintBacktraceFlag
```

Definition at line 1825 of file [structs.h](#).

3.12.2.145 FlintPolyFlag

```
int C_const::FlintPolyFlag
```

Definition at line 1826 of file [structs.h](#).

3.12.2.146 HumanStatsFlag

```
int C_const::HumanStatsFlag
```

Definition at line 1827 of file [structs.h](#).

3.12.2.147 doloopstacksize

```
int C_const::doloopstacksize
```

Definition at line 1828 of file [structs.h](#).

3.12.2.148 dolooplevel

```
int C_const::dolooplevel
```

Definition at line 1829 of file [structs.h](#).

3.12.2.149 CheckpointFlag

```
int C_const::CheckpointFlag
```

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

Definition at line 1830 of file [structs.h](#).

3.12.2.150 SizeCommuteInSet

```
int C_const::SizeCommuteInSet
```

Definition at line 1834 of file [structs.h](#).

3.12.2.151 origin

```
int C_const::origin
```

Definition at line 1839 of file [structs.h](#).

3.12.2.152 vectorlikeLHS

```
int C_const::vectorlikeLHS
```

Definition at line 1840 of file [structs.h](#).

3.12.2.153 nummodels

```
int C_const::nummodels
```

Definition at line 1841 of file [structs.h](#).

3.12.2.154 modelspace

```
int C_const::modelspace
```

Definition at line 1842 of file [structs.h](#).

3.12.2.155 ModelLevel

```
int C_const::ModelLevel
```

Definition at line 1843 of file [structs.h](#).

3.12.2.156 ShortStatsMax

```
int C_const::ShortStatsMax
```

Definition at line 1844 of file [structs.h](#).

3.12.2.157 InnerTest

```
int C_const::InnerTest
```

Definition at line 1845 of file [structs.h](#).

3.12.2.158 argsumcheck

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

Definition at line 1846 of file [structs.h](#).

3.12.2.159 insidesumcheck

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

Definition at line 1847 of file [structs.h](#).

3.12.2.160 inexprsumcheck

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

Definition at line 1848 of file [structs.h](#).

3.12.2.161 RepSumCheck

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

Definition at line 1849 of file [structs.h](#).

3.12.2.162 lUniTrace

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

Definition at line 1850 of file [structs.h](#).

3.12.2.163 RepLevel

WORD C_const::RepLevel

Definition at line 1851 of file [structs.h](#).

3.12.2.164 arglevel

WORD C_const::arglevel

Definition at line 1852 of file [structs.h](#).

3.12.2.165 insidelevel

WORD C_const::insidelevel

Definition at line 1853 of file [structs.h](#).

3.12.2.166 inexprlevel

WORD C_const::inexprlevel

Definition at line 1854 of file [structs.h](#).

3.12.2.167 termlevel

WORD C_const::termlevel

Definition at line 1855 of file [structs.h](#).

3.12.2.168 MustTestTable

WORD C_const::MustTestTable

Definition at line 1856 of file [structs.h](#).

3.12.2.169 DumNum

WORD C_const::DumNum

Definition at line 1857 of file [structs.h](#).

3.12.2.170 ncmmod

WORD C_const::ncmod

Definition at line 1858 of file [structs.h](#).

3.12.2.171 npowmod

WORD C_const::npowmod

Definition at line 1859 of file [structs.h](#).

3.12.2.172 modmode

WORD C_const::modmode

Definition at line 1860 of file [structs.h](#).

3.12.2.173 nhalfmod

WORD C_const::nhalfmod

Definition at line 1861 of file [structs.h](#).

3.12.2.174 DirtPow

WORD C_const::DirtPow

Definition at line 1862 of file [structs.h](#).

3.12.2.175 lUnitTrace

WORD C_const::lUnitTrace

Definition at line 1863 of file [structs.h](#).

3.12.2.176 NwildC

WORD C_const::NwildC

Definition at line 1864 of file [structs.h](#).

3.12.2.177 ComDefer

WORD C_const::ComDefer

Definition at line 1865 of file [structs.h](#).

3.12.2.178 CollectFun

WORD C_const::CollectFun

Definition at line 1866 of file [structs.h](#).

3.12.2.179 AltCollectFun

WORD C_const::AltCollectFun

Definition at line 1867 of file [structs.h](#).

3.12.2.180 OutputMode

WORD C_const::OutputMode

Definition at line 1868 of file [structs.h](#).

3.12.2.181 Cnumpows

WORD C_const::Cnumpows

Definition at line 1869 of file [structs.h](#).

3.12.2.182 OutputSpaces

WORD C_const::OutputSpaces

Definition at line 1870 of file [structs.h](#).

3.12.2.183 OutNumberType

WORD C_const::OutNumberType

Definition at line 1871 of file [structs.h](#).

3.12.2.184 DidClean

WORD C_const::DidClean

Definition at line 1872 of file [structs.h](#).

3.12.2.185 IfLevel

WORD C_const::IfLevel

Definition at line 1873 of file [structs.h](#).

3.12.2.186 WhileLevel

WORD C_const::WhileLevel

Definition at line 1874 of file [structs.h](#).

3.12.2.187 SwitchLevel

WORD C_const::SwitchLevel

Definition at line 1875 of file [structs.h](#).

3.12.2.188 SwitchInArray

WORD C_const::SwitchInArray

Definition at line 1876 of file [structs.h](#).

3.12.2.189 MaxSwitch

WORD C_const::MaxSwitch

Definition at line 1877 of file [structs.h](#).

3.12.2.190 LogHandle

WORD C_const::LogHandle

Definition at line 1878 of file [structs.h](#).

3.12.2.191 LineLength

WORD C_const::LineLength

Definition at line 1879 of file [structs.h](#).

3.12.2.192 StoreHandle

WORD C_const::StoreHandle

Definition at line 1880 of file [structs.h](#).

3.12.2.193 HideLevel

WORD C_const::HideLevel

Definition at line 1881 of file [structs.h](#).

3.12.2.194 lPolyFun

WORD C_const::lPolyFun

Definition at line 1882 of file [structs.h](#).

3.12.2.195 IPolyFunInv

WORD C_const::lPolyFunInv

Definition at line 1883 of file [structs.h](#).

3.12.2.196 IPolyFunType

WORD C_const::lPolyFunType

Definition at line 1884 of file [structs.h](#).

3.12.2.197 IPolyFunExp

WORD C_const::lPolyFunExp

Definition at line 1885 of file [structs.h](#).

3.12.2.198 IPolyFunVar

WORD C_const::lPolyFunVar

Definition at line 1886 of file [structs.h](#).

3.12.2.199 IPolyFunPow

WORD C_const::lPolyFunPow

Definition at line 1887 of file [structs.h](#).

3.12.2.200 SymChangeFlag

WORD C_const::SymChangeFlag

Definition at line 1888 of file [structs.h](#).

3.12.2.201 CollectPercentage

WORD C_const::CollectPercentage

Definition at line 1889 of file [structs.h](#).

3.12.2.202 extrasymbols

WORD C_const::extrasymbols

Definition at line 1890 of file [structs.h](#).

3.12.2.203 PolyRatFunChanged

```
WORD C_const::PolyRatFunChanged
```

Definition at line [1891](#) of file [structs.h](#).

3.12.2.204 ToBeInFactors

```
WORD C_const::ToBeInFactors
```

Definition at line [1892](#) of file [structs.h](#).

3.12.2.205 Commercial

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

Definition at line [1896](#) of file [structs.h](#).

3.12.2.206 debugFlags

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

Definition at line [1897](#) of file [structs.h](#).

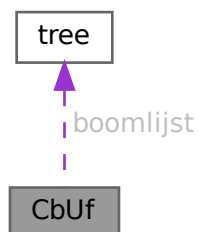
The documentation for this struct was generated from the following file:

- [structs.h](#)

3.13 CbUf Struct Reference

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

Collaboration diagram for CbUf:



Data Fields

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

3.13.1 Detailed Description

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

Definition at line [971](#) of file [structs.h](#).

3.13.2 Field Documentation

3.13.2.1 Buffer

WORD* CbUf::Buffer

[D] Size in BufferSize

Definition at line [972](#) of file [structs.h](#).

Referenced by [CleanupArgCache\(\)](#), [clearcbuf\(\)](#), [DoubleCbuffer\(\)](#), [finishcbuf\(\)](#), [inicbufs\(\)](#), [PF_BroadcastCBuf\(\)](#), and [Processor\(\)](#).

3.13.2.2 Top

WORD* CbUf::Top

pointer to the end of the Buffer memory

Definition at line [973](#) of file [structs.h](#).

Referenced by [AddNtoC\(\)](#), [AddNtoL\(\)](#), [DoubleCbuffer\(\)](#), [finishcbuf\(\)](#), [inicbufs\(\)](#), and [PF_BroadcastCBuf\(\)](#).

3.13.2.3 Pointer

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

pointer into the Buffer memory

Definition at line 974 of file [structs.h](#).

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

3.13.2.4 lhs

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

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

Definition at line 975 of file [structs.h](#).

Referenced by [AddLHS\(\)](#), [DoubleCbuffer\(\)](#), [finishcbuf\(\)](#), [Generator\(\)](#), [inicbufs\(\)](#), [PF_BroadcastCBuf\(\)](#), [Processor\(\)](#), and [TestMatch\(\)](#).

3.13.2.5 rhs

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

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

Definition at line 976 of file [structs.h](#).

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

3.13.2.6 CanCommu

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

points into rhs memory behind WORD* area.

Definition at line 977 of file [structs.h](#).

Referenced by [AddRHS\(\)](#), [finishcbuf\(\)](#), [inicbufs\(\)](#), [IniFbuffer\(\)](#), and [PF_BroadcastCBuf\(\)](#).

3.13.2.7 NumTerms

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

points into rhs memory behind CanCommu area

Definition at line 978 of file [structs.h](#).

Referenced by [AddRHS\(\)](#), [finishcbuf\(\)](#), [inicbufs\(\)](#), [IniFbuffer\(\)](#), and [PF_BroadcastCBuf\(\)](#).

3.13.2.8 numdum

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

points into rhs memory behind NumTerms

Definition at line 979 of file [structs.h](#).

Referenced by [AddRHS\(\)](#), [inicbufs\(\)](#), [IniFbuffer\(\)](#), and [PF_BroadcastCBuf\(\)](#).

3.13.2.9 dimension

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

points into rhs memory behind numdum

Definition at line 980 of file [structs.h](#).

Referenced by [AddRHS\(\)](#), [inicbufs\(\)](#), [IniFbuffer\(\)](#), and [PF_BroadcastCBuf\(\)](#).

3.13.2.10 boomlijst

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

[D] Number elements in MaxTreeSize

Definition at line 981 of file [structs.h](#).

Referenced by [CleanupArgCache\(\)](#), [clearcbuf\(\)](#), [finishcbuf\(\)](#), [inicbufs\(\)](#), [IniFbuffer\(\)](#), and [PF_BroadcastCBuf\(\)](#).

3.13.2.11 BufferSize

```
LONG CbUf::BufferSize
```

Number of allocated WORD's in Buffer

Definition at line 982 of file [structs.h](#).

Referenced by [DoubleCbuffer\(\)](#), [finishcbuf\(\)](#), [inicbufs\(\)](#), and [PF_BroadcastCBuf\(\)](#).

3.13.2.12 numlhs

```
int CbUf::numlhs
```

Definition at line 983 of file [structs.h](#).

3.13.2.13 numrhs

```
int CbUf::numrhs
```

Definition at line 984 of file [structs.h](#).

3.13.2.14 maxlhs

```
int CbUf::maxlhs
```

Definition at line 985 of file [structs.h](#).

3.13.2.15 maxrhs

```
int CbUf::maxrhs
```

Definition at line 986 of file [structs.h](#).

3.13.2.16 mnumlhs

```
int CbUf::mnumlhs
```

Definition at line 987 of file [structs.h](#).

3.13.2.17 mnumrhs

```
int CbUf::mnumrhs
```

Definition at line 988 of file [structs.h](#).

3.13.2.18 numtree

```
int CbUf::numtree
```

Definition at line 989 of file [structs.h](#).

3.13.2.19 rootnum

```
int CbUf::rootnum
```

Definition at line 990 of file [structs.h](#).

3.13.2.20 MaxTreeSize

```
int CbUf::MaxTreeSize
```

Definition at line 991 of file [structs.h](#).

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

- [structs.h](#)

3.14 ChAnNeL Struct Reference

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

Data Fields

- char * [name](#)
- int [handle](#)

3.14.1 Detailed Description

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

Definition at line [1001](#) of file [structs.h](#).

3.14.2 Field Documentation

3.14.2.1 name

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

[D] Name of the channel

Definition at line [1002](#) of file [structs.h](#).

3.14.2.2 handle

```
int ChAnNeL::handle
```

File handle

Definition at line [1003](#) of file [structs.h](#).

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

- [structs.h](#)

3.15 COST Struct Reference

Data Fields

- LONG [add](#)
- LONG [mul](#)
- LONG [div](#)
- LONG [pow](#)

3.15.1 Detailed Description

Definition at line [1261](#) of file [structs.h](#).

3.15.2 Field Documentation

3.15.2.1 add

```
LONG COST::add
```

Definition at line [1262](#) of file [structs.h](#).

3.15.2.2 mul

```
LONG COST::mul
```

Definition at line [1263](#) of file [structs.h](#).

3.15.2.3 div

```
LONG COST::div
```

Definition at line [1264](#) of file [structs.h](#).

3.15.2.4 pow

```
LONG COST::pow
```

Definition at line [1265](#) of file [structs.h](#).

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

- [structs.h](#)

3.16 CSEEq Struct Reference

Public Member Functions

- bool [operator\(\)](#) (const vector< WORD > &lhs, const vector< WORD > &rhs) const

3.16.1 Detailed Description

Definition at line [1059](#) of file [optimize.cc](#).

3.16.2 Member Function Documentation

3.16.2.1 operator()

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

Definition at line 1060 of file [optimize.cc](#).

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

- [optimize.cc](#)

3.17 CSEHash Struct Reference

Public Member Functions

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

3.17.1 Detailed Description

Definition at line 1053 of file [optimize.cc](#).

3.17.2 Member Function Documentation

3.17.2.1 operator()

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

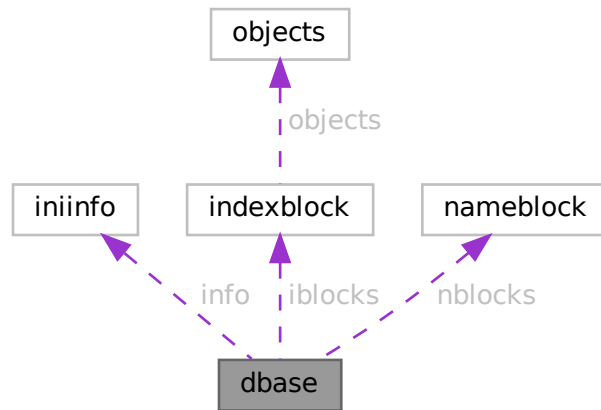
Definition at line 1054 of file [optimize.cc](#).

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

- [optimize.cc](#)

3.18 dbase Struct Reference

Collaboration diagram for dbase:



Data Fields

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

3.18.1 Detailed Description

Definition at line 123 of file `minos.h`.

3.18.2 Field Documentation

3.18.2.1 `info`

`INIINFO` `dbase::info`

Definition at line 124 of file `minos.h`.

3.18.2.2 mode

MLONG dbase::mode

Definition at line 125 of file [minos.h](#).

3.18.2.3 tablenamessize

MLONG dbase::tablenamessize

Definition at line 126 of file [minos.h](#).

3.18.2.4 topnumber

MLONG dbase::topnumber

Definition at line 127 of file [minos.h](#).

3.18.2.5 tablenameefill

MLONG dbase::tablenameefill

Definition at line 128 of file [minos.h](#).

3.18.2.6 rwmode

MLONG dbase::rwmode

Definition at line 129 of file [minos.h](#).

3.18.2.7 iblocks

INDEXBLOCK** dbase::iblocks

Definition at line 130 of file [minos.h](#).

3.18.2.8 nblocks

NAMESBLOCK** dbase::nblocks

Definition at line 131 of file [minos.h](#).

3.18.2.9 handle

FILE* dbase::handle

Definition at line 132 of file [minos.h](#).

3.18.2.10 name

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

Definition at line 133 of file [minos.h](#).

3.18.2.11 fullname

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

Definition at line 134 of file [minos.h](#).

3.18.2.12 tablenames

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

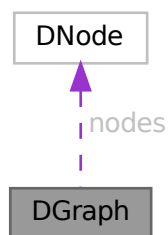
Definition at line 135 of file [minos.h](#).

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

- [minos.h](#)

3.19 DGraph Struct Reference

Collaboration diagram for DGraph:



Data Fields

- int [nnodes](#)
- int [nedges](#)
- [DNode](#) * [nodes](#)
- [DEdge](#) * [edges](#)

3.19.1 Detailed Description

Definition at line 255 of file [grccparam.h](#).

3.19.2 Field Documentation

3.19.2.1 nnodes

```
int DGraph::nnodes
```

Definition at line 256 of file [grccparam.h](#).

3.19.2.2 nedges

```
int DGraph::nedges
```

Definition at line 257 of file [grccparam.h](#).

3.19.2.3 nodes

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

Definition at line 258 of file [grccparam.h](#).

3.19.2.4 edges

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

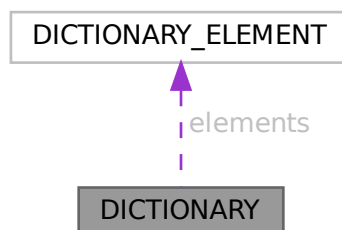
Definition at line 259 of file [grccparam.h](#).

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

- [grccparam.h](#)

3.20 DICTIONARY Struct Reference

Collaboration diagram for DICTIONARY:



Data Fields

- [DICTIONARY_ELEMENT](#) ** [elements](#)
- UBYTE * [name](#)
- int [sizeelements](#)
- int [numelements](#)
- int [numbers](#)
- int [variables](#)
- int [characters](#)
- int [funwith](#)
- int [gnumelements](#)
- int [ranges](#)

3.20.1 Detailed Description

Definition at line [1326](#) of file [structs.h](#).

3.20.2 Field Documentation

3.20.2.1 elements

```
DICTIONARY\_ELEMENT** DICTIONARY::elements
```

Definition at line [1327](#) of file [structs.h](#).

3.20.2.2 name

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

Definition at line [1328](#) of file [structs.h](#).

3.20.2.3 sizeelements

```
int DICTIONARY::sizeelements
```

Definition at line [1329](#) of file [structs.h](#).

3.20.2.4 numelements

```
int DICTIONARY::numelements
```

Definition at line [1330](#) of file [structs.h](#).

3.20.2.5 numbers

```
int DICTIONARY::numbers
```

Definition at line [1331](#) of file [structs.h](#).

3.20.2.6 variables

```
int DICTIONARY::variables
```

Definition at line 1332 of file [structs.h](#).

3.20.2.7 characters

```
int DICTIONARY::characters
```

Definition at line 1333 of file [structs.h](#).

3.20.2.8 funwith

```
int DICTIONARY::funwith
```

Definition at line 1334 of file [structs.h](#).

3.20.2.9 gnumelements

```
int DICTIONARY::gnumelements
```

Definition at line 1335 of file [structs.h](#).

3.20.2.10 ranges

```
int DICTIONARY::ranges
```

Definition at line 1336 of file [structs.h](#).

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

- [structs.h](#)

3.21 DICTIONARY_ELEMENT Struct Reference

Data Fields

- WORD * [lhs](#)
- WORD * [rhs](#)
- int [type](#)
- int [size](#)

3.21.1 Detailed Description

Definition at line 1319 of file [structs.h](#).

3.21.2 Field Documentation

3.21.2.1 lhs

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

Definition at line 1320 of file [structs.h](#).

3.21.2.2 rhs

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

Definition at line 1321 of file [structs.h](#).

3.21.2.3 type

```
int DICTIONARY_ELEMENT::type
```

Definition at line 1322 of file [structs.h](#).

3.21.2.4 size

```
int DICTIONARY_ELEMENT::size
```

Definition at line 1323 of file [structs.h](#).

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

- [structs.h](#)

3.22 DiStRiBuTe Struct Reference

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

Data Fields

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

3.22.1 Detailed Description

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

Definition at line [1080](#) of file [structs.h](#).

3.22.2 Field Documentation

3.22.2.1 obj1

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

Definition at line [1081](#) of file [structs.h](#).

3.22.2.2 obj2

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

Definition at line [1082](#) of file [structs.h](#).

3.22.2.3 out

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

Definition at line [1083](#) of file [structs.h](#).

3.22.2.4 sign

```
WORD DiStRiBuTe::sign
```

Definition at line [1084](#) of file [structs.h](#).

3.22.2.5 n1

```
WORD DiStRiBuTe::n1
```

Definition at line [1085](#) of file [structs.h](#).

3.22.2.6 n2

```
WORD DiStRiBuTe::n2
```

Definition at line [1086](#) of file [structs.h](#).

3.22.2.7 n

```
WORD DiStRiBuTe::n
```

Definition at line 1087 of file [structs.h](#).

3.22.2.8 cycle

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

Definition at line 1088 of file [structs.h](#).

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

- [structs.h](#)

3.23 DNode Struct Reference

Data Fields

- int [extloop](#)
- int [intrct](#)

3.23.1 Detailed Description

Definition at line 248 of file [grccparam.h](#).

3.23.2 Field Documentation

3.23.2.1 extloop

```
int DNode::extloop
```

Definition at line 249 of file [grccparam.h](#).

3.23.2.2 intrct

```
int DNode::intrct
```

Definition at line 250 of file [grccparam.h](#).

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

- [grccparam.h](#)

3.24 dollar_buf Struct Reference

3.24.1 Detailed Description

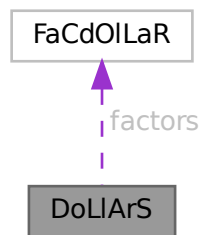
Definition at line 2481 of file [parallel.c](#).

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

- [parallel.c](#)

3.25 DoLIaRS Struct Reference

Collaboration diagram for DoLIaRS:



Data Fields

- WORD * [where](#)
- [FACDOLLAR](#) * [factors](#)
- LONG [size](#)
- LONG [name](#)
- WORD [type](#)
- WORD [node](#)
- WORD [index](#)
- WORD [zero](#)
- WORD [numdummies](#)
- WORD [nfactors](#)

3.25.1 Detailed Description

Definition at line 541 of file [structs.h](#).

3.25.2 Field Documentation

3.25.2.1 where

`WORD* DoLLArS::where`

Definition at line 542 of file [structs.h](#).

3.25.2.2 factors

`FACDOLLAR* DoLLArS::factors`

Definition at line 543 of file [structs.h](#).

3.25.2.3 size

`LONG DoLLArS::size`

Definition at line 548 of file [structs.h](#).

3.25.2.4 name

`LONG DoLLArS::name`

Definition at line 549 of file [structs.h](#).

3.25.2.5 type

`WORD DoLLArS::type`

Definition at line 550 of file [structs.h](#).

3.25.2.6 node

`WORD DoLLArS::node`

Definition at line 551 of file [structs.h](#).

3.25.2.7 index

`WORD DoLLArS::index`

Definition at line 552 of file [structs.h](#).

3.25.2.8 zero

```
WORD DoLlArS::zero
```

Definition at line 553 of file [structs.h](#).

3.25.2.9 numdummies

```
WORD DoLlArS::numdummies
```

Definition at line 554 of file [structs.h](#).

3.25.2.10 nfactors

```
WORD DoLlArS::nfactors
```

Definition at line 555 of file [structs.h](#).

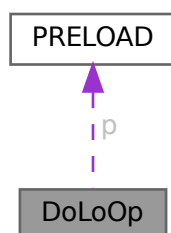
The documentation for this struct was generated from the following file:

- [structs.h](#)

3.26 DoLoOp Struct Reference

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

Collaboration diagram for DoLoOp:



Data Fields

- [PRELOAD](#) [p](#)
- UBYTE * [name](#)
- UBYTE * [vars](#)
- UBYTE * [contents](#)
- UBYTE * [dollaname](#)
- LONG [startlinenumber](#)
- LONG [firstnum](#)
- LONG [lastnum](#)
- LONG [incnum](#)
- int [type](#)
- int [NoShowInput](#)
- int [errorsinloop](#)
- int [firstloopcall](#)
- WORD [firstdollar](#)
- WORD [lastdollar](#)
- WORD [incdollar](#)
- WORD [NumPreTypes](#)
- WORD [PrefLevel](#)
- WORD [PreSwitchLevel](#)

3.26.1 Detailed Description

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

Definition at line [877](#) of file [structs.h](#).

3.26.2 Field Documentation

3.26.2.1 [p](#)

[PRELOAD](#) DoLoOp::p

size, name and buffer

Definition at line [878](#) of file [structs.h](#).

3.26.2.2 [name](#)

UBYTE* DoLoOp::name

pointer into [PRELOAD](#) buffer

Definition at line [879](#) of file [structs.h](#).

3.26.2.3 vars

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

Definition at line 880 of file [structs.h](#).

3.26.2.4 contents

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

Definition at line 881 of file [structs.h](#).

3.26.2.5 dollaname

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

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

Definition at line 882 of file [structs.h](#).

3.26.2.6 startlinenumber

```
LONG DoLoOp::startlinenumber
```

Definition at line 883 of file [structs.h](#).

3.26.2.7 firstnum

```
LONG DoLoOp::firstnum
```

Definition at line 884 of file [structs.h](#).

3.26.2.8 lastnum

```
LONG DoLoOp::lastnum
```

Definition at line 885 of file [structs.h](#).

3.26.2.9 incnum

```
LONG DoLoOp::incnum
```

Definition at line 886 of file [structs.h](#).

3.26.2.10 type

```
int DoLoOp::type
```

Definition at line 887 of file [structs.h](#).

3.26.2.11 NoShowInput

```
int DoLoOp::NoShowInput
```

Definition at line 888 of file [structs.h](#).

3.26.2.12 errorsinloop

```
int DoLoOp::errorsinloop
```

Definition at line 889 of file [structs.h](#).

3.26.2.13 firstloopcall

```
int DoLoOp::firstloopcall
```

Definition at line 890 of file [structs.h](#).

3.26.2.14 firstdollar

```
WORD DoLoOp::firstdollar
```

Definition at line 891 of file [structs.h](#).

3.26.2.15 lastdollar

```
WORD DoLoOp::lastdollar
```

Definition at line 892 of file [structs.h](#).

3.26.2.16 incdollar

```
WORD DoLoOp::incdollar
```

Definition at line 893 of file [structs.h](#).

3.26.2.17 NumPreTypes

```
WORD DoLoOp::NumPreTypes
```

Definition at line 894 of file [structs.h](#).

3.26.2.18 PreIfLevel

WORD DoLoOp::PreIfLevel

Definition at line 895 of file [structs.h](#).

3.26.2.19 PreSwitchLevel

WORD DoLoOp::PreSwitchLevel

Definition at line 896 of file [structs.h](#).

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

- [structs.h](#)

3.27 DuBiOuS Struct Reference

Data Fields

- LONG [name](#)
- WORD [node](#)
- WORD [dummy](#)

3.27.1 Detailed Description

Definition at line 528 of file [structs.h](#).

3.27.2 Field Documentation

3.27.2.1 name

LONG DuBiOuS::name

Definition at line 529 of file [structs.h](#).

3.27.2.2 node

WORD DuBiOuS::node

Definition at line 530 of file [structs.h](#).

3.27.2.3 dummy

WORD DuBiOuS::dummy

Definition at line 531 of file [structs.h](#).

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

- [structs.h](#)

3.28 ECand Class Reference

Public Member Functions

- [ECand](#) (int dt, int nplist, int *plst)
- void [prECand](#) (const char *msg)

Data Fields

- Bool [det](#)
- int [nplist](#)
- int [plist](#) [GRCC_MAXMPARTICLES2]

3.28.1 Detailed Description

Definition at line 1124 of file [grcc.h](#).

3.28.2 Constructor & Destructor Documentation

3.28.2.1 ECand()

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

Definition at line 8186 of file [grcc.cc](#).

3.28.2.2 ~ECand()

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

Definition at line 8206 of file [grcc.cc](#).

Public Member Functions

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

Data Fields

- [EGraph](#) * [egraph](#)
- int [id](#)
- int [ext](#)
- int [ptcl](#)
- int [deleted](#)
- int [nodes](#) [2]
- int [nlegs](#) [2]
- int * [emom](#)
- int * [lmom](#)
- int * [extMom](#)
- ULong [momset](#)
- int [momdir](#)
- Bool [cut](#)
- int [visited](#)
- int [conid](#)
- int [edtype](#)
- int [opicomp](#)
- int [dir](#)
- int [DUMMY_PADDING](#)

3.29.1 Detailed Description

Definition at line [562](#) of file [grcc.h](#).

3.29.2 Constructor & Destructor Documentation

3.29.2.1 EEdge() [1/2]

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

Definition at line [6143](#) of file [grcc.cc](#).

3.29.2.2 EEdge() [2/2]

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

Definition at line [6165](#) of file [grcc.cc](#).

3.29.2.3 ~EEdge()

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

Definition at line 6184 of file [grcc.cc](#).

3.29.3 Member Function Documentation

3.29.3.1 copy()

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

Definition at line 6195 of file [grcc.cc](#).

3.29.3.2 setId()

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

Definition at line 6207 of file [grcc.cc](#).

3.29.3.3 print()

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

Definition at line 6243 of file [grcc.cc](#).

3.29.3.4 setType()

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

Definition at line 6214 of file [grcc.cc](#).

3.29.3.5 setLMom()

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

Definition at line 6237 of file [grcc.cc](#).

3.29.3.6 setEMom()

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

Definition at line [6225](#) of file [grcc.cc](#).

3.29.4 Field Documentation

3.29.4.1 egraph

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

Definition at line [565](#) of file [grcc.h](#).

3.29.4.2 id

```
int EEdge::id
```

Definition at line [567](#) of file [grcc.h](#).

3.29.4.3 ext

```
int EEdge::ext
```

Definition at line [568](#) of file [grcc.h](#).

3.29.4.4 ptcl

```
int EEdge::ptcl
```

Definition at line [569](#) of file [grcc.h](#).

3.29.4.5 deleted

```
int EEdge::deleted
```

Definition at line [570](#) of file [grcc.h](#).

3.29.4.6 nodes

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

Definition at line [571](#) of file [grcc.h](#).

3.29.4.7 nlegs

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

Definition at line 572 of file [grcc.h](#).

3.29.4.8 emom

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

Definition at line 575 of file [grcc.h](#).

3.29.4.9 lmom

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

Definition at line 576 of file [grcc.h](#).

3.29.4.10 extMom

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

Definition at line 577 of file [grcc.h](#).

3.29.4.11 momset

```
ULong EEdge::momset
```

Definition at line 580 of file [grcc.h](#).

3.29.4.12 momdir

```
int EEdge::momdir
```

Definition at line 581 of file [grcc.h](#).

3.29.4.13 cut

```
Bool EEdge::cut
```

Definition at line 583 of file [grcc.h](#).

3.29.4.14 visited

```
int EEdge::visited
```

Definition at line 584 of file [grcc.h](#).

3.29.4.15 conid

```
int EEdge::conid
```

Definition at line 585 of file [grcc.h](#).

3.29.4.16 edtype

```
int EEdge::edtype
```

Definition at line 586 of file [grcc.h](#).

3.29.4.17 opicomp

```
int EEdge::opicomp
```

Definition at line 587 of file [grcc.h](#).

3.29.4.18 dir

```
int EEdge::dir
```

Definition at line 588 of file [grcc.h](#).

3.29.4.19 DUMMYPADDING

```
int EEdge::DUMMYPADDING
```

Definition at line 590 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.30 EFLine Class Reference

Public Member Functions

- void [print](#) (const char *msg)

Data Fields

- int [elist](#) [GRCC_MAXNODES]
- FLType [ftype](#)
- int [fkind](#)
- int [nlist](#)

3.30.1 Detailed Description

Definition at line 612 of file [grcc.h](#).

3.30.2 Constructor & Destructor Documentation

3.30.2.1 EFLine()

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

Definition at line 6294 of file [grcc.cc](#).

3.30.3 Member Function Documentation

3.30.3.1 print()

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

Definition at line 6302 of file [grcc.cc](#).

3.30.4 Field Documentation

3.30.4.1 elist

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

Definition at line 614 of file [grcc.h](#).

3.30.4.2 ftype

```
FLType EFLine::ftype
```

Definition at line 615 of file [grcc.h](#).

3.30.4.3 fkind

```
int EFLine::fkind
```

Definition at line 616 of file [grcc.h](#).

- int [findRoot](#) (void)
- int [dirEdge](#) (int n, int e)
- void [extMomConsv](#) (void)
- int [cmpMom](#) (int *lm0, int *em0, int *lm1, int *em1)
- int [groupLMom](#) (int *grp, int *ed2gr)
- void [chkMomConsv](#) (void)
- void [prFLines](#) (void)
- void [getFLines](#) (void)
- int [fltrace](#) (int fk, int nd0, int *fl)
- void [addFLine](#) (const FLType ft, int fk, int nfl, int *fl)
- int [legParticle](#) (int ed, int lg)

Data Fields

- [Options](#) * [opt](#)
- [Model](#) * [model](#)
- [Process](#) * [proc](#)
- [SProcess](#) * [sproc](#)
- [MGraph](#) * [mgraph](#)
- [MConn](#) * [econn](#)
- [ENode](#) ** [nodes](#)
- [EEdge](#) ** [edges](#)
- [BigInt](#) [mld](#)
- [BigInt](#) [ald](#)
- [BigInt](#) [sld](#)
- [BigInt](#) [gSubld](#)
- [Bool](#) [assigned](#)
- [int](#) [fsign](#)
- [BigInt](#) [nsym](#)
- [BigInt](#) [esym](#)
- [BigInt](#) [nsym1](#)
- [BigInt](#) [extperm](#)
- [BigInt](#) [multp](#)
- [int](#) [pld](#)
- [int](#) [sNodes](#)
- [int](#) [sEdges](#)
- [int](#) [sMaxdeg](#)
- [int](#) [sLoops](#)
- [int](#) [nNodes](#)
- [int](#) [nEdges](#)
- [int](#) [nExtern](#)
- [int](#) [maxdeg](#)
- [int](#) [nLoops](#)
- [int](#) [totalc](#)
- [int](#) [nopicomp](#)
- [int](#) [opi2plp](#)
- [int](#) [nopi2p](#)
- [int](#) [nadj2ptv](#)
- [int](#) [DUMMY_PADDING](#)
- [int](#) * [bidef](#)
- [int](#) * [bilow](#)
- [int](#) * [extMom](#)
- [int](#) [bconn](#)
- [int](#) [bicount](#)

- int [loopm](#)
- int [opiCount](#)
- [EFLine](#) * [flines](#) [GRCC_MAXFLINES]
- int [nFlines](#)
- int [DUMMYPADDING1](#)

3.31.1 Detailed Description

Definition at line [624](#) of file [grcc.h](#).

3.31.2 Constructor & Destructor Documentation

3.31.2.1 EGraph()

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

Definition at line [6327](#) of file [grcc.cc](#).

3.31.2.2 ~EGraph()

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

Definition at line [6387](#) of file [grcc.cc](#).

3.31.3 Member Function Documentation

3.31.3.1 copy()

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

Definition at line [6418](#) of file [grcc.cc](#).

3.31.3.2 print()

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

Definition at line [6817](#) of file [grcc.cc](#).

3.31.3.3 printPy()

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

Definition at line 6883 of file [grcc.cc](#).

3.31.3.4 fromDGraph()

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

Definition at line 6482 of file [grcc.cc](#).

3.31.3.5 fromMGraph()

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

Definition at line 6588 of file [grcc.cc](#).

3.31.3.6 optQGrafM()

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

Definition at line 7044 of file [grcc.cc](#).

3.31.3.7 optQGrafA()

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

Definition at line 7263 of file [grcc.cc](#).

3.31.3.8 isOptE()

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

Definition at line 7288 of file [grcc.cc](#).

3.31.3.9 setExtern()

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

Definition at line 6770 of file [grcc.cc](#).

3.31.3.10 isExternal()

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

Definition at line 700 of file [grcc.h](#).

3.31.3.11 isFermion()

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

Definition at line 7905 of file [grcc.cc](#).

3.31.3.12 setExtLoop()

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

Definition at line 6460 of file [grcc.cc](#).

3.31.3.13 endSetExtLoop()

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

Definition at line 6467 of file [grcc.cc](#).

3.31.3.14 connComp()

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

Definition at line 7440 of file [grcc.cc](#).

3.31.3.15 connVisit()

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

Definition at line 7472 of file [grcc.cc](#).

3.31.3.16 biconnE()

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

Definition at line 7493 of file [grcc.cc](#).

3.31.3.17 biinitE()

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

Definition at line 7557 of file [grcc.cc](#).

3.31.3.18 bisearchE()

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

Definition at line 7593 of file [grcc.cc](#).

3.31.3.19 findRoot()

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

Definition at line 7391 of file [grcc.cc](#).

3.31.3.20 dirEdge()

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

Definition at line 6951 of file [grcc.cc](#).

3.31.3.21 extMomConsv()

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

Definition at line 7798 of file [grcc.cc](#).

3.31.3.22 cmpMom()

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

Definition at line 6961 of file [grcc.cc](#).

3.31.3.23 groupLMom()

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

Definition at line [7011](#) of file [grcc.cc](#).

3.31.3.24 chkMomConsv()

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

Definition at line [7826](#) of file [grcc.cc](#).

3.31.3.25 prFLines()

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

Definition at line [6938](#) of file [grcc.cc](#).

3.31.3.26 getFLines()

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

Definition at line [8022](#) of file [grcc.cc](#).

3.31.3.27 fltrace()

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

Definition at line [7919](#) of file [grcc.cc](#).

3.31.3.28 addFLine()

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

Definition at line [8116](#) of file [grcc.cc](#).

3.31.3.29 legParticle()

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

Definition at line 7897 of file [grcc.cc](#).

3.31.4 Field Documentation

3.31.4.1 opt

[Options*](#) EGraph::opt

Definition at line 626 of file [grcc.h](#).

3.31.4.2 model

[Model*](#) EGraph::model

Definition at line 627 of file [grcc.h](#).

3.31.4.3 proc

[Process*](#) EGraph::proc

Definition at line 628 of file [grcc.h](#).

3.31.4.4 sproc

[SProcess*](#) EGraph::sproc

Definition at line 629 of file [grcc.h](#).

3.31.4.5 mgraph

[MGraph*](#) EGraph::mgraph

Definition at line 630 of file [grcc.h](#).

3.31.4.6 econn

[MConn*](#) EGraph::econn

Definition at line 631 of file [grcc.h](#).

3.31.4.7 nodes

`ENode** EGraph::nodes`

Definition at line 633 of file [grcc.h](#).

3.31.4.8 edges

`EEdge** EGraph::edges`

Definition at line 634 of file [grcc.h](#).

3.31.4.9 mId

`BigInt EGraph::mId`

Definition at line 636 of file [grcc.h](#).

3.31.4.10 aId

`BigInt EGraph::aId`

Definition at line 637 of file [grcc.h](#).

3.31.4.11 sId

`BigInt EGraph::sId`

Definition at line 638 of file [grcc.h](#).

3.31.4.12 gSubId

`BigInt EGraph::gSubId`

Definition at line 639 of file [grcc.h](#).

3.31.4.13 assigned

`Bool EGraph::assigned`

Definition at line 640 of file [grcc.h](#).

3.31.4.14 fsign

`int EGraph::fsign`

Definition at line 642 of file [grcc.h](#).

3.31.4.15 nsym

`BigInt EGraph::nsym`

Definition at line 643 of file [grcc.h](#).

3.31.4.16 esym

`BigInt EGraph::esym`

Definition at line 643 of file [grcc.h](#).

3.31.4.17 nsym1

`BigInt EGraph::nsym1`

Definition at line 644 of file [grcc.h](#).

3.31.4.18 extperm

`BigInt EGraph::extperm`

Definition at line 645 of file [grcc.h](#).

3.31.4.19 multp

`BigInt EGraph::multp`

Definition at line 646 of file [grcc.h](#).

3.31.4.20 pld

`int EGraph::pId`

Definition at line 648 of file [grcc.h](#).

3.31.4.21 sNodes

`int EGraph::sNodes`

Definition at line 650 of file [grcc.h](#).

3.31.4.22 sEdges

`int EGraph::sEdges`

Definition at line 651 of file [grcc.h](#).

3.31.4.23 sMaxdeg

```
int EGraph::sMaxdeg
```

Definition at line 652 of file [grcc.h](#).

3.31.4.24 sLoops

```
int EGraph::sLoops
```

Definition at line 653 of file [grcc.h](#).

3.31.4.25 nNodes

```
int EGraph::nNodes
```

Definition at line 655 of file [grcc.h](#).

3.31.4.26 nEdges

```
int EGraph::nEdges
```

Definition at line 656 of file [grcc.h](#).

3.31.4.27 nExtern

```
int EGraph::nExtern
```

Definition at line 657 of file [grcc.h](#).

3.31.4.28 maxdeg

```
int EGraph::maxdeg
```

Definition at line 658 of file [grcc.h](#).

3.31.4.29 nLoops

```
int EGraph::nLoops
```

Definition at line 659 of file [grcc.h](#).

3.31.4.30 totalc

```
int EGraph::totalc
```

Definition at line 660 of file [grcc.h](#).

3.31.4.31 nopicmp

```
int EGraph::nopicmp
```

Definition at line 663 of file [grcc.h](#).

3.31.4.32 opi2plp

```
int EGraph::opi2plp
```

Definition at line 664 of file [grcc.h](#).

3.31.4.33 nopi2p

```
int EGraph::nopi2p
```

Definition at line 665 of file [grcc.h](#).

3.31.4.34 nadj2ptv

```
int EGraph::nadj2ptv
```

Definition at line 666 of file [grcc.h](#).

3.31.4.35 DUMMY_PADDING

```
int EGraph::DUMMY_PADDING
```

Definition at line 668 of file [grcc.h](#).

3.31.4.36 bidef

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

Definition at line 670 of file [grcc.h](#).

3.31.4.37 bilow

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

Definition at line 671 of file [grcc.h](#).

3.31.4.38 extMom

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

Definition at line 672 of file [grcc.h](#).

3.31.4.39 bconn

```
int EGraph::bconn
```

Definition at line 673 of file [grcc.h](#).

3.31.4.40 bicount

```
int EGraph::bicount
```

Definition at line 674 of file [grcc.h](#).

3.31.4.41 loopm

```
int EGraph::loopm
```

Definition at line 675 of file [grcc.h](#).

3.31.4.42 opiCount

```
int EGraph::opiCount
```

Definition at line 676 of file [grcc.h](#).

3.31.4.43 flines

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

Definition at line 679 of file [grcc.h](#).

3.31.4.44 nFlines

```
int EGraph::nFlines
```

Definition at line 680 of file [grcc.h](#).

3.31.4.45 DUMMY_PADDING1

```
int EGraph::DUMMY_PADDING1
```

Definition at line 682 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.32.2 Constructor & Destructor Documentation

3.32.2.1 ENode() [1/2]

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

Definition at line 6023 of file [grcc.cc](#).

3.32.2.2 ENode() [2/2]

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

Definition at line 6039 of file [grcc.cc](#).

3.32.2.3 ~ENode()

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

Definition at line 6059 of file [grcc.cc](#).

3.32.3 Member Function Documentation

3.32.3.1 initAss()

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

Definition at line 6083 of file [grcc.cc](#).

3.32.3.2 setId()

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

Definition at line 6091 of file [grcc.cc](#).

3.32.3.3 copy()

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

Definition at line 6070 of file [grcc.cc](#).

3.32.3.4 setExtern()

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

Definition at line 6098 of file [grcc.cc](#).

3.32.3.5 setType()

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

Definition at line 6111 of file [grcc.cc](#).

3.32.3.6 print()

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

Definition at line 6124 of file [grcc.cc](#).

3.32.4 Field Documentation

3.32.4.1 egraph

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

Definition at line 530 of file [grcc.h](#).

3.32.4.2 edges

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

Definition at line 531 of file [grcc.h](#).

3.32.4.3 id

```
int ENode::id
```

Definition at line 534 of file [grcc.h](#).

3.32.4.4 maxdeg

```
int ENode::maxdeg
```

Definition at line 535 of file [grcc.h](#).

3.32.4.5 deg

```
int ENode::deg
```

Definition at line 536 of file [grcc.h](#).

3.32.4.6 extloop

```
int ENode::extloop
```

Definition at line 537 of file [grcc.h](#).

3.32.4.7 ndtype

```
int ENode::ndtype
```

Definition at line 538 of file [grcc.h](#).

3.32.4.8 intrct

```
int ENode::intrct
```

Definition at line 539 of file [grcc.h](#).

3.32.4.9 klow

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

Definition at line 542 of file [grcc.h](#).

3.32.4.10 visited

```
int ENode::visited
```

Definition at line 543 of file [grcc.h](#).

3.32.4.11 DUMMYPADDING

```
int ENode::DUMMYPADDING
```

Definition at line 545 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.33 EStack Class Reference

Public Member Functions

- void [print](#) (const char *msg)

Data Fields

- int [edgen](#)
- int [det](#)
- int [nplist](#)
- int [plist](#) [GRCC_MAXMPARTICLES2]

3.33.1 Detailed Description

Definition at line [1350](#) of file [grcc.h](#).

3.33.2 Constructor & Destructor Documentation

3.33.2.1 EStack()

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

Definition at line [1359](#) of file [grcc.h](#).

3.33.2.2 ~EStack()

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

Definition at line [1360](#) of file [grcc.h](#).

3.33.3 Member Function Documentation

3.33.3.1 print()

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

Definition at line [10232](#) of file [grcc.cc](#).

3.33.4 Field Documentation

3.33.4.1 edgen

```
int EStack::edgen
```

Definition at line [1354](#) of file [grcc.h](#).

3.33.4.2 det

```
int EStack::det
```

Definition at line 1355 of file [grcc.h](#).

3.33.4.3 nplist

```
int EStack::nplist
```

Definition at line 1356 of file [grcc.h](#).

3.33.4.4 plist

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

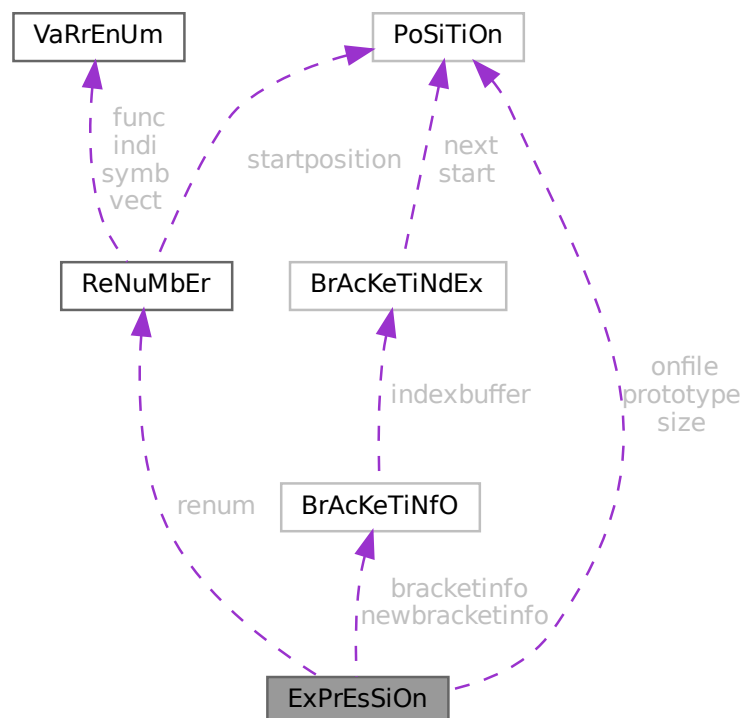
Definition at line 1357 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.34 ExPrEsSiOn Struct Reference

Collaboration diagram for ExPrEsSiOn:



Data Fields

- [POSITION onfile](#)
- [POSITION prototype](#)
- [POSITION size](#)
- [RENUMBER renum](#)
- [BRACKETINFO * bracketinfo](#)
- [BRACKETINFO * newbracketinfo](#)
- [WORD * renumlists](#)
- [WORD * inmem](#)
- [LONG counter](#)
- [LONG name](#)
- [WORD hidelevel](#)
- [WORD vflags](#)
- [WORD printflag](#)
- [WORD status](#)
- [WORD replace](#)
- [WORD node](#)
- [WORD whichbuffer](#)
- [WORD namesize](#)
- [WORD compression](#)
- [WORD numdummies](#)
- [WORD numfactors](#)
- [WORD sizeprototype](#)
- [WORD uflags](#)

3.34.1 Detailed Description

Definition at line [382](#) of file [structs.h](#).

3.34.2 Field Documentation

3.34.2.1 onfile

[POSITION](#) ExPrEsSiOn::onfile

Definition at line [383](#) of file [structs.h](#).

3.34.2.2 prototype

[POSITION](#) ExPrEsSiOn::prototype

Definition at line [384](#) of file [structs.h](#).

3.34.2.3 size

[POSITION](#) ExPrEsSiOn::size

Definition at line [385](#) of file [structs.h](#).

3.34.2.4 renum

`RENUMBER ExPrEsSiOn::renum`

Definition at line 386 of file [structs.h](#).

3.34.2.5 bracketinfo

`BRACKETINFO* ExPrEsSiOn::bracketinfo`

Definition at line 387 of file [structs.h](#).

3.34.2.6 newbracketinfo

`BRACKETINFO* ExPrEsSiOn::newbracketinfo`

Definition at line 388 of file [structs.h](#).

3.34.2.7 renumlists

`WORD* ExPrEsSiOn::renumlists`

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

Definition at line 389 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#).

3.34.2.8 inmem

`WORD* ExPrEsSiOn::inmem`

Definition at line 391 of file [structs.h](#).

3.34.2.9 counter

`LONG ExPrEsSiOn::counter`

Definition at line 392 of file [structs.h](#).

3.34.2.10 name

`LONG ExPrEsSiOn::name`

Definition at line 393 of file [structs.h](#).

3.34.2.11 hidelevel

WORD ExPrEsSiOn::hidelevel

Definition at line 394 of file [structs.h](#).

3.34.2.12 vflags

WORD ExPrEsSiOn::vflags

Definition at line 395 of file [structs.h](#).

3.34.2.13 printflag

WORD ExPrEsSiOn::printflag

Definition at line 396 of file [structs.h](#).

3.34.2.14 status

WORD ExPrEsSiOn::status

Definition at line 397 of file [structs.h](#).

3.34.2.15 replace

WORD ExPrEsSiOn::replace

Definition at line 398 of file [structs.h](#).

3.34.2.16 node

WORD ExPrEsSiOn::node

Definition at line 399 of file [structs.h](#).

3.34.2.17 whichbuffer

WORD ExPrEsSiOn::whichbuffer

Definition at line 400 of file [structs.h](#).

3.34.2.18 namesize

WORD ExPrEsSiOn::namesize

Definition at line 401 of file [structs.h](#).

3.34.2.19 compression

WORD ExPrEsSiOn::compression

Definition at line 402 of file [structs.h](#).

3.34.2.20 numdummies

WORD ExPrEsSiOn::numdummies

Definition at line 403 of file [structs.h](#).

3.34.2.21 numfactors

WORD ExPrEsSiOn::numfactors

Definition at line 404 of file [structs.h](#).

3.34.2.22 sizeprototype

WORD ExPrEsSiOn::sizeprototype

Definition at line 405 of file [structs.h](#).

3.34.2.23 uflags

WORD ExPrEsSiOn::uflags

Definition at line 406 of file [structs.h](#).

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

- [structs.h](#)

3.35 FaCdOILaR Struct Reference

Data Fields

- WORD * [where](#)
- LONG [size](#)
- WORD [type](#)
- WORD [value](#)

3.35.1 Detailed Description

Definition at line 534 of file [structs.h](#).

3.35.2 Field Documentation

3.35.2.1 where

WORD* FaCdOlLaR::where

Definition at line 535 of file [structs.h](#).

3.35.2.2 size

LONG FaCdOlLaR::size

Definition at line 536 of file [structs.h](#).

3.35.2.3 type

WORD FaCdOlLaR::type

Definition at line 537 of file [structs.h](#).

3.35.2.4 value

WORD FaCdOlLaR::value

Definition at line 538 of file [structs.h](#).

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

- [structs.h](#)

3.36 factorized_poly Class Reference

Public Member Functions

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

Data Fields

- std::vector< [poly](#) > [factor](#)
- std::vector< int > [power](#)

Friends

- std::ostream & [operator](#)<< (std::ostream &out, const [poly](#) &p)

3.36.1 Detailed Description

Definition at line 54 of file [polyfact.h](#).

3.36.2 Member Function Documentation

3.36.2.1 `add_factor()`

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

Definition at line 125 of file [polyfact.cc](#).

3.36.2.2 `tostring()`

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

Definition at line 59 of file [polyfact.cc](#).

3.36.3 Field Documentation

3.36.3.1 `factor`

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

Definition at line 59 of file [polyfact.h](#).

3.36.3.2 `power`

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

Definition at line 60 of file [polyfact.h](#).

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

- [polyfact.h](#)
- [polyfact.cc](#)

3.37 FGInput Struct Reference

Data Fields

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

3.37.1 Detailed Description

Definition at line 221 of file [grccparam.h](#).

3.37.2 Field Documentation

3.37.2.1 ninitl

```
long FGInput::ninitl
```

Definition at line 222 of file [grccparam.h](#).

3.37.2.2 initln

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

Definition at line 223 of file [grccparam.h](#).

3.37.2.3 initlc

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

Definition at line 224 of file [grccparam.h](#).

3.37.2.4 nfinal

```
long FGInput::nfinal
```

Definition at line 225 of file [grccparam.h](#).

3.37.2.5 finaln

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

Definition at line 226 of file [grccparam.h](#).

3.37.2.6 finalc

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

Definition at line 227 of file [grccparam.h](#).

3.37.2.7 coupl

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

Definition at line 228 of file [grccparam.h](#).

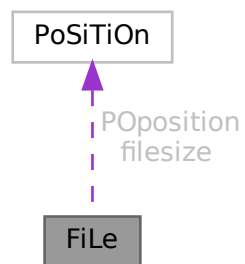
The documentation for this struct was generated from the following file:

- [grccparam.h](#)

3.38 FiLe Struct Reference

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

Collaboration diagram for FiLe:



Data Fields

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

3.38.1 Detailed Description

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

Definition at line 682 of file [structs.h](#).

3.38.2 Field Documentation

3.38.2.1 PPosition

`POSITION FiLe::PPosition`

Definition at line 683 of file [structs.h](#).

3.38.2.2 filesize

`POSITION FiLe::filesize`

Definition at line 684 of file [structs.h](#).

3.38.2.3 PObuffer

`WORD* FiLe::PObuffer`

Definition at line 685 of file [structs.h](#).

3.38.2.4 PStop

`WORD* FiLe::PStop`

Definition at line 686 of file [structs.h](#).

3.38.2.5 POfill

`WORD* FiLe::POfill`

Definition at line 687 of file [structs.h](#).

3.38.2.6 POfull

`WORD* FiLe::POfull`

Definition at line 688 of file [structs.h](#).

3.38.2.7 name

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

Definition at line 695 of file [structs.h](#).

3.38.2.8 numblocks

```
ULONG FiLe::numblocks
```

Definition at line 700 of file [structs.h](#).

3.38.2.9 inbuffer

```
ULONG FiLe::inbuffer
```

Definition at line 701 of file [structs.h](#).

3.38.2.10 PSize

```
LONG FiLe::PSize
```

Definition at line 702 of file [structs.h](#).

3.38.2.11 handle

```
int FiLe::handle
```

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

Definition at line 710 of file [structs.h](#).

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

3.38.2.12 active

```
int FiLe::active
```

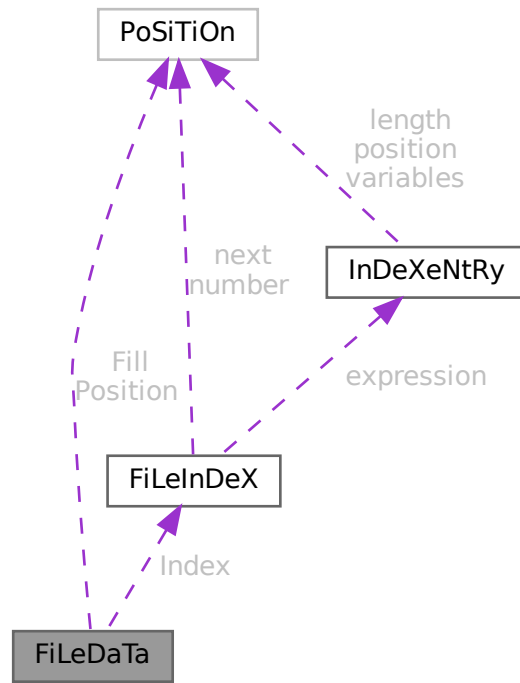
Definition at line 711 of file [structs.h](#).

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

- [structs.h](#)

3.39 FiLeDaTa Struct Reference

Collaboration diagram for FiLeDaTa:



Data Fields

- [FILEINDEX](#) `Index`
- [POSITION](#) `Fill`
- [POSITION](#) `Position`
- [WORD](#) `Handle`
- [WORD](#) `dirtyflag`

3.39.1 Detailed Description

Definition at line 150 of file [structs.h](#).

3.39.2 Field Documentation

3.39.2.1 Index

[FILEINDEX](#) `FiLeDaTa::Index`

Definition at line 151 of file [structs.h](#).

3.39.2.2 Fill

```
POSITION FileDaTa::Fill
```

Definition at line 152 of file [structs.h](#).

3.39.2.3 Position

```
POSITION FileDaTa::Position
```

Definition at line 153 of file [structs.h](#).

3.39.2.4 Handle

```
WORD FileDaTa::Handle
```

Definition at line 154 of file [structs.h](#).

3.39.2.5 dirtyflag

```
WORD FileDaTa::dirtyflag
```

Definition at line 155 of file [structs.h](#).

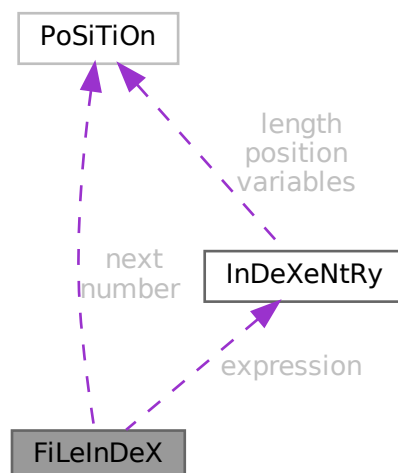
The documentation for this struct was generated from the following file:

- [structs.h](#)

3.40 FiLeInDeX Struct Reference

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

Collaboration diagram for FiLeInDeX:



Data Fields

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

3.40.1 Detailed Description

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

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

The variable number has been made of type [POSITION](#) to avoid padding problems with some types of computers/↔ OS and keep system independence of the .sav files.

This struct is always 512 bytes long.

Definition at line [137](#) of file [structs.h](#).

3.40.2 Field Documentation

3.40.2.1 next

[POSITION](#) [FiLeInDeX::next](#)

Position of next [FILEINDEX](#) if any

Definition at line [138](#) of file [structs.h](#).

Referenced by [ReadSaveIndex\(\)](#).

3.40.2.2 number

[POSITION](#) [FiLeInDeX::number](#)

Number of used entries in this index

Definition at line [139](#) of file [structs.h](#).

Referenced by [ReadSaveIndex\(\)](#).

3.40.2.3 expression

[INDEXENTRY](#) [FiLeInDeX::expression](#) [[INFILEINDEX](#)]

File index entries

Definition at line [140](#) of file [structs.h](#).

3.40.2.4 empty

```
SBYTE FileInDeX::empty[EMPTYINDEX]
```

Padding to 512 bytes

Definition at line 141 of file [structs.h](#).

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

- [structs.h](#)

3.41 FixedGlobals Struct Reference

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

Data Fields

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

3.41.1 Detailed Description

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

Definition at line 2516 of file [structs.h](#).

3.41.2 Field Documentation

3.41.2.1 Operation

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

Definition at line 2517 of file [structs.h](#).

3.41.2.2 OperaFind

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

Definition at line 2518 of file [structs.h](#).

3.41.2.3 VarType

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

Definition at line 2519 of file [structs.h](#).

3.41.2.4 ExprStat

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

Definition at line 2520 of file [structs.h](#).

3.41.2.5 FunNam

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

Definition at line 2521 of file [structs.h](#).

3.41.2.6 swmes

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

Definition at line 2522 of file [structs.h](#).

3.41.2.7 fname

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

Definition at line 2523 of file [structs.h](#).

3.41.2.8 fname2

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

Definition at line 2524 of file [structs.h](#).

3.41.2.9 s_one

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

Definition at line 2525 of file [structs.h](#).

3.41.2.10 fnamebase

WORD FixedGlobals::fnamebase

Definition at line 2526 of file [structs.h](#).

3.41.2.11 fname2base

WORD FixedGlobals::fname2base

Definition at line 2527 of file [structs.h](#).

3.41.2.12 fname2size

WORD FixedGlobals::fname2size

Definition at line 2528 of file [structs.h](#).

3.41.2.13 fname2size

WORD FixedGlobals::fname2size

Definition at line 2529 of file [structs.h](#).

3.41.2.14 cTable

UINT FixedGlobals::cTable[256]

Definition at line 2530 of file [structs.h](#).

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

- [structs.h](#)

3.42 fixedset Struct Reference

Data Fields

- char * [name](#)
- char * [description](#)
- int [type](#)
- int [dimension](#)

3.42.1 Detailed Description

Definition at line 574 of file [structs.h](#).

3.42.2 Field Documentation

3.42.2.1 name

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

Definition at line 575 of file [structs.h](#).

3.42.2.2 description

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

Definition at line 576 of file [structs.h](#).

3.42.2.3 type

```
int fixedset::type
```

Definition at line 577 of file [structs.h](#).

3.42.2.4 dimension

```
int fixedset::dimension
```

Definition at line 578 of file [structs.h](#).

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

- [structs.h](#)

3.43 flint::fmpz Class Reference

Public Member Functions

- void [print](#) (const string &text)

Data Fields

- [fmpz_t](#) d

3.43.1 Detailed Description

Definition at line 80 of file [flintinterface.h](#).

3.43.2 Constructor & Destructor Documentation

3.43.2.1 fmpz()

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

Definition at line 83 of file [flintinterface.h](#).

3.43.2.2 ~fmpz()

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

Definition at line 84 of file [flintinterface.h](#).

3.43.3 Member Function Documentation

3.43.3.1 print()

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

Definition at line 85 of file [flintinterface.h](#).

3.43.4 Field Documentation

3.43.4.1 d

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

Definition at line 82 of file [flintinterface.h](#).

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

- [flintinterface.h](#)

3.44 Fraction Class Reference

Public Member Functions

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

Data Fields

- BigInt [num](#)
- BigInt [den](#)
- double [ratio](#)

3.44.1 Detailed Description

Definition at line [204](#) of file [grcc.h](#).

3.44.2 Constructor & Destructor Documentation

3.44.2.1 Fraction() [1/2]

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

Definition at line [209](#) of file [grcc.h](#).

3.44.2.2 Fraction() [2/2]

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

Definition at line [10449](#) of file [grcc.cc](#).

3.44.3 Member Function Documentation

3.44.3.1 print()

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

Definition at line [10460](#) of file [grcc.cc](#).

3.44.3.2 setValue() [1/2]

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

Definition at line [10472](#) of file [grcc.cc](#).

3.44.3.3 setValue() [2/2]

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

Definition at line 10480 of file [grcc.cc](#).

3.44.3.4 add() [1/2]

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

Definition at line 10525 of file [grcc.cc](#).

3.44.3.5 add() [2/2]

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

Definition at line 10548 of file [grcc.cc](#).

3.44.3.6 sub()

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

Definition at line 10557 of file [grcc.cc](#).

3.44.3.7 gcd()

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

Definition at line 10488 of file [grcc.cc](#).

3.44.3.8 normal()

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

Definition at line 10508 of file [grcc.cc](#).

3.44.3.9 isEq()

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

Definition at line 10566 of file [grcc.cc](#).

3.44.4 Field Documentation

3.44.4.1 num

```
BigInt Fraction::num
```

Definition at line 206 of file [grcc.h](#).

3.44.4.2 den

```
BigInt Fraction::den
```

Definition at line 206 of file [grcc.h](#).

3.44.4.3 ratio

```
double Fraction::ratio
```

Definition at line 207 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.45 FUN_INFO Struct Reference

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

Data Fields

- WORD * [location](#)
- int [numargs](#)
- int [numfunnies](#)
- int [numwildcards](#)
- int [symmet](#)
- int [tensor](#)
- int [commute](#)

3.45.1 Detailed Description

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

Definition at line 609 of file [structs.h](#).

3.45.2 Field Documentation

3.45.2.1 location

`WORD* FUN_INFO::location`

Definition at line 610 of file [structs.h](#).

3.45.2.2 numargs

`int FUN_INFO::numargs`

Definition at line 611 of file [structs.h](#).

3.45.2.3 numfunnies

`int FUN_INFO::numfunnies`

Definition at line 612 of file [structs.h](#).

3.45.2.4 numwildcards

`int FUN_INFO::numwildcards`

Definition at line 613 of file [structs.h](#).

3.45.2.5 symmet

`int FUN_INFO::symmet`

Definition at line 614 of file [structs.h](#).

3.45.2.6 tensor

`int FUN_INFO::tensor`

Definition at line 615 of file [structs.h](#).

3.45.2.7 commute

`int FUN_INFO::commute`

Definition at line 616 of file [structs.h](#).

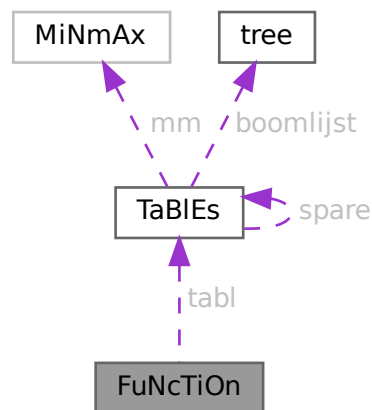
The documentation for this struct was generated from the following file:

- [structs.h](#)

3.46 FuNcTiOn Struct Reference

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

Collaboration diagram for FuNcTiOn:



Data Fields

- [TABLES](#) [tabl](#)
- LONG [symminfo](#)
- LONG [name](#)
- WORD [commute](#)
- WORD [complex](#)
- WORD [number](#)
- WORD [flags](#)
- WORD [spec](#)
- WORD [symmetric](#)
- WORD [node](#)
- WORD [namesize](#)
- WORD [dimension](#)
- WORD [maxnumargs](#)
- WORD [minnumargs](#)

3.46.1 Detailed Description

Contains all information about a function. Also used for tables. It is used in the [LIST](#) elements of AC.

Definition at line [487](#) of file [structs.h](#).

3.46.2 Field Documentation

3.46.2.1 `tabl`

`TABLES FuNcTiOn::tabl`

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

Definition at line 488 of file [structs.h](#).

3.46.2.2 `symminfo`

`LONG FuNcTiOn::symminfo`

Info regarding symm properties offset in buffer

Definition at line 489 of file [structs.h](#).

3.46.2.3 `name`

`LONG FuNcTiOn::name`

Location in namebuffer of [NAMETREE](#)

Definition at line 490 of file [structs.h](#).

Referenced by [StartVariables\(\)](#).

3.46.2.4 `commute`

`WORD FuNcTiOn::commute`

Commutation properties

Definition at line 491 of file [structs.h](#).

3.46.2.5 `complex`

`WORD FuNcTiOn::complex`

Properties under complex conjugation

Definition at line 492 of file [structs.h](#).

3.46.2.6 number

WORD FuNcTiOn::number

Number when stored in file

Definition at line 493 of file [structs.h](#).

Referenced by [ReadSaveIndex\(\)](#).

3.46.2.7 flags

WORD FuNcTiOn::flags

Used to indicate usage when storing

Definition at line 494 of file [structs.h](#).

3.46.2.8 spec

WORD FuNcTiOn::spec

Regular, Tensor, etc. See [FunSpecs](#).

Definition at line 495 of file [structs.h](#).

Referenced by [ReadSaveVariables\(\)](#).

3.46.2.9 symmetric

WORD FuNcTiOn::symmetric

0 if symmetric properties

Definition at line 496 of file [structs.h](#).

Referenced by [StartVariables\(\)](#).

3.46.2.10 node

WORD FuNcTiOn::node

Location in namenode of [NAMETREE](#)

Definition at line 497 of file [structs.h](#).

3.46.2.11 namesize

WORD FuNcTiOn::namesize

Length of the name

Definition at line 498 of file [structs.h](#).

3.46.2.12 dimension

WORD FuNcTiOn::dimension

Definition at line 499 of file [structs.h](#).

3.46.2.13 maxnumargs

WORD FuNcTiOn::maxnumargs

Definition at line 500 of file [structs.h](#).

3.46.2.14 minnumargs

WORD FuNcTiOn::minnumargs

Definition at line 501 of file [structs.h](#).

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

- [structs.h](#)

3.47 gcd_heuristic_failed Class Reference

3.47.1 Detailed Description

Definition at line 33 of file [polygcd.h](#).

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

- [polygcd.h](#)

3.48 HANDLERS Struct Reference

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

Data Fields

- WORD [newlogonly](#)
- WORD [newhandle](#)
- WORD [oldhandle](#)
- WORD [oldlogonly](#)
- WORD [oldprinttype](#)
- WORD [oldsilent](#)

3.48.1 Detailed Description

The struct [HANDLERS](#) is used in the communication with external channels.

Definition at line [943](#) of file [structs.h](#).

3.48.2 Field Documentation

3.48.2.1 newlogonly

```
WORD HANDLERS::newlogonly
```

Definition at line [944](#) of file [structs.h](#).

3.48.2.2 newhandle

```
WORD HANDLERS::newhandle
```

Definition at line [945](#) of file [structs.h](#).

3.48.2.3 oldhandle

```
WORD HANDLERS::oldhandle
```

Definition at line [946](#) of file [structs.h](#).

3.48.2.4 oldlogonly

```
WORD HANDLERS::oldlogonly
```

Definition at line [947](#) of file [structs.h](#).

3.48.2.5 oldprinttype

```
WORD HANDLERS::oldprinttype
```

Definition at line [948](#) of file [structs.h](#).

3.48.2.6 oldsilent

```
WORD HANDLERS::oldsilent
```

Definition at line 949 of file [structs.h](#).

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

- [structs.h](#)

3.49 Input Struct Reference

Data Fields

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

3.49.1 Detailed Description

Definition at line 193 of file [grccparam.h](#).

3.49.2 Field Documentation

3.49.2.1 name

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

Definition at line 194 of file [grccparam.h](#).

3.49.2.2 icode

```
int IInput::icode
```

Definition at line 195 of file [grccparam.h](#).

3.49.2.3 nplistn

```
int IInput::nplistn
```

Definition at line 196 of file [grccparam.h](#).

3.49.2.4 plistn

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

Definition at line 197 of file [grccparam.h](#).

3.49.2.5 plistc

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

Definition at line 198 of file [grccparam.h](#).

3.49.2.6 cvallist

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

Definition at line 199 of file [grccparam.h](#).

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

- [grccparam.h](#)

3.50 InDeX Struct Reference

Data Fields

- LONG [name](#)
- WORD [type](#)
- WORD [dimension](#)
- WORD [number](#)
- WORD [flags](#)
- WORD [nmin4](#)
- WORD [node](#)
- WORD [namesize](#)

3.50.1 Detailed Description

Definition at line 442 of file [structs.h](#).

3.50.2 Field Documentation

3.50.2.1 name

```
LONG InDeX::name
```

Definition at line 443 of file [structs.h](#).

3.50.2.2 type

WORD InDeX::type

Definition at line 444 of file [structs.h](#).

3.50.2.3 dimension

WORD InDeX::dimension

Definition at line 445 of file [structs.h](#).

3.50.2.4 number

WORD InDeX::number

Definition at line 446 of file [structs.h](#).

3.50.2.5 flags

WORD InDeX::flags

Definition at line 447 of file [structs.h](#).

3.50.2.6 nmin4

WORD InDeX::nmin4

Definition at line 448 of file [structs.h](#).

3.50.2.7 node

WORD InDeX::node

Definition at line 449 of file [structs.h](#).

3.50.2.8 namesize

WORD InDeX::namesize

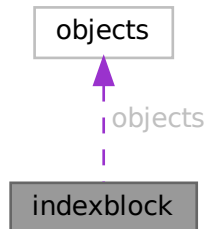
Definition at line 450 of file [structs.h](#).

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

- [structs.h](#)

3.51 indexblock Struct Reference

Collaboration diagram for indexblock:



Data Fields

- MLONG [flags](#)
- MLONG [previousblock](#)
- MLONG [position](#)
- OBJECTS [objects](#) [NUMOBJECTS]

3.51.1 Detailed Description

Definition at line [110](#) of file [minos.h](#).

3.51.2 Field Documentation

3.51.2.1 flags

MLONG `indexblock::flags`

Definition at line [111](#) of file [minos.h](#).

3.51.2.2 previousblock

MLONG `indexblock::previousblock`

Definition at line [112](#) of file [minos.h](#).

3.51.2.3 position

MLONG `indexblock::position`

Definition at line [113](#) of file [minos.h](#).

3.51.2.4 objects

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

Definition at line 114 of file [minos.h](#).

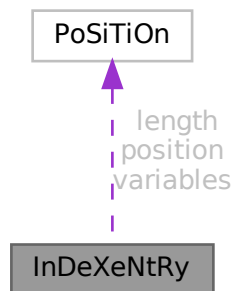
The documentation for this struct was generated from the following file:

- [minos.h](#)

3.52 InDeXeNtRy Struct Reference

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

Collaboration diagram for InDeXeNtRy:



Data Fields

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

3.52.1 Detailed Description

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

It represents one expression in the file.

Definition at line 100 of file [structs.h](#).

3.52.2 Field Documentation

3.52.2.1 position

`POSITION InDeXeNtRy::position`

Position of the expression itself

Definition at line 101 of file [structs.h](#).

3.52.2.2 length

`POSITION InDeXeNtRy::length`

Length of the expression itself

Definition at line 102 of file [structs.h](#).

3.52.2.3 variables

`POSITION InDeXeNtRy::variables`

Position of the list with variables

Definition at line 103 of file [structs.h](#).

3.52.2.4 CompressSize

`LONG InDeXeNtRy::CompressSize`

Size of buffer before compress

Definition at line 104 of file [structs.h](#).

3.52.2.5 nsymbols

`WORD InDeXeNtRy::nsymbols`

Number of symbols in the list

Definition at line 105 of file [structs.h](#).

Referenced by [ReadSaveVariables\(\)](#).

3.52.2.6 nindices

WORD InDeXeNtRy::nindices

Number of indices in the list

Definition at line 106 of file [structs.h](#).

Referenced by [ReadSaveVariables\(\)](#).

3.52.2.7 nvectors

WORD InDeXeNtRy::nvectors

Number of vectors in the list

Definition at line 107 of file [structs.h](#).

Referenced by [ReadSaveVariables\(\)](#).

3.52.2.8 nfunctions

WORD InDeXeNtRy::nfunctions

Number of functions in the list

Definition at line 108 of file [structs.h](#).

Referenced by [ReadSaveVariables\(\)](#).

3.52.2.9 size

WORD InDeXeNtRy::size

Size of variables field

Definition at line 109 of file [structs.h](#).

3.52.2.10 name

SBYTE InDeXeNtRy::name[MAXENAME+1]

Name of expression

Definition at line 110 of file [structs.h](#).

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

- [structs.h](#)

3.53 iniinfo Struct Reference

Data Fields

- MLONG [entriesinindex](#)
- MLONG [numberofindexblocks](#)
- MLONG [firstindexblock](#)
- MLONG [lastindexblock](#)
- MLONG [numberoftables](#)
- MLONG [numberofnamesblocks](#)
- MLONG [firstnameblock](#)
- MLONG [lastnameblock](#)

3.53.1 Detailed Description

Definition at line [85](#) of file [minos.h](#).

3.53.2 Field Documentation

3.53.2.1 entriesinindex

```
MLONG iniinfo::entriesinindex
```

Definition at line [87](#) of file [minos.h](#).

3.53.2.2 numberofindexblocks

```
MLONG iniinfo::numberofindexblocks
```

Definition at line [88](#) of file [minos.h](#).

3.53.2.3 firstindexblock

```
MLONG iniinfo::firstindexblock
```

Definition at line [89](#) of file [minos.h](#).

3.53.2.4 lastindexblock

```
MLONG iniinfo::lastindexblock
```

Definition at line [90](#) of file [minos.h](#).

3.53.2.5 numberoftables

```
MLONG iniinfo::numberoftables
```

Definition at line 91 of file [minos.h](#).

3.53.2.6 numberofnamesblocks

```
MLONG iniinfo::numberofnamesblocks
```

Definition at line 92 of file [minos.h](#).

3.53.2.7 firstnameblock

```
MLONG iniinfo::firstnameblock
```

Definition at line 93 of file [minos.h](#).

3.53.2.8 lastnameblock

```
MLONG iniinfo::lastnameblock
```

Definition at line 94 of file [minos.h](#).

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

- [minos.h](#)

3.54 INSIDEINFO Struct Reference

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

Data Fields

- WORD * [buffer](#)
- int [oldcompiletype](#)
- int [oldparallelflag](#)
- int [oldnumpotmoddollars](#)
- WORD [size](#)
- WORD [numdollars](#)
- WORD [oldcbuf](#)
- WORD [oldrbuf](#)
- WORD [inscbuf](#)
- WORD [oldcnumlhs](#)

3.54.1 Detailed Description

Used for #inside

Definition at line 836 of file [structs.h](#).

3.54.2 Field Documentation

3.54.2.1 buffer

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

Definition at line 837 of file [structs.h](#).

3.54.2.2 oldcompiletype

```
int INSIDEINFO::oldcompiletype
```

Definition at line 838 of file [structs.h](#).

3.54.2.3 oldparallelflag

```
int INSIDEINFO::oldparallelflag
```

Definition at line 839 of file [structs.h](#).

3.54.2.4 oldnumpotmoddollars

```
int INSIDEINFO::oldnumpotmoddollars
```

Definition at line 840 of file [structs.h](#).

3.54.2.5 size

```
WORD INSIDEINFO::size
```

Definition at line 841 of file [structs.h](#).

3.54.2.6 numdollars

```
WORD INSIDEINFO::numdollars
```

Definition at line 842 of file [structs.h](#).

3.54.2.7 oldcbuf

WORD INSIDEINFO::oldcbuf

Definition at line 843 of file [structs.h](#).

3.54.2.8 oldrbuf

WORD INSIDEINFO::oldrbuf

Definition at line 844 of file [structs.h](#).

3.54.2.9 inscbuf

WORD INSIDEINFO::inscbuf

Definition at line 845 of file [structs.h](#).

3.54.2.10 oldcnumlhs

WORD INSIDEINFO::oldcnumlhs

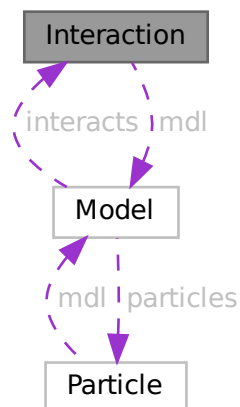
Definition at line 846 of file [structs.h](#).

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

- [structs.h](#)

3.55 Interaction Class Reference

Collaboration diagram for Interaction:



Public Member Functions

- [Interaction](#) ([Model](#) *modl, int iid, const char *nam, int icode, int *cpl, int nlgs, int *plst, int csm, int lp)
- void [prlInteraction](#) (void)

Data Fields

- [Model](#) * [mdl](#)
- char * [name](#)
- int * [plist](#)
- int * [clist](#)
- int * [slist](#)
- int [id](#)
- int [csum](#)
- int [nlegs](#)
- int [loop](#)
- int [icode](#)
- int [nplist](#)
- int [nclist](#)
- int [nslist](#)

3.55.1 Detailed Description

Definition at line 258 of file [grcc.h](#).

3.55.2 Constructor & Destructor Documentation

3.55.2.1 Interaction()

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

Definition at line 1820 of file [grcc.cc](#).

3.55.2.2 ~Interaction()

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

Definition at line 1944 of file [grcc.cc](#).

3.55.3 Member Function Documentation

3.55.3.1 prInteraction()

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

Definition at line 1959 of file [grcc.cc](#).

3.55.4 Field Documentation

3.55.4.1 mdl

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

Definition at line 260 of file [grcc.h](#).

3.55.4.2 name

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

Definition at line 261 of file [grcc.h](#).

3.55.4.3 plist

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

Definition at line 262 of file [grcc.h](#).

3.55.4.4 clist

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

Definition at line 263 of file [grcc.h](#).

3.55.4.5 slist

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

Definition at line 264 of file [grcc.h](#).

3.55.4.6 id

```
int Interaction::id
```

Definition at line 266 of file [grcc.h](#).

3.55.4.7 csum

```
int Interaction::csum
```

Definition at line 267 of file [grcc.h](#).

3.55.4.8 nlegs

```
int Interaction::nlegs
```

Definition at line 268 of file [grcc.h](#).

3.55.4.9 loop

```
int Interaction::loop
```

Definition at line 269 of file [grcc.h](#).

3.55.4.10 icode

```
int Interaction::icode
```

Definition at line 270 of file [grcc.h](#).

3.55.4.11 nplist

```
int Interaction::nplist
```

Definition at line 272 of file [grcc.h](#).

3.55.4.12 nclist

```
int Interaction::nclist
```

Definition at line 273 of file [grcc.h](#).

3.55.4.13 nslist

```
int Interaction::nslist
```

Definition at line 274 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.56 KEYWORD Struct Reference

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

Data Fields

- char * [name](#)
- TFUN [func](#)
- int [type](#)
- int [flags](#)

3.56.1 Detailed Description

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

Definition at line [218](#) of file [structs.h](#).

3.56.2 Field Documentation

3.56.2.1 name

```
char* KEYWORD::name
```

Definition at line [219](#) of file [structs.h](#).

3.56.2.2 func

```
TFUN KEYWORD::func
```

Definition at line [220](#) of file [structs.h](#).

3.56.2.3 type

```
int KEYWORD::type
```

Definition at line [221](#) of file [structs.h](#).

3.56.2.4 flags

```
int KEYWORD::flags
```

Definition at line [222](#) of file [structs.h](#).

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

- [structs.h](#)

3.57 KEYWORDV Struct Reference

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

Data Fields

- char * [name](#)
- int * [var](#)
- int [type](#)
- int [flags](#)

3.57.1 Detailed Description

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

Definition at line [230](#) of file [structs.h](#).

3.57.2 Field Documentation

3.57.2.1 name

```
char* KEYWORDV::name
```

Definition at line [231](#) of file [structs.h](#).

3.57.2.2 var

```
int* KEYWORDV::var
```

Definition at line [232](#) of file [structs.h](#).

3.57.2.3 type

```
int KEYWORDV::type
```

Definition at line [233](#) of file [structs.h](#).

3.57.2.4 flags

```
int KEYWORDV::flags
```

Definition at line [234](#) of file [structs.h](#).

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

- [structs.h](#)

3.58 LIST Struct Reference

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

Data Fields

- void * [lijst](#)
- char * [message](#)
- int [num](#)
- int [maxnum](#)
- int [size](#)
- int [numglobal](#)
- int [numtemp](#)
- int [numclear](#)

3.58.1 Detailed Description

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

Definition at line [202](#) of file [structs.h](#).

3.58.2 Field Documentation

3.58.2.1 lijst

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

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

Definition at line [203](#) of file [structs.h](#).

3.58.2.2 message

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

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

Definition at line [204](#) of file [structs.h](#).

3.58.2.3 num

```
int LIST::num
```

Number of elements in lijst.

Definition at line [205](#) of file [structs.h](#).

3.58.2.4 maxnum

```
int LIST::maxnum
```

Maximum number of elements in lijst.

Definition at line 206 of file [structs.h](#).

3.58.2.5 size

```
int LIST::size
```

Size of one element in lijst.

Definition at line 207 of file [structs.h](#).

3.58.2.6 numglobal

```
int LIST::numglobal
```

Marker for position when .global is executed.

Definition at line 208 of file [structs.h](#).

3.58.2.7 numtemp

```
int LIST::numtemp
```

At the moment only needed for sets and setstore.

Definition at line 209 of file [structs.h](#).

3.58.2.8 numclear

```
int LIST::numclear
```

Only for the clear instruction.

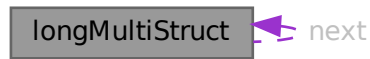
Definition at line 210 of file [structs.h](#).

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

- [structs.h](#)

3.59 longMultiStruct Struct Reference

Collaboration diagram for longMultiStruct:



Data Fields

- UBYTE * [buffer](#)
- int [bufpos](#)
- int [packpos](#)
- int [nPacks](#)
- int [lastLen](#)
- struct [longMultiStruct](#) * [next](#)

3.59.1 Detailed Description

Definition at line [1013](#) of file [mpi.c](#).

3.59.2 Field Documentation

3.59.2.1 buffer

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

Definition at line [1014](#) of file [mpi.c](#).

3.59.2.2 bufpos

```
int longMultiStruct::bufpos
```

Definition at line [1015](#) of file [mpi.c](#).

3.59.2.3 packpos

```
int longMultiStruct::packpos
```

Definition at line [1016](#) of file [mpi.c](#).

3.59.2.4 nPacks

```
int longMultiStruct::nPacks
```

Definition at line 1017 of file [mpi.c](#).

3.59.2.5 lastLen

```
int longMultiStruct::lastLen
```

Definition at line 1018 of file [mpi.c](#).

3.59.2.6 next

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

Definition at line 1021 of file [mpi.c](#).

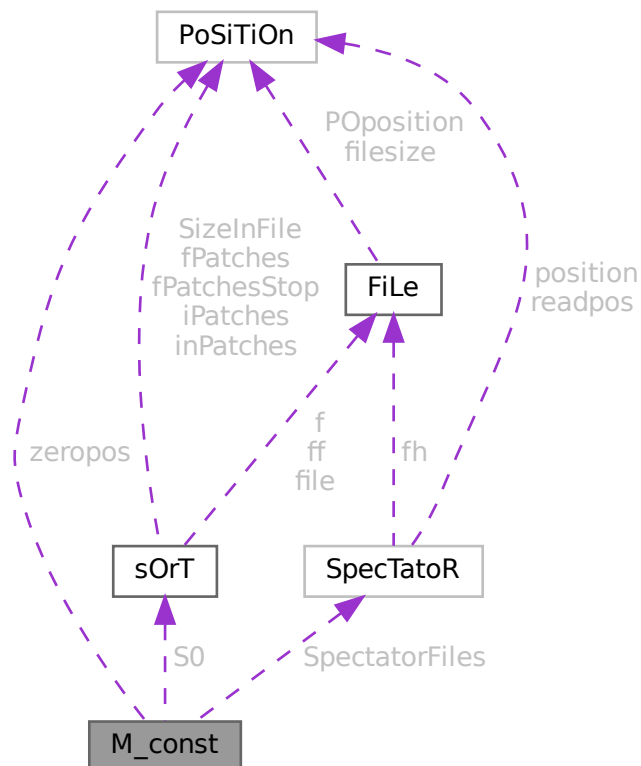
The documentation for this struct was generated from the following file:

- [mpi.c](#)

3.60 M_const Struct Reference

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

Collaboration diagram for M_const:



Data Fields

- [POSITION](#) [zeropos](#)
- [SORTING](#) * [S0](#)
- [UWORD](#) * [gcmmod](#)
- [UWORD](#) * [gpowmod](#)
- [UBYTE](#) * [TempDir](#)
- [UBYTE](#) * [TempSortDir](#)
- [UBYTE](#) * [IncDir](#)
- [UBYTE](#) * [InputFileName](#)
- [UBYTE](#) * [LogFileName](#)
- [UBYTE](#) * [OutBuffer](#)
- [UBYTE](#) * [Path](#)
- [UBYTE](#) * [SetupDir](#)
- [UBYTE](#) * [SetupFile](#)
- [UBYTE](#) * [gFortran90Kind](#)
- [UBYTE](#) * [gextrasym](#)
- [UBYTE](#) * [ggextrasym](#)
- [UBYTE](#) * [oldnumextrasymbols](#)
- [SPECTATOR](#) * [SpectatorFiles](#)
- [LONG](#) [MaxTer](#)
- [LONG](#) [CompressSize](#)
- [LONG](#) [ScratSize](#)
- [LONG](#) [HideSize](#)
- [LONG](#) [SizeStoreCache](#)
- [LONG](#) [MaxStreamSize](#)
- [LONG](#) [SIOsize](#)
- [LONG](#) [SLargeSize](#)
- [LONG](#) [SSmallEsize](#)
- [LONG](#) [SSmallSize](#)
- [LONG](#) [STermsInSmall](#)
- [LONG](#) [MaxBracketBufferSize](#)
- [LONG](#) [hProcessBucketSize](#)
- [LONG](#) [gProcessBucketSize](#)
- [LONG](#) [shmWinSize](#)
- [LONG](#) [OldChildTime](#)
- [LONG](#) [OldSecTime](#)
- [LONG](#) [OldMilliTime](#)
- [LONG](#) [WorkSize](#)
- [LONG](#) [gThreadBucketSize](#)
- [LONG](#) [ggThreadBucketSize](#)
- [LONG](#) [SumTime](#)
- [LONG](#) [SpectatorSize](#)
- [LONG](#) [TimeLimit](#)
- [int](#) [FileOnlyFlag](#)
- [int](#) [Interact](#)
- [int](#) [MaxParLevel](#)
- [int](#) [OutBufSize](#)
- [int](#) [SMaxFpatches](#)
- [int](#) [SMaxPatches](#)
- [int](#) [StdOut](#)
- [int](#) [ginsidefirst](#)
- [int](#) [gDefDim](#)
- [int](#) [gDefDim4](#)
- [int](#) [NumFixedSets](#)

- int NumFixedFunctions
- int rbufnum
- int dbufnum
- int sbufnum
- int zbufnum
- int SkipClears
- int gTokensWriteFlag
- int gfunpowers
- int gStatsFlag
- int gNamesFlag
- int gCodesFlag
- int gSortType
- int gproperorderflag
- int hparallelflag
- int gparallelflag
- int totalnumberofthreads
- int gSizeCommuteInSet
- int gThreadStats
- int ggThreadStats
- int gFinalStats
- int ggFinalStats
- int gThreadsFlag
- int ggThreadsFlag
- int gThreadBalancing
- int ggThreadBalancing
- int gThreadSortFileSynch
- int ggThreadSortFileSynch
- int gProcessStats
- int ggProcessStats
- int gOldParallelStats
- int ggOldParallelStats
- int maxFlevels
- int resetTimeOnClear
- int gcNumDollars
- int MultiRun
- int gNoSpacesInNumbers
- int ggNoSpacesInNumbers
- int glsFortran90
- int PrintTotalSize
- int fbufferize
- int gOldFactArgFlag
- int ggOldFactArgFlag
- int gnumextrasym
- int ggnumextrasym
- int NumSpectatorFiles
- int SizeForSpectatorFiles
- int gOldGCDflag
- int ggOldGCDflag
- int gWTimeStatsFlag
- int ggWTimeStatsFlag
- int jumpratio
- WORD MaxTal
- WORD IndDum
- WORD DumInd
- WORD Willnd

- WORD [gncmod](#)
- WORD [gnpowmod](#)
- WORD [gmodmode](#)
- WORD [gUnitTrace](#)
- WORD [gOutputMode](#)
- WORD [gOutputSpaces](#)
- WORD [gOutNumberType](#)
- WORD [gCnumpows](#)
- WORD [gUniTrace](#) [4]
- WORD [MaxWildcards](#)
- WORD [mTraceDum](#)
- WORD [OffsetIndex](#)
- WORD [OffsetVector](#)
- WORD [RepMax](#)
- WORD [LogType](#)
- WORD [ggStatsFlag](#)
- WORD [gLineLength](#)
- WORD [qError](#)
- WORD [FortranCont](#)
- WORD [HoldFlag](#)
- WORD [Ordering](#) [15]
- WORD [silent](#)
- WORD [tracebackflag](#)
- WORD [expnum](#)
- WORD [denomnum](#)
- WORD [facnum](#)
- WORD [invfacnum](#)
- WORD [sumnum](#)
- WORD [sumpnum](#)
- WORD [OldOrderFlag](#)
- WORD [termfunnum](#)
- WORD [matchfunnum](#)
- WORD [countfunnum](#)
- WORD [gPolyFun](#)
- WORD [gPolyFunInv](#)
- WORD [gPolyFunType](#)
- WORD [gPolyFunExp](#)
- WORD [gPolyFunVar](#)
- WORD [gPolyFunPow](#)
- WORD [dollarzero](#)
- WORD [atstartup](#)
- WORD [exitflag](#)
- WORD [NumStoreCaches](#)
- WORD [gIndentSpace](#)
- WORD [ggIndentSpace](#)
- WORD [gShortStatsMax](#)
- WORD [ggShortStatsMax](#)
- WORD [gextrasymbols](#)
- WORD [ggextrasymbols](#)
- WORD [zerorhs](#)
- WORD [onerhs](#)
- WORD [havesortdir](#)
- WORD [vectorzero](#)
- WORD [ClearStore](#)
- WORD [BracketFactors](#) [8]
- BOOL [FromStdin](#)
- BOOL [IgnoreDeprecation](#)

3.60.1 Detailed Description

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

Definition at line [1389](#) of file [structs.h](#).

3.60.2 Field Documentation

3.60.2.1 zeropos

```
POSITION M_const::zeropos
```

Definition at line [1390](#) of file [structs.h](#).

3.60.2.2 S0

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

[D] The main sort buffer

Definition at line [1391](#) of file [structs.h](#).

3.60.2.3 gcmmod

```
UWORD* M_const::gcmmod
```

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

Definition at line [1392](#) of file [structs.h](#).

3.60.2.4 gpowmod

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

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

Definition at line [1393](#) of file [structs.h](#).

3.60.2.5 TempDir

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

Definition at line [1394](#) of file [structs.h](#).

3.60.2.6 TempSortDir

UBYTE* M_const::TempSortDir

Definition at line 1395 of file [structs.h](#).

3.60.2.7 IncDir

UBYTE* M_const::IncDir

Definition at line 1396 of file [structs.h](#).

3.60.2.8 InputFileName

UBYTE* M_const::InputFileName

Definition at line 1397 of file [structs.h](#).

3.60.2.9 LogFileName

UBYTE* M_const::LogFileName

Definition at line 1398 of file [structs.h](#).

3.60.2.10 OutBuffer

UBYTE* M_const::OutBuffer

Definition at line 1399 of file [structs.h](#).

3.60.2.11 Path

UBYTE* M_const::Path

Definition at line 1400 of file [structs.h](#).

3.60.2.12 SetupDir

UBYTE* M_const::SetupDir

Definition at line 1401 of file [structs.h](#).

3.60.2.13 SetupFile

UBYTE* M_const::SetupFile

Definition at line 1402 of file [structs.h](#).

3.60.2.14 gFortran90Kind

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

Definition at line 1403 of file [structs.h](#).

3.60.2.15 gextrasym

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

Definition at line 1404 of file [structs.h](#).

3.60.2.16 ggextrasym

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

Definition at line 1405 of file [structs.h](#).

3.60.2.17 oldnumextrasymbols

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

Definition at line 1406 of file [structs.h](#).

3.60.2.18 SpectatorFiles

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

Definition at line 1407 of file [structs.h](#).

3.60.2.19 MaxTer

```
LONG M_const::MaxTer
```

Definition at line 1415 of file [structs.h](#).

3.60.2.20 CompressSize

```
LONG M_const::CompressSize
```

Definition at line 1416 of file [structs.h](#).

3.60.2.21 ScratSize

```
LONG M_const::SkratSize
```

Definition at line 1417 of file [structs.h](#).

3.60.2.22 HideSize

`LONG M_const::HideSize`

Definition at line [1418](#) of file [structs.h](#).

3.60.2.23 SizeStoreCache

`LONG M_const::SizeStoreCache`

Definition at line [1419](#) of file [structs.h](#).

3.60.2.24 MaxStreamSize

`LONG M_const::MaxStreamSize`

Definition at line [1420](#) of file [structs.h](#).

3.60.2.25 SIOsize

`LONG M_const::SIOsize`

Definition at line [1421](#) of file [structs.h](#).

3.60.2.26 SLargeSize

`LONG M_const::SLargeSize`

Definition at line [1422](#) of file [structs.h](#).

3.60.2.27 SSmallEsize

`LONG M_const::SSmallEsize`

Definition at line [1423](#) of file [structs.h](#).

3.60.2.28 SSmallSize

`LONG M_const::SSmallSize`

Definition at line [1424](#) of file [structs.h](#).

3.60.2.29 STermsInSmall

`LONG M_const::STermsInSmall`

Definition at line [1425](#) of file [structs.h](#).

3.60.2.30 MaxBracketBufferSize

LONG M_const::MaxBracketBufferSize

Definition at line 1426 of file [structs.h](#).

3.60.2.31 hProcessBucketSize

LONG M_const::hProcessBucketSize

Definition at line 1427 of file [structs.h](#).

3.60.2.32 gProcessBucketSize

LONG M_const::gProcessBucketSize

Definition at line 1428 of file [structs.h](#).

3.60.2.33 shmWinSize

LONG M_const::shmWinSize

Definition at line 1429 of file [structs.h](#).

3.60.2.34 OldChildTime

LONG M_const::OldChildTime

Definition at line 1430 of file [structs.h](#).

3.60.2.35 OldSecTime

LONG M_const::OldSecTime

Definition at line 1431 of file [structs.h](#).

3.60.2.36 OldMilliTime

LONG M_const::OldMilliTime

Definition at line 1432 of file [structs.h](#).

3.60.2.37 WorkSize

LONG M_const::WorkSize

Definition at line 1433 of file [structs.h](#).

3.60.2.38 gThreadBucketSize

LONG M_const::gThreadBucketSize

Definition at line 1434 of file [structs.h](#).

3.60.2.39 ggThreadBucketSize

LONG M_const::ggThreadBucketSize

Definition at line 1435 of file [structs.h](#).

3.60.2.40 SumTime

LONG M_const::SumTime

Definition at line 1436 of file [structs.h](#).

3.60.2.41 SpectatorSize

LONG M_const::SpectatorSize

Definition at line 1437 of file [structs.h](#).

3.60.2.42 TimeLimit

LONG M_const::TimeLimit

Definition at line 1438 of file [structs.h](#).

3.60.2.43 FileOnlyFlag

int M_const::FileOnlyFlag

Definition at line 1445 of file [structs.h](#).

3.60.2.44 Interact

int M_const::Interact

Definition at line 1446 of file [structs.h](#).

3.60.2.45 MaxParLevel

int M_const::MaxParLevel

Definition at line 1447 of file [structs.h](#).

3.60.2.46 OutBufSize

```
int M_const::OutBufSize
```

Definition at line 1448 of file [structs.h](#).

3.60.2.47 SMaxFpatches

```
int M_const::SMaxFpatches
```

Definition at line 1449 of file [structs.h](#).

3.60.2.48 SMaxPatches

```
int M_const::SMaxPatches
```

Definition at line 1450 of file [structs.h](#).

3.60.2.49 StdOut

```
int M_const::StdOut
```

Definition at line 1451 of file [structs.h](#).

3.60.2.50 ginsidefirst

```
int M_const::ginsidefirst
```

Definition at line 1452 of file [structs.h](#).

3.60.2.51 gDefDim

```
int M_const::gDefDim
```

Definition at line 1453 of file [structs.h](#).

3.60.2.52 gDefDim4

```
int M_const::gDefDim4
```

Definition at line 1454 of file [structs.h](#).

3.60.2.53 NumFixedSets

```
int M_const::NumFixedSets
```

Definition at line 1455 of file [structs.h](#).

3.60.2.54 NumFixedFunctions

```
int M_const::NumFixedFunctions
```

Definition at line [1456](#) of file [structs.h](#).

3.60.2.55 rbufnum

```
int M_const::rbufnum
```

Definition at line [1457](#) of file [structs.h](#).

3.60.2.56 dbufnum

```
int M_const::dbufnum
```

Definition at line [1458](#) of file [structs.h](#).

3.60.2.57 sbufnum

```
int M_const::sbufnum
```

Definition at line [1459](#) of file [structs.h](#).

3.60.2.58 zbufnum

```
int M_const::zbufnum
```

Definition at line [1460](#) of file [structs.h](#).

3.60.2.59 SkipClears

```
int M_const::SkipClears
```

Definition at line [1461](#) of file [structs.h](#).

3.60.2.60 gTokensWriteFlag

```
int M_const::gTokensWriteFlag
```

Definition at line [1462](#) of file [structs.h](#).

3.60.2.61 gfunpowers

```
int M_const::gfunpowers
```

Definition at line [1463](#) of file [structs.h](#).

3.60.2.62 gStatsFlag

```
int M_const::gStatsFlag
```

Definition at line 1464 of file [structs.h](#).

3.60.2.63 gNamesFlag

```
int M_const::gNamesFlag
```

Definition at line 1465 of file [structs.h](#).

3.60.2.64 gCodesFlag

```
int M_const::gCodesFlag
```

Definition at line 1466 of file [structs.h](#).

3.60.2.65 gSortType

```
int M_const::gSortType
```

Definition at line 1467 of file [structs.h](#).

3.60.2.66 gproperorderflag

```
int M_const::gproperorderflag
```

Definition at line 1468 of file [structs.h](#).

3.60.2.67 hparallelflag

```
int M_const::hparallelflag
```

Definition at line 1469 of file [structs.h](#).

3.60.2.68 gparallelflag

```
int M_const::gparallelflag
```

Definition at line 1470 of file [structs.h](#).

3.60.2.69 totalnumberofthreads

```
int M_const::totalnumberofthreads
```

Definition at line 1471 of file [structs.h](#).

3.60.2.70 gSizeCommuteInSet

```
int M_const::gSizeCommuteInSet
```

Definition at line 1472 of file [structs.h](#).

3.60.2.71 gThreadStats

```
int M_const::gThreadStats
```

Definition at line 1473 of file [structs.h](#).

3.60.2.72 ggThreadStats

```
int M_const::ggThreadStats
```

Definition at line 1474 of file [structs.h](#).

3.60.2.73 gFinalStats

```
int M_const::gFinalStats
```

Definition at line 1475 of file [structs.h](#).

3.60.2.74 ggFinalStats

```
int M_const::ggFinalStats
```

Definition at line 1476 of file [structs.h](#).

3.60.2.75 gThreadsFlag

```
int M_const::gThreadsFlag
```

Definition at line 1477 of file [structs.h](#).

3.60.2.76 ggThreadsFlag

```
int M_const::ggThreadsFlag
```

Definition at line 1478 of file [structs.h](#).

3.60.2.77 gThreadBalancing

```
int M_const::gThreadBalancing
```

Definition at line 1479 of file [structs.h](#).

3.60.2.78 ggThreadBalancing

```
int M_const::ggThreadBalancing
```

Definition at line 1480 of file [structs.h](#).

3.60.2.79 gThreadSortFileSynch

```
int M_const::gThreadSortFileSynch
```

Definition at line 1481 of file [structs.h](#).

3.60.2.80 ggThreadSortFileSynch

```
int M_const::ggThreadSortFileSynch
```

Definition at line 1482 of file [structs.h](#).

3.60.2.81 gProcessStats

```
int M_const::gProcessStats
```

Definition at line 1483 of file [structs.h](#).

3.60.2.82 ggProcessStats

```
int M_const::ggProcessStats
```

Definition at line 1484 of file [structs.h](#).

3.60.2.83 gOldParallelStats

```
int M_const::gOldParallelStats
```

Definition at line 1485 of file [structs.h](#).

3.60.2.84 ggOldParallelStats

```
int M_const::ggOldParallelStats
```

Definition at line 1486 of file [structs.h](#).

3.60.2.85 maxFlevels

```
int M_const::maxFlevels
```

Definition at line 1487 of file [structs.h](#).

3.60.2.86 resetTimeOnClear

```
int M_const::resetTimeOnClear
```

Definition at line 1488 of file [structs.h](#).

3.60.2.87 gcNumDollars

```
int M_const::gcNumDollars
```

Definition at line 1489 of file [structs.h](#).

3.60.2.88 MultiRun

```
int M_const::MultiRun
```

Definition at line 1490 of file [structs.h](#).

3.60.2.89 gNoSpacesInNumbers

```
int M_const::gNoSpacesInNumbers
```

Definition at line 1491 of file [structs.h](#).

3.60.2.90 ggNoSpacesInNumbers

```
int M_const::ggNoSpacesInNumbers
```

Definition at line 1492 of file [structs.h](#).

3.60.2.91 gIsFortran90

```
int M_const::gIsFortran90
```

Definition at line 1493 of file [structs.h](#).

3.60.2.92 PrintTotalSize

```
int M_const::PrintTotalSize
```

Definition at line 1494 of file [structs.h](#).

3.60.2.93 fbufferSize

```
int M_const::fbufferSize
```

Definition at line 1495 of file [structs.h](#).

3.60.2.94 gOldFactArgFlag

```
int M_const::gOldFactArgFlag
```

Definition at line 1496 of file [structs.h](#).

3.60.2.95 ggOldFactArgFlag

```
int M_const::ggOldFactArgFlag
```

Definition at line 1497 of file [structs.h](#).

3.60.2.96 gnumextrasym

```
int M_const::gnumextrasym
```

Definition at line 1498 of file [structs.h](#).

3.60.2.97 ggnumextrasym

```
int M_const::ggnumextrasym
```

Definition at line 1499 of file [structs.h](#).

3.60.2.98 NumSpectatorFiles

```
int M_const::NumSpectatorFiles
```

Definition at line 1500 of file [structs.h](#).

3.60.2.99 SizeForSpectatorFiles

```
int M_const::SizeForSpectatorFiles
```

Definition at line 1501 of file [structs.h](#).

3.60.2.100 gOldGCDflag

```
int M_const::gOldGCDflag
```

Definition at line 1502 of file [structs.h](#).

3.60.2.101 ggOldGCDflag

```
int M_const::ggOldGCDflag
```

Definition at line 1503 of file [structs.h](#).

3.60.2.102 gWTimeStatsFlag

```
int M_const::gWTimeStatsFlag
```

Definition at line 1504 of file [structs.h](#).

3.60.2.103 ggWTimeStatsFlag

```
int M_const::ggWTimeStatsFlag
```

Definition at line 1505 of file [structs.h](#).

3.60.2.104 jumpratio

```
int M_const::jumpratio
```

Definition at line 1506 of file [structs.h](#).

3.60.2.105 MaxTal

```
WORD M_const::MaxTal
```

Definition at line 1507 of file [structs.h](#).

3.60.2.106 IndDum

```
WORD M_const::IndDum
```

Definition at line 1508 of file [structs.h](#).

3.60.2.107 DumInd

```
WORD M_const::DumInd
```

Definition at line 1509 of file [structs.h](#).

3.60.2.108 Willnd

```
WORD M_const::WilInd
```

Definition at line 1510 of file [structs.h](#).

3.60.2.109 gncmod

```
WORD M_const::gncmod
```

Definition at line 1511 of file [structs.h](#).

3.60.2.110 gnpowmod

WORD M_const::gnpowmod

Definition at line 1512 of file [structs.h](#).

3.60.2.111 gmodmode

WORD M_const::gmodmode

Definition at line 1513 of file [structs.h](#).

3.60.2.112 gUnitTrace

WORD M_const::gUnitTrace

Definition at line 1514 of file [structs.h](#).

3.60.2.113 gOutputMode

WORD M_const::gOutputMode

Definition at line 1515 of file [structs.h](#).

3.60.2.114 gOutputSpaces

WORD M_const::gOutputSpaces

Definition at line 1516 of file [structs.h](#).

3.60.2.115 gOutNumberType

WORD M_const::gOutNumberType

Definition at line 1517 of file [structs.h](#).

3.60.2.116 gCnumpows

WORD M_const::gCnumpows

Definition at line 1518 of file [structs.h](#).

3.60.2.117 gUniTrace

WORD M_const::gUniTrace[4]

Definition at line 1519 of file [structs.h](#).

3.60.2.118 MaxWildcards

WORD M_const::MaxWildcards

Definition at line 1520 of file [structs.h](#).

3.60.2.119 mTraceDum

WORD M_const::mTraceDum

Definition at line 1521 of file [structs.h](#).

3.60.2.120 OffsetIndex

WORD M_const::OffsetIndex

Definition at line 1522 of file [structs.h](#).

3.60.2.121 OffsetVector

WORD M_const::OffsetVector

Definition at line 1523 of file [structs.h](#).

3.60.2.122 RepMax

WORD M_const::RepMax

Definition at line 1524 of file [structs.h](#).

3.60.2.123 LogType

WORD M_const::LogType

Definition at line 1525 of file [structs.h](#).

3.60.2.124 ggStatsFlag

WORD M_const::ggStatsFlag

Definition at line 1526 of file [structs.h](#).

3.60.2.125 gLineLength

WORD M_const::gLineLength

Definition at line 1527 of file [structs.h](#).

3.60.2.126 qError

WORD M_const::qError

Definition at line 1528 of file [structs.h](#).

3.60.2.127 FortranCont

WORD M_const::FortranCont

Definition at line 1529 of file [structs.h](#).

3.60.2.128 HoldFlag

WORD M_const::HoldFlag

Definition at line 1530 of file [structs.h](#).

3.60.2.129 Ordering

WORD M_const::Ordering[15]

Definition at line 1531 of file [structs.h](#).

3.60.2.130 silent

WORD M_const::silent

Definition at line 1532 of file [structs.h](#).

3.60.2.131 tracebackflag

WORD M_const::tracebackflag

Definition at line 1533 of file [structs.h](#).

3.60.2.132 expnum

WORD M_const::expnum

Definition at line 1534 of file [structs.h](#).

3.60.2.133 denomnum

WORD M_const::denomnum

Definition at line 1535 of file [structs.h](#).

3.60.2.134 facnum

WORD M_const::facnum

Definition at line 1536 of file [structs.h](#).

3.60.2.135 invfacnum

WORD M_const::invfacnum

Definition at line 1537 of file [structs.h](#).

3.60.2.136 sumnum

WORD M_const::sumnum

Definition at line 1538 of file [structs.h](#).

3.60.2.137 sumpnum

WORD M_const::sumpnum

Definition at line 1539 of file [structs.h](#).

3.60.2.138 OldOrderFlag

WORD M_const::OldOrderFlag

Definition at line 1540 of file [structs.h](#).

3.60.2.139 termfunnum

WORD M_const::termfunnum

Definition at line 1541 of file [structs.h](#).

3.60.2.140 matchfunnum

WORD M_const::matchfunnum

Definition at line 1542 of file [structs.h](#).

3.60.2.141 countfunnum

WORD M_const::countfunnum

Definition at line 1543 of file [structs.h](#).

3.60.2.142 gPolyFun

WORD M_const::gPolyFun

Definition at line 1544 of file [structs.h](#).

3.60.2.143 gPolyFunInv

WORD M_const::gPolyFunInv

Definition at line 1545 of file [structs.h](#).

3.60.2.144 gPolyFunType

WORD M_const::gPolyFunType

Definition at line 1546 of file [structs.h](#).

3.60.2.145 gPolyFunExp

WORD M_const::gPolyFunExp

Definition at line 1547 of file [structs.h](#).

3.60.2.146 gPolyFunVar

WORD M_const::gPolyFunVar

Definition at line 1548 of file [structs.h](#).

3.60.2.147 gPolyFunPow

WORD M_const::gPolyFunPow

Definition at line 1549 of file [structs.h](#).

3.60.2.148 dollarzero

WORD M_const::dollarzero

Definition at line 1550 of file [structs.h](#).

3.60.2.149 atstartup

WORD M_const::atstartup

Definition at line 1551 of file [structs.h](#).

3.60.2.150 exitflag

WORD M_const::exitflag

Definition at line 1552 of file [structs.h](#).

3.60.2.151 NumStoreCaches

WORD M_const::NumStoreCaches

Definition at line 1553 of file [structs.h](#).

3.60.2.152 gIndentSpace

WORD M_const::gIndentSpace

Definition at line 1554 of file [structs.h](#).

3.60.2.153 ggIndentSpace

WORD M_const::ggIndentSpace

Definition at line 1555 of file [structs.h](#).

3.60.2.154 gShortStatsMax

WORD M_const::gShortStatsMax

For On FewerStatistics 10;

Definition at line 1556 of file [structs.h](#).

3.60.2.155 ggShortStatsMax

WORD M_const::ggShortStatsMax

For On FewerStatistics 10;

Definition at line 1557 of file [structs.h](#).

3.60.2.156 gextrasymbols

WORD M_const::gextrasymbols

Definition at line 1558 of file [structs.h](#).

3.60.2.157 ggextrasympols

WORD M_const::ggextrasympols

Definition at line 1559 of file [structs.h](#).

3.60.2.158 zerorhs

WORD M_const::zerorhs

Definition at line 1560 of file [structs.h](#).

3.60.2.159 onerhs

WORD M_const::onerhs

Definition at line 1561 of file [structs.h](#).

3.60.2.160 havesortdir

WORD M_const::havesortdir

Definition at line 1562 of file [structs.h](#).

3.60.2.161 vectorzero

WORD M_const::vectorzero

Definition at line 1563 of file [structs.h](#).

3.60.2.162 ClearStore

WORD M_const::ClearStore

Definition at line 1564 of file [structs.h](#).

3.60.2.163 BracketFactors

WORD M_const::BracketFactors[8]

Definition at line 1565 of file [structs.h](#).

3.60.2.164 FromStdin

BOOL M_const::FromStdin

Definition at line 1566 of file [structs.h](#).

3.60.2.165 IgnoreDeprecation

```
BOOL M_const::IgnoreDeprecation
```

Definition at line 1567 of file [structs.h](#).

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

- [structs.h](#)

3.61 MCBLOCK Class Reference

Public Member Functions

- void [init](#) (void)

Data Fields

- Edge2n * [edges](#)
- int [nedges](#)
- int [nartps](#)
- int [loop](#)
- int [DUMMYPADDING](#)

3.61.1 Detailed Description

Definition at line 985 of file [grcc.h](#).

3.61.2 Constructor & Destructor Documentation

3.61.2.1 MCBLOCK()

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

Definition at line 5511 of file [grcc.cc](#).

3.61.2.2 ~MCBlock()

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

Definition at line 5517 of file [grcc.cc](#).

3.61.3 Member Function Documentation

3.61.3.1 init()

```
void MCBLOCK::init (
    void )
```

Definition at line 5522 of file [grcc.cc](#).

3.61.4 Field Documentation

3.61.4.1 edges

```
Edge2n* MCBLOCK::edges
```

Definition at line 987 of file [grcc.h](#).

3.61.4.2 nmedges

```
int MCBLOCK::nmedges
```

Definition at line 988 of file [grcc.h](#).

3.61.4.3 nartps

```
int MCBLOCK::nartps
```

Definition at line 989 of file [grcc.h](#).

3.61.4.4 loop

```
int MCBLOCK::loop
```

Definition at line 990 of file [grcc.h](#).

3.61.4.5 DUMMYPADDING

```
int MCBLOCK::DUMMYPADDING
```

Definition at line 992 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.62 MCBridge Class Reference

Data Fields

- Edge2n [nodes](#)
- int [next](#)

3.62.1 Detailed Description

Definition at line [973](#) of file [grcc.h](#).

3.62.2 Constructor & Destructor Documentation

3.62.2.1 MCBridge()

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

Definition at line [5499](#) of file [grcc.cc](#).

3.62.2.2 ~MCBridge()

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

Definition at line [5504](#) of file [grcc.cc](#).

3.62.3 Field Documentation

3.62.3.1 nodes

```
Edge2n MCBridge::nodes
```

Definition at line [975](#) of file [grcc.h](#).

3.62.3.2 next

```
int MCBridge::next
```

Definition at line [976](#) of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.63 MCEdge Class Reference

Data Fields

- Edge2n [nodes](#)
- ULong [momset](#)
- int [momdir](#)
- int [DUMMYPADDING](#)

3.63.1 Detailed Description

Definition at line [936](#) of file [grcc.h](#).

3.63.2 Constructor & Destructor Documentation

3.63.2.1 MCEdge()

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

Definition at line [5458](#) of file [grcc.cc](#).

3.63.2.2 ~MCEdge()

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

Definition at line [5464](#) of file [grcc.cc](#).

3.63.3 Field Documentation

3.63.3.1 nodes

```
Edge2n MCEdge::nodes
```

Definition at line [938](#) of file [grcc.h](#).

3.63.3.2 momset

```
ULong MCEdge::momset
```

Definition at line [939](#) of file [grcc.h](#).

3.63.3.3 momdir

```
int MCEdge::momdir
```

Definition at line 940 of file [grcc.h](#).

3.63.3.4 DUMMYPADDING

```
int MCEdge::DUMMYPADDING
```

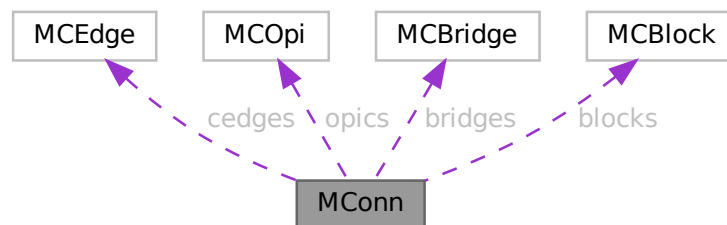
Definition at line 942 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.64 MConn Class Reference

Collaboration diagram for MConn:



Public Member Functions

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

Data Fields

- [MCEdge](#) * [cedges](#)
- [MCOpi](#) * [opics](#)
- [MCBridge](#) * [bridges](#)
- [MCBlock](#) * [blocks](#)
- int * [articuls](#)
- int * [opisp](#)
- int * [opistk](#)
- Edge2n * [blksp](#)
- Edge2n * [blkstk](#)
- int [snodes](#)
- int [sedges](#)
- int [nopic](#)
- int [nlpopic](#)
- int [nctopic](#)
- int [nbacked](#)
- int [nbridges](#)
- int [ne0bridges](#)
- int [ne1bridges](#)
- int [nselfloops](#)
- int [nmultiedges](#)
- int [nblocks](#)
- int [na1blocks](#)
- int [narticuls](#)
- int [neblocks](#)
- int [nopisp](#)
- int [opistkptr](#)
- int [nblksp](#)
- int [blkstkptr](#)
- int [DUMMYPADDING](#)

3.64.1 Detailed Description

Definition at line [1002](#) of file [grcc.h](#).

3.64.2 Constructor & Destructor Documentation**3.64.2.1 MConn()**

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

Definition at line [5533](#) of file [grcc.cc](#).

3.64.2.2 ~MConn()

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

Definition at line [5556](#) of file [grcc.cc](#).

3.64.3 Member Function Documentation

3.64.3.1 init()

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

Definition at line 5571 of file [grcc.cc](#).

3.64.3.2 initCEdges()

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

Definition at line 5628 of file [grcc.cc](#).

3.64.3.3 pushNode()

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

Definition at line 5602 of file [grcc.cc](#).

3.64.3.4 pushEdge()

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

Definition at line 5611 of file [grcc.cc](#).

3.64.3.5 addCEdge()

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

Definition at line 5654 of file [grcc.cc](#).

3.64.3.6 addOPlc()

```
void MConn::addOPlc (
    MOpI * mopi,
    int stp )
```

Definition at line 5678 of file [grcc.cc](#).

3.64.3.7 addBridge()

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

Definition at line 5705 of file [grcc.cc](#).

3.64.3.8 addArtic()

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

Definition at line 5722 of file [grcc.cc](#).

3.64.3.9 addBlock()

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

Definition at line 5733 of file [grcc.cc](#).

3.64.3.10 addBlockSelf()

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

Definition at line 5757 of file [grcc.cc](#).

3.64.3.11 print()

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

Definition at line 5775 of file [grcc.cc](#).

3.64.3.12 prEdges()

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

Definition at line 5848 of file [grcc.cc](#).

3.64.4 Field Documentation

3.64.4.1 cedges

`MCEdge*` `MConn::cedges`

Definition at line 1005 of file [grcc.h](#).

3.64.4.2 opics

`MCOp*` `MConn::opics`

Definition at line 1006 of file [grcc.h](#).

3.64.4.3 bridges

`MCBridge*` `MConn::bridges`

Definition at line 1007 of file [grcc.h](#).

3.64.4.4 blocks

`MCBlock*` `MConn::blocks`

Definition at line 1008 of file [grcc.h](#).

3.64.4.5 articuls

`int*` `MConn::articuls`

Definition at line 1009 of file [grcc.h](#).

3.64.4.6 opisp

`int*` `MConn::opisp`

Definition at line 1011 of file [grcc.h](#).

3.64.4.7 opistk

`int*` `MConn::opistk`

Definition at line 1012 of file [grcc.h](#).

3.64.4.8 blksp

Edge2n* MConn::blksp

Definition at line 1013 of file [grcc.h](#).

3.64.4.9 blkstk

Edge2n* MConn::blkstk

Definition at line 1014 of file [grcc.h](#).

3.64.4.10 snodes

int MConn::snodes

Definition at line 1015 of file [grcc.h](#).

3.64.4.11 sedges

int MConn::sedges

Definition at line 1016 of file [grcc.h](#).

3.64.4.12 nopic

int MConn::nopic

Definition at line 1019 of file [grcc.h](#).

3.64.4.13 nlpopic

int MConn::nlpopic

Definition at line 1020 of file [grcc.h](#).

3.64.4.14 nctopic

int MConn::nctopic

Definition at line 1021 of file [grcc.h](#).

3.64.4.15 nbacked

int MConn::nbacked

Definition at line 1024 of file [grcc.h](#).

3.64.4.16 nbridges

```
int MConn::nbridges
```

Definition at line 1027 of file [grcc.h](#).

3.64.4.17 ne0bridges

```
int MConn::ne0bridges
```

Definition at line 1028 of file [grcc.h](#).

3.64.4.18 ne1bridges

```
int MConn::ne1bridges
```

Definition at line 1029 of file [grcc.h](#).

3.64.4.19 nselfloops

```
int MConn::nselfloops
```

Definition at line 1030 of file [grcc.h](#).

3.64.4.20 nmultiedges

```
int MConn::nmultiedges
```

Definition at line 1031 of file [grcc.h](#).

3.64.4.21 nblocks

```
int MConn::nblocks
```

Definition at line 1034 of file [grcc.h](#).

3.64.4.22 na1blocks

```
int MConn::na1blocks
```

Definition at line 1035 of file [grcc.h](#).

3.64.4.23 narticuls

```
int MConn::narticuls
```

Definition at line 1036 of file [grcc.h](#).

3.64.4.24 neblocks

```
int MConn::neblocks
```

Definition at line 1037 of file [grcc.h](#).

3.64.4.25 nopisp

```
int MConn::nopisp
```

Definition at line 1040 of file [grcc.h](#).

3.64.4.26 opistkptr

```
int MConn::opistkptr
```

Definition at line 1041 of file [grcc.h](#).

3.64.4.27 nblksp

```
int MConn::nblksp
```

Definition at line 1042 of file [grcc.h](#).

3.64.4.28 blkstkptr

```
int MConn::blkstkptr
```

Definition at line 1043 of file [grcc.h](#).

3.64.4.29 DUMMY_PADDING

```
int MConn::DUMMY_PADDING
```

Definition at line 1045 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.65 MCOpI Class Reference

Public Member Functions

- void [init](#) (void)

Data Fields

- int * [nodes](#)
- int [nnodes](#)
- int [nlegs](#)
- int [next](#)
- int [nedges](#)
- int [loop](#)
- int [ctloop](#)
- int [mom0lg](#)
- int [DUMMY_PADDING](#)

3.65.1 Detailed Description

Definition at line [951](#) of file [grcc.h](#).

3.65.2 Constructor & Destructor Documentation

3.65.2.1 MCOpI()

```
MCOpI::MCOpI (  
    void )
```

Definition at line [5472](#) of file [grcc.cc](#).

3.65.2.2 ~MCOpI()

```
MCOpI::~~MCOpI (  
    void )
```

Definition at line [5478](#) of file [grcc.cc](#).

3.65.3 Member Function Documentation

3.65.3.1 init()

```
void MCOpI::init (  
    void )
```

Definition at line [5484](#) of file [grcc.cc](#).

3.65.4 Field Documentation

3.65.4.1 nodes

```
int* MCOpI::nodes
```

Definition at line [953](#) of file [grcc.h](#).

3.65.4.2 nnodes

```
int MCOpI::nnodes
```

Definition at line 954 of file [grcc.h](#).

3.65.4.3 nlegs

```
int MCOpI::nlegs
```

Definition at line 955 of file [grcc.h](#).

3.65.4.4 next

```
int MCOpI::next
```

Definition at line 956 of file [grcc.h](#).

3.65.4.5 nedges

```
int MCOpI::nedges
```

Definition at line 957 of file [grcc.h](#).

3.65.4.6 loop

```
int MCOpI::loop
```

Definition at line 958 of file [grcc.h](#).

3.65.4.7 ctloop

```
int MCOpI::ctloop
```

Definition at line 959 of file [grcc.h](#).

3.65.4.8 mom0lg

```
int MCOpI::mom0lg
```

Definition at line 960 of file [grcc.h](#).

3.65.4.9 DUMMYPADDING

```
int MCOpi::DUMMYPADDING
```

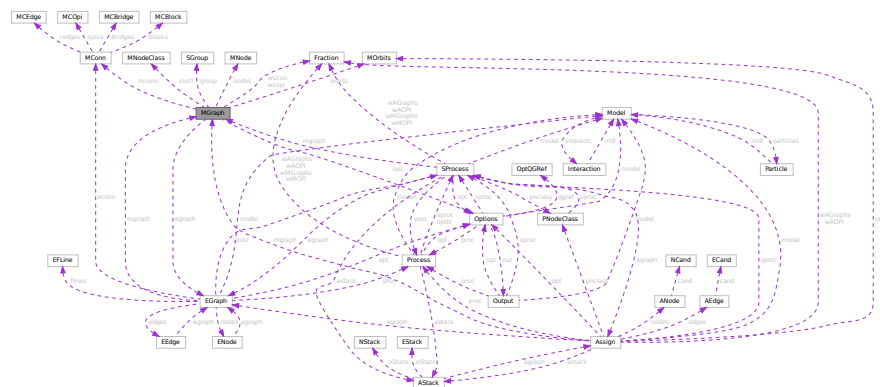
Definition at line 962 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.66 MGraph Class Reference

Collaboration diagram for MGraph:



Public Member Functions

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

Data Fields

- [Options](#) * [opt](#)
- [SGroup](#) * [group](#)
- [MOrbits](#) * [orbits](#)
- [MNode](#) ** [nodes](#)
- [BigInt](#) [mld](#)
- [int](#) * [clist](#)
- [int](#) [pld](#)
- [int](#) [nNodes](#)
- [int](#) [nEdges](#)
- [int](#) [nLoops](#)
- [int](#) [nExtern](#)
- [int](#) [nClasses](#)
- [int](#) [mindeg](#)
- [int](#) [maxdeg](#)
- [int](#) ** [adjMat](#)
- [MNodeClass](#) * [curcl](#)
- [EGraph](#) * [egraph](#)
- [BigInt](#) [nsym](#)
- [BigInt](#) [esym](#)
- [BigInt](#) [cDiag](#)
- [BigInt](#) [c1PI](#)
- [BigInt](#) [cNoTadpole](#)
- [BigInt](#) [cNoTadBlock](#)
- [BigInt](#) [c1PINoTadBlock](#)
- [Fraction](#) [wscon](#)
- [Fraction](#) [wsopi](#)
- [BigInt](#) [ngen](#)
- [BigInt](#) [ngconn](#)
- [BigInt](#) [nCallRefine](#)
- [BigInt](#) [discardRefine](#)
- [BigInt](#) [discardDisc](#)
- [BigInt](#) [discardIso](#)
- [Bool](#) [opi](#)
- [Bool](#) [opiloop](#)
- [Bool](#) [extself](#)
- [Bool](#) [selfloop](#)
- [Bool](#) [multiedge](#)
- [Bool](#) [tadpole](#)
- [Bool](#) [tadblock](#)
- [Bool](#) [block](#)
- [Bool](#) [bipart](#)
- [int](#) [DUMMYPADDING1](#)
- [MConn](#) * [mconn](#)
- [int](#) ** [modmat](#)
- [int](#) * [bidef](#)
- [int](#) * [bilow](#)
- [int](#) * [bicol](#)
- [int](#) [bicount](#)
- [int](#) [DUMMYPADDING2](#)

3.66.1 Detailed Description

Definition at line 793 of file [grcc.h](#).

3.66.2 Constructor & Destructor Documentation

3.66.2.1 MGraph() [1/2]

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

Definition at line 4073 of file [grcc.cc](#).

3.66.2.2 MGraph() [2/2]

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

Definition at line 4005 of file [grcc.cc](#).

3.66.2.3 ~MGraph()

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

Definition at line 4189 of file [grcc.cc](#).

3.66.3 Member Function Documentation

3.66.3.1 init()

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

Definition at line 4145 of file [grcc.cc](#).

3.66.3.2 generate()

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

Definition at line 4960 of file [grcc.cc](#).

3.66.3.3 isExternal()

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

Definition at line 876 of file [grcc.h](#).

3.66.3.4 printAdjMat()

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

Definition at line 4219 of file [grcc.cc](#).

3.66.3.5 print()

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

Definition at line 4238 of file [grcc.cc](#).

3.66.3.6 printPy()

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

Definition at line 4262 of file [grcc.cc](#).

3.66.3.7 isConnected()

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

Definition at line 4293 of file [grcc.cc](#).

3.66.3.8 visit()

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

Definition at line 4317 of file [grcc.cc](#).

3.66.3.9 isIsomorphic()

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

Definition at line 4342 of file [grcc.cc](#).

3.66.3.10 permMat()

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

Definition at line 4400 of file [grcc.cc](#).

3.66.3.11 compMat()

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

Definition at line 4414 of file [grcc.cc](#).

3.66.3.12 refineClass()

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

Definition at line 4432 of file [grcc.cc](#).

3.66.3.13 bisearchME()

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

Definition at line 4639 of file [grcc.cc](#).

3.66.3.14 biconnME()

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

Definition at line 4514 of file [grcc.cc](#).

3.66.3.15 isOptM()

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

Definition at line 5171 of file [grcc.cc](#).

3.66.3.16 connectClass()

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

Definition at line 4979 of file [grcc.cc](#).

3.66.3.17 connectNode()

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

Definition at line 5007 of file [grcc.cc](#).

3.66.3.18 connectLeg()

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

Definition at line 5024 of file [grcc.cc](#).

3.66.3.19 newGraph()

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

Definition at line 5237 of file [grcc.cc](#).

3.66.4 Field Documentation

3.66.4.1 opt

[Options*](#) MGraph::opt

Definition at line 798 of file [grcc.h](#).

3.66.4.2 group

`SGroup* MGraph::group`

Definition at line 801 of file [grcc.h](#).

3.66.4.3 orbits

`MOrbits* MGraph::orbits`

Definition at line 802 of file [grcc.h](#).

3.66.4.4 nodes

`MNode** MGraph::nodes`

Definition at line 804 of file [grcc.h](#).

3.66.4.5 mId

`BigInt MGraph::mId`

Definition at line 805 of file [grcc.h](#).

3.66.4.6 clist

`int* MGraph::clist`

Definition at line 806 of file [grcc.h](#).

3.66.4.7 pId

`int MGraph::pId`

Definition at line 807 of file [grcc.h](#).

3.66.4.8 nNodes

`int MGraph::nNodes`

Definition at line 808 of file [grcc.h](#).

3.66.4.9 nEdges

`int MGraph::nEdges`

Definition at line 809 of file [grcc.h](#).

3.66.4.10 nLoops

```
int MGraph::nLoops
```

Definition at line 810 of file [grcc.h](#).

3.66.4.11 nExtern

```
int MGraph::nExtern
```

Definition at line 811 of file [grcc.h](#).

3.66.4.12 nClasses

```
int MGraph::nClasses
```

Definition at line 812 of file [grcc.h](#).

3.66.4.13 mindeg

```
int MGraph::mindeg
```

Definition at line 813 of file [grcc.h](#).

3.66.4.14 maxdeg

```
int MGraph::maxdeg
```

Definition at line 814 of file [grcc.h](#).

3.66.4.15 adjMat

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

Definition at line 817 of file [grcc.h](#).

3.66.4.16 curcl

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

Definition at line 818 of file [grcc.h](#).

3.66.4.17 egraph

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

Definition at line 819 of file [grcc.h](#).

3.66.4.18 nsym

`BigInt MGraph::nsym`

Definition at line 820 of file [grcc.h](#).

3.66.4.19 esym

`BigInt MGraph::esym`

Definition at line 821 of file [grcc.h](#).

3.66.4.20 cDiag

`BigInt MGraph::cDiag`

Definition at line 824 of file [grcc.h](#).

3.66.4.21 c1PI

`BigInt MGraph::c1PI`

Definition at line 825 of file [grcc.h](#).

3.66.4.22 cNoTadpole

`BigInt MGraph::cNoTadpole`

Definition at line 826 of file [grcc.h](#).

3.66.4.23 cNoTadBlock

`BigInt MGraph::cNoTadBlock`

Definition at line 827 of file [grcc.h](#).

3.66.4.24 c1PINoTadBlock

`BigInt MGraph::c1PINoTadBlock`

Definition at line 828 of file [grcc.h](#).

3.66.4.25 wscon

`Fraction MGraph::wscon`

Definition at line 829 of file [grcc.h](#).

3.66.4.26 wsopi

`Fraction MGraph::wsopi`

Definition at line 830 of file [grcc.h](#).

3.66.4.27 ngen

`BigInt MGraph::ngen`

Definition at line 833 of file [grcc.h](#).

3.66.4.28 ngconn

`BigInt MGraph::ngconn`

Definition at line 834 of file [grcc.h](#).

3.66.4.29 nCallRefine

`BigInt MGraph::nCallRefine`

Definition at line 836 of file [grcc.h](#).

3.66.4.30 discardRefine

`BigInt MGraph::discardRefine`

Definition at line 837 of file [grcc.h](#).

3.66.4.31 discardDisc

`BigInt MGraph::discardDisc`

Definition at line 838 of file [grcc.h](#).

3.66.4.32 discardIso

`BigInt MGraph::discardIso`

Definition at line 839 of file [grcc.h](#).

3.66.4.33 opi

`Bool MGraph::opi`

Definition at line 842 of file [grcc.h](#).

3.66.4.34 opiloop

Bool MGraph::opiloop

Definition at line 843 of file [grcc.h](#).

3.66.4.35 extself

Bool MGraph::extself

Definition at line 844 of file [grcc.h](#).

3.66.4.36 selfloop

Bool MGraph::selfloop

Definition at line 845 of file [grcc.h](#).

3.66.4.37 multiedge

Bool MGraph::multiedge

Definition at line 846 of file [grcc.h](#).

3.66.4.38 tadpole

Bool MGraph::tadpole

Definition at line 847 of file [grcc.h](#).

3.66.4.39 tadblocK

Bool MGraph::tadblocK

Definition at line 848 of file [grcc.h](#).

3.66.4.40 block

Bool MGraph::block

Definition at line 849 of file [grcc.h](#).

3.66.4.41 bipart

Bool MGraph::bipart

Definition at line 850 of file [grcc.h](#).

3.66.4.42 DUMMYPADDING1

```
int MGraph::DUMMYPADDING1
```

Definition at line 852 of file [grcc.h](#).

3.66.4.43 mconn

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

Definition at line 855 of file [grcc.h](#).

3.66.4.44 modmat

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

Definition at line 858 of file [grcc.h](#).

3.66.4.45 bidef

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

Definition at line 861 of file [grcc.h](#).

3.66.4.46 bilow

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

Definition at line 862 of file [grcc.h](#).

3.66.4.47 bicol

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

Definition at line 863 of file [grcc.h](#).

3.66.4.48 bicount

```
int MGraph::bicount
```

Definition at line 864 of file [grcc.h](#).

3.66.4.49 DUMMYPADDING2

```
int MGraph::DUMMYPADDING2
```

Definition at line 866 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.67 MiNmAx Struct Reference

Data Fields

- WORD [mini](#)
- WORD [maxi](#)
- WORD [size](#)

3.67.1 Detailed Description

Definition at line 301 of file [structs.h](#).

3.67.2 Field Documentation

3.67.2.1 mini

```
WORD MiNmAx::mini
```

Minimum value

Definition at line 302 of file [structs.h](#).

3.67.2.2 maxi

```
WORD MiNmAx::maxi
```

Maximum value

Definition at line 303 of file [structs.h](#).

3.67.2.3 size

```
WORD MinMax::size
```

Value of one unit in this position.

Definition at line 304 of file [structs.h](#).

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

- [structs.h](#)

3.68 MInput Struct Reference

Data Fields

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

3.68.1 Detailed Description

Definition at line 175 of file [grccparam.h](#).

3.68.2 Field Documentation

3.68.2.1 name

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

Definition at line 176 of file [grccparam.h](#).

3.68.2.2 defpart

```
int MInput::defpart
```

Definition at line 177 of file [grccparam.h](#).

3.68.2.3 ncouple

```
int MInput::ncouple
```

Definition at line 179 of file [grccparam.h](#).

3.68.2.4 cnamlist

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

Definition at line 180 of file [grccparam.h](#).

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

- [grccparam.h](#)

3.69 MNode Class Reference

Public Member Functions

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

Data Fields

- int [id](#)
- int [deg](#)
- int [clss](#)
- int [extloop](#)
- int [cminddeg](#)
- int [cmaxdeg](#)
- int [freelg](#)
- int [visited](#)

3.69.1 Detailed Description

Definition at line 771 of file [grcc.h](#).

3.69.2 Constructor & Destructor Documentation

3.69.2.1 MNode() [1/2]

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

Definition at line 3967 of file [grcc.cc](#).

3.69.2.2 MNode() [2/2]

```
MNode::MNode (
    int id,
    int deg,
    int extloop,
    int cls,
    int cmin,
    int cmax )
```

Definition at line [3985](#) of file [grcc.cc](#).

3.69.3 Field Documentation

3.69.3.1 id

```
int MNode::id
```

Definition at line [773](#) of file [grcc.h](#).

3.69.3.2 deg

```
int MNode::deg
```

Definition at line [774](#) of file [grcc.h](#).

3.69.3.3 cls

```
int MNode::cls
```

Definition at line [775](#) of file [grcc.h](#).

3.69.3.4 extloop

```
int MNode::extloop
```

Definition at line [776](#) of file [grcc.h](#).

3.69.3.5 cmindeg

```
int MNode::cmindeg
```

Definition at line [777](#) of file [grcc.h](#).

3.69.3.6 cmaxdeg

```
int MNode::cmaxdeg
```

Definition at line 778 of file [grcc.h](#).

3.69.3.7 freelg

```
int MNode::freelg
```

Definition at line 780 of file [grcc.h](#).

3.69.3.8 visited

```
int MNode::visited
```

Definition at line 781 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.70 MNodeClass Class Reference

Public Member Functions

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

Data Fields

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

3.70.1 Detailed Description

Definition at line 900 of file [grcc.h](#).

3.70.2 Constructor & Destructor Documentation

3.70.2.1 MNodeClass()

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

Definition at line 3720 of file [grcc.cc](#).

3.70.2.2 ~MNodeClass()

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

Definition at line 3749 of file [grcc.cc](#).

3.70.3 Member Function Documentation

3.70.3.1 init()

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

Definition at line 3756 of file [grcc.cc](#).

3.70.3.2 copy()

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

Definition at line 3780 of file [grcc.cc](#).

3.70.3.3 clCmp()

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

Definition at line 3836 of file [grcc.cc](#).

3.70.3.4 printMat()

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

Definition at line [3887](#) of file [grcc.cc](#).

3.70.3.5 mkFlist()

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

Definition at line [3804](#) of file [grcc.cc](#).

3.70.3.6 mkNdCl()

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

Definition at line [3820](#) of file [grcc.cc](#).

3.70.3.7 mkClMat()

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

Definition at line [3858](#) of file [grcc.cc](#).

3.70.3.8 incMat()

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

Definition at line [3878](#) of file [grcc.cc](#).

3.70.3.9 cmpMNCArray()

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

Definition at line [3918](#) of file [grcc.cc](#).

3.70.3.10 reorder()

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

Definition at line 3934 of file [grcc.cc](#).

3.70.4 Field Documentation

3.70.4.1 clmat

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

Definition at line 902 of file [grcc.h](#).

3.70.4.2 clist

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

Definition at line 903 of file [grcc.h](#).

3.70.4.3 ndcl

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

Definition at line 904 of file [grcc.h](#).

3.70.4.4 flist

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

Definition at line 905 of file [grcc.h](#).

3.70.4.5 clord

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

Definition at line 906 of file [grcc.h](#).

3.70.4.6 cmindeg

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

Definition at line 907 of file [grcc.h](#).

3.70.4.7 cmaxdeg

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

Definition at line 908 of file [grcc.h](#).

3.70.4.8 nNodes

```
int MNodeClass::nNodes
```

Definition at line 909 of file [grcc.h](#).

3.70.4.9 nClasses

```
int MNodeClass::nClasses
```

Definition at line 910 of file [grcc.h](#).

3.70.4.10 maxdeg

```
int MNodeClass::maxdeg
```

Definition at line 911 of file [grcc.h](#).

3.70.4.11 flg0

```
int MNodeClass::flg0
```

Definition at line 913 of file [grcc.h](#).

3.70.4.12 flg1

```
int MNodeClass::flg1
```

Definition at line 914 of file [grcc.h](#).

3.70.4.13 flg2

```
int MNodeClass::flg2
```

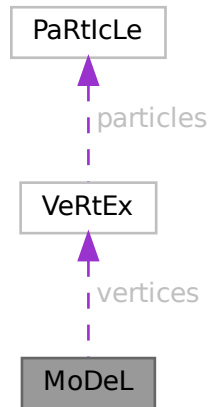
Definition at line 915 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.71 MoDeL Struct Reference

Collaboration diagram for MoDeL:



Data Fields

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

3.71.1 Detailed Description

Definition at line 637 of file [structs.h](#).

3.71.2 Field Documentation

3.71.2.1 vertices

VERTEX** MoDeL::vertices

Definition at line 638 of file [structs.h](#).

3.71.2.2 couplings

`WORD* MoDeL::couplings`

Definition at line 639 of file [structs.h](#).

3.71.2.3 name

`UBYTE* MoDeL::name`

Definition at line 640 of file [structs.h](#).

3.71.2.4 grccmodel

`void* MoDeL::grccmodel`

Definition at line 641 of file [structs.h](#).

3.71.2.5 legcouple

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

Definition at line 642 of file [structs.h](#).

3.71.2.6 nparticles

`WORD MoDeL::nparticles`

Definition at line 643 of file [structs.h](#).

3.71.2.7 nvertices

`WORD MoDeL::nvertices`

Definition at line 644 of file [structs.h](#).

3.71.2.8 invertices

`WORD MoDeL::invertices`

Definition at line 645 of file [structs.h](#).

3.71.2.9 sizevertices

`WORD MoDeL::sizevertices`

Definition at line 646 of file [structs.h](#).

3.71.2.10 sizecouplings

WORD MoDeL::sizecouplings

Definition at line 647 of file [structs.h](#).

3.71.2.11 ncouplings

WORD MoDeL::ncouplings

Definition at line 648 of file [structs.h](#).

3.71.2.12 error

WORD MoDeL::error

Definition at line 649 of file [structs.h](#).

3.71.2.13 dummy

WORD MoDeL::dummy

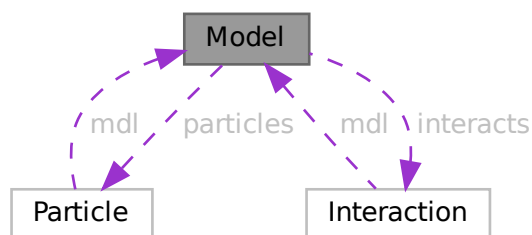
Definition at line 650 of file [structs.h](#).

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

- [structs.h](#)

3.72 Model Class Reference

Collaboration diagram for Model:



Public Member Functions

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

Static Public Member Functions

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

Data Fields

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

3.72.1 Detailed Description

Definition at line 284 of file [grcc.h](#).

3.72.2 Constructor & Destructor Documentation

3.72.2.1 Model()

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

Definition at line 1979 of file [grcc.cc](#).

3.72.2.2 ~Model()

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

Definition at line 2040 of file [grcc.cc](#).

3.72.3 Member Function Documentation

3.72.3.1 prModel()

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

Definition at line 2082 of file [grcc.cc](#).

3.72.3.2 addParticle()

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

Definition at line 2131 of file [grcc.cc](#).

3.72.3.3 addParticleEnd()

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

Definition at line 2173 of file [grcc.cc](#).

3.72.3.4 addInteraction()

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

Definition at line 2211 of file [grcc.cc](#).

3.72.3.5 addInteractionEnd()

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

Definition at line 2339 of file [grcc.cc](#).

3.72.3.6 findParticleName()

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

Definition at line 2409 of file [grcc.cc](#).

3.72.3.7 findParticleCode()

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

Definition at line 2427 of file [grcc.cc](#).

3.72.3.8 findInteractionName()

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

Definition at line 2546 of file [grcc.cc](#).

3.72.3.9 findInteractionCode()

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

Definition at line 2559 of file [grcc.cc](#).

3.72.3.10 particleName()

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

Definition at line 2442 of file [grcc.cc](#).

3.72.3.11 particleCode()

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

Definition at line [2462](#) of file [grcc.cc](#).

3.72.3.12 normalParticle()

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

Definition at line [2508](#) of file [grcc.cc](#).

3.72.3.13 antiParticle()

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

Definition at line [2525](#) of file [grcc.cc](#).

3.72.3.14 allParticles()

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

Definition at line [2539](#) of file [grcc.cc](#).

3.72.3.15 findMClass()

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

Definition at line [2495](#) of file [grcc.cc](#).

3.72.3.16 prParticleArray()

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

Definition at line [2480](#) of file [grcc.cc](#).

3.72.3.17 printMInput()

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

Definition at line 2572 of file [grcc.cc](#).

3.72.3.18 printPInput()

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

Definition at line 2592 of file [grcc.cc](#).

3.72.3.19 printIInput()

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

Definition at line 2600 of file [grcc.cc](#).

3.72.4 Field Documentation

3.72.4.1 name

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

Definition at line 286 of file [grcc.h](#).

3.72.4.2 cnlist

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

Definition at line 287 of file [grcc.h](#).

3.72.4.3 ncouple

```
int Model::ncouple
```

Definition at line 288 of file [grcc.h](#).

3.72.4.4 nParticles

```
int Model::nParticles
```

Definition at line 290 of file [grcc.h](#).

3.72.4.5 particles

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

Definition at line 291 of file [grcc.h](#).

3.72.4.6 pdef

```
int Model::pdef
```

Definition at line 292 of file [grcc.h](#).

3.72.4.7 nallPart

```
int Model::nallPart
```

Definition at line 295 of file [grcc.h](#).

3.72.4.8 allPart

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

Definition at line 296 of file [grcc.h](#).

3.72.4.9 nintPart

```
int Model::nintPart
```

Definition at line 299 of file [grcc.h](#).

3.72.4.10 intPart

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

Definition at line 300 of file [grcc.h](#).

3.72.4.11 nInteracts

```
int Model::nInteracts
```

Definition at line 302 of file [grcc.h](#).

3.72.4.12 interacts

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

Definition at line 303 of file [grcc.h](#).

3.72.4.13 vdef

```
int Model::vdef
```

Definition at line 304 of file [grcc.h](#).

3.72.4.14 maxnlegs

```
int Model::maxnlegs
```

Definition at line 306 of file [grcc.h](#).

3.72.4.15 maxcpl

```
int Model::maxcpl
```

Definition at line 307 of file [grcc.h](#).

3.72.4.16 maxloop

```
int Model::maxloop
```

Definition at line 308 of file [grcc.h](#).

3.72.4.17 cplgcp

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

Definition at line 311 of file [grcc.h](#).

3.72.4.18 cplglg

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

Definition at line 312 of file [grcc.h](#).

3.72.4.19 cplgnvl

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

Definition at line 313 of file [grcc.h](#).

3.72.4.20 cplgvl

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

Definition at line 314 of file [grcc.h](#).

3.72.4.21 ncplgcp

```
int Model::ncplgcp
```

Definition at line 315 of file [grcc.h](#).

3.72.4.22 defpart

```
int Model::defpart
```

Definition at line 317 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.73 MODNUM Struct Reference

Data Fields

- UWORD * [a](#)
- UWORD * [m](#)
- WORD [na](#)
- WORD [nm](#)

3.73.1 Detailed Description

Definition at line 1268 of file [structs.h](#).

3.73.2 Field Documentation

3.73.2.1 a

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

Definition at line 1269 of file [structs.h](#).

3.73.2.2 m

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

Definition at line 1270 of file [structs.h](#).

3.73.2.3 na

WORD MODNUM::na

Definition at line 1271 of file [structs.h](#).

3.73.2.4 nm

WORD MODNUM::nm

Definition at line 1272 of file [structs.h](#).

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

- [structs.h](#)

3.74 MoDoPtDoLIaRS Struct Reference

Data Fields

- WORD [number](#)
- WORD [type](#)

3.74.1 Detailed Description

Definition at line 562 of file [structs.h](#).

3.74.2 Field Documentation

3.74.2.1 number

WORD MoDoPtDoLIaRS::number

Definition at line 566 of file [structs.h](#).

3.74.2.2 type

WORD MoDoPtDoLIaRS::type

Definition at line 567 of file [structs.h](#).

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

- [structs.h](#)

3.75 monomial_larger Class Reference

Public Member Functions

- bool [operator\(\)](#) (const WORD *a, const WORD *b)

3.75.1 Detailed Description

Definition at line 177 of file [poly.h](#).

3.75.2 Constructor & Destructor Documentation

3.75.2.1 monomial_larger()

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

Definition at line 182 of file [poly.h](#).

3.75.3 Member Function Documentation

3.75.3.1 operator()

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

Definition at line 188 of file [poly.h](#).

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

- [poly.h](#)

3.76 MOrbits Class Reference

Public Member Functions

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

Data Fields

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

3.76.1 Detailed Description

Definition at line [1067](#) of file [grcc.h](#).

3.76.2 Constructor & Destructor Documentation

3.76.2.1 MOrbits()

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

Definition at line [5355](#) of file [grcc.cc](#).

3.76.2.2 ~MOrbits()

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

Definition at line [5362](#) of file [grcc.cc](#).

3.76.3 Member Function Documentation

3.76.3.1 print()

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

Definition at line [5367](#) of file [grcc.cc](#).

3.76.3.2 initPerm()

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

Definition at line [5380](#) of file [grcc.cc](#).

3.76.3.3 fromPerm()

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

Definition at line [5391](#) of file [grcc.cc](#).

3.76.3.4 toOrbits()

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

Definition at line 5430 of file [grcc.cc](#).

3.76.4 Field Documentation

3.76.4.1 nOrbits

```
int MOrbits::nOrbits
```

Definition at line 1079 of file [grcc.h](#).

3.76.4.2 nNodes

```
int MOrbits::nNodes
```

Definition at line 1080 of file [grcc.h](#).

3.76.4.3 nd2or

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

Definition at line 1081 of file [grcc.h](#).

3.76.4.4 or2nd

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

Definition at line 1082 of file [grcc.h](#).

3.76.4.5 flist

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

Definition at line 1083 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.77 flint::mpoly Class Reference

Public Member Functions

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

Data Fields

- fmpz_mpoly_t [d](#)

3.77.1 Detailed Description

Definition at line [102](#) of file [flintinterface.h](#).

3.77.2 Constructor & Destructor Documentation

3.77.2.1 mpoly()

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

Definition at line [107](#) of file [flintinterface.h](#).

3.77.2.2 ~mpoly()

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

Definition at line [108](#) of file [flintinterface.h](#).

3.77.3 Member Function Documentation

3.77.3.1 print()

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

Definition at line [109](#) of file [flintinterface.h](#).

3.77.4 Field Documentation

3.77.4.1 d

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

Definition at line [106](#) of file [flintinterface.h](#).

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

- [flintinterface.h](#)

3.78 flint::mpoly_ctx Class Reference

Public Member Functions

- [mpoly_ctx](#) (int64_t nvars)

Data Fields

- [fmpz_mpoly_ctx_t d](#)

3.78.1 Detailed Description

Definition at line 125 of file [flintinterface.h](#).

3.78.2 Constructor & Destructor Documentation

3.78.2.1 mpoly_ctx()

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

Definition at line 128 of file [flintinterface.h](#).

3.78.2.2 ~mpoly_ctx()

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

Definition at line 129 of file [flintinterface.h](#).

3.78.3 Field Documentation

3.78.3.1 d

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

Definition at line 127 of file [flintinterface.h](#).

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

- [flintinterface.h](#)

3.79 flint::mpoly_factor Class Reference

Public Member Functions

- [mpoly_factor](#) (fmpz_mpoly_ctx_struct *ctx_in)

Data Fields

- `fmprz_mpoly_factor_t` [d](#)

3.79.1 Detailed Description

Definition at line [115](#) of file [flintinterface.h](#).

3.79.2 Constructor & Destructor Documentation

3.79.2.1 `mpoly_factor()`

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

Definition at line [120](#) of file [flintinterface.h](#).

3.79.2.2 `~mpoly_factor()`

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

Definition at line [123](#) of file [flintinterface.h](#).

3.79.3 Field Documentation

3.79.3.1 `d`

```
fmprz_mpoly_factor_t flint::mpoly_factor::d
```

Definition at line [119](#) of file [flintinterface.h](#).

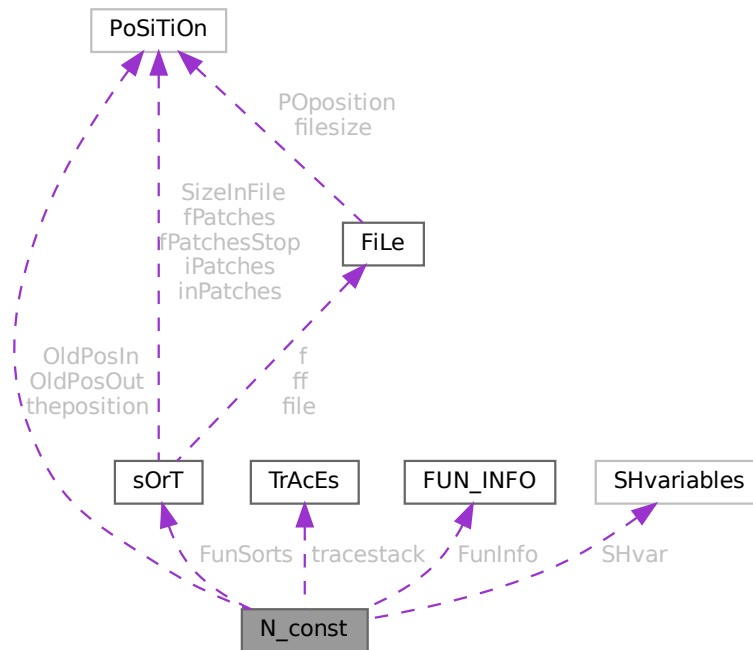
The documentation for this class was generated from the following file:

- [flintinterface.h](#)

3.80 N_const Struct Reference

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

Collaboration diagram for N_const:



Data Fields

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

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

- int [numfuninfo](#)
- int [NumFunSorts](#)
- int [MaxFunSorts](#)
- int [arglistsize](#)
- int [tlistsize](#)
- int [filenum](#)
- int [compressSize](#)
- int [polysortflag](#)
- int [nogroundlevel](#)
- int [subsubveto](#)
- WORD [MaxRenumScratch](#)
- WORD [oldtype](#)
- WORD [oldvalue](#)
- WORD [NumWild](#)
- WORD [RepFunNum](#)
- WORD [DisOrderFlag](#)
- WORD [WildDirt](#)
- WORD [NumFound](#)
- WORD [WildReserve](#)
- WORD [TelInFun](#)
- WORD [TeSuOut](#)
- WORD [WildArgs](#)
- WORD [WildEat](#)
- WORD [PolyNormFlag](#)
- WORD [PolyFunTodo](#)
- WORD [sizeselecttermundo](#)
- WORD [patternbuffersize](#)
- WORD [numlistingprint](#)
- WORD [ncmod](#)
- WORD [ExpectedSign](#)
- WORD [SignCheck](#)
- WORD [IndDum](#)
- WORD [poly_num_vars](#)
- WORD [idfunctionflag](#)
- WORD [poly_vars_type](#)
- WORD [tryterm](#)

3.80.1 Detailed Description

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

Definition at line [2169](#) of file [structs.h](#).

3.80.2 Field Documentation

3.80.2.1 OldPosIn

[POSITION](#) [N_const::OldPosIn](#)

Definition at line [2170](#) of file [structs.h](#).

3.80.2.2 OldPosOut

`POSITION N_const::OldPosOut`

Definition at line 2171 of file [structs.h](#).

3.80.2.3 theposition

`POSITION N_const::theposition`

Definition at line 2172 of file [structs.h](#).

3.80.2.4 EndNest

`WORD* N_const::EndNest`

Definition at line 2173 of file [structs.h](#).

3.80.2.5 Frozen

`WORD* N_const::Frozen`

Definition at line 2174 of file [structs.h](#).

3.80.2.6 FullProto

`WORD* N_const::FullProto`

Definition at line 2175 of file [structs.h](#).

3.80.2.7 cTerm

`WORD* N_const::cTerm`

Definition at line 2176 of file [structs.h](#).

3.80.2.8 RepPoint

`int* N_const::RepPoint`

Definition at line 2177 of file [structs.h](#).

3.80.2.9 WildValue

`WORD* N_const::WildValue`

Definition at line 2178 of file [structs.h](#).

3.80.2.10 WildStop

WORD* N_const::WildStop

Definition at line 2179 of file [structs.h](#).

3.80.2.11 argaddress

WORD* N_const::argaddress

Definition at line 2180 of file [structs.h](#).

3.80.2.12 RepFunList

WORD* N_const::RepFunList

Definition at line 2181 of file [structs.h](#).

3.80.2.13 patstop

WORD* N_const::patstop

Definition at line 2182 of file [structs.h](#).

3.80.2.14 terstop

WORD* N_const::terstop

Definition at line 2183 of file [structs.h](#).

3.80.2.15 terstart

WORD* N_const::terstart

Definition at line 2184 of file [structs.h](#).

3.80.2.16 terfirstcomm

WORD* N_const::terfirstcomm

Definition at line 2185 of file [structs.h](#).

3.80.2.17 DumFound

WORD* N_const::DumFound

Definition at line 2186 of file [structs.h](#).

3.80.2.18 DumPlace

WORD** N_const::DumPlace

Definition at line 2187 of file [structs.h](#).

3.80.2.19 DumFunPlace

WORD** N_const::DumFunPlace

Definition at line 2188 of file [structs.h](#).

3.80.2.20 UsedSymbol

WORD* N_const::UsedSymbol

Definition at line 2189 of file [structs.h](#).

3.80.2.21 UsedVector

WORD* N_const::UsedVector

Definition at line 2190 of file [structs.h](#).

3.80.2.22 UsedIndex

WORD* N_const::UsedIndex

Definition at line 2191 of file [structs.h](#).

3.80.2.23 UsedFunction

WORD* N_const::UsedFunction

Definition at line 2192 of file [structs.h](#).

3.80.2.24 MaskPointer

WORD* N_const::MaskPointer

Definition at line 2193 of file [structs.h](#).

3.80.2.25 ForFindOnly

WORD* N_const::ForFindOnly

Definition at line 2194 of file [structs.h](#).

3.80.2.26 findTerm

WORD* N_const::findTerm

Definition at line 2195 of file [structs.h](#).

3.80.2.27 findPattern

WORD* N_const::findPattern

Definition at line 2196 of file [structs.h](#).

3.80.2.28 dummyrenumlist

WORD* N_const::dummyrenumlist

Definition at line 2201 of file [structs.h](#).

3.80.2.29 funargs

int* N_const::funargs

Definition at line 2202 of file [structs.h](#).

3.80.2.30 funlocs

WORD** N_const::funlocs

Definition at line 2203 of file [structs.h](#).

3.80.2.31 funinds

int* N_const::funinds

Definition at line 2204 of file [structs.h](#).

3.80.2.32 NoScrat2

UWORD* N_const::NoScrat2

Definition at line 2205 of file [structs.h](#).

3.80.2.33 ReplaceScrat

WORD* N_const::ReplaceScrat

Definition at line 2206 of file [structs.h](#).

3.80.2.34 **tracestack**

`TRACES* N_const::tracestack`

Definition at line 2207 of file [structs.h](#).

3.80.2.35 **selecttermundo**

`WORD* N_const::selecttermundo`

Definition at line 2208 of file [structs.h](#).

3.80.2.36 **patternbuffer**

`WORD* N_const::patternbuffer`

Definition at line 2209 of file [structs.h](#).

3.80.2.37 **termbuffer**

`WORD* N_const::termbuffer`

Definition at line 2210 of file [structs.h](#).

3.80.2.38 **PoinScratch**

`WORD** N_const::PoinScratch`

Definition at line 2211 of file [structs.h](#).

3.80.2.39 **FunScratch**

`WORD** N_const::FunScratch`

Definition at line 2212 of file [structs.h](#).

3.80.2.40 **RenumScratch**

`WORD* N_const::RenumScratch`

Definition at line 2213 of file [structs.h](#).

3.80.2.41 **FunInfo**

`FUN_INFO* N_const::FunInfo`

Definition at line 2214 of file [structs.h](#).

3.80.2.42 SplitScratch

WORD** N_const::SplitScratch

Definition at line 2215 of file [structs.h](#).

3.80.2.43 SplitScratch1

WORD** N_const::SplitScratch1

Definition at line 2216 of file [structs.h](#).

3.80.2.44 FunSorts

[SORTING](#)** N_const::FunSorts

Definition at line 2217 of file [structs.h](#).

3.80.2.45 SoScratC

UWORD* N_const::SoScratC

Definition at line 2218 of file [structs.h](#).

3.80.2.46 listinprint

WORD* N_const::listinprint

Definition at line 2219 of file [structs.h](#).

3.80.2.47 currentTerm

WORD* N_const::currentTerm

Definition at line 2220 of file [structs.h](#).

3.80.2.48 arglist

WORD** N_const::arglist

Definition at line 2221 of file [structs.h](#).

3.80.2.49 tlistbuf

int* N_const::tlistbuf

Definition at line 2222 of file [structs.h](#).

3.80.2.50 compressSpace

WORD* N_const::compressSpace

Definition at line 2226 of file [structs.h](#).

3.80.2.51 SHcombi

UWORD* N_const::SHcombi

Definition at line 2231 of file [structs.h](#).

3.80.2.52 poly_vars

WORD* N_const::poly_vars

Definition at line 2232 of file [structs.h](#).

3.80.2.53 cmod

UWORD* N_const::cmod

Definition at line 2233 of file [structs.h](#).

3.80.2.54 SHvar

[SHvariables](#) N_const::SHvar

Definition at line 2234 of file [structs.h](#).

3.80.2.55 deferskipped

LONG N_const::deferskipped

Definition at line 2235 of file [structs.h](#).

3.80.2.56 InScratch

LONG N_const::InScratch

Definition at line 2236 of file [structs.h](#).

3.80.2.57 SplitScratchSize

LONG N_const::SplitScratchSize

Definition at line 2237 of file [structs.h](#).

3.80.2.58 InScratch1

```
LONG N_const::InScratch1
```

Definition at line 2238 of file [structs.h](#).

3.80.2.59 SplitScratchSize1

```
LONG N_const::SplitScratchSize1
```

Definition at line 2239 of file [structs.h](#).

3.80.2.60 ninterms

```
LONG N_const::ninterms
```

Definition at line 2240 of file [structs.h](#).

3.80.2.61 SHcombisize

```
LONG N_const::SHcombisize
```

Definition at line 2249 of file [structs.h](#).

3.80.2.62 NumTotWildArgs

```
int N_const::NumTotWildArgs
```

Definition at line 2250 of file [structs.h](#).

3.80.2.63 UseFindOnly

```
int N_const::UseFindOnly
```

Definition at line 2251 of file [structs.h](#).

3.80.2.64 UsedOtherFind

```
int N_const::UsedOtherFind
```

Definition at line 2252 of file [structs.h](#).

3.80.2.65 ErrorInDollar

```
int N_const::ErrorInDollar
```

Definition at line 2253 of file [structs.h](#).

3.80.2.66 numfargs

```
int N_const::numfargs
```

Definition at line [2254](#) of file [structs.h](#).

3.80.2.67 numflocs

```
int N_const::numflocs
```

Definition at line [2255](#) of file [structs.h](#).

3.80.2.68 nargs

```
int N_const::nargs
```

Definition at line [2256](#) of file [structs.h](#).

3.80.2.69 tohunt

```
int N_const::tohunt
```

Definition at line [2257](#) of file [structs.h](#).

3.80.2.70 numoffuns

```
int N_const::numoffuns
```

Definition at line [2258](#) of file [structs.h](#).

3.80.2.71 funisize

```
int N_const::funisize
```

Definition at line [2259](#) of file [structs.h](#).

3.80.2.72 RSize

```
int N_const::RSize
```

Definition at line [2260](#) of file [structs.h](#).

3.80.2.73 numtracesctack

```
int N_const::numtracesctack
```

Definition at line [2261](#) of file [structs.h](#).

3.80.2.74 intracestack

```
int N_const::intracestack
```

Definition at line 2262 of file [structs.h](#).

3.80.2.75 numfuninfo

```
int N_const::numfuninfo
```

Definition at line 2263 of file [structs.h](#).

3.80.2.76 NumFunSorts

```
int N_const::NumFunSorts
```

Definition at line 2264 of file [structs.h](#).

3.80.2.77 MaxFunSorts

```
int N_const::MaxFunSorts
```

Definition at line 2265 of file [structs.h](#).

3.80.2.78 arglistsize

```
int N_const::arglistsize
```

Definition at line 2266 of file [structs.h](#).

3.80.2.79 tlistsize

```
int N_const::tlistsize
```

Definition at line 2267 of file [structs.h](#).

3.80.2.80 filenum

```
int N_const::filenum
```

Definition at line 2268 of file [structs.h](#).

3.80.2.81 compressSize

```
int N_const::compressSize
```

Definition at line 2269 of file [structs.h](#).

3.80.2.82 polysortflag

```
int N_const::polysortflag
```

Definition at line 2270 of file [structs.h](#).

3.80.2.83 nogroundlevel

```
int N_const::nogroundlevel
```

Definition at line 2271 of file [structs.h](#).

3.80.2.84 subsubveto

```
int N_const::subsubveto
```

Definition at line 2272 of file [structs.h](#).

3.80.2.85 MaxRenumScratch

```
WORD N_const::MaxRenumScratch
```

Definition at line 2273 of file [structs.h](#).

3.80.2.86 oldtype

```
WORD N_const::oldtype
```

Definition at line 2274 of file [structs.h](#).

3.80.2.87 oldvalue

```
WORD N_const::oldvalue
```

Definition at line 2275 of file [structs.h](#).

3.80.2.88 NumWild

```
WORD N_const::NumWild
```

Definition at line 2276 of file [structs.h](#).

3.80.2.89 RepFunNum

```
WORD N_const::RepFunNum
```

Definition at line 2277 of file [structs.h](#).

3.80.2.90 DisOrderFlag

WORD N_const::DisOrderFlag

Definition at line 2278 of file [structs.h](#).

3.80.2.91 WildDirt

WORD N_const::WildDirt

Definition at line 2279 of file [structs.h](#).

3.80.2.92 NumFound

WORD N_const::NumFound

Definition at line 2280 of file [structs.h](#).

3.80.2.93 WildReserve

WORD N_const::WildReserve

Definition at line 2281 of file [structs.h](#).

3.80.2.94 TeInFun

WORD N_const::TeInFun

Definition at line 2282 of file [structs.h](#).

3.80.2.95 TeSuOut

WORD N_const::TeSuOut

Definition at line 2283 of file [structs.h](#).

3.80.2.96 WildArgs

WORD N_const::WildArgs

Definition at line 2284 of file [structs.h](#).

3.80.2.97 WildEat

WORD N_const::WildEat

Definition at line 2285 of file [structs.h](#).

3.80.2.98 PolyNormFlag

WORD N_const::PolyNormFlag

Definition at line 2286 of file [structs.h](#).

3.80.2.99 PolyFunTodo

WORD N_const::PolyFunTodo

Definition at line 2287 of file [structs.h](#).

3.80.2.100 sizeselecttermundo

WORD N_const::sizeselecttermundo

Definition at line 2288 of file [structs.h](#).

3.80.2.101 patternbufferize

WORD N_const::patternbufferize

Definition at line 2289 of file [structs.h](#).

3.80.2.102 numlistinprint

WORD N_const::numlistinprint

Definition at line 2290 of file [structs.h](#).

3.80.2.103 ncmmod

WORD N_const::ncmmod

Definition at line 2291 of file [structs.h](#).

3.80.2.104 ExpectedSign

WORD N_const::ExpectedSign

Definition at line 2292 of file [structs.h](#).

3.80.2.105 SignCheck

WORD N_const::SignCheck

Used in pattern matching of antisymmetric functions

Definition at line 2293 of file [structs.h](#).

3.80.2.106 IndDum

WORD N_const::IndDum

Used in pattern matching of antisymmetric functions

Definition at line 2294 of file [structs.h](#).

3.80.2.107 poly_num_vars

WORD N_const::poly_num_vars

Definition at line 2295 of file [structs.h](#).

3.80.2.108 idfunctionflag

WORD N_const::idfunctionflag

Definition at line 2296 of file [structs.h](#).

3.80.2.109 poly_vars_type

WORD N_const::poly_vars_type

Definition at line 2297 of file [structs.h](#).

3.80.2.110 tryterm

WORD N_const::tryterm

Definition at line 2298 of file [structs.h](#).

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

- [structs.h](#)

3.81 nameblock Struct Reference

Data Fields

- MLONG [previousblock](#)
- MLONG [position](#)
- char [names](#) [NAMETABLESIZE]

3.81.1 Detailed Description

Definition at line [117](#) of file [minos.h](#).

3.81.2 Field Documentation

3.81.2.1 previousblock

```
MLONG nameblock::previousblock
```

Definition at line [118](#) of file [minos.h](#).

3.81.2.2 position

```
MLONG nameblock::position
```

Definition at line [119](#) of file [minos.h](#).

3.81.2.3 names

```
char nameblock::names[NAMETABLESIZE]
```

Definition at line [120](#) of file [minos.h](#).

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

- [minos.h](#)

3.82 NaMeNode Struct Reference

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

Data Fields

- LONG [name](#)
- WORD [parent](#)
- WORD [left](#)
- WORD [right](#)
- WORD [balance](#)
- WORD [type](#)
- WORD [number](#)

3.82.1 Detailed Description

The names of variables are kept in an array. Elements of type [NAMENODE](#) define a tree (that is kept balanced) that make it easy and fast to look for variables. See also [NAMETREE](#).

Definition at line [243](#) of file [structs.h](#).

3.82.2 Field Documentation

3.82.2.1 name

LONG [NaMeNode::name](#)

Offset into [NAMETREE::namebuffer](#).

Definition at line [244](#) of file [structs.h](#).

3.82.2.2 parent

WORD [NaMeNode::parent](#)

=-1 if no parent.

Definition at line [245](#) of file [structs.h](#).

3.82.2.3 left

WORD [NaMeNode::left](#)

=-1 if no child.

Definition at line [246](#) of file [structs.h](#).

3.82.2.4 right

WORD [NaMeNode::right](#)

=-1 if no child.

Definition at line [247](#) of file [structs.h](#).

3.82.2.5 balance

WORD NaMeNode::balance

Used for the balancing of the tree.

Definition at line 248 of file [structs.h](#).

3.82.2.6 type

WORD NaMeNode::type

Type associated with the name. See [compiler types](#).

Definition at line 249 of file [structs.h](#).

3.82.2.7 number

WORD NaMeNode::number

Number of variable in [LIST](#)'s like for example [C_const::SymbolList](#).

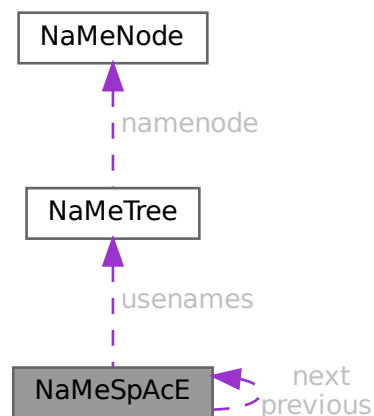
Definition at line 250 of file [structs.h](#).

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

- [structs.h](#)

3.83 NaMeSpAcE Struct Reference

Collaboration diagram for NaMeSpAcE:



Data Fields

- struct [NaMeSpAcE](#) * [previous](#)
- struct [NaMeSpAcE](#) * [next](#)
- [NAMETREE](#) * [usernames](#)
- [UBYTE](#) * [name](#)

3.83.1 Detailed Description

Definition at line [658](#) of file [structs.h](#).

3.83.2 Field Documentation

3.83.2.1 [previous](#)

```
struct NaMeSpAcE* NaMeSpAcE::previous
```

Definition at line [659](#) of file [structs.h](#).

3.83.2.2 [next](#)

```
struct NaMeSpAcE* NaMeSpAcE::next
```

Definition at line [660](#) of file [structs.h](#).

3.83.2.3 [usernames](#)

```
NAMETREE* NaMeSpAcE::usernames
```

Definition at line [661](#) of file [structs.h](#).

3.83.2.4 [name](#)

```
UBYTE* NaMeSpAcE::name
```

Definition at line [662](#) of file [structs.h](#).

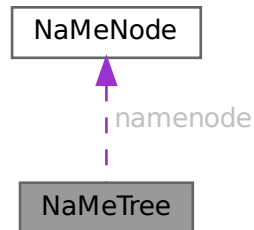
The documentation for this struct was generated from the following file:

- [structs.h](#)

3.84 NaMeTree Struct Reference

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

Collaboration diagram for NaMeTree:



Data Fields

- `NAMENODE * namenode`
- `UBYTE * namebuffer`
- `LONG nodesize`
- `LONG nodefill`
- `LONG namesize`
- `LONG namefill`
- `LONG oldnamefill`
- `LONG oldnodefill`
- `LONG globalnamefill`
- `LONG globalnodefill`
- `LONG clearnamefill`
- `LONG clearnodefill`
- `WORD headnode`

3.84.1 Detailed Description

A struct of type `NAMETREE` controls a complete (balanced) tree of names for the compiler. The compiler maintains several of such trees and the system has been set up in such a way that one could define more of them if we ever want to work with local name spaces.

Definition at line 260 of file `structs.h`.

3.84.2 Field Documentation

3.84.2.1 namenode

`NAMENODE* NaMeTree::namenode`

[D] Vector of `NAMENODE`'s. Number of elements is `nodesize`. =0 if no memory has been allocated.

Definition at line 261 of file `structs.h`.

3.84.2.2 namebuffer

```
UBYTE* NaMeTree::namebuffer
```

[D] Buffer that holds all the name strings referred to by the NAMENODE's. Allocation size is [namesize](#). =0 if no memory has been allocated.

Definition at line [263](#) of file [structs.h](#).

3.84.2.3 nodesize

```
LONG NaMeTree::nodesize
```

Maximum number of elements in [namenode](#).

Definition at line [266](#) of file [structs.h](#).

3.84.2.4 nodefill

```
LONG NaMeTree::nodefill
```

Number of currently used nodes in [namenode](#).

Definition at line [267](#) of file [structs.h](#).

3.84.2.5 namesize

```
LONG NaMeTree::namesize
```

Allocation size of [namebuffer](#) in bytes.

Definition at line [268](#) of file [structs.h](#).

3.84.2.6 namefill

```
LONG NaMeTree::namefill
```

Number of bytes occupied.

Definition at line [269](#) of file [structs.h](#).

3.84.2.7 oldnamefill

```
LONG NaMeTree::oldnamefill
```

UNUSED

Definition at line [270](#) of file [structs.h](#).

3.84.2.8 oldnodefill

```
LONG NaMeTree::oldnodefill
```

UNUSED

Definition at line 271 of file [structs.h](#).

3.84.2.9 globalnamefill

```
LONG NaMeTree::globalnamefill
```

Set by .global statement to the value of [namefill](#). When a .store command is processed, this value will be used to reset namefill.

Definition at line 272 of file [structs.h](#).

3.84.2.10 globalnodefill

```
LONG NaMeTree::globalnodefill
```

Same usage as [globalnamefill](#), but for nodefill.

Definition at line 274 of file [structs.h](#).

3.84.2.11 clearnamefill

```
LONG NaMeTree::clearnamefill
```

Marks the reset point used by the .clear statement.

Definition at line 275 of file [structs.h](#).

3.84.2.12 clearnodefill

```
LONG NaMeTree::clearnodefill
```

Marks the reset point used by the .clear statement.

Definition at line 276 of file [structs.h](#).

3.84.2.13 headnode

```
WORD NaMeTree::headnode
```

Offset in [namenode](#) of head node. ==-1 if tree is empty.

Definition at line 277 of file [structs.h](#).

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

- [structs.h](#)

3.85 NCand Class Reference

Public Member Functions

- [NCand](#) (const NCandSt *sta*, const int *dega*, const int *nilst*, int **ilst*)
- void [prNCand](#) (const char **msg*)

Data Fields

- int [deg](#)
- NCandSt [st](#)
- int [nilist](#)
- int [ilst](#) [GRCC_MAXMINTERACT]

3.85.1 Detailed Description

Definition at line [1106](#) of file [grcc.h](#).

3.85.2 Constructor & Destructor Documentation

3.85.2.1 NCand()

```
NCand::NCand (
    const NCandSt sta,
    const int dega,
    const int nilst,
    int * ilst )
```

Definition at line [8146](#) of file [grcc.cc](#).

3.85.2.2 ~NCand()

```
NCand::~NCand (
    void )
```

Definition at line [8173](#) of file [grcc.cc](#).

3.85.3 Member Function Documentation

3.85.3.1 prNCand()

```
void NCand::prNCand (
    const char * msg )
```

Definition at line [8178](#) of file [grcc.cc](#).

3.85.4 Field Documentation

3.85.4.1 deg

```
int NCand::deg
```

Definition at line 1110 of file [grcc.h](#).

3.85.4.2 st

```
NCandSt NCand::st
```

Definition at line 1111 of file [grcc.h](#).

3.85.4.3 nilist

```
int NCand::nilist
```

Definition at line 1112 of file [grcc.h](#).

3.85.4.4 ilist

```
int NCand::ilist[GRCC_MAXMINTERACT]
```

Definition at line 1113 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.86 NCInput Struct Reference

Data Fields

- int [cldeg](#)
- int [clnum](#)
- int [cltyp](#)
- int [cmind](#)
- int [cmaxd](#)
- int [ptcl](#)
- int [cple](#)

3.86.1 Detailed Description

Definition at line 205 of file [grccparam.h](#).

3.86.2 Field Documentation

3.86.2.1 cldeg

```
int NCInput::cldeg
```

Definition at line 207 of file [grccparam.h](#).

3.86.2.2 clnum

```
int NCInput::clnum
```

Definition at line 208 of file [grccparam.h](#).

3.86.2.3 cltyp

```
int NCInput::cltyp
```

Definition at line 209 of file [grccparam.h](#).

3.86.2.4 cmind

```
int NCInput::cmind
```

Definition at line 210 of file [grccparam.h](#).

3.86.2.5 cmaxd

```
int NCInput::cmaxd
```

Definition at line 211 of file [grccparam.h](#).

3.86.2.6 ptcl

```
int NCInput::ptcl
```

Definition at line 214 of file [grccparam.h](#).

3.86.2.7 cple

```
int NCInput::cple
```

Definition at line 215 of file [grccparam.h](#).

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

- [grccparam.h](#)

3.87 NeStInG Struct Reference

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

Data Fields

- WORD * [termsize](#)
- WORD * [funsize](#)
- WORD * [argsize](#)

3.87.1 Detailed Description

The NESTING struct is used when we enter the argument of functions and there is the possibility that we have to change something there. Because functions can be nested we have to keep track of all levels of functions in case we have to move the outer layers to make room for a larger function argument.

Definition at line [1033](#) of file [structs.h](#).

3.87.2 Field Documentation

3.87.2.1 termsize

```
WORD* NeStInG::termsize
```

Definition at line [1034](#) of file [structs.h](#).

3.87.2.2 funsize

```
WORD* NeStInG::funsize
```

Definition at line [1035](#) of file [structs.h](#).

3.87.2.3 argsize

```
WORD* NeStInG::argsize
```

Definition at line [1036](#) of file [structs.h](#).

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

- [structs.h](#)

3.88 NoDe Struct Reference

Collaboration diagram for NoDe:



Data Fields

- struct [NoDe](#) * [left](#)
- struct [NoDe](#) * [right](#)
- int [lloser](#)
- int [rloser](#)
- int [lsrc](#)
- int [rsrc](#)

3.88.1 Detailed Description

A node for the tree of losers in the final sorting on the master.

Definition at line [265](#) of file [parallel.c](#).

3.88.2 Field Documentation

3.88.2.1 left

```
struct NoDe* NoDe::left
```

Definition at line [266](#) of file [parallel.c](#).

3.88.2.2 right

```
struct NoDe* NoDe::right
```

Definition at line [267](#) of file [parallel.c](#).

3.88.2.3 lloser

```
int NoDe::lloser
```

Definition at line [268](#) of file [parallel.c](#).

3.88.2.4 rloser

```
int NoDe::rloser
```

Definition at line 269 of file [parallel.c](#).

3.88.2.5 lsrc

```
int NoDe::lsrc
```

Definition at line 270 of file [parallel.c](#).

3.88.2.6 rsrc

```
int NoDe::rsrc
```

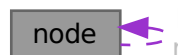
Definition at line 271 of file [parallel.c](#).

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

- [parallel.c](#)

3.89 node Struct Reference

Collaboration diagram for node:



Public Member Functions

- [node](#) (const WORD *data)
- int [cmp](#) (const struct [node](#) *rhs) const
- bool [operator\(\)](#) (const struct [node](#) *lhs, const struct [node](#) *rhs) const
- void [calcHash](#) ()

Data Fields

- const WORD * [data](#)
- struct [node](#) * [l](#)
- struct [node](#) * [r](#)
- WORD [sign](#)
- UWORD [hash](#)

3.89.1 Detailed Description

Definition at line 1495 of file [optimize.cc](#).

3.89.2 Constructor & Destructor Documentation

3.89.2.1 node() [1/2]

```
node::node ( ) [inline]
```

Definition at line 1502 of file [optimize.cc](#).

3.89.2.2 node() [2/2]

```
node::node (
    const WORD * data ) [inline]
```

Definition at line 1503 of file [optimize.cc](#).

3.89.3 Member Function Documentation

3.89.3.1 cmp()

```
int node::cmp (
    const struct node * rhs ) const [inline]
```

Definition at line 1507 of file [optimize.cc](#).

3.89.3.2 operator()()

```
bool node::operator() (
    const struct node * lhs,
    const struct node * rhs ) const [inline]
```

Definition at line 1531 of file [optimize.cc](#).

3.89.3.3 calcHash()

```
void node::calcHash ( ) [inline]
```

Definition at line 1536 of file [optimize.cc](#).

3.89.4 Field Documentation

3.89.4.1 data

```
const WORD* node::data
```

Definition at line 1496 of file [optimize.cc](#).

3.89.4.2 l

```
struct node* node::l
```

Definition at line 1497 of file [optimize.cc](#).

3.89.4.3 r

```
struct node* node::r
```

Definition at line 1498 of file [optimize.cc](#).

3.89.4.4 sign

```
WORD node::sign
```

Definition at line 1499 of file [optimize.cc](#).

3.89.4.5 hash

```
UWORD node::hash
```

Definition at line 1500 of file [optimize.cc](#).

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

- [optimize.cc](#)

3.90 NodeEq Struct Reference

Public Member Functions

- `bool operator() (const NODE *lhs, const NODE *rhs) const`

3.90.1 Detailed Description

Definition at line 1557 of file [optimize.cc](#).

3.90.2 Member Function Documentation

3.90.2.1 operator()

```
bool NodeEq::operator() (
    const NODE * lhs,
    const NODE * rhs ) const [inline]
```

Definition at line 1558 of file [optimize.cc](#).

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

- [optimize.cc](#)

3.91 NodeHash Struct Reference

Public Member Functions

- `size_t operator() (const NODE *n) const`

3.91.1 Detailed Description

Definition at line 1551 of file [optimize.cc](#).

3.91.2 Member Function Documentation

3.91.2.1 operator()

```
size_t NodeHash::operator() (
    const NODE * n ) const [inline]
```

Definition at line 1552 of file [optimize.cc](#).

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

- [optimize.cc](#)

3.92 NStack Class Reference

Public Member Functions

- `void print (const char *msg)`

Data Fields

- int [noden](#)
- int [deg](#)
- NCandSt [st](#)
- int [nilist](#)
- int [ilist](#) [GRCC_MAXMINTERACT]

3.92.1 Detailed Description

Definition at line [1332](#) of file [grcc.h](#).

3.92.2 Constructor & Destructor Documentation

3.92.2.1 NStack()

```
NStack::NStack ( ) [inline]
```

Definition at line [1342](#) of file [grcc.h](#).

3.92.2.2 ~NStack()

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

Definition at line [1343](#) of file [grcc.h](#).

3.92.3 Member Function Documentation

3.92.3.1 print()

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

Definition at line [10225](#) of file [grcc.cc](#).

3.92.4 Field Documentation

3.92.4.1 noden

```
int NStack::noden
```

Definition at line [1336](#) of file [grcc.h](#).

3.92.4.2 deg

```
int NStack::deg
```

Definition at line 1337 of file [grcc.h](#).

3.92.4.3 st

```
NCandSt NStack::st
```

Definition at line 1338 of file [grcc.h](#).

3.92.4.4 nilist

```
int NStack::nilist
```

Definition at line 1339 of file [grcc.h](#).

3.92.4.5 ilist

```
int NStack::ilist[GRCC_MAXMINTERACT]
```

Definition at line 1340 of file [grcc.h](#).

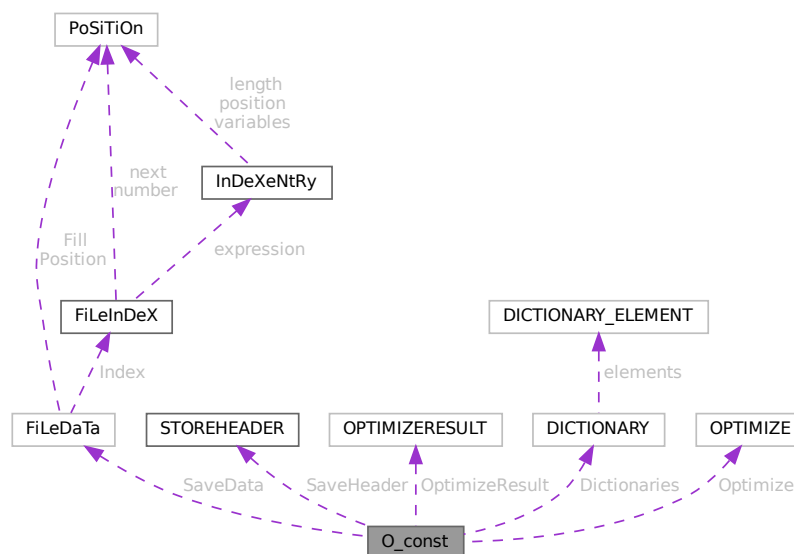
The documentation for this class was generated from the following files:

- [grcc.h](#)
- [grcc.cc](#)

3.93 O_const Struct Reference

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

Collaboration diagram for O_const:



Data Fields

- FILEDATA SaveData
- STOREHEADER SaveHeader
- OPTIMIZERESULT OptimizeResult
- UBYTE * OutputLine
- UBYTE * OutStop
- UBYTE * OutFill
- WORD * bracket
- WORD * termbuf
- WORD * tabstring
- UBYTE * wpos
- UBYTE * wpoin
- UBYTE * DollarOutBuffer
- UBYTE * CurBufWrt
- void(* FlipWORD)(UBYTE *)
- void(* FlipLONG)(UBYTE *)
- void(* FlipPOS)(UBYTE *)
- void(* FlipPOINTER)(UBYTE *)
- void(* ResizeData)(UBYTE *, int, UBYTE *, int)
- void(* ResizeWORD)(UBYTE *, UBYTE *)
- void(* ResizeNCWORD)(UBYTE *, UBYTE *)
- void(* ResizeLONG)(UBYTE *, UBYTE *)
- void(* ResizePOS)(UBYTE *, UBYTE *)
- void(* ResizePOINTER)(UBYTE *, UBYTE *)
- void(* CheckPower)(UBYTE *)
- void(* RenumberVec)(UBYTE *)
- DICTIONARY ** Dictionaries
- UBYTE * tensorList
- WORD * inscheme
- LONG NumInBrack
- LONG wlen
- LONG DollarOutSizeBuffer
- LONG DollarInOutBuffer
- OPTIMIZE Optimize
- int OutInBuffer
- int NoSpacesInNumbers
- int BlockSpaces
- int CurrentDictionary
- int SizeDictionaries
- int NumDictionaries
- int CurDictNumbers
- int CurDictVariables
- int CurDictSpecials
- int CurDictFunWithArgs
- int CurDictNumberWarning
- int CurDictNotInFunctions
- int CurDictInDollars
- int gNumDictionaries
- int IndentSpace
- WORD schemenum
- WORD transFlag
- WORD powerFlag
- WORD mpower
- WORD resizeFlag

- WORD [bufferedInd](#)
- WORD [OutSkip](#)
- WORD [IsBracket](#)
- WORD [InFbrack](#)
- WORD [PrintType](#)
- WORD [FortFirst](#)
- WORD [DoubleFlag](#)
- WORD [FactorMode](#)
- WORD [FactorNum](#)
- WORD [ErrorBlock](#)
- WORD [OptimizationLevel](#)
- UBYTE [FortDotChar](#)

3.93.1 Detailed Description

The [O_const](#) struct is part of the global data and resides in the ALLGLOBALS struct A under the name O We see it used with the macro AO as in AO.OutputLine It contains variables that involve the writing of text output and save/store files.

Definition at line [2317](#) of file [structs.h](#).

3.93.2 Field Documentation

3.93.2.1 SaveData

[FILEDATA](#) [O_const::SaveData](#)

Definition at line [2318](#) of file [structs.h](#).

3.93.2.2 SaveHeader

[STOREHEADER](#) [O_const::SaveHeader](#)

Definition at line [2319](#) of file [structs.h](#).

3.93.2.3 OptimizeResult

[OPTIMIZERESULT](#) [O_const::OptimizeResult](#)

Definition at line [2320](#) of file [structs.h](#).

3.93.2.4 OutputLine

[UBYTE*](#) [O_const::OutputLine](#)

Definition at line [2321](#) of file [structs.h](#).

3.93.2.5 OutStop

UBYTE* O_const::OutStop

Definition at line 2322 of file [structs.h](#).

3.93.2.6 OutFill

UBYTE* O_const::OutFill

Definition at line 2323 of file [structs.h](#).

3.93.2.7 bracket

WORD* O_const::bracket

Definition at line 2324 of file [structs.h](#).

3.93.2.8 termbuf

WORD* O_const::termbuf

Definition at line 2325 of file [structs.h](#).

3.93.2.9 tabstring

WORD* O_const::tabstring

Definition at line 2326 of file [structs.h](#).

3.93.2.10 wpos

UBYTE* O_const::wpos

Definition at line 2327 of file [structs.h](#).

3.93.2.11 wpoin

UBYTE* O_const::wpoin

Definition at line 2328 of file [structs.h](#).

3.93.2.12 DollarOutBuffer

UBYTE* O_const::DollarOutBuffer

Definition at line 2329 of file [structs.h](#).

3.93.2.13 CurBufWrt

```
UBYTE* O_const::CurBufWrt
```

Definition at line 2330 of file [structs.h](#).

3.93.2.14 FlipWORD

```
void(* O_const::FlipWORD) (UBYTE *)
```

Definition at line 2331 of file [structs.h](#).

3.93.2.15 FlipLONG

```
void(* O_const::FlipLONG) (UBYTE *)
```

Definition at line 2332 of file [structs.h](#).

3.93.2.16 FlipPOS

```
void(* O_const::FlipPOS) (UBYTE *)
```

Definition at line 2333 of file [structs.h](#).

3.93.2.17 FlipPOINTER

```
void(* O_const::FlipPOINTER) (UBYTE *)
```

Definition at line 2334 of file [structs.h](#).

3.93.2.18 ResizeData

```
void(* O_const::ResizeData) (UBYTE *, int, UBYTE *, int)
```

Definition at line 2335 of file [structs.h](#).

3.93.2.19 ResizeWORD

```
void(* O_const::ResizeWORD) (UBYTE *, UBYTE *)
```

Definition at line 2336 of file [structs.h](#).

3.93.2.20 ResizeNCWORD

```
void(* O_const::ResizeNCWORD) (UBYTE *, UBYTE *)
```

Definition at line 2337 of file [structs.h](#).

3.93.2.21 ResizeLONG

```
void(* O_const::ResizeLONG) (UBYTE *, UBYTE *)
```

Definition at line 2338 of file [structs.h](#).

3.93.2.22 ResizePOS

```
void(* O_const::ResizePOS) (UBYTE *, UBYTE *)
```

Definition at line 2339 of file [structs.h](#).

3.93.2.23 ResizePOINTER

```
void(* O_const::ResizePOINTER) (UBYTE *, UBYTE *)
```

Definition at line 2340 of file [structs.h](#).

3.93.2.24 CheckPower

```
void(* O_const::CheckPower) (UBYTE *)
```

Definition at line 2341 of file [structs.h](#).

3.93.2.25 RenumberVec

```
void(* O_const::ReNUMBERVec) (UBYTE *)
```

Definition at line 2342 of file [structs.h](#).

3.93.2.26 Dictionaries

```
DICTIONARY** O_const::Dictionaries
```

Definition at line 2343 of file [structs.h](#).

3.93.2.27 tensorList

```
UBYTE* O_const::tensorList
```

Definition at line 2344 of file [structs.h](#).

3.93.2.28 inscheme

```
WORD* O_const::inscheme
```

Definition at line 2345 of file [structs.h](#).

3.93.2.29 NumInBrack

```
LONG O_const::NumInBrack
```

Definition at line 2351 of file [structs.h](#).

3.93.2.30 wlen

```
LONG O_const::wlen
```

Definition at line 2352 of file [structs.h](#).

3.93.2.31 DollarOutSizeBuffer

```
LONG O_const::DollarOutSizeBuffer
```

Definition at line 2353 of file [structs.h](#).

3.93.2.32 DollarInOutBuffer

```
LONG O_const::DollarInOutBuffer
```

Definition at line 2354 of file [structs.h](#).

3.93.2.33 Optimize

```
OPTIMIZE O_const::Optimize
```

Definition at line 2359 of file [structs.h](#).

3.93.2.34 OutInBuffer

```
int O_const::OutInBuffer
```

Definition at line 2360 of file [structs.h](#).

3.93.2.35 NoSpacesInNumbers

```
int O_const::NoSpacesInNumbers
```

Definition at line 2361 of file [structs.h](#).

3.93.2.36 BlockSpaces

```
int O_const::BlockSpaces
```

Definition at line 2362 of file [structs.h](#).

3.93.2.37 CurrentDictionary

```
int O_const::CurrentDictionary
```

Definition at line [2363](#) of file [structs.h](#).

3.93.2.38 SizeDictionaries

```
int O_const::SizeDictionaries
```

Definition at line [2364](#) of file [structs.h](#).

3.93.2.39 NumDictionaries

```
int O_const::NumDictionaries
```

Definition at line [2365](#) of file [structs.h](#).

3.93.2.40 CurDictNumbers

```
int O_const::CurDictNumbers
```

Definition at line [2366](#) of file [structs.h](#).

3.93.2.41 CurDictVariables

```
int O_const::CurDictVariables
```

Definition at line [2367](#) of file [structs.h](#).

3.93.2.42 CurDictSpecials

```
int O_const::CurDictSpecials
```

Definition at line [2368](#) of file [structs.h](#).

3.93.2.43 CurDictFunWithArgs

```
int O_const::CurDictFunWithArgs
```

Definition at line [2369](#) of file [structs.h](#).

3.93.2.44 CurDictNumberWarning

```
int O_const::CurDictNumberWarning
```

Definition at line [2370](#) of file [structs.h](#).

3.93.2.45 CurDictNotInFunctions

```
int O_const::CurDictNotInFunctions
```

Definition at line 2371 of file [structs.h](#).

3.93.2.46 CurDictInDollars

```
int O_const::CurDictInDollars
```

Definition at line 2372 of file [structs.h](#).

3.93.2.47 gNumDictionaries

```
int O_const::gNumDictionaries
```

Definition at line 2373 of file [structs.h](#).

3.93.2.48 IndentSpace

```
int O_const::IndentSpace
```

Definition at line 2374 of file [structs.h](#).

3.93.2.49 schemenum

```
WORD O_const::schemenum
```

Definition at line 2378 of file [structs.h](#).

3.93.2.50 transFlag

```
WORD O_const::transFlag
```

Definition at line 2379 of file [structs.h](#).

3.93.2.51 powerFlag

```
WORD O_const::powerFlag
```

Definition at line 2380 of file [structs.h](#).

3.93.2.52 mpower

```
WORD O_const::mpower
```

Definition at line 2381 of file [structs.h](#).

3.93.2.53 **resizeFlag**

WORD O_const::resizeFlag

Definition at line 2382 of file [structs.h](#).

3.93.2.54 **bufferedInd**

WORD O_const::bufferedInd

Definition at line 2383 of file [structs.h](#).

3.93.2.55 **OutSkip**

WORD O_const::OutSkip

Definition at line 2384 of file [structs.h](#).

3.93.2.56 **IsBracket**

WORD O_const::IsBracket

Definition at line 2385 of file [structs.h](#).

3.93.2.57 **InFbrack**

WORD O_const::InFbrack

Definition at line 2386 of file [structs.h](#).

3.93.2.58 **PrintType**

WORD O_const::PrintType

Definition at line 2387 of file [structs.h](#).

3.93.2.59 **FortFirst**

WORD O_const::FortFirst

Definition at line 2388 of file [structs.h](#).

3.93.2.60 **DoubleFlag**

WORD O_const::DoubleFlag

Definition at line 2389 of file [structs.h](#).

3.93.2.61 FactorMode

WORD O_const::FactorMode

Definition at line 2390 of file [structs.h](#).

3.93.2.62 FactorNum

WORD O_const::FactorNum

Definition at line 2391 of file [structs.h](#).

3.93.2.63 ErrorBlock

WORD O_const::ErrorBlock

Definition at line 2392 of file [structs.h](#).

3.93.2.64 OptimizationLevel

WORD O_const::OptimizationLevel

Definition at line 2393 of file [structs.h](#).

3.93.2.65 FortDotChar

UBYTE O_const::FortDotChar

Definition at line 2394 of file [structs.h](#).

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

- [structs.h](#)

3.94 objects Struct Reference

Data Fields

- MLONG [position](#)
- MLONG [size](#)
- MLONG [date](#)
- MLONG [tablename](#)
- MLONG [uncompressed](#)
- MLONG [spare1](#)
- MLONG [spare2](#)
- MLONG [spare3](#)
- char [element](#) [ELEMENTSIZE]

3.94.1 Detailed Description

Definition at line 97 of file [minos.h](#).

3.94.2 Field Documentation

3.94.2.1 position

```
MLONG objects::position
```

Definition at line 99 of file [minos.h](#).

3.94.2.2 size

```
MLONG objects::size
```

Definition at line 100 of file [minos.h](#).

3.94.2.3 date

```
MLONG objects::date
```

Definition at line 101 of file [minos.h](#).

3.94.2.4 tablenumber

```
MLONG objects::tablenumber
```

Definition at line 102 of file [minos.h](#).

3.94.2.5 uncompressed

```
MLONG objects::uncompressed
```

Definition at line 103 of file [minos.h](#).

3.94.2.6 spare1

```
MLONG objects::spare1
```

Definition at line 104 of file [minos.h](#).

3.94.2.7 spare2

```
MLONG objects::spare2
```

Definition at line 105 of file [minos.h](#).

3.94.2.8 spare3

```
MLONG objects::spare3
```

Definition at line 106 of file [minos.h](#).

3.94.2.9 element

```
char objects::element[ELEMENTSIZE]
```

Definition at line 107 of file [minos.h](#).

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

- [minos.h](#)

3.95 OptDef Struct Reference

Data Fields

- const char * [name](#)
- const char * [mean](#)
- int [defaultv](#)
- int [DUMMYPADDING](#)

3.95.1 Detailed Description

Definition at line 124 of file [grccparam.h](#).

3.95.2 Field Documentation

3.95.2.1 name

```
const char* OptDef::name
```

Definition at line 125 of file [grccparam.h](#).

3.95.2.2 mean

```
const char* OptDef::mean
```

Definition at line 126 of file [grccparam.h](#).

3.95.2.3 defaultv

```
int OptDef::defaultv
```

Definition at line 127 of file [grccparam.h](#).

3.95.2.4 DUMMYPADDING

```
int OptDef::DUMMYPADDING
```

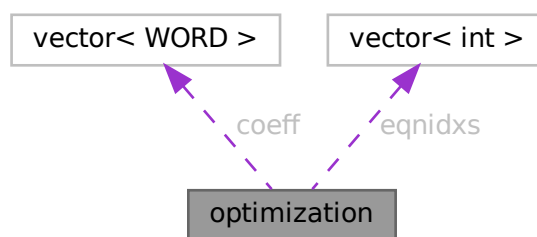
Definition at line 128 of file [grccparam.h](#).

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

- [grccparam.h](#)

3.96 optimization Class Reference

Collaboration diagram for optimization:



Public Member Functions

- `bool operator< (const optimization &a) const`

Data Fields

- int [type](#)
- int [arg1](#)
- int [arg2](#)
- int [improve](#)
- vector< WORD > [coeff](#)
- vector< int > [eqnidxs](#)

3.96.1 Detailed Description

class Optimization

3.96.2 Description

This object represents an optimization. Its type is a number in the range 0 to 5. Depending on this type, the variables `arg1`, `arg2` and `coeff` indicate:

type==0 : optimization of the form $x[\text{arg1}] \wedge \text{arg2}$ (coeff=empty) type==1 : optimization of the form $x[\text{arg1}] * x[\text{arg2}]$ (coeff=empty) type==2 : optimization of the form $x[\text{arg1}] * \text{coeff}$ (arg2=0) type==3 : optimization of the form $x[\text{arg1}] + \text{coeff}$ (arg2=0) type==4 : optimization of the form $x[\text{arg1}] + x[\text{arg2}]$ (coeff=empty) type==5 : optimization of the form $x[\text{arg1}] - x[\text{arg2}]$ (coeff=empty)

Here, "`x[arg]`" represents a symbol (if positive) or an extrasymbol (if negative). The represented symbol's id is `ABS(x[arg])-1`.

"eqns" is a list of equation, where this optimization can be performed.

"improve" is the total improvement of this optimization.

Definition at line [2669](#) of file [optimize.cc](#).

3.96.3 Member Function Documentation

3.96.3.1 operator<()

```
bool optimization::operator< (
    const optimization & a ) const [inline]
```

Definition at line [2675](#) of file [optimize.cc](#).

3.96.4 Field Documentation

3.96.4.1 type

```
int optimization::type
```

Definition at line [2671](#) of file [optimize.cc](#).

3.96.4.2 arg1

```
int optimization::arg1
```

Definition at line 2671 of file [optimize.cc](#).

3.96.4.3 arg2

```
int optimization::arg2
```

Definition at line 2671 of file [optimize.cc](#).

3.96.4.4 improve

```
int optimization::improve
```

Definition at line 2671 of file [optimize.cc](#).

3.96.4.5 coeff

```
vector<WORD> optimization::coeff
```

Definition at line 2672 of file [optimize.cc](#).

3.96.4.6 eqnidxs

```
vector<int> optimization::eqnidxs
```

Definition at line 2673 of file [optimize.cc](#).

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

- [optimize.cc](#)

3.97 OPTIMIZE Struct Reference

Data Fields

- union {
 float [fval](#)
 int [ival](#) [2]
} **mctsconstant**
- int [horner](#)
- int [hornerdirection](#)
- int [method](#)
- int [mctstimelimit](#)
- int [mctsnumexpand](#)
- int [mctsnumkeep](#)
- int [mctsnumrepeat](#)
- int [greedytimelimit](#)
- int [greedyminnum](#)
- int [greedymaxperc](#)
- int [printstats](#)
- int [debugflags](#)
- int [schemeflags](#)
- int [mctsdecaymode](#)
- int [salter](#)
- union {
 float [fval](#)
 int [ival](#) [2]
} **saMaxT**
- union {
 float [fval](#)
 int [ival](#) [2]
} **saMinT**
- int [spare](#)

3.97.1 Detailed Description

Definition at line [1279](#) of file [structs.h](#).

3.97.2 Field Documentation

3.97.2.1 fval

```
float OPTIMIZE::fval
```

Definition at line [1281](#) of file [structs.h](#).

3.97.2.2 ival

```
int OPTIMIZE::ival[2]
```

Definition at line [1282](#) of file [structs.h](#).

3.97.2.3 horner

```
int OPTIMIZE::horner
```

Definition at line 1284 of file [structs.h](#).

3.97.2.4 hornerdirection

```
int OPTIMIZE::hornerdirection
```

Definition at line 1285 of file [structs.h](#).

3.97.2.5 method

```
int OPTIMIZE::method
```

Definition at line 1286 of file [structs.h](#).

3.97.2.6 mctstimelimit

```
int OPTIMIZE::mctstimelimit
```

Definition at line 1287 of file [structs.h](#).

3.97.2.7 mctsnumexpand

```
int OPTIMIZE::mctsnumexpand
```

Definition at line 1288 of file [structs.h](#).

3.97.2.8 mctsnumkeep

```
int OPTIMIZE::mctsnumkeep
```

Definition at line 1289 of file [structs.h](#).

3.97.2.9 mctsnumrepeat

```
int OPTIMIZE::mctsnumrepeat
```

Definition at line 1290 of file [structs.h](#).

3.97.2.10 greedytimelimit

```
int OPTIMIZE::greedytimelimit
```

Definition at line 1291 of file [structs.h](#).

3.97.2.11 greedyminnum

```
int OPTIMIZE::greedyminnum
```

Definition at line 1292 of file [structs.h](#).

3.97.2.12 greedymaxperc

```
int OPTIMIZE::greedymaxperc
```

Definition at line 1293 of file [structs.h](#).

3.97.2.13 printstats

```
int OPTIMIZE::printstats
```

Definition at line 1294 of file [structs.h](#).

3.97.2.14 debugflags

```
int OPTIMIZE::debugflags
```

Definition at line 1295 of file [structs.h](#).

3.97.2.15 schemeflags

```
int OPTIMIZE::schemeflags
```

Definition at line 1296 of file [structs.h](#).

3.97.2.16 mctsdecaymode

```
int OPTIMIZE::mctsdecaymode
```

Definition at line 1297 of file [structs.h](#).

3.97.2.17 salter

```
int OPTIMIZE::saIter
```

Definition at line 1298 of file [structs.h](#).

3.97.2.18 spare

```
int OPTIMIZE::spare
```

Definition at line 1307 of file [structs.h](#).

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

- [structs.h](#)

3.98 OPTIMIZERESULT Struct Reference

Data Fields

- WORD * [code](#)
- UBYTE * [nameofexpr](#)
- LONG [codesize](#)
- WORD [exprnr](#)
- WORD [minvar](#)
- WORD [maxvar](#)

3.98.1 Detailed Description

Definition at line 1310 of file [structs.h](#).

3.98.2 Field Documentation

3.98.2.1 code

```
WORD* OPTIMIZERESULT::code
```

Definition at line 1311 of file [structs.h](#).

3.98.2.2 nameofexpr

```
UBYTE* OPTIMIZERESULT::nameofexpr
```

Definition at line 1312 of file [structs.h](#).

3.98.2.3 codesize

```
LONG OPTIMIZERESULT::codesize
```

Definition at line 1313 of file [structs.h](#).

Public Member Functions

- void [setDefaultValues](#) (void)
- void [setOldDefaultValues](#) (void)
- void [setValue](#) (int ind, int val)
- int [getValue](#) (int ind)
- void [setQGrafOpt](#) (int *qgopt)
- void [print](#) (void)
- void [setOutputF](#) (Bool outgrf, const char *fname)
- void [setOutputP](#) (Bool outgrp, const char *fname)
- void [printLevel](#) (int l)
- void [printModel](#) (void)
- void [setOutMG](#) (OutEGB *omg, void *pt)
- void [setOutAG](#) (OutEG *oag, void *pt)
- void [setEndMG](#) (OutEGB *omg, void *pt)
- void [setErExit](#) (ErExit *ere, void *pt)
- const [OptDef](#) * [getDef](#) (void)
- const [OptDef](#) * [getOldDef](#) (void)
- const [OptQGDef](#) * [getQGDef](#) (void)
- void [begin](#) ([Model](#) *mdl)
- void [end](#) (void)
- void [beginProc](#) ([Process](#) *prc)
- void [endProc](#) (void)
- void [beginSubProc](#) ([SProcess](#) *sprc)
- void [endSubProc](#) (void)
- void [newMGraph](#) ([MGraph](#) *mgr)
- void [newAGraph](#) ([EGraph](#) *egr)
- void [outModel](#) (void)

Data Fields

- [Model](#) * [model](#)
- [Process](#) * [proc](#)
- [SProcess](#) * [sprc](#)
- [Output](#) * [out](#)
- OutEGB * [outmg](#)
- OutEGB * [endmg](#)
- OutEG * [outag](#)
- void * [argmg](#)
- void * [argemg](#)
- void * [argag](#)
- [OptQGRef](#) [qgref](#) [2 *GRCC_QGRAFF_OPT_Size]
- int [values](#) [GRCC_OPT_Size]
- int [qgopt](#) [GRCC_QGRAFF_OPT_Size]
- int [nqgopt](#)
- int [DUMMYPADDING](#)
- double [time0](#)
- double [time1](#)

3.99.1 Detailed Description

Definition at line 101 of file [grcc.h](#).

3.99.2 Constructor & Destructor Documentation

3.99.2.1 Options()

```
Options::Options (
    void )
```

Definition at line 200 of file [grcc.cc](#).

3.99.2.2 ~Options()

```
Options::~~Options (
    void )
```

Definition at line 261 of file [grcc.cc](#).

3.99.3 Member Function Documentation

3.99.3.1 setDefaultValues()

```
void Options::setDefaultValues (
    void )
```

Definition at line 273 of file [grcc.cc](#).

3.99.3.2 setOldDefaultValues()

```
void Options::setOldDefaultValues (
    void )
```

Definition at line 289 of file [grcc.cc](#).

3.99.3.3 setValue()

```
void Options::setValue (
    int ind,
    int val )
```

Definition at line 339 of file [grcc.cc](#).

3.99.3.4 getValue()

```
int Options::getValue (
    int ind )
```

Definition at line 354 of file [grcc.cc](#).

3.99.3.5 setQGrafOpt()

```
void Options::setQGrafOpt (
    int * qgopt )
```

Definition at line [368](#) of file [grcc.cc](#).

3.99.3.6 print()

```
void Options::print (
    void )
```

Definition at line [443](#) of file [grcc.cc](#).

3.99.3.7 setOutputF()

```
void Options::setOutputF (
    Bool outgrf,
    const char * fname )
```

Definition at line [481](#) of file [grcc.cc](#).

3.99.3.8 setOutputP()

```
void Options::setOutputP (
    Bool outgrp,
    const char * fname )
```

Definition at line [488](#) of file [grcc.cc](#).

3.99.3.9 printLevel()

```
void Options::printLevel (
    int l )
```

Definition at line [333](#) of file [grcc.cc](#).

3.99.3.10 printModel()

```
void Options::printModel (
    void )
```

Definition at line [495](#) of file [grcc.cc](#).

3.99.3.11 setOutMG()

```
void Options::setOutMG (
    OutEGB * omg,
    void * pt )
```

Definition at line 305 of file [grcc.cc](#).

3.99.3.12 setOutAG()

```
void Options::setOutAG (
    OutEG * oag,
    void * pt )
```

Definition at line 319 of file [grcc.cc](#).

3.99.3.13 setEndMG()

```
void Options::setEndMG (
    OutEGB * omg,
    void * pt )
```

Definition at line 312 of file [grcc.cc](#).

3.99.3.14 setErExit()

```
void Options::setErExit (
    ErExit * ere,
    void * pt )
```

Definition at line 326 of file [grcc.cc](#).

3.99.3.15 getDef()

```
const OptDef * Options::getDef (
    void )
```

Definition at line 463 of file [grcc.cc](#).

3.99.3.16 getOldDef()

```
const OptDef * Options::getOldDef (
    void )
```

Definition at line 469 of file [grcc.cc](#).

3.99.3.17 getQGDef()

```
const OptQGDef * Options::getQGDef (
    void )
```

Definition at line 475 of file [grcc.cc](#).

3.99.3.18 begin()

```
void Options::begin (
    Model * mdl )
```

Definition at line 516 of file [grcc.cc](#).

3.99.3.19 end()

```
void Options::end (
    void )
```

Definition at line 536 of file [grcc.cc](#).

3.99.3.20 beginProc()

```
void Options::beginProc (
    Process * prc )
```

Definition at line 568 of file [grcc.cc](#).

3.99.3.21 endProc()

```
void Options::endProc (
    void )
```

Definition at line 584 of file [grcc.cc](#).

3.99.3.22 beginSubProc()

```
void Options::beginSubProc (
    SProcess * sprc )
```

Definition at line 673 of file [grcc.cc](#).

3.99.3.23 endSubProc()

```
void Options::endSubProc (
    void )
```

Definition at line 694 of file [grcc.cc](#).

3.99.3.24 newMGraph()

```
void Options::newMGraph (
    MGraph * mgr )
```

Definition at line 762 of file [grcc.cc](#).

3.99.3.25 newAGraph()

```
void Options::newAGraph (
    EGraph * egr )
```

Definition at line 792 of file [grcc.cc](#).

3.99.3.26 outModel()

```
void Options::outModel (
    void )
```

Definition at line 507 of file [grcc.cc](#).

3.99.4 Field Documentation

3.99.4.1 model

```
Model* Options::model
```

Definition at line 104 of file [grcc.h](#).

3.99.4.2 proc

```
Process* Options::proc
```

Definition at line 105 of file [grcc.h](#).

3.99.4.3 sproc

```
SProcess* Options::sproc
```

Definition at line 106 of file [grcc.h](#).

3.99.4.4 out

```
Output* Options::out
```

Definition at line 107 of file [grcc.h](#).

3.99.4.5 outmg

OutEGB* Options::outmg

Definition at line 108 of file [grcc.h](#).

3.99.4.6 endmg

OutEGB* Options::endmg

Definition at line 109 of file [grcc.h](#).

3.99.4.7 outag

OutEG* Options::outag

Definition at line 110 of file [grcc.h](#).

3.99.4.8 argmg

void* Options::argmg

Definition at line 111 of file [grcc.h](#).

3.99.4.9 argemg

void* Options::argemg

Definition at line 112 of file [grcc.h](#).

3.99.4.10 argag

void* Options::argag

Definition at line 113 of file [grcc.h](#).

3.99.4.11 qqref

[OptQGRef](#) Options::qqref[2 *GRCC_QGRAF_OPT_Size]

Definition at line 116 of file [grcc.h](#).

3.99.4.12 values

int Options::values[GRCC_OPT_Size]

Definition at line 117 of file [grcc.h](#).

3.99.4.13 qgopt

```
int Options::qgopt[GRCC_QGRAF_OPT_Size]
```

Definition at line 118 of file [grcc.h](#).

3.99.4.14 nqgopt

```
int Options::nqgopt
```

Definition at line 120 of file [grcc.h](#).

3.99.4.15 DUMMYPADDING

```
int Options::DUMMYPADDING
```

Definition at line 122 of file [grcc.h](#).

3.99.4.16 time0

```
double Options::time0
```

Definition at line 125 of file [grcc.h](#).

3.99.4.17 time1

```
double Options::time1
```

Definition at line 126 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.100 OptQGDef Struct Reference

Data Fields

- const char * [name](#)
- const char * [cname](#)
- const char * [mean](#)

3.100.1 Detailed Description

Definition at line 131 of file [grccparam.h](#).

3.100.2 Field Documentation

3.100.2.1 name

```
const char* OptQGDef::name
```

Definition at line 132 of file [grccparam.h](#).

3.100.2.2 cname

```
const char* OptQGDef::cname
```

Definition at line 133 of file [grccparam.h](#).

3.100.2.3 mean

```
const char* OptQGDef::mean
```

Definition at line 134 of file [grccparam.h](#).

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

- [grccparam.h](#)

3.101 OptQGRef Struct Reference

Data Fields

- const char * [name](#)
- int [index](#)
- int [sign](#)

3.101.1 Detailed Description

Definition at line 137 of file [grccparam.h](#).

3.101.2 Field Documentation

3.101.2.1 name

```
const char* OptQGRef::name
```

Definition at line 138 of file [grccparam.h](#).

```
int OptQGRef::index
```

3.101.2.3 sign

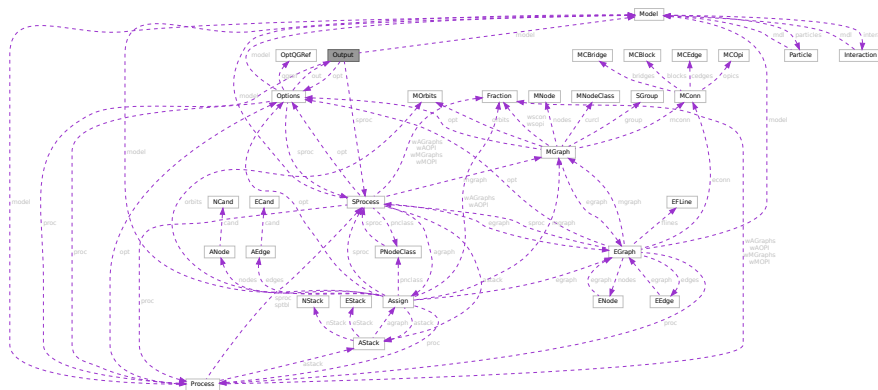
```
int OptQGRef::sign
```

Definition at line 140 of file grccparam.h.

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

- grccparam.h

Collaboration diagram for Output:



- **Output** (*optn)
- void **setOutgrf** (const char *fname)
- void **setOutgrp** (const char *fname)
- Bool **outBeginF** (**Model** *mdl, Bool pr)
- Bool **outBeginP** (**Model** *mdl, Bool pr)
- void **outEndF** (void)
- void **outEndP** (void)
- void **outProcBeginF** (**Process** *prc)
- void **outProcBeginP** (**Process** *prc)
- void **outProcBegin0** (int next, int couple, int loop)
- void **outSProcBeginF** (**SProcess** *sprc)
- void **outSProcBeginP** (**SProcess** *sprc)
- void **outProcEndF** (void)
- void **outProcEndP** (void)
- void **outEGraphF** (**EGraph** *egraph)
- void **outEGraphP** (**EGraph** *egraph)
- void **outModelF** (void)
- void **outModelP** (void)

Data Fields

- [Options](#) * [opt](#)
- [Model](#) * [model](#)
- [Process](#) * [proc](#)
- [SProcess](#) * [sproc](#)
- char * [outgrf](#)
- FILE * [outgrfp](#)
- char * [outgrp](#)
- FILE * [outgrpp](#)
- int [proclid](#)
- Bool [outproc](#)

3.102.1 Detailed Description

Definition at line [166](#) of file [grcc.h](#).

3.102.2 Constructor & Destructor Documentation

3.102.2.1 Output()

```
Output::Output (
    Options * optn )
```

Definition at line [832](#) of file [grcc.cc](#).

3.102.2.2 ~Output()

```
Output::~~Output (
    void )
```

Definition at line [848](#) of file [grcc.cc](#).

3.102.3 Member Function Documentation

3.102.3.1 setOutgrf()

```
void Output::setOutgrf (
    const char * fname )
```

Definition at line [879](#) of file [grcc.cc](#).

3.102.3.2 setOutgrp()

```
void Output::setOutgrp (
    const char * fname )
```

Definition at line [897](#) of file [grcc.cc](#).

3.102.3.3 outBeginF()

```
Bool Output::outBeginF (
    Model * mdl,
    Bool pr )
```

Definition at line 914 of file [grcc.cc](#).

3.102.3.4 outBeginP()

```
Bool Output::outBeginP (
    Model * mdl,
    Bool pr )
```

Definition at line 960 of file [grcc.cc](#).

3.102.3.5 outEndF()

```
void Output::outEndF (
    void )
```

Definition at line 999 of file [grcc.cc](#).

3.102.3.6 outEndP()

```
void Output::outEndP (
    void )
```

Definition at line 1015 of file [grcc.cc](#).

3.102.3.7 outProcBeginF()

```
void Output::outProcBeginF (
    Process * prc )
```

Definition at line 1031 of file [grcc.cc](#).

3.102.3.8 outProcBeginP()

```
void Output::outProcBeginP (
    Process * prc )
```

Definition at line 1076 of file [grcc.cc](#).

3.102.3.9 outProcBegin0()

```
void Output::outProcBegin0 (
    int next,
    int couple,
    int loop )
```

Definition at line 1098 of file [grcc.cc](#).

3.102.3.10 outSProcBeginF()

```
void Output::outSProcBeginF (
    SProcess * sprc )
```

Definition at line 1135 of file [grcc.cc](#).

3.102.3.11 outSProcBeginP()

```
void Output::outSProcBeginP (
    SProcess * sprc )
```

Definition at line 1190 of file [grcc.cc](#).

3.102.3.12 outProcEndF()

```
void Output::outProcEndF (
    void )
```

Definition at line 1210 of file [grcc.cc](#).

3.102.3.13 outProcEndP()

```
void Output::outProcEndP (
    void )
```

Definition at line 1229 of file [grcc.cc](#).

3.102.3.14 outEGraphF()

```
void Output::outEGraphF (
    EGraph * egraph )
```

Definition at line 1238 of file [grcc.cc](#).

3.102.3.15 outEGraphP()

```
void Output::outEGraphP (
    EGraph * egraph )
```

Definition at line 1378 of file [grcc.cc](#).

3.102.3.16 outModelF()

```
void Output::outModelF (  
    void )
```

Definition at line 1491 of file [grcc.cc](#).

3.102.3.17 outModelP()

```
void Output::outModelP (  
    void )
```

Definition at line 1572 of file [grcc.cc](#).

3.102.4 Field Documentation

3.102.4.1 opt

[Options*](#) Output::opt

Definition at line 169 of file [grcc.h](#).

3.102.4.2 model

[Model*](#) Output::model

Definition at line 170 of file [grcc.h](#).

3.102.4.3 proc

[Process*](#) Output::proc

Definition at line 171 of file [grcc.h](#).

3.102.4.4 sproc

[SProcess*](#) Output::sproc

Definition at line 172 of file [grcc.h](#).

3.102.4.5 outgrf

[char*](#) Output::outgrf

Definition at line 173 of file [grcc.h](#).

3.102.4.6 outgrfp

FILE* Output::outgrfp

Definition at line 174 of file [grcc.h](#).

3.102.4.7 outgrp

char* Output::outgrp

Definition at line 175 of file [grcc.h](#).

3.102.4.8 outgrpp

FILE* Output::outgrpp

Definition at line 176 of file [grcc.h](#).

3.102.4.9 proclId

int Output::procId

Definition at line 177 of file [grcc.h](#).

3.102.4.10 outproc

Bool Output::outproc

Definition at line 178 of file [grcc.h](#).

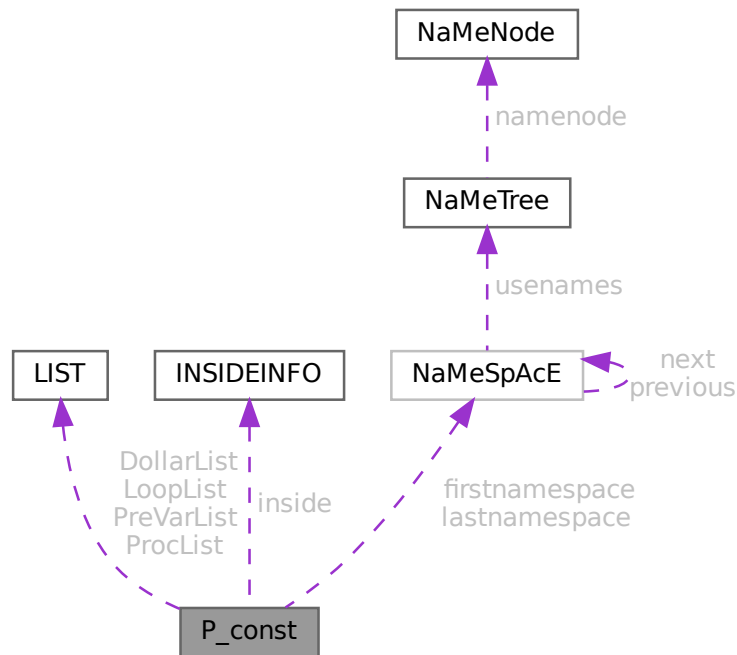
The documentation for this class was generated from the following files:

- [grcc.h](#)
- [grcc.cc](#)

3.103 P_const Struct Reference

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

Collaboration diagram for P_const:



Data Fields

- `LIST DollarList`
- `LIST PreVarList`
- `LIST LoopList`
- `LIST ProcList`
- `INSIDEINFO inside`
- `NAMESPACE * firstnamespace`
- `NAMESPACE * lastnamespace`
- `UBYTE * fullname`
- `UBYTE ** PreSwitchStrings`
- `UBYTE * preStart`
- `UBYTE * preStop`
- `UBYTE * preFill`
- `UBYTE * procedureExtension`
- `UBYTE * cprocedureExtension`
- `LONG * PreAssignStack`
- `int * PreIfStack`
- `int * PreSwitchModes`
- `int * PreTypes`

- LONG [StopWatchZero](#)
- LONG [InOutBuf](#)
- LONG [pSize](#)
- int [PreAssignFlag](#)
- int [PreContinuation](#)
- int [PreproFlag](#)
- int [iBufError](#)
- int [PreOut](#)
- int [PreSwitchLevel](#)
- int [NumPreSwitchStrings](#)
- int [MaxPreTypes](#)
- int [NumPreTypes](#)
- int [MaxPreIfLevel](#)
- int [PreIfLevel](#)
- int [PreInsideLevel](#)
- int [DelayPrevar](#)
- int [AllowDelay](#)
- int [lhdollarerror](#)
- int [eat](#)
- int [gNumPre](#)
- int [PreDebug](#)
- int [OpenDictionary](#)
- int [PreAssignLevel](#)
- int [MaxPreAssignLevel](#)
- int [fullnamesize](#)
- WORD [DebugFlag](#)
- WORD [preError](#)
- UBYTE [ComChar](#)
- UBYTE [cComChar](#)

3.103.1 Detailed Description

The [P_const](#) struct is part of the global data and resides in the ALLGLOBALS struct A under the name P We see it used with the macro AP as in AP.InOutBuf It contains objects that have dealings with the preprocessor.

Definition at line [1580](#) of file [structs.h](#).

3.103.2 Field Documentation

3.103.2.1 DollarList

```
LIST P_const::DollarList
```

Definition at line [1581](#) of file [structs.h](#).

3.103.2.2 PreVarList

```
LIST P_const::PreVarList
```

Definition at line [1583](#) of file [structs.h](#).

3.103.2.3 LoopList

`LIST P_const::LoopList`

Definition at line 1585 of file [structs.h](#).

3.103.2.4 ProcList

`LIST P_const::ProcList`

Definition at line 1586 of file [structs.h](#).

3.103.2.5 inside

`INSIDEINFO P_const::inside`

Definition at line 1587 of file [structs.h](#).

3.103.2.6 firstnamespace

`NAMESPACE* P_const::firstnamespace`

Definition at line 1588 of file [structs.h](#).

3.103.2.7 lastnamespace

`NAMESPACE* P_const::lastnamespace`

Definition at line 1589 of file [structs.h](#).

3.103.2.8 fullname

`UBYTE* P_const::fullname`

Definition at line 1590 of file [structs.h](#).

3.103.2.9 PreSwitchStrings

`UBYTE** P_const::PreSwitchStrings`

Definition at line 1591 of file [structs.h](#).

3.103.2.10 preStart

`UBYTE* P_const::preStart`

Definition at line 1592 of file [structs.h](#).

3.103.2.11 preStop

```
UBYTE* P_const::preStop
```

Definition at line 1593 of file [structs.h](#).

3.103.2.12 preFill

```
UBYTE* P_const::preFill
```

Definition at line 1594 of file [structs.h](#).

3.103.2.13 procedureExtension

```
UBYTE* P_const::procedureExtension
```

Definition at line 1595 of file [structs.h](#).

3.103.2.14 cprocedureExtension

```
UBYTE* P_const::cprocedureExtension
```

Definition at line 1596 of file [structs.h](#).

3.103.2.15 PreAssignStack

```
LONG* P_const::PreAssignStack
```

Definition at line 1597 of file [structs.h](#).

3.103.2.16 PreIfStack

```
int* P_const::PreIfStack
```

Definition at line 1598 of file [structs.h](#).

3.103.2.17 PreSwitchModes

```
int* P_const::PreSwitchModes
```

Definition at line 1599 of file [structs.h](#).

3.103.2.18 PreTypes

```
int* P_const::PreTypes
```

Definition at line 1600 of file [structs.h](#).

3.103.2.19 StopWatchZero

LONG P_const::StopWatchZero

Definition at line 1604 of file [structs.h](#).

3.103.2.20 InOutBuf

LONG P_const::InOutBuf

Definition at line 1605 of file [structs.h](#).

3.103.2.21 pSize

LONG P_const::pSize

Definition at line 1606 of file [structs.h](#).

3.103.2.22 PreAssignFlag

int P_const::PreAssignFlag

Definition at line 1607 of file [structs.h](#).

3.103.2.23 PreContinuation

int P_const::PreContinuation

Definition at line 1608 of file [structs.h](#).

3.103.2.24 PreproFlag

int P_const::PreproFlag

Definition at line 1609 of file [structs.h](#).

3.103.2.25 iBufError

int P_const::iBufError

Definition at line 1610 of file [structs.h](#).

3.103.2.26 PreOut

int P_const::PreOut

Definition at line 1611 of file [structs.h](#).

3.103.2.27 PreSwitchLevel

```
int P_const::PreSwitchLevel
```

Definition at line 1612 of file [structs.h](#).

3.103.2.28 NumPreSwitchStrings

```
int P_const::NumPreSwitchStrings
```

Definition at line 1613 of file [structs.h](#).

3.103.2.29 MaxPreTypes

```
int P_const::MaxPreTypes
```

Definition at line 1614 of file [structs.h](#).

3.103.2.30 NumPreTypes

```
int P_const::NumPreTypes
```

Definition at line 1615 of file [structs.h](#).

3.103.2.31 MaxPreIfLevel

```
int P_const::MaxPreIfLevel
```

Definition at line 1616 of file [structs.h](#).

3.103.2.32 PreIfLevel

```
int P_const::PreIfLevel
```

Definition at line 1617 of file [structs.h](#).

3.103.2.33 PreInsideLevel

```
int P_const::PreInsideLevel
```

Definition at line 1618 of file [structs.h](#).

3.103.2.34 DelayPrevar

```
int P_const::DelayPrevar
```

Definition at line 1619 of file [structs.h](#).

3.103.2.35 AllowDelay

```
int P_const::AllowDelay
```

Definition at line 1620 of file [structs.h](#).

3.103.2.36 lhdollarerror

```
int P_const::lhdollarerror
```

Definition at line 1621 of file [structs.h](#).

3.103.2.37 eat

```
int P_const::eat
```

Definition at line 1622 of file [structs.h](#).

3.103.2.38 gNumPre

```
int P_const::gNumPre
```

Definition at line 1623 of file [structs.h](#).

3.103.2.39 PreDebug

```
int P_const::PreDebug
```

Definition at line 1624 of file [structs.h](#).

3.103.2.40 OpenDictionary

```
int P_const::OpenDictionary
```

Definition at line 1625 of file [structs.h](#).

3.103.2.41 PreAssignLevel

```
int P_const::PreAssignLevel
```

Definition at line 1626 of file [structs.h](#).

3.103.2.42 MaxPreAssignLevel

```
int P_const::MaxPreAssignLevel
```

Definition at line 1627 of file [structs.h](#).

3.103.2.43 fullnamesize

```
int P_const::fullnamesize
```

Definition at line 1628 of file [structs.h](#).

3.103.2.44 DebugFlag

```
WORD P_const::DebugFlag
```

Definition at line 1629 of file [structs.h](#).

3.103.2.45 preError

```
WORD P_const::preError
```

Definition at line 1630 of file [structs.h](#).

3.103.2.46 ComChar

```
UBYTE P_const::ComChar
```

Definition at line 1631 of file [structs.h](#).

3.103.2.47 cComChar

```
UBYTE P_const::cComChar
```

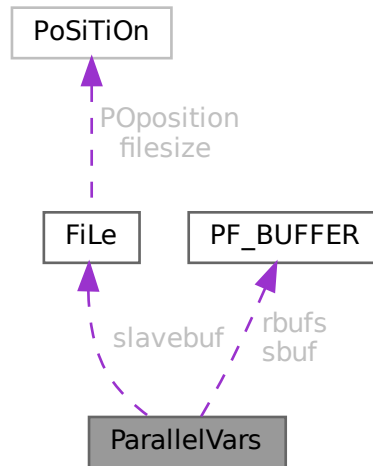
Definition at line 1632 of file [structs.h](#).

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

- [structs.h](#)

3.104 ParallelVars Struct Reference

Collaboration diagram for ParallelVars:



Data Fields

- `FILEHANDLE` `slavebuf`
- `PF_BUFFER` * `sbuf`
- `PF_BUFFER` ** `rbufs`
- `int` `me`
- `int` `numtasks`
- `int` `parallel`
- `int` `rhsInParallel`
- `int` `mkSlaveInfile`
- `int` `exprbufsize`
- `int` `exprtodo`
- `int` `log`
- `WORD` `numsbufs`
- `WORD` `numrbufs`

3.104.1 Detailed Description

Definition at line 168 of file `parallel.h`.

3.104.2 Field Documentation

3.104.2.1 `slavebuf`

`FILEHANDLE` `ParallelVars::slavebuf`

Definition at line 169 of file `parallel.h`.

3.104.2.2 sbuf

`PF_BUFFER* ParallelVars::sbuf`

Definition at line 171 of file [parallel.h](#).

3.104.2.3 rbufs

`PF_BUFFER** ParallelVars::rbufs`

Definition at line 172 of file [parallel.h](#).

3.104.2.4 me

`int ParallelVars::me`

Definition at line 173 of file [parallel.h](#).

3.104.2.5 numtasks

`int ParallelVars::numtasks`

Definition at line 174 of file [parallel.h](#).

3.104.2.6 parallel

`int ParallelVars::parallel`

Definition at line 175 of file [parallel.h](#).

3.104.2.7 rhsInParallel

`int ParallelVars::rhsInParallel`

Definition at line 177 of file [parallel.h](#).

3.104.2.8 mkSlaveInfile

`int ParallelVars::mkSlaveInfile`

Definition at line 178 of file [parallel.h](#).

3.104.2.9 exprbufsize

`int ParallelVars::exprbufsize`

Definition at line 179 of file [parallel.h](#).

3.104.2.10 `exprtodo`

```
int ParallelVars::exprtodo
```

Definition at line 180 of file [parallel.h](#).

3.104.2.11 `log`

```
int ParallelVars::log
```

Definition at line 181 of file [parallel.h](#).

3.104.2.12 `numsbuFs`

```
WORD ParallelVars::numsbuFs
```

Definition at line 182 of file [parallel.h](#).

3.104.2.13 `numrbuFs`

```
WORD ParallelVars::numrbuFs
```

Definition at line 183 of file [parallel.h](#).

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

- [parallel.h](#)

3.105 PaRtl Struct Reference

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

Data Fields

- WORD * [psize](#)
- WORD * [args](#)
- WORD * [nargs](#)
- WORD * [nfun](#)
- WORD [numargs](#)
- WORD [numpart](#)
- WORD [where](#)

3.105.1 Detailed Description

The struct PARTI is used to help determining whether a `partition_` function can be replaced.

Definition at line 1096 of file [structs.h](#).

3.105.2 Field Documentation

3.105.2.1 psize

WORD* PaRtI::psize

Definition at line 1097 of file [structs.h](#).

3.105.2.2 args

WORD* PaRtI::args

Definition at line 1098 of file [structs.h](#).

3.105.2.3 nargs

WORD* PaRtI::nargs

Definition at line 1099 of file [structs.h](#).

3.105.2.4 nfun

WORD* PaRtI::nfun

Definition at line 1100 of file [structs.h](#).

3.105.2.5 numargs

WORD PaRtI::numargs

Definition at line 1101 of file [structs.h](#).

3.105.2.6 numpart

WORD PaRtI::numpart

Definition at line 1102 of file [structs.h](#).

3.105.2.7 where

WORD PaRtI::where

Definition at line 1103 of file [structs.h](#).

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

- [structs.h](#)

3.106 PaRtlcLe Struct Reference

Data Fields

- WORD [number](#)
- WORD [spin](#)
- WORD [mass](#)
- WORD [type](#)

3.106.1 Detailed Description

Definition at line [619](#) of file [structs.h](#).

3.106.2 Field Documentation

3.106.2.1 number

WORD PaRtlcLe::number

Definition at line [620](#) of file [structs.h](#).

3.106.2.2 spin

WORD PaRtlcLe::spin

Definition at line [621](#) of file [structs.h](#).

3.106.2.3 mass

WORD PaRtlcLe::mass

Definition at line [622](#) of file [structs.h](#).

3.106.2.4 type

WORD PaRtlcLe::type

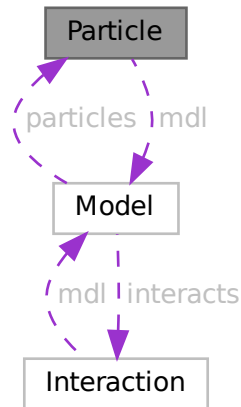
Definition at line [623](#) of file [structs.h](#).

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

- [structs.h](#)

3.107 Particle Class Reference

Collaboration diagram for Particle:



Public Member Functions

- [Particle](#) ([Model](#) *mdl, int pid, [PInput](#) *pinp)
- char * [particleName](#) (int p)
- int [particleCode](#) (int p)
- char * [interactionName](#) (int p)
- char * [aparticle](#) (void)
- void [prParticle](#) (void)
- int [isNeutral](#) (void)
- const char * [typeName](#) (void)
- const char * [typeGName](#) (void)

Data Fields

- [Model](#) * mdl
- char * [name](#)
- char * [aname](#)
- int [id](#)
- int [ptype](#)
- int [neutral](#)
- int [pcode](#)
- int [acode](#)
- int [cmindeg](#)
- int [cmaxdeg](#)
- int [extonly](#)

3.107.1 Detailed Description

Definition at line 227 of file [grcc.h](#).

3.107.2 Constructor & Destructor Documentation

3.107.2.1 Particle()

```
Particle::Particle (
    Model * modl,
    int pid,
    PInput * pinp )
```

Definition at line 1672 of file [grcc.cc](#).

3.107.2.2 ~Particle()

```
Particle::~~Particle (
    void )
```

Definition at line 1749 of file [grcc.cc](#).

3.107.3 Member Function Documentation

3.107.3.1 particleName()

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

Definition at line 1756 of file [grcc.cc](#).

3.107.3.2 particleCode()

```
int Particle::particleCode (
    int p )
```

Definition at line 1766 of file [grcc.cc](#).

3.107.3.3 aparticle()

```
char * Particle::aparticle (
    void )
```

Definition at line 1794 of file [grcc.cc](#).

3.107.3.4 prParticle()

```
void Particle::prParticle (  
    void )
```

Definition at line 1806 of file [grcc.cc](#).

3.107.3.5 isNeutral()

```
int Particle::isNeutral (  
    void )
```

Definition at line 1776 of file [grcc.cc](#).

3.107.3.6 typeName()

```
const char * Particle::typeName (  
    void )
```

Definition at line 1782 of file [grcc.cc](#).

3.107.3.7 typeGName()

```
const char * Particle::typeGName (  
    void )
```

Definition at line 1788 of file [grcc.cc](#).

3.107.4 Field Documentation

3.107.4.1 mdl

```
Model* Particle::mdl
```

Definition at line 229 of file [grcc.h](#).

3.107.4.2 name

```
char* Particle::name
```

Definition at line 230 of file [grcc.h](#).

3.107.4.3 aname

```
char* Particle::aname
```

Definition at line 231 of file [grcc.h](#).

3.107.4.4 id

```
int Particle::id
```

Definition at line 232 of file [grcc.h](#).

3.107.4.5 ptype

```
int Particle::ptype
```

Definition at line 233 of file [grcc.h](#).

3.107.4.6 neutral

```
int Particle::neutral
```

Definition at line 234 of file [grcc.h](#).

3.107.4.7 pcode

```
int Particle::pcode
```

Definition at line 235 of file [grcc.h](#).

3.107.4.8 acode

```
int Particle::acode
```

Definition at line 236 of file [grcc.h](#).

3.107.4.9 cmindeg

```
int Particle::cmindeg
```

Definition at line 237 of file [grcc.h](#).

3.107.4.10 cmaxdeg

```
int Particle::cmaxdeg
```

Definition at line 238 of file [grcc.h](#).

3.107.4.11 extonly

```
int Particle::extonly
```

Definition at line 240 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.108 PeRmUtE Struct Reference

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

Data Fields

- WORD * [objects](#)
- WORD [sign](#)
- WORD [n](#)
- WORD [cycle](#) [MAXMATCH]

3.108.1 Detailed Description

The struct PERM is used to generate all permutations when the pattern matcher has to try to match (anti)symmetric functions.

Definition at line 1057 of file [structs.h](#).

3.108.2 Field Documentation

3.108.2.1 objects

```
WORD* PeRmUtE::objects
```

Definition at line 1058 of file [structs.h](#).

3.108.2.2 sign

```
WORD PeRmUtE::sign
```

Definition at line 1059 of file [structs.h](#).

3.108.2.3 n

WORD PeRmUtE::n

Definition at line 1060 of file [structs.h](#).

3.108.2.4 cycle

WORD PeRmUtE::cycle[MAXMATCH]

Definition at line 1061 of file [structs.h](#).

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

- [structs.h](#)

3.109 PeRmUtEp Struct Reference

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

Data Fields

- WORD ** [objects](#)
- WORD [sign](#)
- WORD [n](#)
- WORD [cycle](#) [MAXMATCH]

3.109.1 Detailed Description

Like struct PERM but works with pointers.

Definition at line 1068 of file [structs.h](#).

3.109.2 Field Documentation

3.109.2.1 objects

WORD** PeRmUtEp::objects

Definition at line 1069 of file [structs.h](#).

3.109.2.2 sign

WORD PeRmUtEp::sign

Definition at line 1070 of file [structs.h](#).

3.109.2.3 n

WORD PeRmUtEp::n

Definition at line 1071 of file [structs.h](#).

3.109.2.4 cycle

WORD PeRmUtEp::cycle[MAXMATCH]

Definition at line 1072 of file [structs.h](#).

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

- [structs.h](#)

3.110 PF_BUFFER Struct Reference

```
#include <parallel.h>
```

Data Fields

- WORD ** [buff](#)
- WORD ** [fill](#)
- WORD ** [full](#)
- WORD ** [stop](#)
- MPI_Status * [status](#)
- MPI_Status * [retstat](#)
- MPI_Request * [request](#)
- MPI_Datatype * [type](#)
- int * [index](#)
- int * [tag](#)
- int * [from](#)
- int [numbufs](#)
- int [active](#)

3.110.1 Detailed Description

A struct for nonblocking, unbuffered send of the sorted terms in the PObuffers back to the master using several "rotating" PObuffers.

Definition at line 147 of file [parallel.h](#).

3.110.2 Field Documentation

3.110.2.1 buff

WORD** PF_BUFFER::buff

Definition at line 148 of file [parallel.h](#).

3.110.2.2 fill

```
WORD** PF_BUFFER::fill
```

Definition at line 149 of file [parallel.h](#).

3.110.2.3 full

```
WORD** PF_BUFFER::full
```

Definition at line 150 of file [parallel.h](#).

3.110.2.4 stop

```
WORD** PF_BUFFER::stop
```

Definition at line 151 of file [parallel.h](#).

3.110.2.5 status

```
MPI_Status* PF_BUFFER::status
```

Definition at line 152 of file [parallel.h](#).

3.110.2.6 retstat

```
MPI_Status* PF_BUFFER::retstat
```

Definition at line 153 of file [parallel.h](#).

3.110.2.7 request

```
MPI_Request* PF_BUFFER::request
```

Definition at line 154 of file [parallel.h](#).

3.110.2.8 type

```
MPI_Datatype* PF_BUFFER::type
```

Definition at line 155 of file [parallel.h](#).

3.110.2.9 index

```
int* PF_BUFFER::index
```

Definition at line 156 of file [parallel.h](#).

3.110.2.10 tag

```
int* PF_BUFFER::tag
```

Definition at line 157 of file [parallel.h](#).

3.110.2.11 from

```
int* PF_BUFFER::from
```

Definition at line 158 of file [parallel.h](#).

3.110.2.12 numbufs

```
int PF_BUFFER::numbufs
```

Definition at line 159 of file [parallel.h](#).

3.110.2.13 active

```
int PF_BUFFER::active
```

Definition at line 160 of file [parallel.h](#).

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

- [parallel.h](#)

3.111 PGInput Struct Reference

Data Fields

- int [cdeg](#)
- int [ctyp](#)
- int [cnum](#)
- int [cple](#)
- const char * [pname](#)
- long [pcode](#)

3.111.1 Detailed Description

Definition at line 232 of file [grccparam.h](#).

3.111.2 Field Documentation

3.111.2.1 cdeg

```
int PGInput::cdeg
```

Definition at line 233 of file [grccparam.h](#).

3.111.2.2 ctyp

```
int PGInput::ctyp
```

Definition at line 234 of file [grccparam.h](#).

3.111.2.3 cnum

```
int PGInput::cnum
```

Definition at line 235 of file [grccparam.h](#).

3.111.2.4 cple

```
int PGInput::cple
```

Definition at line 236 of file [grccparam.h](#).

3.111.2.5 pname

```
const char* PGInput::pname
```

Definition at line 238 of file [grccparam.h](#).

3.111.2.6 pcode

```
long PGInput::pcode
```

Definition at line 240 of file [grccparam.h](#).

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

- [grccparam.h](#)

3.112 PInput Struct Reference

Data Fields

- const char * [name](#)
- const char * [aname](#)
- int [pcode](#)
- int [acode](#)
- const char * [ptypen](#)
- int [ptypec](#)
- int [extonly](#)

3.112.1 Detailed Description

Definition at line [183](#) of file [grccparam.h](#).

3.112.2 Field Documentation

3.112.2.1 name

```
const char* PInput::name
```

Definition at line [184](#) of file [grccparam.h](#).

3.112.2.2 aname

```
const char* PInput::aname
```

Definition at line [185](#) of file [grccparam.h](#).

3.112.2.3 pcode

```
int PInput::pcode
```

Definition at line [186](#) of file [grccparam.h](#).

3.112.2.4 acode

```
int PInput::acode
```

Definition at line [187](#) of file [grccparam.h](#).

3.112.2.5 ptypen

```
const char* PInput::ptypen
```

Definition at line [188](#) of file [grccparam.h](#).

```
int PInput::ptyp
```

3.112.2.7 extonly

```
int PInput::extonly
```

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

- ### 3.113 PNodeClass Class Reference

[illegible]

- `PNodeClass` (`SProcess` *sprc, int nnods, int nclss, `NCInput` *cls)
- `PNodeClass` (`SProcess` *sprc, int nnods, int nclss, int *dgs, int *typ, int *ptcl, int *cpl, int *cnt, int *cmind, int *cmaxd)
- void `prPNodeClass` (void)
- void `prElem` (int e)

- SProcess * sproc
- int * deg
- int * type
- int * particle
- int * couple
- int * cmindeg
- int * cmaxdeg
- int * count
- int * cl2nd
- int * nd2cl
- int * cl2mcl
- int nnodes
- int nclass

3.113.1 Detailed Description

Definition at line [348](#) of file [grcc.h](#).

3.113.2 Constructor & Destructor Documentation

3.113.2.1 PNodeClass() [1/2]

```
PNodeClass::PNodeClass (
    SProcess * sprc,
    int nnods,
    int nclss,
    NCInput * cls )
```

Definition at line [2627](#) of file [grcc.cc](#).

3.113.2.2 PNodeClass() [2/2]

```
PNodeClass::PNodeClass (
    SProcess * sprc,
    int nnods,
    int nclss,
    int * dgs,
    int * typ,
    int * ptcl,
    int * cpl,
    int * cnt,
    int * cmind,
    int * cmaxd )
```

Definition at line [2701](#) of file [grcc.cc](#).

3.113.2.3 ~PNodeClass()

```
PNodeClass::~~PNodeClass (
    void )
```

Definition at line [2779](#) of file [grcc.cc](#).

3.113.3 Member Function Documentation

3.113.3.1 prPNodeClass()

```
void PNodeClass::prPNodeClass (
    void )
```

Definition at line [2794](#) of file [grcc.cc](#).

3.113.3.2 prElem()

```
void PNodeClass::prElem (  
    int e )
```

Definition at line 2806 of file [grcc.cc](#).

3.113.4 Field Documentation

3.113.4.1 sproc

```
SProcess* PNodeClass::sproc
```

Definition at line 350 of file [grcc.h](#).

3.113.4.2 deg

```
int* PNodeClass::deg
```

Definition at line 351 of file [grcc.h](#).

3.113.4.3 type

```
int* PNodeClass::type
```

Definition at line 352 of file [grcc.h](#).

3.113.4.4 particle

```
int* PNodeClass::particle
```

Definition at line 353 of file [grcc.h](#).

3.113.4.5 couple

```
int* PNodeClass::couple
```

Definition at line 354 of file [grcc.h](#).

3.113.4.6 cmindeg

```
int* PNodeClass::cmindeg
```

Definition at line 355 of file [grcc.h](#).

3.113.4.7 cmaxdeg

```
int* PNodeClass::cmaxdeg
```

Definition at line 356 of file [grcc.h](#).

3.113.4.8 count

```
int* PNodeClass::count
```

Definition at line 357 of file [grcc.h](#).

3.113.4.9 cl2nd

```
int* PNodeClass::cl2nd
```

Definition at line 358 of file [grcc.h](#).

3.113.4.10 nd2cl

```
int* PNodeClass::nd2cl
```

Definition at line 359 of file [grcc.h](#).

3.113.4.11 cl2mcl

```
int* PNodeClass::cl2mcl
```

Definition at line 360 of file [grcc.h](#).

3.113.4.12 nnodes

```
int PNodeClass::nnodes
```

Definition at line 361 of file [grcc.h](#).

3.113.4.13 nclass

```
int PNodeClass::nclass
```

Definition at line 362 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.114 flint::poly Class Reference

Public Member Functions

- void [print](#) (const string &text)

Data Fields

- [fmpz_poly_t](#) [d](#)

3.114.1 Detailed Description

Definition at line [87](#) of file [flintinterface.h](#).

3.114.2 Constructor & Destructor Documentation

3.114.2.1 [poly\(\)](#)

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

Definition at line [90](#) of file [flintinterface.h](#).

3.114.2.2 [~poly\(\)](#)

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

Definition at line [91](#) of file [flintinterface.h](#).

3.114.3 Member Function Documentation

3.114.3.1 [print\(\)](#)

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

Definition at line [92](#) of file [flintinterface.h](#).

3.114.4 Field Documentation

3.114.4.1 [d](#)

```
fmpz_poly_t flint::poly::d
```

Definition at line [89](#) of file [flintinterface.h](#).

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

- [flintinterface.h](#)

3.115 poly Class Reference

Public Member Functions

- [poly](#) (PHEAD int, WORD=-1, WORD=1)
- [poly](#) (PHEAD const UWORD *, WORD, WORD=-1, WORD=1)
- [poly](#) (const [poly](#) &, WORD=-1, WORD=1)
- [poly](#) & [operator+=](#) (const [poly](#) &)
- [poly](#) & [operator-=](#) (const [poly](#) &)
- [poly](#) & [operator*=](#) (const [poly](#) &)
- [poly](#) & [operator/=](#) (const [poly](#) &)
- [poly](#) & [operator%=>](#) (const [poly](#) &)
- const [poly](#) [operator+](#) (const [poly](#) &) const
- const [poly](#) [operator-](#) (const [poly](#) &) const
- const [poly](#) [operator*](#) (const [poly](#) &) const
- const [poly](#) [operator/](#) (const [poly](#) &) const
- const [poly](#) [operator%](#) (const [poly](#) &) const
- bool [operator==](#) (const [poly](#) &) const
- bool [operator!=](#) (const [poly](#) &) const
- [poly](#) & [operator=](#) (const [poly](#) &)
- WORD & [operator\[\]](#) (int)
- const WORD & [operator\[\]](#) (int) const
- void [termscopy](#) (const WORD *, int, int)
- void [check_memory](#) (int)
- void [expand_memory](#) (int)
- bool [is_zero](#) () const
- bool [is_one](#) () const
- bool [is_integer](#) () const
- bool [is_monomial](#) () const
- int [is_dense_univariate](#) () const
- int [sign](#) () const
- int [degree](#) (int) const
- int [total_degree](#) () const
- int [first_variable](#) () const
- int [number_of_terms](#) () const
- const std::vector< int > [all_variables](#) () const
- const [poly](#) [integer_lcoeff](#) () const
- const [poly](#) [lcoeff_univar](#) (int) const
- const [poly](#) [lcoeff_multivar](#) (int) const
- const [poly](#) [coefficient](#) (int, int) const
- const [poly](#) [derivative](#) (int) const
- void [setmod](#) (WORD, WORD=1)
- void [coefficients_modulo](#) (UWORD *, WORD, bool)
- int [size_of_form_notation](#) () const
- int [size_of_form_notation_with_den](#) (WORD) const
- const [poly](#) & [normalize](#) ()
- const std::string [to_string](#) () const
- WORD [last_monomial_index](#) () const
- WORD * [last_monomial](#) () const
- int [compare_degree_vector](#) (const [poly](#) &) const
- std::vector< int > [degree_vector](#) () const
- int [compare_degree_vector](#) (const std::vector< int > &) const

Static Public Member Functions

- static const [poly simple_poly](#) (PHEAD int, int=0, int=1, int=0, int=1)
- static const [poly simple_poly](#) (PHEAD int, const [poly](#) &, int=1, int=0, int=1)
- static void [get_variables](#) (PHEAD const std::vector< WORD * > &, bool, bool)
- static const [poly argument_to_poly](#) (PHEAD WORD *, bool, bool, [poly](#) *den=NULL)
- static void [poly_to_argument](#) (const [poly](#) &, WORD *, bool)
- static void [poly_to_argument_with_den](#) (const [poly](#) &, WORD, const UWORD *, WORD *, bool)
- static const [poly from_coefficient_list](#) (PHEAD const std::vector< WORD > &, int, WORD)
- static const std::vector< WORD > [to_coefficient_list](#) (const [poly](#) &)
- static const std::vector< WORD > [coefficient_list_divmod](#) (const std::vector< WORD > &, const std::vector< WORD > &, WORD, int)
- static int [monomial_compare](#) (PHEAD const WORD *, const WORD *)
- static void [add](#) (const [poly](#) &, const [poly](#) &, [poly](#) &)
- static void [sub](#) (const [poly](#) &, const [poly](#) &, [poly](#) &)
- static void [mul](#) (const [poly](#) &, const [poly](#) &, [poly](#) &)
- static void [div](#) (const [poly](#) &, const [poly](#) &, [poly](#) &)
- static void [mod](#) (const [poly](#) &, const [poly](#) &, [poly](#) &)
- static void [divmod](#) (const [poly](#) &, const [poly](#) &, [poly](#) &, [poly](#) &, bool only_divides)
- static bool [divides](#) (const [poly](#) &, const [poly](#) &)
- static void [mul_one_term](#) (const [poly](#) &, const [poly](#) &, [poly](#) &)
- static void [mul_univar](#) (const [poly](#) &, const [poly](#) &, [poly](#) &, int)
- static void [mul_heap](#) (const [poly](#) &, const [poly](#) &, [poly](#) &)
- static void [divmod_one_term](#) (const [poly](#) &, const [poly](#) &, [poly](#) &, [poly](#) &, bool)
- static void [divmod_univar](#) (const [poly](#) &, const [poly](#) &, [poly](#) &, [poly](#) &, int, bool)
- static void [divmod_heap](#) (const [poly](#) &, const [poly](#) &, [poly](#) &, [poly](#) &, bool, bool, bool &)
- static void [push_heap](#) (PHEAD WORD **, int)
- static void [pop_heap](#) (PHEAD WORD **, int)

Data Fields

- WORD * [terms](#)
- LONG [size_of_terms](#)
- WORD [modp](#)
- WORD [modn](#)

3.115.1 Detailed Description

Definition at line 53 of file [poly.h](#).

3.115.2 Constructor & Destructor Documentation

3.115.2.1 [poly\(\)](#) [1/3]

```
poly::poly (
    PHEAD int a,
    WORD modp = -1,
    WORD modn = 1 )
```

Definition at line 54 of file [poly.cc](#).

3.115.2.2 poly() [2/3]

```
poly::poly (
    PHEAD const UWORD * a,
    WORD na,
    WORD modp = -1,
    WORD modn = 1 )
```

Definition at line 83 of file [poly.cc](#).

3.115.2.3 poly() [3/3]

```
poly::poly (
    const poly & a,
    WORD modp = -1,
    WORD modn = 1 )
```

Definition at line 111 of file [poly.cc](#).

3.115.2.4 ~poly()

```
poly::~~poly ( )
```

Definition at line 133 of file [poly.cc](#).

3.115.3 Member Function Documentation

3.115.3.1 operator+=()

```
poly & poly::operator+= (
    const poly & a )
```

Definition at line 1966 of file [poly.cc](#).

3.115.3.2 operator-=()

```
poly & poly::operator-= (
    const poly & a )
```

Definition at line 1967 of file [poly.cc](#).

3.115.3.3 operator*=()

```
poly & poly::operator*= (
    const poly & a )
```

Definition at line 1968 of file [poly.cc](#).

3.115.3.4 operator/=()

```
poly & poly::operator/= (
    const poly & a )
```

Definition at line 1969 of file [poly.cc](#).

3.115.3.5 operator%=()

```
poly & poly::operator%= (
    const poly & a )
```

Definition at line 1970 of file [poly.cc](#).

3.115.3.6 operator+()

```
const poly poly::operator+ (
    const poly & a ) const
```

Definition at line 1959 of file [poly.cc](#).

3.115.3.7 operator-()

```
const poly poly::operator- (
    const poly & a ) const
```

Definition at line 1960 of file [poly.cc](#).

3.115.3.8 operator*()

```
const poly poly::operator* (
    const poly & a ) const
```

Definition at line 1961 of file [poly.cc](#).

3.115.3.9 operator/()

```
const poly poly::operator/ (
    const poly & a ) const
```

Definition at line 1962 of file [poly.cc](#).

3.115.3.10 operator%()

```
const poly poly::operator% (
    const poly & a ) const
```

Definition at line 1963 of file [poly.cc](#).

3.115.3.11 operator==()

```
bool poly::operator== (
    const poly & a ) const
```

Definition at line 1973 of file [poly.cc](#).

3.115.3.12 operator"!=()

```
bool poly::operator!= (
    const poly & a ) const
```

Definition at line 1979 of file [poly.cc](#).

3.115.3.13 operator=()

```
poly & poly::operator= (
    const poly & a )
```

Definition at line 1939 of file [poly.cc](#).

3.115.3.14 operator[]() [1/2]

```
WORD & poly::operator[] (
    int i ) [inline]
```

Definition at line 247 of file [poly.h](#).

3.115.3.15 operator[]() [2/2]

```
const WORD & poly::operator[] (
    int i ) const [inline]
```

Definition at line 251 of file [poly.h](#).

3.115.3.16 termscopy()

```
void poly::termscopy (
    const WORD * source,
    int dest,
    int num ) [inline]
```

Definition at line 258 of file [poly.h](#).

3.115.3.17 check_memory()

```
void poly::check_memory (
    int i ) [inline]
```

Definition at line 240 of file [poly.h](#).

3.115.3.18 expand_memory()

```
void poly::expand_memory (
    int i )
```

Definition at line 149 of file [poly.cc](#).

3.115.3.19 is_zero()

```
bool poly::is_zero ( ) const
```

Definition at line 2209 of file [poly.cc](#).

3.115.3.20 is_one()

```
bool poly::is_one ( ) const
```

Definition at line 2219 of file [poly.cc](#).

3.115.3.21 is_integer()

```
bool poly::is_integer ( ) const
```

Definition at line 2239 of file [poly.cc](#).

3.115.3.22 is_monomial()

```
bool poly::is_monomial ( ) const
```

Definition at line 2259 of file [poly.cc](#).

3.115.3.23 is_dense_univariate()

```
int poly::is_dense_univariate ( ) const
```

Dense univariate detection

3.115.4 Description

This method returns whether the polynomial is dense and univariate. The possible return values are:

-2 is not dense univariate -1 is no variables $n \geq 0$ is univariate in n

3.115.5 Notes

A univariate polynomial is considered dense iff more than half of the coefficients $a_0 \dots a_{\text{deg}}$ are non-zero.

Definition at line 2284 of file [poly.cc](#).

Referenced by [divmod\(\)](#), and [mul\(\)](#).

3.115.5.1 sign()

```
int poly::sign ( ) const
```

Definition at line 2039 of file [poly.cc](#).

3.115.5.2 degree()

```
int poly::degree (
    int x ) const
```

Definition at line 2050 of file [poly.cc](#).

3.115.5.3 total_degree()

```
int poly::total_degree ( ) const
```

Definition at line 2063 of file [poly.cc](#).

3.115.5.4 first_variable()

```
int poly::first_variable ( ) const
```

Definition at line 2000 of file [poly.cc](#).

3.115.5.5 number_of_terms()

```
int poly::number_of_terms ( ) const
```

Definition at line 1986 of file [poly.cc](#).

3.115.5.6 all_variables()

```
const vector< int > poly::all_variables ( ) const
```

Definition at line 2016 of file [poly.cc](#).

3.115.5.7 integer_lcoeff()

```
const poly poly::integer_lcoeff ( ) const
```

Definition at line 2083 of file [poly.cc](#).

3.115.5.8 lcoeff_univar()

```
const poly poly::lcoeff_univar (
    int x ) const
```

Definition at line 2133 of file [poly.cc](#).

3.115.5.9 lcoeff_multivar()

```
const poly poly::lcoeff_multivar (
    int x ) const
```

Definition at line 2140 of file [poly.cc](#).

3.115.5.10 coefficient()

```
const poly poly::coefficient (
    int x,
    int n ) const
```

Definition at line 2105 of file [poly.cc](#).

3.115.5.11 derivative()

```
const poly poly::derivative (
    int x ) const
```

Definition at line 2172 of file [poly.cc](#).

3.115.5.12 setmod()

```
void poly::setmod (
    WORD _modp,
    WORD _modn = 1 )
```

Definition at line 179 of file [poly.cc](#).

3.115.5.13 coefficients_modulo()

```
void poly::coefficients_modulo (
    UWORD * a,
    WORD na,
    bool small )
```

Definition at line 215 of file [poly.cc](#).

3.115.5.14 simple_poly() [1/2]

```
const poly poly::simple_poly (
    PHEAD int x,
    int a = 0,
    int b = 1,
    int p = 0,
    int n = 1 ) [static]
```

Definition at line 2317 of file [poly.cc](#).

3.115.5.15 simple_poly() [2/2]

```
const poly poly::simple_poly (
    PHEAD int x,
    const poly & a,
    int b = 1,
    int p = 0,
    int n = 1 ) [static]
```

Definition at line 2356 of file [poly.cc](#).

3.115.5.16 get_variables()

```
void poly::get_variables (
    PHEAD const std::vector< WORD * > & ,
    bool ,
    bool ) [static]
```

Definition at line 2395 of file [poly.cc](#).

3.115.5.17 argument_to_poly()

```
const poly poly::argument_to_poly (
    PHEAD WORD * e,
    bool with_arghead,
    bool sort_univar,
    poly * den = NULL ) [static]
```

Definition at line 2498 of file [poly.cc](#).

3.115.5.18 poly_to_argument()

```
void poly::poly_to_argument (
    const poly & a,
    WORD * res,
    bool with_arghead ) [static]
```

Definition at line 2611 of file [poly.cc](#).

3.115.5.19 poly_to_argument_with_den()

```
void poly::poly_to_argument_with_den (
    const poly & a,
    WORD nden,
    const UWORD * den,
    WORD * res,
    bool with_arghead ) [static]
```

Definition at line 2683 of file [poly.cc](#).

3.115.5.20 size_of_form_notation()

```
int poly::size_of_form_notation ( ) const
```

Definition at line 2766 of file [poly.cc](#).

3.115.5.21 size_of_form_notation_with_den()

```
int poly::size_of_form_notation_with_den (
    WORD nden ) const
```

Definition at line 2795 of file [poly.cc](#).

3.115.5.22 normalize()

```
const poly & poly::normalize ( )
```

Definition at line 362 of file [poly.cc](#).

3.115.5.23 from_coefficient_list()

```
const poly poly::from_coefficient_list (
    PHEAD const std::vector< WORD > & ,
    int ,
    WORD ) [static]
```

Definition at line 2885 of file [poly.cc](#).

3.115.5.24 to_coefficient_list()

```
const vector< WORD > poly::to_coefficient_list (
    const poly & a ) [static]
```

Definition at line 2823 of file [poly.cc](#).

3.115.5.25 coefficient_list_divmod()

```
const vector< WORD > poly::coefficient_list_divmod (
    const std::vector< WORD > & ,
    const std::vector< WORD > & ,
    WORD ,
    int ) [static]
```

Definition at line 2846 of file [poly.cc](#).

3.115.5.26 to_string()

```
const string poly::to_string ( ) const
```

Definition at line 267 of file [poly.cc](#).

3.115.5.27 monomial_compare()

```
int poly::monomial_compare (
    PHEAD const WORD * a,
    const WORD * b ) [static]
```

Definition at line 350 of file [poly.cc](#).

3.115.5.28 last_monomial_index()

```
WORD poly::last_monomial_index ( ) const
```

Definition at line 465 of file [poly.cc](#).

3.115.5.29 last_monomial()

```
WORD * poly::last_monomial ( ) const
```

Definition at line 472 of file [poly.cc](#).

3.115.5.30 compare_degree_vector()

```
int poly::compare_degree_vector (
    const poly & b ) const
```

Definition at line 481 of file [poly.cc](#).

3.115.5.31 degree_vector()

```
std::vector< int > poly::degree_vector ( ) const
```

Definition at line 503 of file [poly.cc](#).

3.115.5.32 add()

```
void poly::add (
    const poly & a,
    const poly & b,
    poly & c ) [static]
```

Definition at line 538 of file [poly.cc](#).

3.115.5.33 sub()

```
void poly::sub (
    const poly & a,
    const poly & b,
    poly & c ) [static]
```

Definition at line 622 of file [poly.cc](#).

3.115.5.34 mul()

```
void poly::mul (
    const poly & a,
    const poly & b,
    poly & c ) [static]
```

Polynomial multiplication

3.115.6 Description

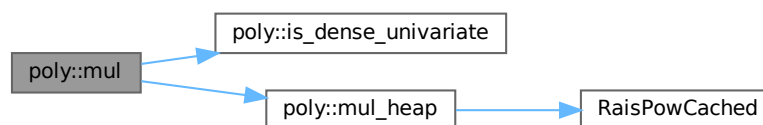
This routine determines which multiplication routine to use for multiplying two polynomials. The logic is as follows:

- If a or b consists of only one term, call `mul_one_term`;
- Otherwise, if both are univariate and dense, call `mul_univar`;
- Otherwise, call `mul_heap`.

Definition at line 1195 of file [poly.cc](#).

References [is_dense_univariate\(\)](#), and [mul_heap\(\)](#).

Here is the call graph for this function:



3.115.6.1 div()

```
void poly::div (
    const poly & a,
    const poly & b,
    poly & c ) [static]
```

Definition at line 1915 of file [poly.cc](#).

3.115.6.2 mod()

```
void poly::mod (
    const poly & a,
    const poly & b,
    poly & c ) [static]
```

Definition at line 1927 of file [poly.cc](#).

3.115.6.3 divmod()

```
void poly::divmod (
    const poly & a,
    const poly & b,
    poly & q,
    poly & r,
    bool only_divides ) [static]
```

Polynomial division

3.115.7 Description

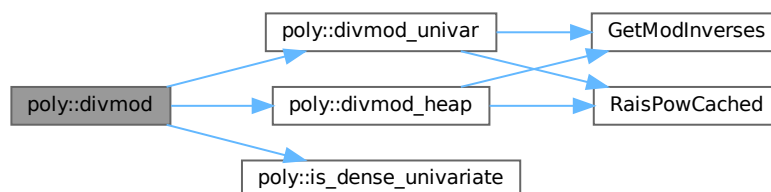
This routine determines which division routine to use for dividing two polynomials. The logic is as follows:

- If b consists of only one term, call `divmod_one_term`;
- Otherwise, if both are univariate and dense, call `divmod_univar`;
- Otherwise, call `divmod_heap`.

Definition at line 1856 of file [poly.cc](#).

References [divmod_heap\(\)](#), [divmod_univar\(\)](#), and [is_dense_univariate\(\)](#).

Here is the call graph for this function:



3.115.7.1 divides()

```
bool poly::divides (
    const poly & a,
    const poly & b ) [static]
```

Definition at line 1902 of file [poly.cc](#).

3.115.7.2 mul_one_term()

```
void poly::mul_one_term (
    const poly & a,
    const poly & b,
    poly & c ) [static]
```

Definition at line 755 of file [poly.cc](#).

3.115.7.3 mul_univar()

```
void poly::mul_univar (
    const poly & a,
    const poly & b,
    poly & c,
    int var ) [static]
```

Definition at line 829 of file [poly.cc](#).

3.115.7.4 mul_heap()

```
void poly::mul_heap (
    const poly & a,
    const poly & b,
    poly & c ) [static]
```

Multiplication of polynomials with a heap

3.115.8 Description

Multiplies two multivariate polynomials. The next element of the product is efficiently determined by using a heap. If the product of the maximum power in all variables is small, a hash table is used to add equal terms for extra speed.

A heap element h is formatted as follows:

- h[0] = index in a
- h[1] = index in b
- h[2] = hash code (-1 if no hash is used)
- h[3] = length of coefficient with sign
- h[4...4+AN.poly_num_vars-1] = powers

- $h[4+AN.poly\_num\_vars...4+h[3]-1]$ = coefficient

Definition at line 961 of file [poly.cc](#).

References [RaisPowCached\(\)](#).

Referenced by [mul\(\)](#).

Here is the call graph for this function:



3.115.8.1 `divmod_one_term()`

```

void poly::divmod_one_term (
    const poly & a,
    const poly & b,
    poly & q,
    poly & r,
    bool only_divides ) [static]
  
```

Definition at line 1245 of file [poly.cc](#).

3.115.8.2 `divmod_univar()`

```

void poly::divmod_univar (
    const poly & a,
    const poly & b,
    poly & q,
    poly & r,
    int var,
    bool only_divides ) [static]
  
```

Division of dense univariate polynomials.

3.115.9 Description

Divides two dense univariate polynomials. For each power, the method collects all terms that result in that power.

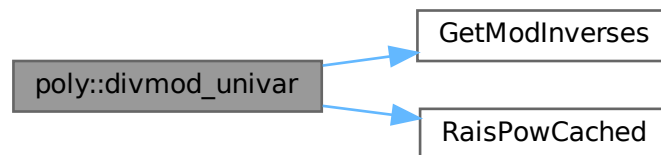
Relevant formula $[Q=A/B, P=\text{SUM}(p_i \cdot x^i), n=\text{deg}(A), m=\text{deg}(B)]: q_k = [a_{m+k} - \text{SUM}(i=k+1 \dots n-m, b_{m+k-i} \cdot q_i)] / b_m$

Definition at line 1376 of file [poly.cc](#).

References [GetModInverses\(\)](#), and [RaisPowCached\(\)](#).

Referenced by [divmod\(\)](#).

Here is the call graph for this function:



3.115.9.1 divmod_heap()

```

void poly::divmod_heap (
    const poly & a,
    const poly & b,
    poly & q,
    poly & r,
    bool only_divides,
    bool check_div,
    bool & div_fail ) [static]
  
```

Division of polynomials with a heap

3.115.10 Description

Divides two multivariate polynomials. The next element of the quotient/remainder is efficiently determined by using a heap. If the product of the maximum power in all variables is small, a hash table is used to add equal terms for extra speed.

If the input flag `check_div` is set then if the result of any coefficient division results in a non-zero remainder (indicating that division has failed over the integers) the output flag `div_fail` will be set and the division will be terminated early (`q`, `r` will be incorrect). If `check_div` is not set then non-zero remainders from coefficient division will be written into `r`.

A heap element `h` is formatted as follows:

- $h[0]$ = index in a
- $h[1]$ = index in b
- $h[2] = -1$ (no hash is used)
- $h[3]$ = length of coefficient with sign
- $h[4 \dots 4 + \text{AN.poly\_num\_vars} - 1]$ = powers
- $h[4 + \text{AN.poly\_num\_vars} \dots 4 + h[3] - 1]$ = coefficient

Note: the hashing trick as in multiplication cannot be used easily, since there is no tight upperbound on the exponents in the answer.

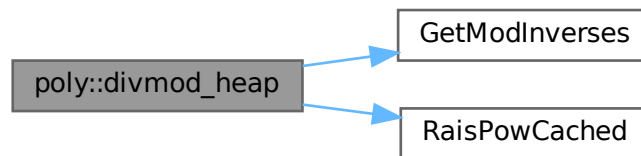
For details, see M. Monagan, "Polynomial Division using Dynamic Array, Heaps, and Packed Exponent Vectors"

Definition at line 1579 of file [poly.cc](#).

References [GetModInverses\(\)](#), and [RaisPowCached\(\)](#).

Referenced by [divmod\(\)](#).

Here is the call graph for this function:



3.115.10.1 `push_heap()`

```
void poly::push_heap (
    PHEAD WORD ** heap,
    int n ) [static]
```

Definition at line 739 of file [poly.cc](#).

3.115.10.2 `pop_heap()`

```
void poly::pop_heap (
    PHEAD WORD ** heap,
    int n ) [static]
```

Definition at line 707 of file [poly.cc](#).

3.115.11 Field Documentation

3.115.11.1 terms

WORD* poly::terms

Definition at line 62 of file [poly.h](#).

3.115.11.2 size_of_terms

LONG poly::size_of_terms

Definition at line 63 of file [poly.h](#).

3.115.11.3 modp

WORD poly::modp

Definition at line 64 of file [poly.h](#).

3.115.11.4 modn

WORD poly::modn

Definition at line 64 of file [poly.h](#).

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

- [poly.h](#)
- [poly.cc](#)

3.116 flint::poly_factor Class Reference

Data Fields

- [fmpz_poly_factor_t](#) [d](#)

3.116.1 Detailed Description

Definition at line 96 of file [flintinterface.h](#).

3.116.2 Constructor & Destructor Documentation

3.116.2.1 `poly_factor()`

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

Definition at line 99 of file [flintinterface.h](#).

3.116.2.2 `~poly_factor()`

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

Definition at line 100 of file [flintinterface.h](#).

3.116.3 Field Documentation

3.116.3.1 `d`

```
mpz_poly_factor_t flint::poly_factor::d
```

Definition at line 98 of file [flintinterface.h](#).

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

- [flintinterface.h](#)

3.117 POLYMOD Struct Reference

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

Data Fields

- WORD * [coefs](#)
- WORD [numsym](#)
- WORD [arraysize](#)
- WORD [polysize](#)
- WORD [modnum](#)

3.117.1 Detailed Description

The [POLYMOD](#) struct controls one univariate polynomial of which the coefficients have been taken modulus a (prime) number that fits inside a variable of type WORD. The polynomial is stored as an array of coefficients of size WORD.

Definition at line 1236 of file [structs.h](#).

3.117.2 Field Documentation

3.117.2.1 coefs

WORD* POLYMOD::coefs

Definition at line 1237 of file [structs.h](#).

3.117.2.2 numsym

WORD POLYMOD::numsym

Definition at line 1238 of file [structs.h](#).

3.117.2.3 arraysize

WORD POLYMOD::arraysize

Definition at line 1239 of file [structs.h](#).

3.117.2.4 polysize

WORD POLYMOD::polysize

Definition at line 1240 of file [structs.h](#).

3.117.2.5 modnum

WORD POLYMOD::modnum

Definition at line 1241 of file [structs.h](#).

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

- [structs.h](#)

3.118 PoSiTiOn Struct Reference

Data Fields

- [off_t p1](#)

3.118.1 Detailed Description

Definition at line 59 of file [structs.h](#).

3.118.2 Field Documentation

3.118.2.1 p1

```
off_t PoSiTiOn::p1
```

Definition at line 60 of file [structs.h](#).

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

- [structs.h](#)

3.119 PRELOAD Struct Reference

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

Data Fields

- UBYTE * [buffer](#)
- LONG [size](#)

3.119.1 Detailed Description

Used by the preprocessor to load the contents of a doloop or a procedure. The struct [PRELOAD](#) is used both in the DOLOOP and [PROCEDURE](#) structs.

Definition at line 854 of file [structs.h](#).

3.119.2 Field Documentation

3.119.2.1 buffer

```
UBYTE* PRELOAD::buffer
```

Definition at line 855 of file [structs.h](#).

3.119.2.2 size

```
LONG PRELOAD::size
```

Definition at line 856 of file [structs.h](#).

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

- [structs.h](#)

3.120 pReVaR Struct Reference

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

Data Fields

- UBYTE * [name](#)
- UBYTE * [value](#)
- UBYTE * [argnames](#)
- int [nargs](#)
- int [wildarg](#)

3.120.1 Detailed Description

An element of the type PREVAR is needed for each preprocessor variable.

Definition at line [824](#) of file [structs.h](#).

3.120.2 Field Documentation

3.120.2.1 name

```
UBYTE* pReVaR::name
```

allocated

Definition at line [825](#) of file [structs.h](#).

3.120.2.2 value

```
UBYTE* pReVaR::value
```

points into memory of name

Definition at line [826](#) of file [structs.h](#).

3.120.2.3 argnames

```
UBYTE* pReVaR::argnames
```

names of arguments, zero separated. points into memory of name

Definition at line [827](#) of file [structs.h](#).

3.120.2.4 nargs

```
int pReVaR::nargs
```

0 = regular, >= 1: number of macro arguments. total number

Definition at line 828 of file [structs.h](#).

3.120.2.5 wildarg

```
int pReVaR::wildarg
```

The number of a potential ?var. If none: 0. wildarg<nargs

Definition at line 829 of file [structs.h](#).

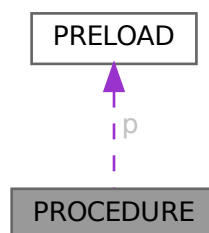
The documentation for this struct was generated from the following file:

- [structs.h](#)

3.121 PROCEDURE Struct Reference

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

Collaboration diagram for PROCEDURE:



Data Fields

- [PRELOAD](#) p
- UBYTE * [name](#)
- int [loadmode](#)
- int [mustfree](#)

Public Member Functions

- [Process](#) (int pid, [Model](#) *model, [Options](#) *opt, int nin, int *initlPart, int nfin, int *finalPart, int *coupling)
- **Process** (int pid, [Model](#) *model, [Options](#) *opt, [FGInput](#) *fgi)
- void [prProcess](#) (void)
- void [outProcP](#) (FILE *fp)
- void [prProcessP](#) (const char *fname)
- void [mkSProcess](#) (void)

Data Fields

- [Model](#) * [model](#)
- [Options](#) * [opt](#)
- [BigInt](#) [mgrcount](#)
- [BigInt](#) [agrcount](#)
- int * [initlPart](#)
- int * [finalPart](#)
- int [id](#)
- int [ninitl](#)
- int [nfinal](#)
- int [ctotal](#)
- int [nExtern](#)
- int [loop](#)
- int [maxnlegs](#)
- int [clist](#) [GRCC_MAXNCPLG]
- int [nSubproc](#)
- [SProcess](#) * [sptbl](#) [GRCC_MAXSUBPROCS]
- [SProcess](#) * [sproc](#)
- [AStack](#) * [astack](#)
- [BigInt](#) [ngraphs](#)
- [BigInt](#) [nopi](#)
- [BigInt](#) [wgraphs](#)
- [BigInt](#) [wopi](#)
- [BigInt](#) [nMGraphs](#)
- [BigInt](#) [nMOPI](#)
- [Fraction](#) [wMGraphs](#)
- [Fraction](#) [wMOPI](#)
- [BigInt](#) [nAGraphs](#)
- [BigInt](#) [nAOPI](#)
- [Fraction](#) [wAGraphs](#)
- [Fraction](#) [wAOPI](#)
- double [sec](#)

3.122.1 Detailed Description

Definition at line 447 of file [grcc.h](#).

3.122.2 Constructor & Destructor Documentation

3.122.2.1 Process()

```
Process::Process (
    int pid,
    Model * model,
    Options * opt,
    int nin,
    int * initlPart,
    int nfin,
    int * finalPart,
    int * coupling )
```

Definition at line 3339 of file [grcc.cc](#).

3.122.2.2 ~Process()

```
Process::~~Process (
    void )
```

Definition at line 3438 of file [grcc.cc](#).

3.122.3 Member Function Documentation

3.122.3.1 prProcess()

```
void Process::prProcess (
    void )
```

Definition at line 3449 of file [grcc.cc](#).

3.122.3.2 outProcP()

```
void Process::outProcP (
    FILE * fp )
```

Definition at line 3472 of file [grcc.cc](#).

3.122.3.3 prProcessP()

```
void Process::prProcessP (
    const char * fname )
```

Definition at line 3457 of file [grcc.cc](#).

3.122.3.4 mkSProcess()

```
void Process::mkSProcess (
    void )
```

Definition at line [3527](#) of file [grcc.cc](#).

3.122.4 Field Documentation

3.122.4.1 model

```
Model* Process::model
```

Definition at line [449](#) of file [grcc.h](#).

3.122.4.2 opt

```
Options* Process::opt
```

Definition at line [450](#) of file [grcc.h](#).

3.122.4.3 mgrcount

```
BigInt Process::mgrcount
```

Definition at line [451](#) of file [grcc.h](#).

3.122.4.4 agrcount

```
BigInt Process::agrcount
```

Definition at line [452](#) of file [grcc.h](#).

3.122.4.5 initlPart

```
int* Process::initlPart
```

Definition at line [453](#) of file [grcc.h](#).

3.122.4.6 finalPart

```
int* Process::finalPart
```

Definition at line [454](#) of file [grcc.h](#).

3.122.4.7 id

```
int Process::id
```

Definition at line 456 of file [grcc.h](#).

3.122.4.8 ninitl

```
int Process::ninitl
```

Definition at line 457 of file [grcc.h](#).

3.122.4.9 nfinal

```
int Process::nfinal
```

Definition at line 458 of file [grcc.h](#).

3.122.4.10 ctotat

```
int Process::ctotat
```

Definition at line 459 of file [grcc.h](#).

3.122.4.11 nExtern

```
int Process::nExtern
```

Definition at line 460 of file [grcc.h](#).

3.122.4.12 loop

```
int Process::loop
```

Definition at line 461 of file [grcc.h](#).

3.122.4.13 maxnlegs

```
int Process::maxnlegs
```

Definition at line 462 of file [grcc.h](#).

3.122.4.14 clist

```
int Process::clist[GRCC_MAXNCPLG]
```

Definition at line 463 of file [grcc.h](#).

3.122.4.15 nSubproc

```
int Process::nSubproc
```

Definition at line 466 of file [grcc.h](#).

3.122.4.16 sptbl

```
SProcess* Process::sptbl[GRCC_MAXSUBPROCS]
```

Definition at line 467 of file [grcc.h](#).

3.122.4.17 sproc

```
SProcess* Process::sproc
```

Definition at line 468 of file [grcc.h](#).

3.122.4.18 astack

```
AStack* Process::astack
```

Definition at line 471 of file [grcc.h](#).

3.122.4.19 ngraphs

```
BigInt Process::ngraphs
```

Definition at line 474 of file [grcc.h](#).

3.122.4.20 nopi

```
BigInt Process::nopi
```

Definition at line 475 of file [grcc.h](#).

3.122.4.21 wgraphs

```
BigInt Process::wgraphs
```

Definition at line 476 of file [grcc.h](#).

3.122.4.22 wopi

```
BigInt Process::wopi
```

Definition at line 477 of file [grcc.h](#).

3.122.4.23 nMGraphs

`BigInt Process::nMGraphs`

Definition at line 480 of file [grcc.h](#).

3.122.4.24 nMOPI

`BigInt Process::nMOPI`

Definition at line 481 of file [grcc.h](#).

3.122.4.25 wMGraphs

`Fraction Process::wMGraphs`

Definition at line 482 of file [grcc.h](#).

3.122.4.26 wMOPI

`Fraction Process::wMOPI`

Definition at line 483 of file [grcc.h](#).

3.122.4.27 nAGraphs

`BigInt Process::nAGraphs`

Definition at line 485 of file [grcc.h](#).

3.122.4.28 nAOPI

`BigInt Process::nAOPI`

Definition at line 486 of file [grcc.h](#).

3.122.4.29 wAGraphs

`Fraction Process::wAGraphs`

Definition at line 487 of file [grcc.h](#).

3.122.4.30 wAOPI

`Fraction Process::wAOPI`

Definition at line 488 of file [grcc.h](#).

3.122.4.31 sec

```
double Process::sec
```

Definition at line 490 of file [grcc.h](#).

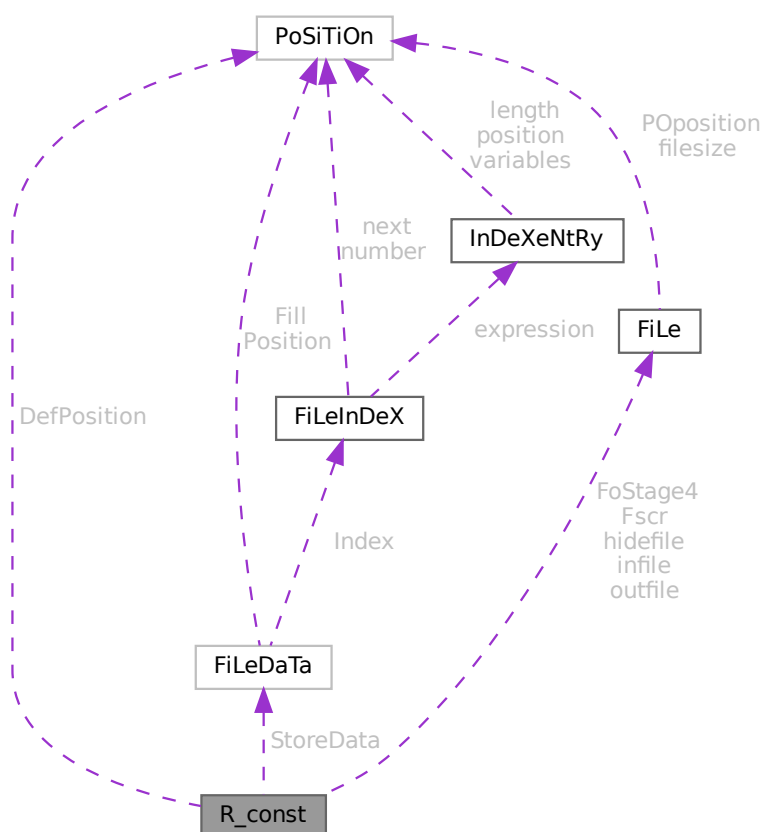
The documentation for this class was generated from the following files:

- [grcc.h](#)
- [grcc.cc](#)

3.123 R_const Struct Reference

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

Collaboration diagram for R_const:



Data Fields

- FILEDATA StoreData
- FILEHANDLE Fscr [3]
- FILEHANDLE FoStage4 [2]
- POSITION DefPosition
- FILEHANDLE * infile
- FILEHANDLE * outfile
- FILEHANDLE * hidefile
- WORD * CompressBuffer
- WORD * ComprTop
- WORD * CompressPointer
- COMPAREDUMMY CompareRoutine
- ULONG * wranfia
- SBYTE * moebiustable
- LONG OldTime
- LONG InInBuf
- LONG InHiBuf
- LONG pWorkSize
- LONG IWorkSize
- LONG posWorkSize
- ULONG wranfseed
- int NoCompress
- int gzipCompress
- int Cnumlhs
- int outtohide
- int wranfcall
- int wranfnpair1
- int wranfnpair2
- WORD GetFile
- WORD KeptInHold
- WORD BracketOn
- WORD MaxBracket
- WORD CurDum
- WORD DeferFlag
- WORD TePos
- WORD sLevel
- WORD Stage4Name
- WORD GetOneFile
- WORD PolyFun
- WORD PolyFunInv
- WORD PolyFunType
- WORD PolyFunExp
- WORD PolyFunVar
- WORD PolyFunPow
- WORD Eside
- WORD MaxDum
- WORD level
- WORD expchanged
- WORD expflags
- WORD CurExpr
- WORD SortType
- WORD ShortSortCount
- WORD modeloptions
- WORD funoffset
- WORD moebiustablesize

3.123.1 Detailed Description

The `R_const` struct is part of the global data and resides either in the ALLGLOBALS struct A, or the ALLPRIVATES struct B (TFORM) under the name R. We see it used with the macro AR as in AR.infile. It has the variables that define the running environment and that should be transferred with a term in a multithreaded run.

Definition at line 1956 of file [structs.h](#).

3.123.2 Field Documentation

3.123.2.1 StoreData

`FILEDATA R_const::StoreData`

Definition at line 1957 of file [structs.h](#).

3.123.2.2 Fscr

`FILEHANDLE R_const::Fscr[3]`

Definition at line 1958 of file [structs.h](#).

3.123.2.3 FoStage4

`FILEHANDLE R_const::FoStage4[2]`

Definition at line 1959 of file [structs.h](#).

3.123.2.4 DefPosition

`POSITION R_const::DefPosition`

Definition at line 1960 of file [structs.h](#).

3.123.2.5 infile

`FILEHANDLE* R_const::infile`

Definition at line 1961 of file [structs.h](#).

3.123.2.6 outfile

`FILEHANDLE* R_const::outfile`

Definition at line 1962 of file [structs.h](#).

3.123.2.7 hidefile

`FILEHANDLE* R_const::hidefile`

Definition at line 1963 of file [structs.h](#).

3.123.2.8 CompressBuffer

`WORD* R_const::CompressBuffer`

Definition at line 1965 of file [structs.h](#).

3.123.2.9 ComprTop

`WORD* R_const::ComprTop`

Definition at line 1966 of file [structs.h](#).

3.123.2.10 CompressPointer

`WORD* R_const::CompressPointer`

Definition at line 1967 of file [structs.h](#).

3.123.2.11 CompareRoutine

`COMPAREDUMMY R_const::CompareRoutine`

Definition at line 1968 of file [structs.h](#).

3.123.2.12 wranfia

`ULONG* R_const::wranfia`

Definition at line 1969 of file [structs.h](#).

3.123.2.13 moebiustable

`SBYTE* R_const::moebiustable`

Definition at line 1970 of file [structs.h](#).

3.123.2.14 OldTime

`LONG R_const::OldTime`

Definition at line 1972 of file [structs.h](#).

3.123.2.15 InInBuf

LONG R_const::InInBuf

Definition at line 1973 of file [structs.h](#).

3.123.2.16 InHiBuf

LONG R_const::InHiBuf

Definition at line 1974 of file [structs.h](#).

3.123.2.17 pWorkSize

LONG R_const::pWorkSize

Definition at line 1975 of file [structs.h](#).

3.123.2.18 lWorkSize

LONG R_const::lWorkSize

Definition at line 1976 of file [structs.h](#).

3.123.2.19 posWorkSize

LONG R_const::posWorkSize

Definition at line 1977 of file [structs.h](#).

3.123.2.20 wranfseed

ULONG R_const::wranfseed

Definition at line 1978 of file [structs.h](#).

3.123.2.21 NoCompress

int R_const::NoCompress

Definition at line 1979 of file [structs.h](#).

3.123.2.22 gzipCompress

int R_const::gzipCompress

Definition at line 1980 of file [structs.h](#).

3.123.2.23 Cnumlhs

```
int R_const::Cnumlhs
```

Definition at line 1981 of file [structs.h](#).

3.123.2.24 outtohide

```
int R_const::outtohide
```

Definition at line 1982 of file [structs.h](#).

3.123.2.25 wranfcall

```
int R_const::wranfcall
```

Definition at line 1986 of file [structs.h](#).

3.123.2.26 wranfnpair1

```
int R_const::wranfnpair1
```

Definition at line 1987 of file [structs.h](#).

3.123.2.27 wranfnpair2

```
int R_const::wranfnpair2
```

Definition at line 1988 of file [structs.h](#).

3.123.2.28 GetFile

```
WORD R_const::GetFile
```

Definition at line 1994 of file [structs.h](#).

3.123.2.29 KeptInHold

```
WORD R_const::KeptInHold
```

Definition at line 1995 of file [structs.h](#).

3.123.2.30 BracketOn

```
WORD R_const::BracketOn
```

Definition at line 1996 of file [structs.h](#).

3.123.2.31 MaxBracket

WORD R_const::MaxBracket

Definition at line 1997 of file [structs.h](#).

3.123.2.32 CurDum

WORD R_const::CurDum

Definition at line 1998 of file [structs.h](#).

3.123.2.33 DeferFlag

WORD R_const::DeferFlag

Definition at line 1999 of file [structs.h](#).

3.123.2.34 TePos

WORD R_const::TePos

Definition at line 2000 of file [structs.h](#).

3.123.2.35 sLevel

WORD R_const::sLevel

Definition at line 2001 of file [structs.h](#).

3.123.2.36 Stage4Name

WORD R_const::Stage4Name

Definition at line 2002 of file [structs.h](#).

3.123.2.37 GetOneFile

WORD R_const::GetOneFile

Definition at line 2003 of file [structs.h](#).

3.123.2.38 PolyFun

WORD R_const::PolyFun

Definition at line 2004 of file [structs.h](#).

3.123.2.39 PolyFunInv

WORD R_const::PolyFunInv

Definition at line 2005 of file [structs.h](#).

3.123.2.40 PolyFunType

WORD R_const::PolyFunType

Definition at line 2006 of file [structs.h](#).

3.123.2.41 PolyFunExp

WORD R_const::PolyFunExp

Definition at line 2007 of file [structs.h](#).

3.123.2.42 PolyFunVar

WORD R_const::PolyFunVar

Definition at line 2008 of file [structs.h](#).

3.123.2.43 PolyFunPow

WORD R_const::PolyFunPow

Definition at line 2009 of file [structs.h](#).

3.123.2.44 Eside

WORD R_const::Eside

Definition at line 2010 of file [structs.h](#).

3.123.2.45 MaxDum

WORD R_const::MaxDum

Definition at line 2011 of file [structs.h](#).

3.123.2.46 level

WORD R_const::level

Definition at line 2012 of file [structs.h](#).

3.123.2.47 expchanged

WORD R_const::expchanged

Definition at line 2013 of file [structs.h](#).

3.123.2.48 expflags

WORD R_const::expflags

Definition at line 2014 of file [structs.h](#).

3.123.2.49 CurExpr

WORD R_const::CurExpr

Definition at line 2015 of file [structs.h](#).

3.123.2.50 SortType

WORD R_const::SortType

Definition at line 2016 of file [structs.h](#).

3.123.2.51 ShortSortCount

WORD R_const::ShortSortCount

Definition at line 2017 of file [structs.h](#).

3.123.2.52 modeloptions

WORD R_const::modeloptions

Definition at line 2018 of file [structs.h](#).

3.123.2.53 funoffset

WORD R_const::funoffset

Definition at line 2019 of file [structs.h](#).

3.123.2.54 moebiustablesize

```
WORD R_const::moebiustablesize
```

Definition at line 2020 of file [structs.h](#).

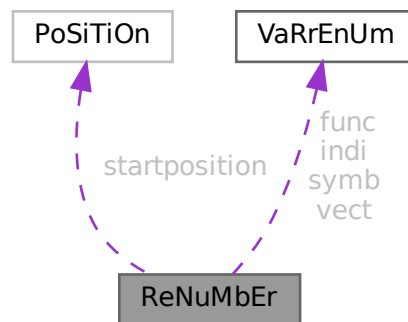
The documentation for this struct was generated from the following file:

- [structs.h](#)

3.124 ReNuMbEr Struct Reference

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

Collaboration diagram for ReNuMbEr:



Data Fields

- [POSITION](#) startposition
- [VARRENUM](#) symb
- [VARRENUM](#) indi
- [VARRENUM](#) vect
- [VARRENUM](#) func
- [WORD](#) * symnum
- [WORD](#) * indnum
- [WORD](#) * vecnum
- [WORD](#) * funnum

3.124.1 Detailed Description

Only symb.lo gets dynamically allocated. All other pointers points into this memory.

Definition at line 176 of file [structs.h](#).

3.124.2 Field Documentation

3.124.2.1 startposition

`POSITION ReNuMbEr::startposition`

Definition at line 177 of file [structs.h](#).

3.124.2.2 symb

`VARRENUM ReNuMbEr::symb`

Symbols

Definition at line 179 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#), [Generator\(\)](#), [GetFirstTerm\(\)](#), and [TermRenumbrer\(\)](#).

3.124.2.3 indi

`VARRENUM ReNuMbEr::indi`

Indices

Definition at line 180 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#), and [TermRenumbrer\(\)](#).

3.124.2.4 vect

`VARRENUM ReNuMbEr::vect`

Vectors

Definition at line 181 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#), and [TermRenumbrer\(\)](#).

3.124.2.5 func

`VARRENUM ReNuMbEr::func`

Functions

Definition at line 182 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#), and [TermRenumbrer\(\)](#).

3.124.2.6 symnum

WORD* ReNuMbEr::symnum

Renumbered symbols

Definition at line 184 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#), and [TermRenumber\(\)](#).

3.124.2.7 indnum

WORD* ReNuMbEr::indnum

Renumbered indices

Definition at line 185 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#), and [TermRenumber\(\)](#).

3.124.2.8 vecnum

WORD* ReNuMbEr::vecnum

Renumbered vectors

Definition at line 186 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#), and [TermRenumber\(\)](#).

3.124.2.9 funnum

WORD* ReNuMbEr::funnum

Renumbered functions

Definition at line 187 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#), and [TermRenumber\(\)](#).

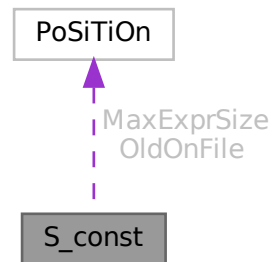
The documentation for this struct was generated from the following file:

- [structs.h](#)

3.125 S_const Struct Reference

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

Collaboration diagram for S_const:



Data Fields

- [POSITION](#) [MaxExprSize](#)
- [POSITION](#) * [OldOnFile](#)
- [WORD](#) * [OldNumFactors](#)
- [WORD](#) * [Oldvflags](#)
- [WORD](#) * [Olduflags](#)
- [int](#) [NumOldOnFile](#)
- [int](#) [NumOldNumFactors](#)
- [int](#) [MultiThreaded](#)
- [int](#) [Balancing](#)
- [WORD](#) [ExecMode](#)
- [WORD](#) [CollectOverFlag](#)

3.125.1 Detailed Description

The [S_const](#) struct is part of the global data and resides in the ALLGLOBALS struct A under the name S We see it used with the macro AS as in AS.ExecMode It has some variables used by the master in multithreaded runs

Definition at line [1911](#) of file [structs.h](#).

3.125.2 Field Documentation

3.125.2.1 MaxExprSize

```
POSITION S_const::MaxExprSize
```

Definition at line [1912](#) of file [structs.h](#).

3.125.2.2 OldOnFile

`POSITION* S_const::OldOnFile`

Definition at line 1918 of file [structs.h](#).

3.125.2.3 OldNumFactors

`WORD* S_const::OldNumFactors`

Definition at line 1919 of file [structs.h](#).

3.125.2.4 Oldvflags

`WORD* S_const::Oldvflags`

Definition at line 1920 of file [structs.h](#).

3.125.2.5 Olduflags

`WORD* S_const::Olduflags`

Definition at line 1921 of file [structs.h](#).

3.125.2.6 NumOldOnFile

`int S_const::NumOldOnFile`

Definition at line 1925 of file [structs.h](#).

3.125.2.7 NumOldNumFactors

`int S_const::NumOldNumFactors`

Definition at line 1926 of file [structs.h](#).

3.125.2.8 MultiThreaded

`int S_const::MultiThreaded`

Definition at line 1927 of file [structs.h](#).

3.125.2.9 Balancing

`int S_const::Balancing`

Definition at line 1934 of file [structs.h](#).

3.125.2.10 ExecMode

WORD S_const::ExecMode

Definition at line 1935 of file [structs.h](#).

3.125.2.11 CollectOverFlag

WORD S_const::CollectOverFlag

Definition at line 1937 of file [structs.h](#).

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

- [structs.h](#)

3.126 SeTs Struct Reference

Data Fields

- LONG [name](#)
- WORD [type](#)
- WORD [first](#)
- WORD [last](#)
- WORD [node](#)
- WORD [namesize](#)
- WORD [dimension](#)
- WORD [flags](#)

3.126.1 Detailed Description

Definition at line 513 of file [structs.h](#).

3.126.2 Field Documentation

3.126.2.1 name

LONG SeTs::name

Definition at line 514 of file [structs.h](#).

3.126.2.2 type

WORD SeTs::type

Definition at line 515 of file [structs.h](#).

3.126.2.3 first

WORD SeTs::first

Definition at line 516 of file [structs.h](#).

3.126.2.4 last

WORD SeTs::last

Definition at line 517 of file [structs.h](#).

3.126.2.5 node

WORD SeTs::node

Definition at line 518 of file [structs.h](#).

3.126.2.6 namesize

WORD SeTs::namesize

Definition at line 519 of file [structs.h](#).

3.126.2.7 dimension

WORD SeTs::dimension

Definition at line 520 of file [structs.h](#).

3.126.2.8 flags

WORD SeTs::flags

Definition at line 521 of file [structs.h](#).

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

- [structs.h](#)

3.127 SETUPPARAMETERS Struct Reference

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

Data Fields

- UBYTE * [parameter](#)
- int [type](#)
- int [flags](#)
- LONG [value](#)

3.127.1 Detailed Description

Each setup parameter has one element of the struct [SETUPPARAMETERS](#) assigned to it. By binary search in the array of them we can then locate the proper element by name. We have to assume that two ints make a long and either one or two longs make a pointer. The long before the ints and padding gives a problem in the initialization.

Definition at line [1015](#) of file [structs.h](#).

3.127.2 Field Documentation

3.127.2.1 parameter

```
UBYTE* SETUPPARAMETERS::parameter
```

Definition at line [1016](#) of file [structs.h](#).

3.127.2.2 type

```
int SETUPPARAMETERS::type
```

Definition at line [1017](#) of file [structs.h](#).

3.127.2.3 flags

```
int SETUPPARAMETERS::flags
```

Definition at line [1018](#) of file [structs.h](#).

3.127.2.4 value

```
LONG SETUPPARAMETERS::value
```

Definition at line [1022](#) of file [structs.h](#).

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

- [structs.h](#)

3.128 SGroup Class Reference

Public Member Functions

- void [print](#) (void)
- void [newGroup](#) (int nelm, int nclss, int *clss)
- void [clearGroup](#) (void)
- void [delGroup](#) (void)
- int * [genNext](#) (void)
- void [addGroup](#) (int *p)
- BigInt [nElem](#) (void)
- int * [nextElem](#) (void)

Data Fields

- BigInt [size](#)
- BigInt [nelem](#)
- int ** [elem](#)
- int [nnodes](#)
- int [neclass](#)
- int [eclass](#) [GRCC_MAXNODES]
- int [cgen](#)
- int [csav](#)
- int [permg](#) [GRCC_MAXNODES]
- int [perms](#) [GRCC_MAXNODES]
- int [pgr](#) [GRCC_MAXNODES]
- int [pgq](#) [GRCC_MAXNODES]
- int [psr](#) [GRCC_MAXNODES]
- int [psq](#) [GRCC_MAXNODES]

3.128.1 Detailed Description

Definition at line [732](#) of file [grcc.h](#).

3.128.2 Constructor & Destructor Documentation

3.128.2.1 SGroup()

```
SGroup::SGroup (  
    void )
```

Definition at line [5873](#) of file [grcc.cc](#).

3.128.2.2 ~SGroup()

```
SGroup::~~SGroup (  
    void )
```

Definition at line [5882](#) of file [grcc.cc](#).

3.128.3 Member Function Documentation

3.128.3.1 print()

```
void SGroup::print (
    void )
```

Definition at line 5899 of file [grcc.cc](#).

3.128.3.2 newGroup()

```
void SGroup::newGroup (
    int nelm,
    int nclss,
    int * clss )
```

Definition at line 5914 of file [grcc.cc](#).

3.128.3.3 clearGroup()

```
void SGroup::clearGroup (
    void )
```

Definition at line 5939 of file [grcc.cc](#).

3.128.3.4 delGroup()

```
void SGroup::delGroup (
    void )
```

Definition at line 5947 of file [grcc.cc](#).

3.128.3.5 genNext()

```
int * SGroup::genNext (
    void )
```

Definition at line 5963 of file [grcc.cc](#).

3.128.3.6 addGroup()

```
void SGroup::addGroup (
    int * p )
```

Definition at line 5973 of file [grcc.cc](#).

3.128.3.7 nElem()

```
BigInt SGroup::nElem (  
    void )
```

Definition at line 5990 of file [grcc.cc](#).

3.128.3.8 nextElem()

```
int * SGroup::nextElem (  
    void )
```

Definition at line 5996 of file [grcc.cc](#).

3.128.4 Field Documentation

3.128.4.1 size

```
BigInt SGroup::size
```

Definition at line 734 of file [grcc.h](#).

3.128.4.2 nelem

```
BigInt SGroup::nelem
```

Definition at line 735 of file [grcc.h](#).

3.128.4.3 elem

```
int** SGroup::elem
```

Definition at line 736 of file [grcc.h](#).

3.128.4.4 nnodes

```
int SGroup::nnodes
```

Definition at line 737 of file [grcc.h](#).

3.128.4.5 neclass

```
int SGroup::neclass
```

Definition at line 738 of file [grcc.h](#).

3.128.4.6 **eclass**

```
int SGroup::eclass[GRCC_MAXNODES]
```

Definition at line 739 of file [grcc.h](#).

3.128.4.7 **cgen**

```
int SGroup::cgen
```

Definition at line 740 of file [grcc.h](#).

3.128.4.8 **csav**

```
int SGroup::csav
```

Definition at line 741 of file [grcc.h](#).

3.128.4.9 **permg**

```
int SGroup::permg[GRCC_MAXNODES]
```

Definition at line 742 of file [grcc.h](#).

3.128.4.10 **perms**

```
int SGroup::perms[GRCC_MAXNODES]
```

Definition at line 743 of file [grcc.h](#).

3.128.4.11 **pgr**

```
int SGroup::pgr[GRCC_MAXNODES]
```

Definition at line 745 of file [grcc.h](#).

3.128.4.12 **pgq**

```
int SGroup::pgq[GRCC_MAXNODES]
```

Definition at line 746 of file [grcc.h](#).

3.128.4.13 **psr**

```
int SGroup::psr[GRCC_MAXNODES]
```

Definition at line 747 of file [grcc.h](#).

3.128.4.14 psq

```
int SGroup::psq[GRCC_MAXNODES]
```

Definition at line 748 of file [grcc.h](#).

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

- [grcc.h](#)
- [grcc.cc](#)

3.129 SHvariables Struct Reference

Data Fields

- WORD * [outterm](#)
- WORD * [outfun](#)
- WORD * [incoef](#)
- WORD * [stop1](#)
- WORD * [stop2](#)
- WORD * [ststop1](#)
- WORD * [ststop2](#)
- FINISHUFFLE [finishuf](#)
- DO_UFFLE [do_uffle](#)
- LONG [combilast](#)
- WORD [nincoef](#)
- WORD [level](#)
- WORD [thefunction](#)
- WORD [option](#)

3.129.1 Detailed Description

Definition at line 1244 of file [structs.h](#).

3.129.2 Field Documentation

3.129.2.1 outterm

```
WORD* SHvariables::outterm
```

Definition at line 1245 of file [structs.h](#).

3.129.2.2 outfun

```
WORD* SHvariables::outfun
```

Definition at line 1246 of file [structs.h](#).

3.129.2.3 incoef

WORD* SHvariables::incoef

Definition at line 1247 of file [structs.h](#).

3.129.2.4 stop1

WORD* SHvariables::stop1

Definition at line 1248 of file [structs.h](#).

3.129.2.5 stop2

WORD* SHvariables::stop2

Definition at line 1249 of file [structs.h](#).

3.129.2.6 ststop1

WORD* SHvariables::ststop1

Definition at line 1250 of file [structs.h](#).

3.129.2.7 ststop2

WORD* SHvariables::ststop2

Definition at line 1251 of file [structs.h](#).

3.129.2.8 finishuf

FINISHUFFLE SHvariables::finishuf

Definition at line 1252 of file [structs.h](#).

3.129.2.9 do_uffle

DO_UFFLE SHvariables::do_uffle

Definition at line 1253 of file [structs.h](#).

3.129.2.10 combilast

LONG SHvariables::combilast

Definition at line 1254 of file [structs.h](#).

3.129.2.11 nincoef

WORD SHvariables::nincoef

Definition at line 1255 of file [structs.h](#).

3.129.2.12 level

WORD SHvariables::level

Definition at line 1256 of file [structs.h](#).

3.129.2.13 thefunction

WORD SHvariables::thefunction

Definition at line 1257 of file [structs.h](#).

3.129.2.14 option

WORD SHvariables::option

Definition at line 1258 of file [structs.h](#).

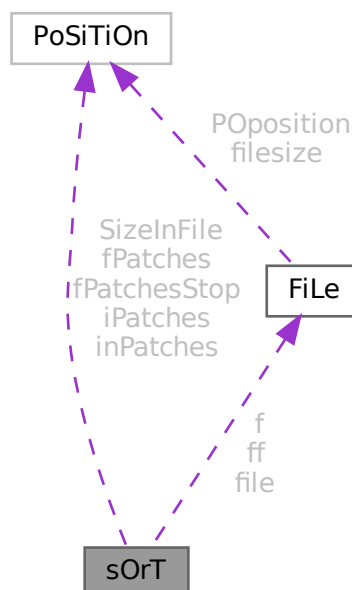
The documentation for this struct was generated from the following file:

- [structs.h](#)

3.130 sOrT Struct Reference

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

Collaboration diagram for sOrT:



Data Fields

- [FILEHANDLE](#) [file](#)
- [POSITION](#) [SizeInFile](#) [3]
- [WORD](#) * [lBuffer](#)
- [WORD](#) * [lTop](#)
- [WORD](#) * [lFill](#)
- [WORD](#) * [used](#)
- [WORD](#) * [sBuffer](#)
- [WORD](#) * [sTop](#)
- [WORD](#) * [sTop2](#)
- [WORD](#) * [sHalf](#)
- [WORD](#) * [sFill](#)
- [WORD](#) ** [sPointer](#)
- [WORD](#) ** [PoinFill](#)
- [WORD](#) * [cBuffer](#)
- [WORD](#) ** [Patches](#)
- [WORD](#) ** [pStop](#)
- [WORD](#) ** [poina](#)
- [WORD](#) ** [poin2a](#)
- [WORD](#) * [ktoi](#)
- [WORD](#) * [tree](#)
- [POSITION](#) * [fPatches](#)
- [POSITION](#) * [inPatches](#)
- [POSITION](#) * [fPatchesStop](#)
- [POSITION](#) * [iPatches](#)
- [FILEHANDLE](#) * [f](#)
- [FILEHANDLE](#) ** [ff](#)
- [LONG](#) [sTerms](#)
- [LONG](#) [LargeSize](#)
- [LONG](#) [SmallSize](#)
- [LONG](#) [SmallEsize](#)
- [LONG](#) [TermsInSmall](#)
- [LONG](#) [Terms2InSmall](#)
- [LONG](#) [GenTerms](#)
- [LONG](#) [TermsLeft](#)
- [LONG](#) [GenSpace](#)
- [LONG](#) [SpaceLeft](#)
- [LONG](#) [putinsize](#)
- [LONG](#) [ninterms](#)
- [int](#) [MaxPatches](#)
- [int](#) [MaxFpatches](#)
- [int](#) [type](#)
- [int](#) [lPatch](#)
- [int](#) [fPatchN1](#)
- [int](#) [PolyWise](#)
- [int](#) [PolyFlag](#)
- [int](#) [cBufferSize](#)
- [int](#) [maxtermsize](#)
- [int](#) [newmaxtermsize](#)
- [int](#) [outputmode](#)
- [int](#) [stagelevel](#)
- [WORD](#) [fPatchN](#)
- [WORD](#) [inNum](#)
- [WORD](#) [stage4](#)

3.130.1 Detailed Description

The struct SORTING is used to control a sort operation. It includes a small and a large buffer and arrays for keeping track of various stages of the (merge) sorts. Each sort level has its own struct and different levels can have different sizes for its arrays. Also different threads have their own set of SORTING structs.

Definition at line 1115 of file [structs.h](#).

3.130.2 Field Documentation

3.130.2.1 file

```
FILEHANDLE sOrT::file
```

Definition at line 1116 of file [structs.h](#).

3.130.2.2 SizeInFile

```
POSITION sOrT::SizeInFile[3]
```

Definition at line 1117 of file [structs.h](#).

3.130.2.3 lBuffer

```
WORD* sOrT::lBuffer
```

Definition at line 1118 of file [structs.h](#).

3.130.2.4 lTop

```
WORD* sOrT::lTop
```

Definition at line 1119 of file [structs.h](#).

3.130.2.5 lFill

```
WORD* sOrT::lFill
```

Definition at line 1120 of file [structs.h](#).

3.130.2.6 used

```
WORD* sOrT::used
```

Definition at line 1121 of file [structs.h](#).

3.130.2.7 sBuffer

```
WORD* sOrT::sBuffer
```

Definition at line 1122 of file [structs.h](#).

3.130.2.8 sTop

```
WORD* sOrT::sTop
```

Definition at line 1123 of file [structs.h](#).

3.130.2.9 sTop2

```
WORD* sOrT::sTop2
```

Definition at line 1124 of file [structs.h](#).

3.130.2.10 sHalf

```
WORD* sOrT::sHalf
```

Definition at line 1125 of file [structs.h](#).

3.130.2.11 sFill

```
WORD* sOrT::sFill
```

Definition at line 1126 of file [structs.h](#).

3.130.2.12 sPointer

```
WORD** sOrT::sPointer
```

Definition at line 1127 of file [structs.h](#).

3.130.2.13 PoinFill

```
WORD** sOrT::PoinFill
```

Definition at line 1128 of file [structs.h](#).

3.130.2.14 cBuffer

```
WORD* sOrT::cBuffer
```

Definition at line 1129 of file [structs.h](#).

3.130.2.15 Patches

WORD** sOrT::Patches

Definition at line 1130 of file [structs.h](#).

3.130.2.16 pStop

WORD** sOrT::pStop

Definition at line 1131 of file [structs.h](#).

3.130.2.17 poina

WORD** sOrT::poina

Definition at line 1132 of file [structs.h](#).

3.130.2.18 poin2a

WORD** sOrT::poin2a

Definition at line 1133 of file [structs.h](#).

3.130.2.19 ktoi

WORD* sOrT::ktoi

Definition at line 1134 of file [structs.h](#).

3.130.2.20 tree

WORD* sOrT::tree

Definition at line 1135 of file [structs.h](#).

3.130.2.21 fPatches

POSITION* sOrT::fPatches

Definition at line 1141 of file [structs.h](#).

3.130.2.22 inPatches

POSITION* sOrT::inPatches

Definition at line 1142 of file [structs.h](#).

3.130.2.23 fPatchesStop

`POSITION* sOrT::fPatchesStop`

Definition at line 1143 of file [structs.h](#).

3.130.2.24 iPatches

`POSITION* sOrT::iPatches`

Definition at line 1144 of file [structs.h](#).

3.130.2.25 f

`FILEHANDLE* sOrT::f`

Definition at line 1145 of file [structs.h](#).

3.130.2.26 ff

`FILEHANDLE** sOrT::ff`

Definition at line 1146 of file [structs.h](#).

3.130.2.27 sTerms

`LONG sOrT::sTerms`

Definition at line 1147 of file [structs.h](#).

3.130.2.28 LargeSize

`LONG sOrT::LargeSize`

Definition at line 1148 of file [structs.h](#).

3.130.2.29 SmallSize

`LONG sOrT::SmallSize`

Definition at line 1149 of file [structs.h](#).

3.130.2.30 SmallEsize

`LONG sOrT::SmallEsize`

Definition at line 1150 of file [structs.h](#).

3.130.2.31 TermsInSmall

LONG sOrT::TermsInSmall

Definition at line 1151 of file [structs.h](#).

3.130.2.32 Terms2InSmall

LONG sOrT::Terms2InSmall

Definition at line 1152 of file [structs.h](#).

3.130.2.33 GenTerms

LONG sOrT::GenTerms

Definition at line 1153 of file [structs.h](#).

3.130.2.34 TermsLeft

LONG sOrT::TermsLeft

Definition at line 1154 of file [structs.h](#).

3.130.2.35 GenSpace

LONG sOrT::GenSpace

Definition at line 1155 of file [structs.h](#).

3.130.2.36 SpaceLeft

LONG sOrT::SpaceLeft

Definition at line 1156 of file [structs.h](#).

3.130.2.37 putinsize

LONG sOrT::putinsize

Definition at line 1157 of file [structs.h](#).

3.130.2.38 ninterms

LONG sOrT::ninterms

Definition at line 1158 of file [structs.h](#).

3.130.2.39 MaxPatches

```
int sOrT::MaxPatches
```

Definition at line [1159](#) of file [structs.h](#).

3.130.2.40 MaxFpatches

```
int sOrT::MaxFpatches
```

Definition at line [1160](#) of file [structs.h](#).

3.130.2.41 type

```
int sOrT::type
```

Definition at line [1161](#) of file [structs.h](#).

3.130.2.42 lPatch

```
int sOrT::lPatch
```

Definition at line [1162](#) of file [structs.h](#).

3.130.2.43 fPatchN1

```
int sOrT::fPatchN1
```

Definition at line [1163](#) of file [structs.h](#).

3.130.2.44 PolyWise

```
int sOrT::PolyWise
```

Definition at line [1164](#) of file [structs.h](#).

3.130.2.45 PolyFlag

```
int sOrT::PolyFlag
```

Definition at line [1165](#) of file [structs.h](#).

3.130.2.46 cBufferSize

```
int sOrT::cBufferSize
```

Definition at line [1166](#) of file [structs.h](#).

3.130.2.47 maxtermsize

```
int sOrT::maxtermsize
```

Definition at line 1167 of file [structs.h](#).

3.130.2.48 newmaxtermsize

```
int sOrT::newmaxtermsize
```

Definition at line 1168 of file [structs.h](#).

3.130.2.49 outputmode

```
int sOrT::outputmode
```

Definition at line 1169 of file [structs.h](#).

3.130.2.50 stagelevel

```
int sOrT::stagelevel
```

Definition at line 1170 of file [structs.h](#).

3.130.2.51 fPatchN

```
WORD sOrT::fPatchN
```

Definition at line 1171 of file [structs.h](#).

3.130.2.52 inNum

```
WORD sOrT::inNum
```

Definition at line 1172 of file [structs.h](#).

3.130.2.53 stage4

```
WORD sOrT::stage4
```

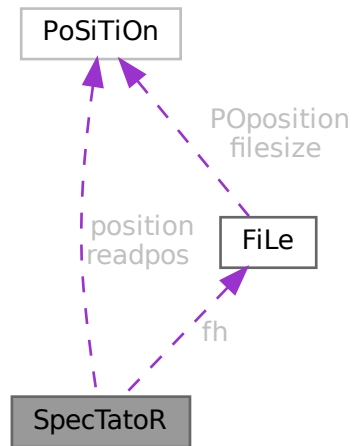
Definition at line 1173 of file [structs.h](#).

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

- [structs.h](#)

3.131 SpecTatoR Struct Reference

Collaboration diagram for SpecTatoR:



Data Fields

- `POSITION` `position`
- `POSITION` `readpos`
- `FILEHANDLE` * `fh`
- `char` * `name`
- `WORD` `exprnumber`
- `WORD` `flags`

3.131.1 Detailed Description

Definition at line 751 of file [structs.h](#).

3.131.2 Field Documentation

3.131.2.1 position

`POSITION` `SpecTatoR::position`

Definition at line 752 of file [structs.h](#).

3.131.2.2 readpos

`POSITION` `SpecTatoR::readpos`

Definition at line 753 of file [structs.h](#).

```
FILEHANDLE* SpecTatoR::fh
```

3.131.2.4 name

```
char* SpecTatoR::name
```

3.131.2.5 exprnumber

```
WORD  SpecTatoR::exprnumber
```

3.131.2.6 flags

WORD SpecTatoR::flags

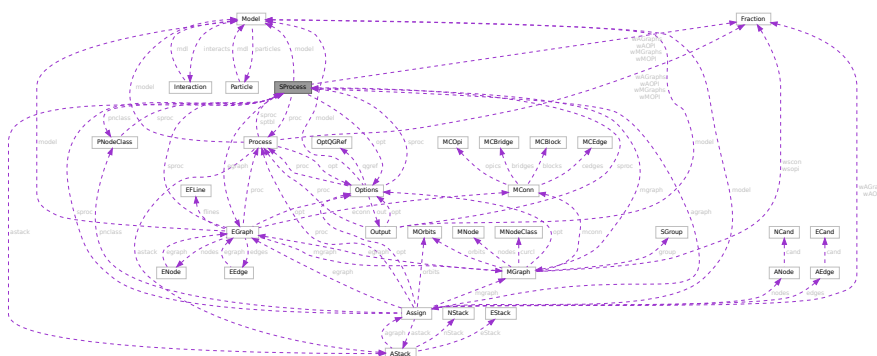
Definition at line 757 of file structs.h.

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

- `structs.h`

3.132 SProcess Class Reference

Collaboration diagram for SProcess:



Public Member Functions

- [SProcess](#) ([Model](#) *mdl, [Process](#) *prc, [Options](#) *opts, int sid, int *clst, int ncls, [NCInput](#) *cls)
- [SProcess](#) ([Model](#) *mdl, [Process](#) *prc, [Options](#) *opts, int sid, int *clst, int ncls, int *cdeg, int *ctyp, int *ptcl, int *cpl, int *cnum, int *cmind, int *cmaxd)
- void [prSProcess](#) (void)
- [BigInt](#) [generate](#) (void)
- void [assign](#) ([MGraph](#) *mgr)
- int [toMNodeClass](#) (int *ctyp, int *cldeg, int *clnum, int *cmind, int *cmaxd)
- [PNodeClass](#) * [match](#) ([MGraph](#) *mgr)
- void [endMGraph](#) ([MGraph](#) *mgr)
- void [endAGraph](#) ([EGraph](#) *egr)
- void [resultMGraph](#) ([BigInt](#) nmgraphs, [Fraction](#) mwsum, [BigInt](#) nmopi, [Fraction](#) mwopi)
- void [resultAGraph](#) ([BigInt](#) nagraphs, [Fraction](#) awsum, [BigInt](#) naopi, [Fraction](#) awopi)

Data Fields

- [Model](#) * [model](#)
- [Process](#) * [proc](#)
- [Options](#) * [opt](#)
- [PNodeClass](#) * [pnclass](#)
- [AStack](#) * [astack](#)
- [MGraph](#) * [mgraph](#)
- [EGraph](#) * [egraph](#)
- [Assign](#) * [agraph](#)
- int * [cl2nd](#)
- int * [nd2cl](#)
- int [nclass](#)
- int [ninitl](#)
- int [nfinal](#)
- int [nvert](#)
- int [clist](#) [GRCC_MAXNCPLG]
- int [id](#)
- int [loop](#)
- int [nNodes](#)
- int [nEdges](#)
- int [nExtern](#)
- int [ncouple](#)
- int [tCouple](#)
- int [DUMMY_PADDING](#)
- [BigInt](#) [mgrcount](#)
- [BigInt](#) [agrcount](#)
- [BigInt](#) [extperm](#)
- [BigInt](#) [nMGraphs](#)
- [BigInt](#) [nMOPI](#)
- [Fraction](#) [wMGraphs](#)
- [Fraction](#) [wMOPI](#)
- [BigInt](#) [nAGraphs](#)
- [BigInt](#) [nAOPI](#)
- [Fraction](#) [wAGraphs](#)
- [Fraction](#) [wAOPI](#)

3.132.1 Detailed Description

Definition at line 373 of file [grcc.h](#).

3.132.2 Constructor & Destructor Documentation

3.132.2.1 SProcess() [1/2]

```
SProcess::SProcess (
    Model * mdl,
    Process * prc,
    Options * opts,
    int sid,
    int * clst,
    int ncls,
    NCInput * cls )
```

Definition at line 2997 of file [grcc.cc](#).

3.132.2.2 SProcess() [2/2]

```
SProcess::SProcess (
    Model * mdl,
    Process * prc,
    Options * opts,
    int sid,
    int * clst,
    int ncls,
    int * cdeg,
    int * ctyp,
    int * ptcl,
    int * cpl,
    int * cnum,
    int * cmind,
    int * cmaxd )
```

Definition at line 2828 of file [grcc.cc](#).

3.132.2.3 ~SProcess()

```
SProcess::~~SProcess (
    void )
```

Definition at line 3159 of file [grcc.cc](#).

3.132.3 Member Function Documentation

3.132.3.1 prSProcess()

```
void SProcess::prSProcess (
    void )
```

Definition at line 3168 of file [grcc.cc](#).

3.132.3.2 generate()

```
BigInt SProcess::generate (
    void )
```

Definition at line [3180](#) of file [grcc.cc](#).

3.132.3.3 assign()

```
void SProcess::assign (
    MGraph * mgr )
```

Definition at line [3224](#) of file [grcc.cc](#).

3.132.3.4 toMNodeClass()

```
int SProcess::toMNodeClass (
    int * ctyp,
    int * cldeg,
    int * clnum,
    int * cmind,
    int * cmaxd )
```

Definition at line [3205](#) of file [grcc.cc](#).

3.132.3.5 match()

```
PNodeClass * SProcess::match (
    MGraph * mgr )
```

Definition at line [3241](#) of file [grcc.cc](#).

3.132.3.6 endMGraph()

```
void SProcess::endMGraph (
    MGraph * mgr )
```

Definition at line [3320](#) of file [grcc.cc](#).

3.132.3.7 endAGraph()

```
void SProcess::endAGraph (
    EGraph * egr )
```

Definition at line [3330](#) of file [grcc.cc](#).

3.132.3.8 resultMGraph()

```
void SProcess::resultMGraph (
    BigInt nmgraphs,
    Fraction mwsum,
    BigInt nmopi,
    Fraction mwopi )
```

Definition at line 3302 of file [grcc.cc](#).

3.132.3.9 resultAGraph()

```
void SProcess::resultAGraph (
    BigInt nagraphs,
    Fraction awsum,
    BigInt naopi,
    Fraction awopi )
```

Definition at line 3311 of file [grcc.cc](#).

3.132.4 Field Documentation

3.132.4.1 model

```
Model* SProcess::model
```

Definition at line 380 of file [grcc.h](#).

3.132.4.2 proc

```
Process* SProcess::proc
```

Definition at line 381 of file [grcc.h](#).

3.132.4.3 opt

```
Options* SProcess::opt
```

Definition at line 382 of file [grcc.h](#).

3.132.4.4 pnclass

```
PNodeClass* SProcess::pnclass
```

Definition at line 383 of file [grcc.h](#).

3.132.4.5 astack

`AStack* SProcess::astack`

Definition at line 384 of file [grcc.h](#).

3.132.4.6 mgraph

`MGraph* SProcess::mgraph`

Definition at line 386 of file [grcc.h](#).

3.132.4.7 egraph

`EGraph* SProcess::egraph`

Definition at line 387 of file [grcc.h](#).

3.132.4.8 agraph

`Assign* SProcess::agraph`

Definition at line 388 of file [grcc.h](#).

3.132.4.9 cl2nd

`int* SProcess::cl2nd`

Definition at line 390 of file [grcc.h](#).

3.132.4.10 nd2cl

`int* SProcess::nd2cl`

Definition at line 392 of file [grcc.h](#).

3.132.4.11 nclass

`int SProcess::nclass`

Definition at line 393 of file [grcc.h](#).

3.132.4.12 ninitl

`int SProcess::ninitl`

Definition at line 395 of file [grcc.h](#).

3.132.4.13 nfinal

```
int SProcess::nfinal
```

Definition at line 396 of file [grcc.h](#).

3.132.4.14 nvert

```
int SProcess::nvert
```

Definition at line 397 of file [grcc.h](#).

3.132.4.15 clist

```
int SProcess::clist[GRCC_MAXNCPLG]
```

Definition at line 399 of file [grcc.h](#).

3.132.4.16 id

```
int SProcess::id
```

Definition at line 400 of file [grcc.h](#).

3.132.4.17 loop

```
int SProcess::loop
```

Definition at line 401 of file [grcc.h](#).

3.132.4.18 nNodes

```
int SProcess::nNodes
```

Definition at line 402 of file [grcc.h](#).

3.132.4.19 nEdges

```
int SProcess::nEdges
```

Definition at line 403 of file [grcc.h](#).

3.132.4.20 nExtern

```
int SProcess::nExtern
```

Definition at line 404 of file [grcc.h](#).

3.132.4.21 ncouple

```
int SProcess::ncouple
```

Definition at line 405 of file [grcc.h](#).

3.132.4.22 tCouple

```
int SProcess::tCouple
```

Definition at line 406 of file [grcc.h](#).

3.132.4.23 DUMMYPADDING

```
int SProcess::DUMMYPADDING
```

Definition at line 408 of file [grcc.h](#).

3.132.4.24 mgrcount

```
BigInt SProcess::mgrcount
```

Definition at line 410 of file [grcc.h](#).

3.132.4.25 agrcount

```
BigInt SProcess::agrcount
```

Definition at line 411 of file [grcc.h](#).

3.132.4.26 extperm

```
BigInt SProcess::extperm
```

Definition at line 412 of file [grcc.h](#).

3.132.4.27 nMGraphs

```
BigInt SProcess::nMGraphs
```

Definition at line 415 of file [grcc.h](#).

3.132.4.28 nMOPI

```
BigInt SProcess::nMOPI
```

Definition at line 416 of file [grcc.h](#).

3.132.4.29 wMGraphs

`Fraction SProcess::wMGraphs`

Definition at line 417 of file [grcc.h](#).

3.132.4.30 wMOPI

`Fraction SProcess::wMOPI`

Definition at line 418 of file [grcc.h](#).

3.132.4.31 nAGraphs

`BigInt SProcess::nAGraphs`

Definition at line 420 of file [grcc.h](#).

3.132.4.32 nAOPI

`BigInt SProcess::nAOPI`

Definition at line 421 of file [grcc.h](#).

3.132.4.33 wAGraphs

`Fraction SProcess::wAGraphs`

Definition at line 422 of file [grcc.h](#).

3.132.4.34 wAOPI

`Fraction SProcess::wAOPI`

Definition at line 423 of file [grcc.h](#).

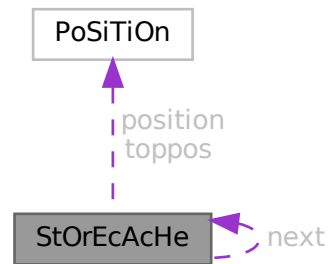
The documentation for this class was generated from the following files:

- [grcc.h](#)
- [grcc.cc](#)

3.133 StOrEcAcHe Struct Reference

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

Collaboration diagram for StOrEcAcHe:



Data Fields

- `POSITION position`
- `POSITION toppos`
- `struct StOrEcAcHe * next`
- `WORD buffer [2]`

3.133.1 Detailed Description

The struct of type STORECACHE is used by a caching system for reading terms from stored expressions. Each thread should have its own system of caches.

Definition at line 1045 of file [structs.h](#).

3.133.2 Field Documentation

3.133.2.1 position

```
POSITION StOrEcAcHe::position
```

Definition at line 1046 of file [structs.h](#).

3.133.2.2 toppos

```
POSITION StOrEcAcHe::toppos
```

Definition at line 1047 of file [structs.h](#).

3.133.2.3 next

```
struct StOrEcAcHe* StOrEcAcHe::next
```

Definition at line 1048 of file [structs.h](#).

3.133.2.4 buffer

```
WORD StOrEcAcHe::buffer[2]
```

Definition at line 1049 of file [structs.h](#).

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

- [structs.h](#)

3.134 STOREHEADER Struct Reference

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

Data Fields

- UBYTE [headermark](#) [8]
- UBYTE [lenWORD](#)
- UBYTE [lenLONG](#)
- UBYTE [lenPOS](#)
- UBYTE [lenPOINTER](#)
- UBYTE [endianness](#) [16]
- UBYTE [sSym](#)
- UBYTE [sInd](#)
- UBYTE [sVec](#)
- UBYTE [sFun](#)
- UBYTE [maxpower](#) [16]
- UBYTE [wildoffset](#) [16]
- UBYTE [revision](#)
- UBYTE [reserved](#) [512-8-4-16-4-16-16-1]

3.134.1 Detailed Description

Defines the structure of the file header for store-files and save-files.

The first 8 bytes serve as a unique mark to identity save-files that contain such a header. Older versions of FORM don't have this header and will write the POSITION of the next file index (struct [FiLeInDeX](#)) here, which is always different from this pattern.

It is always 512 bytes long.

Definition at line 75 of file [structs.h](#).

3.134.2 Field Documentation

3.134.2.1 headermark

```
UBYTE STOREHEADER::headermark[8]
```

Pattern for header identification. Old versions of FORM have a maximum sizeof(POSITION) of 8

Definition at line 76 of file [structs.h](#).

3.134.2.2 lenWORD

```
UBYTE STOREHEADER::lenWORD
```

Number of bytes for WORD

Definition at line 78 of file [structs.h](#).

Referenced by [WriteStoreHeader\(\)](#).

3.134.2.3 lenLONG

```
UBYTE STOREHEADER::lenLONG
```

Number of bytes for LONG

Definition at line 79 of file [structs.h](#).

Referenced by [WriteStoreHeader\(\)](#).

3.134.2.4 lenPOS

```
UBYTE STOREHEADER::lenPOS
```

Number of bytes for POSITION

Definition at line 80 of file [structs.h](#).

Referenced by [WriteStoreHeader\(\)](#).

3.134.2.5 lenPOINTER

```
UBYTE STOREHEADER::lenPOINTER
```

Number of bytes for void *

Definition at line 81 of file [structs.h](#).

Referenced by [WriteStoreHeader\(\)](#).

3.134.2.6 endianness

```
UBYTE STOREHEADER::endianness[16]
```

Used to determine endianness, sizeof(int) should be ≤ 16

Definition at line 82 of file [structs.h](#).

Referenced by [WriteStoreHeader\(\)](#).

3.134.2.7 sSym

```
UBYTE STOREHEADER::sSym
```

sizeof(struct SyMbOl)

Definition at line 83 of file [structs.h](#).

Referenced by [WriteStoreHeader\(\)](#).

3.134.2.8 sInd

```
UBYTE STOREHEADER::sInd
```

sizeof(struct InDeX)

Definition at line 84 of file [structs.h](#).

Referenced by [WriteStoreHeader\(\)](#).

3.134.2.9 sVec

```
UBYTE STOREHEADER::sVec
```

sizeof(struct VeCtOr)

Definition at line 85 of file [structs.h](#).

Referenced by [WriteStoreHeader\(\)](#).

3.134.2.10 sFun

```
UBYTE STOREHEADER::sFun
```

sizeof(struct FuNcTiOn)

Definition at line 86 of file [structs.h](#).

Referenced by [WriteStoreHeader\(\)](#).

3.134.2.11 maxpower

```
UBYTE STOREHEADER::maxpower[16]
```

Maximum power, see `#MAXPOWER`

Definition at line 87 of file [structs.h](#).

Referenced by [WriteStoreHeader\(\)](#).

3.134.2.12 wildoffset

```
UBYTE STOREHEADER::wildoffset[16]
```

`#WILDOFFSET` macro

Definition at line 88 of file [structs.h](#).

Referenced by [WriteStoreHeader\(\)](#).

3.134.2.13 revision

```
UBYTE STOREHEADER::revision
```

Revision number of save-file system

Definition at line 89 of file [structs.h](#).

3.134.2.14 reserved

```
UBYTE STOREHEADER::reserved[512-8-4-16-4-16-16-1]
```

Padding to 512 bytes

Definition at line 90 of file [structs.h](#).

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

- [structs.h](#)

3.135 StreaM Struct Reference

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

Data Fields

- off_t [fileposition](#)
- off_t [linenumber](#)
- off_t [prevline](#)
- UBYTE * [buffer](#)
- UBYTE * [pointer](#)
- UBYTE * [top](#)
- UBYTE * [FoldName](#)
- UBYTE * [name](#)
- UBYTE * [pname](#)
- LONG [buffersize](#)
- LONG [bufferposition](#)
- LONG [inbuffer](#)
- int [previous](#)
- int [handle](#)
- int [type](#)
- int [prevars](#)
- int [previousNoShowInput](#)
- int [eqnum](#)
- int [afterwards](#)
- int [olddelay](#)
- int [oldnoshowinput](#)
- UBYTE [isnextchar](#)
- UBYTE [nextchar](#) [2]
- UBYTE [reserved](#)

3.135.1 Detailed Description

Input is read from 'streams' which are represented by objects of type STREAM. A stream can be a file, a do-loop, a procedure, the string value of a preprocessor variable When a new stream is opened we have to keep information about where to fall back in the parent stream to allow this to happen even in the middle of reading names etc as would be the case with a`i`b

Definition at line [723](#) of file [structs.h](#).

3.135.2 Field Documentation

3.135.2.1 fileposition

```
off_t Stream::fileposition
```

Definition at line [724](#) of file [structs.h](#).

3.135.2.2 linenumber

```
off_t Stream::linenumber
```

Definition at line [725](#) of file [structs.h](#).

3.135.2.3 prevline

```
off_t Stream::prevline
```

Definition at line 726 of file [structs.h](#).

3.135.2.4 buffer

```
UBYTE* Stream::buffer
```

[D] Size in buffersize

Definition at line 727 of file [structs.h](#).

3.135.2.5 pointer

```
UBYTE* Stream::pointer
```

pointer into buffer memory

Definition at line 728 of file [structs.h](#).

3.135.2.6 top

```
UBYTE* Stream::top
```

pointer into buffer memory

Definition at line 729 of file [structs.h](#).

3.135.2.7 FoldName

```
UBYTE* Stream::FoldName
```

[D]

Definition at line 730 of file [structs.h](#).

3.135.2.8 name

```
UBYTE* Stream::name
```

[D]

Definition at line 731 of file [structs.h](#).

3.135.2.9 pname

```
UBYTE* Stream::pname
```

for DOLLARSTREAM and PREVARSTREAM it points always to name, else it is undefined

Definition at line 732 of file [structs.h](#).

3.135.2.10 buffersize

```
LONG Stream::buffersize
```

Definition at line 734 of file [structs.h](#).

3.135.2.11 bufferposition

```
LONG Stream::bufferposition
```

Definition at line 735 of file [structs.h](#).

3.135.2.12 inbuffer

```
LONG Stream::inbuffer
```

Definition at line 736 of file [structs.h](#).

3.135.2.13 previous

```
int Stream::previous
```

Definition at line 737 of file [structs.h](#).

3.135.2.14 handle

```
int Stream::handle
```

Definition at line 738 of file [structs.h](#).

3.135.2.15 type

```
int Stream::type
```

Definition at line 739 of file [structs.h](#).

3.135.2.16 prevars

```
int Stream::prevars
```

Definition at line 740 of file [structs.h](#).

3.135.2.17 previousNoShowInput

```
int Stream::previousNoShowInput
```

Definition at line 741 of file [structs.h](#).

3.135.2.18 eqnum

```
int Stream::eqnum
```

Definition at line 742 of file [structs.h](#).

3.135.2.19 afterwards

```
int Stream::afterwards
```

Definition at line 743 of file [structs.h](#).

3.135.2.20 olddelay

```
int Stream::olddelay
```

Definition at line 744 of file [structs.h](#).

3.135.2.21 oldnoshowinput

```
int Stream::oldnoshowinput
```

Definition at line 745 of file [structs.h](#).

3.135.2.22 isnextchar

```
UBYTE Stream::isnextchar
```

Definition at line 746 of file [structs.h](#).

3.135.2.23 nextchar

```
UBYTE Stream::nextchar[2]
```

Definition at line 747 of file [structs.h](#).

3.135.2.24 reserved

UBYTE Stream::reserved

Definition at line 748 of file [structs.h](#).

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

- [structs.h](#)

3.136 SuBbUf Struct Reference

Data Fields

- WORD [subexpnum](#)
- WORD [buffernum](#)

3.136.1 Detailed Description

Definition at line 264 of file [compiler.c](#).

3.136.2 Field Documentation

3.136.2.1 subexpnum

WORD SuBbUf::subexpnum

Definition at line 265 of file [compiler.c](#).

3.136.2.2 buffernum

WORD SuBbUf::buffernum

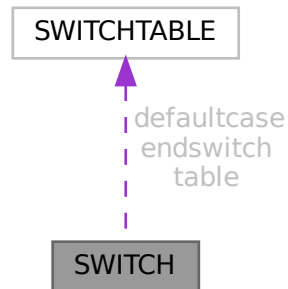
Definition at line 266 of file [compiler.c](#).

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

- [compiler.c](#)

3.137 SWITCH Struct Reference

Collaboration diagram for SWITCH:



Data Fields

- [SWICHTABLE * table](#)
- [SWICHTABLE defaultcase](#)
- [SWICHTABLE endswitch](#)
- WORD [typetable](#)
- WORD [maxcase](#)
- WORD [mincase](#)
- WORD [numcases](#)
- WORD [tablesize](#)
- WORD [caseoffset](#)
- WORD [iflevel](#)
- WORD [whilelevel](#)
- WORD [nestingsum](#)
- WORD [padding](#)

3.137.1 Detailed Description

Definition at line [1345](#) of file [structs.h](#).

3.137.2 Field Documentation

3.137.2.1 table

[SWICHTABLE*](#) [SWITCH::table](#)

Definition at line [1346](#) of file [structs.h](#).

3.137.2.2 defaultcase

`SWITCHTABLE SWITCH::defaultcase`

Definition at line 1347 of file [structs.h](#).

3.137.2.3 endswitch

`SWITCHTABLE SWITCH::endswitch`

Definition at line 1348 of file [structs.h](#).

3.137.2.4 typetable

`WORD SWITCH::typetable`

Definition at line 1349 of file [structs.h](#).

3.137.2.5 maxcase

`WORD SWITCH::maxcase`

Definition at line 1350 of file [structs.h](#).

3.137.2.6 mincase

`WORD SWITCH::mincase`

Definition at line 1351 of file [structs.h](#).

3.137.2.7 numcases

`WORD SWITCH::numcases`

Definition at line 1352 of file [structs.h](#).

3.137.2.8 tablesize

`WORD SWITCH::tablesize`

Definition at line 1353 of file [structs.h](#).

3.137.2.9 caseoffset

`WORD SWITCH::caseoffset`

Definition at line 1354 of file [structs.h](#).

3.137.2.10 iflevel

WORD SWITCH::iflevel

Definition at line 1355 of file [structs.h](#).

3.137.2.11 whilelevel

WORD SWITCH::whilelevel

Definition at line 1356 of file [structs.h](#).

3.137.2.12 nestingsum

WORD SWITCH::nestingsum

Definition at line 1357 of file [structs.h](#).

3.137.2.13 padding

WORD SWITCH::padding

Definition at line 1358 of file [structs.h](#).

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

- [structs.h](#)

3.138 SWITCHTABLE Struct Reference

Data Fields

- WORD [ncase](#)
- WORD [value](#)
- WORD [compbuffer](#)

3.138.1 Detailed Description

Definition at line 1339 of file [structs.h](#).

3.138.2 Field Documentation

3.138.2.1 ncase

WORD SWITCHTABLE::ncase

Definition at line 1340 of file [structs.h](#).

3.138.2.2 value

WORD SWITCHTABLE::value

Definition at line 1341 of file [structs.h](#).

3.138.2.3 compbuffer

WORD SWITCHTABLE::compbuffer

Definition at line 1342 of file [structs.h](#).

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

- [structs.h](#)

3.139 SyMbOI Struct Reference

Data Fields

- LONG [name](#)
- WORD [minpower](#)
- WORD [maxpower](#)
- WORD [complex](#)
- WORD [number](#)
- WORD [flags](#)
- WORD [node](#)
- WORD [namesize](#)
- WORD [dimension](#)

3.139.1 Detailed Description

Definition at line 416 of file [structs.h](#).

3.139.2 Field Documentation

3.139.2.1 name

LONG SyMbOl::name

Definition at line 417 of file [structs.h](#).

3.139.2.2 minpower

WORD SyMbOl::minpower

Definition at line 418 of file [structs.h](#).

3.139.2.3 maxpower

WORD SyMbOl::maxpower

Definition at line 419 of file [structs.h](#).

3.139.2.4 complex

WORD SyMbOl::complex

Definition at line 420 of file [structs.h](#).

3.139.2.5 number

WORD SyMbOl::number

Definition at line 421 of file [structs.h](#).

3.139.2.6 flags

WORD SyMbOl::flags

Definition at line 422 of file [structs.h](#).

3.139.2.7 node

WORD SyMbOl::node

Definition at line 423 of file [structs.h](#).

3.139.2.8 namesize

WORD SyMbOl::namesize

Definition at line 424 of file [structs.h](#).

3.139.2.9 dimension

WORD SyMbOl::dimension

Definition at line 425 of file [structs.h](#).

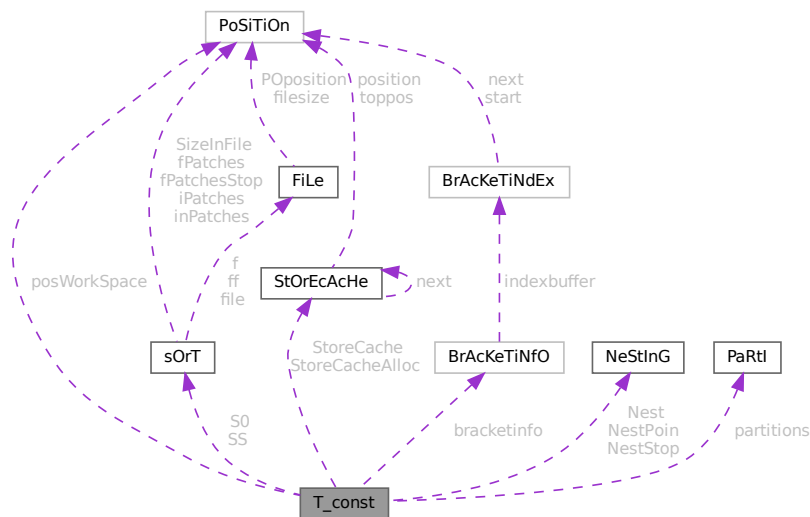
The documentation for this struct was generated from the following file:

- [structs.h](#)

3.140 T_const Struct Reference

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

Collaboration diagram for T_const:



Data Fields

- `SORTING * S0`
- `SORTING * SS`
- `NESTING Nest`
- `NESTING NestStop`
- `NESTING NestPoin`
- `WORD * BrackBuf`
- `STORECACHE StoreCache`
- `STORECACHE StoreCacheAlloc`
- `WORD ** pWorkspace`
- `LONG * lWorkspace`
- `POSITION * posWorkspace`
- `WORD * Workspace`
- `WORD * WorkTop`
- `WORD * WorkPointer`
- `int * RepCount`
- `int * RepTop`
- `WORD * WildArgTaken`
- `UWORD * factorials`
- `WORD * small_power_n`
- `UWORD ** small_power`
- `UWORD * bernoullis`
- `WORD * primelist`
- `LONG * pfac`
- `LONG * pBer`
- `WORD * TMaddr`

- WORD * WildMask
- WORD * previousEfactor
- WORD ** TermMemHeap
- UWORD ** NumberMemHeap
- UWORD ** CacheNumberMemHeap
- BRACKETINFO * bracketinfo
- WORD ** ListPoly
- WORD * ListSymbols
- UWORD * NumMem
- WORD * TopologiesTerm
- WORD * TopologiesStart
- PARTI partitions
- LONG sBer
- LONG pWorkPointer
- LONG IWorkPointer
- LONG posWorkPointer
- LONG lnNumMem
- int sfact
- int mfac
- int ebufnum
- int fbufnum
- int allbufnum
- int aebufnum
- int idallflag
- int idallnum
- int idallmaxnum
- int WildcardBufferSize
- int TermMemMax
- int TermMemTop
- int NumberMemMax
- int NumberMemTop
- int CacheNumberMemMax
- int CacheNumberMemTop
- int bracketindexflag
- int optentimes
- int ListSymbolsSize
- int NumListSymbols
- int numpoly
- int LeaveNegative
- int TrimPower
- WORD small_power_maxx
- WORD small_power_maxn
- WORD dummysubexp [SUBEXPSIZE+4]
- WORD comsym [8]
- WORD comnum [4]
- WORD comfun [FUNHEAD+4]
- WORD comind [7]
- WORD MinVecArg [7+ARGHEAD]
- WORD FunArg [4+ARGHEAD+FUNHEAD]
- WORD locwildvalue [SUBEXPSIZE]
- WORD mulpat [SUBEXPSIZE+5]
- WORD proexp [SUBEXPSIZE+5]
- WORD TMout [40]
- WORD TMbuff
- WORD TMdolfac

- WORD [nfac](#)
- WORD [nBer](#)
- WORD [mBer](#)
- WORD [PolyAct](#)
- WORD [RecFlag](#)
- WORD [inprimelist](#)
- WORD [sizeprimelist](#)
- WORD [fromindex](#)
- WORD [setinterntopo](#)
- WORD [setexterntopo](#)
- WORD [TopologiesLevel](#)
- WORD [TopologiesOptions](#) [2]

3.140.1 Detailed Description

The [T_const](#) struct is part of the global data and resides either in the ALLGLOBALS struct A, or the ALLPRIVATEs struct B (TFORM) under the name T We see it used with the macro AT as in AT.WorkPointer It has variables that are private to each thread, most of which have to be defined at startup.

Definition at line [2035](#) of file [structs.h](#).

3.140.2 Field Documentation

3.140.2.1 S0

[SORTING](#)* [T_const::S0](#)

Definition at line [2039](#) of file [structs.h](#).

3.140.2.2 SS

[SORTING](#)* [T_const::SS](#)

Definition at line [2040](#) of file [structs.h](#).

3.140.2.3 Nest

[NESTING](#) [T_const::Nest](#)

Definition at line [2041](#) of file [structs.h](#).

3.140.2.4 NestStop

[NESTING](#) [T_const::NestStop](#)

Definition at line [2042](#) of file [structs.h](#).

3.140.2.5 NestPoin

`NESTING T_const::NestPoin`

Definition at line 2043 of file [structs.h](#).

3.140.2.6 BrackBuf

`WORD* T_const::BrackBuf`

Definition at line 2044 of file [structs.h](#).

3.140.2.7 StoreCache

`STORECACHE T_const::StoreCache`

Definition at line 2045 of file [structs.h](#).

3.140.2.8 StoreCacheAlloc

`STORECACHE T_const::StoreCacheAlloc`

Definition at line 2046 of file [structs.h](#).

3.140.2.9 pWorkSpace

`WORD** T_const::pWorkSpace`

Definition at line 2047 of file [structs.h](#).

3.140.2.10 lWorkSpace

`LONG* T_const::lWorkSpace`

Definition at line 2048 of file [structs.h](#).

3.140.2.11 posWorkSpace

`POSITION* T_const::posWorkSpace`

Definition at line 2049 of file [structs.h](#).

3.140.2.12 WorkSpace

`WORD* T_const::WorkSpace`

Definition at line 2050 of file [structs.h](#).

3.140.2.13 WorkTop

WORD* T_const::WorkTop

Definition at line 2051 of file [structs.h](#).

3.140.2.14 WorkPointer

WORD* T_const::WorkPointer

Definition at line 2052 of file [structs.h](#).

3.140.2.15 RepCount

int* T_const::RepCount

Definition at line 2053 of file [structs.h](#).

3.140.2.16 RepTop

int* T_const::RepTop

Definition at line 2054 of file [structs.h](#).

3.140.2.17 WildArgTaken

WORD* T_const::WildArgTaken

Definition at line 2055 of file [structs.h](#).

3.140.2.18 factorials

UWORD* T_const::factorials

Definition at line 2056 of file [structs.h](#).

3.140.2.19 small_power_n

WORD* T_const::small_power_n

Definition at line 2057 of file [structs.h](#).

3.140.2.20 small_power

UWORD** T_const::small_power

Definition at line 2058 of file [structs.h](#).

3.140.2.21 bernoullis

UWORD* T_const::bernoullis

Definition at line 2059 of file [structs.h](#).

3.140.2.22 primelist

WORD* T_const::primelist

Definition at line 2060 of file [structs.h](#).

3.140.2.23 pfac

LONG* T_const::pfac

Definition at line 2061 of file [structs.h](#).

3.140.2.24 pBer

LONG* T_const::pBer

Definition at line 2062 of file [structs.h](#).

3.140.2.25 TMaddr

WORD* T_const::TMaddr

Definition at line 2063 of file [structs.h](#).

3.140.2.26 WildMask

WORD* T_const::WildMask

Definition at line 2064 of file [structs.h](#).

3.140.2.27 previousEfactor

WORD* T_const::previousEfactor

Definition at line 2065 of file [structs.h](#).

3.140.2.28 TermMemHeap

WORD** T_const::TermMemHeap

Definition at line 2066 of file [structs.h](#).

3.140.2.29 NumberMemHeap

UWORD** T_const::NumberMemHeap

Definition at line 2067 of file [structs.h](#).

3.140.2.30 CacheNumberMemHeap

UWORD** T_const::CacheNumberMemHeap

Definition at line 2068 of file [structs.h](#).

3.140.2.31 bracketinfo

BRACKETINFO* T_const::bracketinfo

Definition at line 2069 of file [structs.h](#).

3.140.2.32 ListPoly

WORD** T_const::ListPoly

Definition at line 2070 of file [structs.h](#).

3.140.2.33 ListSymbols

WORD* T_const::ListSymbols

Definition at line 2071 of file [structs.h](#).

3.140.2.34 NumMem

UWORD* T_const::NumMem

Definition at line 2072 of file [structs.h](#).

3.140.2.35 TopologiesTerm

WORD* T_const::TopologiesTerm

Definition at line 2073 of file [structs.h](#).

3.140.2.36 TopologiesStart

WORD* T_const::TopologiesStart

Definition at line 2074 of file [structs.h](#).

3.140.2.37 partitions

`PARTI T_const::partitions`

Definition at line 2083 of file [structs.h](#).

3.140.2.38 sBer

`LONG T_const::sBer`

Definition at line 2084 of file [structs.h](#).

3.140.2.39 pWorkPointer

`LONG T_const::pWorkPointer`

Definition at line 2085 of file [structs.h](#).

3.140.2.40 lWorkPointer

`LONG T_const::lWorkPointer`

Definition at line 2086 of file [structs.h](#).

3.140.2.41 posWorkPointer

`LONG T_const::posWorkPointer`

Definition at line 2087 of file [structs.h](#).

3.140.2.42 InNumMem

`LONG T_const::InNumMem`

Definition at line 2088 of file [structs.h](#).

3.140.2.43 sfact

`int T_const::sfact`

Definition at line 2089 of file [structs.h](#).

3.140.2.44 mfac

`int T_const::mfac`

Definition at line 2090 of file [structs.h](#).

3.140.2.45 ebufnum

```
int T_const::ebufnum
```

Definition at line 2091 of file [structs.h](#).

3.140.2.46 fbufnum

```
int T_const::fbufnum
```

Definition at line 2092 of file [structs.h](#).

3.140.2.47 allbufnum

```
int T_const::allbufnum
```

Definition at line 2093 of file [structs.h](#).

3.140.2.48 aebufnum

```
int T_const::aebufnum
```

Definition at line 2094 of file [structs.h](#).

3.140.2.49 idallflag

```
int T_const::idallflag
```

Definition at line 2095 of file [structs.h](#).

3.140.2.50 idallnum

```
int T_const::idallnum
```

Definition at line 2096 of file [structs.h](#).

3.140.2.51 idallmaxnum

```
int T_const::idallmaxnum
```

Definition at line 2097 of file [structs.h](#).

3.140.2.52 WildcardBufferSize

```
int T_const::WildcardBufferSize
```

Definition at line 2098 of file [structs.h](#).

3.140.2.53 TermMemMax

```
int T_const::TermMemMax
```

Definition at line [2107](#) of file [structs.h](#).

3.140.2.54 TermMemTop

```
int T_const::TermMemTop
```

Definition at line [2108](#) of file [structs.h](#).

3.140.2.55 NumberMemMax

```
int T_const::NumberMemMax
```

Definition at line [2109](#) of file [structs.h](#).

3.140.2.56 NumberMemTop

```
int T_const::NumberMemTop
```

Definition at line [2110](#) of file [structs.h](#).

3.140.2.57 CacheNumberMemMax

```
int T_const::CacheNumberMemMax
```

Definition at line [2111](#) of file [structs.h](#).

3.140.2.58 CacheNumberMemTop

```
int T_const::CacheNumberMemTop
```

Definition at line [2112](#) of file [structs.h](#).

3.140.2.59 bracketindexflag

```
int T_const::bracketindexflag
```

Definition at line [2113](#) of file [structs.h](#).

3.140.2.60 optimitimes

```
int T_const::optimitimes
```

Definition at line [2114](#) of file [structs.h](#).

3.140.2.61 ListSymbolsSize

```
int T_const::ListSymbolsSize
```

Definition at line 2115 of file [structs.h](#).

3.140.2.62 NumListSymbols

```
int T_const::NumListSymbols
```

Definition at line 2116 of file [structs.h](#).

3.140.2.63 numpoly

```
int T_const::numpoly
```

Definition at line 2117 of file [structs.h](#).

3.140.2.64 LeaveNegative

```
int T_const::LeaveNegative
```

Definition at line 2118 of file [structs.h](#).

3.140.2.65 TrimPower

```
int T_const::TrimPower
```

Definition at line 2119 of file [structs.h](#).

3.140.2.66 small_power_maxx

```
WORD T_const::small_power_maxx
```

Definition at line 2120 of file [structs.h](#).

3.140.2.67 small_power_maxn

```
WORD T_const::small_power_maxn
```

Definition at line 2121 of file [structs.h](#).

3.140.2.68 dummysubexp

```
WORD T_const::dummysubexp[SUBEXPSIZE+4]
```

Definition at line 2122 of file [structs.h](#).

3.140.2.69 comsym

```
WORD T_const::comsym[8]
```

Definition at line 2123 of file [structs.h](#).

3.140.2.70 comnum

```
WORD T_const::comnum[4]
```

Definition at line 2124 of file [structs.h](#).

3.140.2.71 comfun

```
WORD T_const::comfun[FUNHEAD+4]
```

Definition at line 2125 of file [structs.h](#).

3.140.2.72 comind

```
WORD T_const::comind[7]
```

Definition at line 2127 of file [structs.h](#).

3.140.2.73 MinVecArg

```
WORD T_const::MinVecArg[7+ARGHEAD]
```

Definition at line 2128 of file [structs.h](#).

3.140.2.74 FunArg

```
WORD T_const::FunArg[4+ARGHEAD+FUNHEAD]
```

Definition at line 2129 of file [structs.h](#).

3.140.2.75 locwildvalue

```
WORD T_const::locwildvalue[SUBEXPSIZE]
```

Definition at line 2130 of file [structs.h](#).

3.140.2.76 mulpat

```
WORD T_const::mulpat[SUBEXPSIZE+5]
```

Definition at line 2131 of file [structs.h](#).

3.140.2.77 proexp

WORD T_const::proexp[SUBEXPSIZE+5]

Definition at line 2133 of file [structs.h](#).

3.140.2.78 TMout

WORD T_const::TMout[40]

Definition at line 2134 of file [structs.h](#).

3.140.2.79 TMbuff

WORD T_const::TMbuff

Definition at line 2135 of file [structs.h](#).

3.140.2.80 TMdolfac

WORD T_const::TMdolfac

Definition at line 2136 of file [structs.h](#).

3.140.2.81 nfac

WORD T_const::nfac

Definition at line 2137 of file [structs.h](#).

3.140.2.82 nBer

WORD T_const::nBer

Definition at line 2138 of file [structs.h](#).

3.140.2.83 mBer

WORD T_const::mBer

Definition at line 2139 of file [structs.h](#).

3.140.2.84 PolyAct

WORD T_const::PolyAct

Definition at line 2140 of file [structs.h](#).

3.140.2.85 RecFlag

WORD T_const::RecFlag

Definition at line [2141](#) of file [structs.h](#).

3.140.2.86 inprimelist

WORD T_const::inprimelist

Definition at line [2142](#) of file [structs.h](#).

3.140.2.87 sizeprimelist

WORD T_const::sizeprimelist

Definition at line [2143](#) of file [structs.h](#).

3.140.2.88 fromindex

WORD T_const::fromindex

Definition at line [2144](#) of file [structs.h](#).

3.140.2.89 setinterntopo

WORD T_const::setinterntopo

Definition at line [2145](#) of file [structs.h](#).

3.140.2.90 setexterntopo

WORD T_const::setexterntopo

Definition at line [2146](#) of file [structs.h](#).

3.140.2.91 TopologiesLevel

WORD T_const::TopologiesLevel

Definition at line [2147](#) of file [structs.h](#).

3.140.2.92 TopologiesOptions

WORD T_const::TopologiesOptions[2]

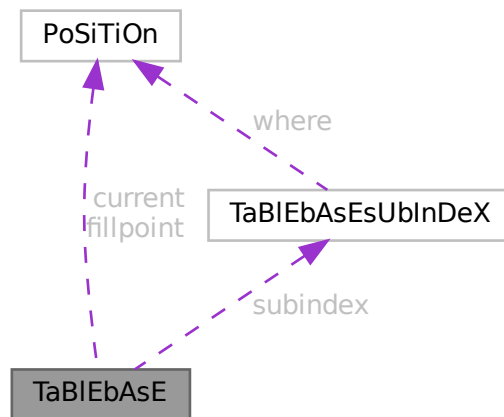
Definition at line 2148 of file [structs.h](#).

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

- [structs.h](#)

3.141 TaBIEbAsE Struct Reference

Collaboration diagram for TaBIEbAsE:



Data Fields

- [POSITION](#) fillpoint
- [POSITION](#) current
- UBYTE * [name](#)
- int * [tablenumbers](#)
- [TABLEBASESUBINDEX](#) * [subindex](#)
- int [numtables](#)

3.141.1 Detailed Description

Definition at line 594 of file [structs.h](#).

3.141.2 Field Documentation

3.141.2.1 fillpoint

`POSITION TableEbAsE::fillpoint`

Definition at line 595 of file [structs.h](#).

3.141.2.2 current

`POSITION TableEbAsE::current`

Definition at line 596 of file [structs.h](#).

3.141.2.3 name

`UBYTE* TableEbAsE::name`

Definition at line 597 of file [structs.h](#).

3.141.2.4 tablenumbers

`int* TableEbAsE::tablenumbers`

Definition at line 598 of file [structs.h](#).

3.141.2.5 subindex

`TABLEBASESUBINDEX* TableEbAsE::subindex`

Definition at line 599 of file [structs.h](#).

3.141.2.6 numtables

`int TableEbAsE::numtables`

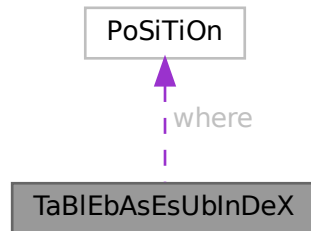
Definition at line 600 of file [structs.h](#).

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

- [structs.h](#)

3.142 TaBlEbAsEsUbInDeX Struct Reference

Collaboration diagram for TaBlEbAsEsUbInDeX:



Data Fields

- [POSITION](#) [where](#)
- [LONG](#) [size](#)

3.142.1 Detailed Description

Definition at line [585](#) of file [structs.h](#).

3.142.2 Field Documentation

3.142.2.1 where

[POSITION](#) `TaBlEbAsEsUbInDeX::where`

Definition at line [586](#) of file [structs.h](#).

3.142.2.2 size

[LONG](#) `TaBlEbAsEsUbInDeX::size`

Definition at line [587](#) of file [structs.h](#).

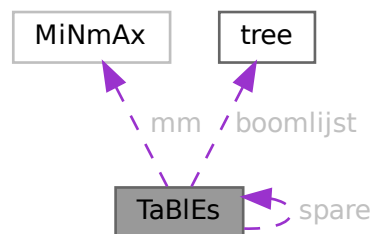
The documentation for this struct was generated from the following file:

- [structs.h](#)

3.143 TaBlEs Struct Reference

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

Collaboration diagram for TaBlEs:



Data Fields

- WORD * [tablepointers](#)
- WORD * [prototype](#)
- WORD * [pattern](#)
- [MINMAX](#) * [mm](#)
- WORD * [flags](#)
- [COMPTREE](#) * [boomlijst](#)
- UBYTE * [argtail](#)
- struct [TaBlEs](#) * [spare](#)
- WORD * [buffers](#)
- LONG [totind](#)
- LONG [reserved](#)
- LONG [defined](#)
- LONG [mdefined](#)
- int [prototypeSize](#)
- int [numind](#)
- int [bounds](#)
- int [strict](#)
- int [sparse](#)
- int [numtree](#)
- int [rootnum](#)
- int [MaxTreeSize](#)
- WORD [bufnum](#)
- WORD [bufferssize](#)
- WORD [buffersfill](#)
- WORD [tablenum](#)
- WORD [mode](#)
- WORD [numdummies](#)

3.143.1 Detailed Description

buffers, mm, flags, and prototype are always dynamically allocated, tablepointers only if needed (=0 if unallocated), boomlijst and argtail only for sparse tables.

Allocation is done for both the normal and the stub instance (spare), except for prototype and argtail which share memory.

Definition at line 342 of file [structs.h](#).

3.143.2 Field Documentation

3.143.2.1 tablepointers

```
WORD* TaBlEs::tablepointers
```

[D] Start in tablepointers table.

Definition at line 343 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#).

3.143.2.2 prototype

```
WORD* TaBlEs::prototype
```

[D] The wildcard prototyping for arguments

Definition at line 348 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#).

3.143.2.3 pattern

```
WORD* TaBlEs::pattern
```

The pattern with which to match the arguments

Definition at line 349 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#).

3.143.2.4 mm

```
MINMAX* TaBlEs::mm
```

[D] Array bounds, dimension by dimension. # elements = numind.

Definition at line 351 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#).

3.143.2.5 flags

```
WORD* Tables::flags
```

[D] Is element in use ? etc. # elements = numind.

Definition at line 352 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#).

3.143.2.6 boomlijst

```
COMPTREE* Tables::boomlijst
```

[D] Tree for searching in sparse tables

Definition at line 353 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#).

3.143.2.7 argtail

```
UBYTE* Tables::argtail
```

[D] The arguments in characters. Starts for tablebase with parenthesis to indicate tail

Definition at line 354 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#).

3.143.2.8 spare

```
struct Tables* Tables::spare
```

[D] For tablebase. Alternatingly stubs and real

Definition at line 356 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#).

3.143.2.9 buffers

```
WORD* Tables::buffers
```

[D] When we use more than one compiler buffer.

Definition at line 357 of file [structs.h](#).

Referenced by [AddRHS\(\)](#), and [DoRecovery\(\)](#).

3.143.2.10 totind

```
LONG TaBlEs::totind
```

Total number requested

Definition at line 358 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#).

3.143.2.11 reserved

```
LONG TaBlEs::reserved
```

Total reservation in tablepointers for sparse

Definition at line 359 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#).

3.143.2.12 defined

```
LONG TaBlEs::defined
```

Number of table elements that are defined

Definition at line 360 of file [structs.h](#).

3.143.2.13 mdefined

```
LONG TaBlEs::mdefined
```

Same as defined but after .global

Definition at line 361 of file [structs.h](#).

3.143.2.14 prototypeSize

```
int TaBlEs::prototypeSize
```

Size of allocated memory for prototype in bytes.

Definition at line 362 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#).

3.143.2.15 numind

```
int TaBles::numind
```

Number of array indices

Definition at line 363 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#).

3.143.2.16 bounds

```
int TaBles::bounds
```

Array bounds check on/off.

Definition at line 364 of file [structs.h](#).

3.143.2.17 strict

```
int TaBles::strict
```

>0: all must be defined. <0: undefined not substitute

Definition at line 365 of file [structs.h](#).

3.143.2.18 sparse

```
int TaBles::sparse
```

0 --> sparse table

Definition at line 366 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#).

3.143.2.19 numtree

```
int TaBles::numtree
```

For the tree for sparse tables

Definition at line 367 of file [structs.h](#).

3.143.2.20 rootnum

```
int TaBlEs::rootnum
```

For the tree for sparse tables

Definition at line 368 of file [structs.h](#).

3.143.2.21 MaxTreeSize

```
int TaBlEs::MaxTreeSize
```

For the tree for sparse tables

Definition at line 369 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#).

3.143.2.22 bufnum

```
WORD TaBlEs::bufnum
```

Each table potentially its own buffer

Definition at line 370 of file [structs.h](#).

Referenced by [AddRHS\(\)](#).

3.143.2.23 bufferssize

```
WORD TaBlEs::bufferssize
```

When we use more than one compiler buffer

Definition at line 371 of file [structs.h](#).

Referenced by [AddRHS\(\)](#), and [DoRecovery\(\)](#).

3.143.2.24 buffersfill

```
WORD TaBlEs::buffersfill
```

When we use more than one compiler buffer

Definition at line 372 of file [structs.h](#).

Referenced by [AddRHS\(\)](#).

3.143.2.25 tablenum

WORD TaBles::tablenum

For testing of tableuse

Definition at line 373 of file [structs.h](#).

3.143.2.26 mode

WORD TaBles::mode

0: normal, 1: stub

Definition at line 374 of file [structs.h](#).

3.143.2.27 numdummies

WORD TaBles::numdummies

Definition at line 375 of file [structs.h](#).

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

- [structs.h](#)

3.144 TERMINFO Struct Reference

Data Fields

- WORD * [term](#)
- void * [currentModel](#)
- void * [currentMODEL](#)
- WORD * [legcouple](#) [MAXLEGS+1]
- LONG [numtopo](#)
- LONG [numdia](#)
- WORD [diaoffset](#)
- WORD [level](#)
- WORD [externalset](#)
- WORD [internalset](#)
- WORD [numextern](#)
- WORD [flags](#)

3.144.1 Detailed Description

Definition at line 1361 of file [structs.h](#).

3.144.2 Field Documentation

3.144.2.1 term

```
WORD* TERMINFO::term
```

Definition at line 1362 of file [structs.h](#).

3.144.2.2 currentModel

```
void* TERMINFO::currentModel
```

Definition at line 1363 of file [structs.h](#).

3.144.2.3 currentMODEL

```
void* TERMINFO::currentMODEL
```

Definition at line 1364 of file [structs.h](#).

3.144.2.4 legcouple

```
WORD* TERMINFO::legcouple[MAXLEGS+1]
```

Definition at line 1365 of file [structs.h](#).

3.144.2.5 numtopo

```
LONG TERMINFO::numtopo
```

Definition at line 1366 of file [structs.h](#).

3.144.2.6 numdia

```
LONG TERMINFO::numdia
```

Definition at line 1367 of file [structs.h](#).

3.144.2.7 diaoffset

```
WORD TERMINFO::diaoffset
```

Definition at line 1368 of file [structs.h](#).

Data Fields

- WORD * [vert](#)
- WORD * [vertmax](#)
- [Options](#) * [opt](#)
- int [cldeg](#) [MAXPOINTS]
- int [clnum](#) [MAXPOINTS]
- int [clex](#) [MAXPOINTS]
- int [cmin](#) [MAXLEGS+1]
- int [cmax](#) [MAXLEGS+1]
- int [ncl](#)
- int [nvert](#)

3.145.1 Detailed Description

Definition at line [42](#) of file [diawrap.cc](#).

3.145.2 Field Documentation

3.145.2.1 [vert](#)

```
WORD* ToPoTyPe::vert
```

Definition at line [43](#) of file [diawrap.cc](#).

3.145.2.2 [vertmax](#)

```
WORD* ToPoTyPe::vertmax
```

Definition at line [44](#) of file [diawrap.cc](#).

3.145.2.3 [opt](#)

```
Options* ToPoTyPe::opt
```

Definition at line [45](#) of file [diawrap.cc](#).

3.145.2.4 [cldeg](#)

```
int ToPoTyPe::cldeg[MAXPOINTS]
```

Definition at line [46](#) of file [diawrap.cc](#).

3.145.2.5 [clnum](#)

```
int ToPoTyPe::clnum[MAXPOINTS]
```

Definition at line [46](#) of file [diawrap.cc](#).

3.145.2.6 clext

```
int ToPoTyPe::clext[MAXPOINTS]
```

Definition at line 46 of file [diawrap.cc](#).

3.145.2.7 cmind

```
int ToPoTyPe::cmind[MAXLEGS+1]
```

Definition at line 47 of file [diawrap.cc](#).

3.145.2.8 cmaxd

```
int ToPoTyPe::cmaxd[MAXLEGS+1]
```

Definition at line 47 of file [diawrap.cc](#).

3.145.2.9 ncl

```
int ToPoTyPe::ncl
```

Definition at line 48 of file [diawrap.cc](#).

3.145.2.10 nvert

```
int ToPoTyPe::nvert
```

Definition at line 48 of file [diawrap.cc](#).

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

- [diawrap.cc](#)

3.146 TrAcEn Struct Reference

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

Data Fields

- WORD * [accu](#)
- WORD * [accup](#)
- WORD * [termp](#)
- WORD * [perm](#)
- WORD * [inlist](#)
- WORD [sgn](#)
- WORD [num](#)
- WORD [level](#)
- WORD [factor](#)
- WORD [allsign](#)

3.146.1 Detailed Description

The struct TRACEN keeps track of the progress during the expansion of a 4-dimensional trace. Each time a term gets generated the expansion tree continues in the next statement. When it returns it has to know where to continue.

Definition at line 806 of file [structs.h](#).

3.146.2 Field Documentation

3.146.2.1 `accu`

```
WORD* TrAcEn::accu
```

Definition at line 807 of file [structs.h](#).

3.146.2.2 `accup`

```
WORD* TrAcEn::accup
```

Definition at line 808 of file [structs.h](#).

3.146.2.3 `termp`

```
WORD* TrAcEn::termp
```

Definition at line 809 of file [structs.h](#).

3.146.2.4 `perm`

```
WORD* TrAcEn::perm
```

Definition at line 810 of file [structs.h](#).

3.146.2.5 `inlist`

```
WORD* TrAcEn::inlist
```

Definition at line 811 of file [structs.h](#).

3.146.2.6 `sgn`

```
WORD TrAcEn::sgn
```

Definition at line 812 of file [structs.h](#).

3.146.2.7 num

WORD TrAcEn::num

Definition at line 812 of file [structs.h](#).

3.146.2.8 level

WORD TrAcEn::level

Definition at line 812 of file [structs.h](#).

3.146.2.9 factor

WORD TrAcEn::factor

Definition at line 812 of file [structs.h](#).

3.146.2.10 allsign

WORD TrAcEn::allsign

Definition at line 812 of file [structs.h](#).

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

- [structs.h](#)

3.147 TrAcEs Struct Reference

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

Data Fields

- WORD * [accu](#)
- WORD * [accup](#)
- WORD * [termp](#)
- WORD * [perm](#)
- WORD * [inlist](#)
- WORD * [nt3](#)
- WORD * [nt4](#)
- WORD * [j3](#)
- WORD * [j4](#)
- WORD * [e3](#)
- WORD * [e4](#)
- WORD * [eers](#)
- WORD * [mepf](#)
- WORD * [mdel](#)
- WORD * [pepf](#)
- WORD * [pdel](#)
- WORD [sgn](#)
- WORD [stap](#)
- WORD [step1](#)
- WORD [kstep](#)
- WORD [mdum](#)
- WORD [gamm](#)
- WORD [ad](#)
- WORD [a3](#)
- WORD [a4](#)
- WORD [lc3](#)
- WORD [lc4](#)
- WORD [sign1](#)
- WORD [sign2](#)
- WORD [gamma5](#)
- WORD [num](#)
- WORD [level](#)
- WORD [factor](#)
- WORD [allsign](#)
- WORD [finalstep](#)

3.147.1 Detailed Description

The struct TRACES keeps track of the progress during the expansion of a 4-dimensional trace. Each time a term gets generated the expansion tree continues in the next statement. When it returns it has to know where to continue. The 4-dimensional traces are more complicated than the n-dimensional traces (see TRACEN) because of the extra tricks that can be used. They are responsible for the shorter final expressions.

Definition at line 774 of file [structs.h](#).

3.147.2 Field Documentation

3.147.2.1 [accu](#)

WORD* [TrAcEs::accu](#)

Definition at line 775 of file [structs.h](#).

3.147.2.2 **accup**

WORD* TrAcEs::accup

Definition at line 776 of file [structs.h](#).

3.147.2.3 **term**

WORD* TrAcEs::term

Definition at line 777 of file [structs.h](#).

3.147.2.4 **perm**

WORD* TrAcEs::perm

Definition at line 778 of file [structs.h](#).

3.147.2.5 **inlist**

WORD* TrAcEs::inlist

Definition at line 779 of file [structs.h](#).

3.147.2.6 **nt3**

WORD* TrAcEs::nt3

Definition at line 780 of file [structs.h](#).

3.147.2.7 **nt4**

WORD* TrAcEs::nt4

Definition at line 781 of file [structs.h](#).

3.147.2.8 **j3**

WORD* TrAcEs::j3

Definition at line 782 of file [structs.h](#).

3.147.2.9 **j4**

WORD* TrAcEs::j4

Definition at line 783 of file [structs.h](#).

3.147.2.10 e3

WORD* TrAcEs::e3

Definition at line 784 of file [structs.h](#).

3.147.2.11 e4

WORD* TrAcEs::e4

Definition at line 785 of file [structs.h](#).

3.147.2.12 eers

WORD* TrAcEs::eers

Definition at line 786 of file [structs.h](#).

3.147.2.13 mepf

WORD* TrAcEs::mepf

Definition at line 787 of file [structs.h](#).

3.147.2.14 mdel

WORD* TrAcEs::mdel

Definition at line 788 of file [structs.h](#).

3.147.2.15 pepf

WORD* TrAcEs::pepf

Definition at line 789 of file [structs.h](#).

3.147.2.16 pdel

WORD* TrAcEs::pdel

Definition at line 790 of file [structs.h](#).

3.147.2.17 sgn

WORD TrAcEs::sgn

Definition at line 791 of file [structs.h](#).

3.147.2.18 stap

WORD TrAcEs::stap

Definition at line 792 of file [structs.h](#).

3.147.2.19 step1

WORD TrAcEs::step1

Definition at line 793 of file [structs.h](#).

3.147.2.20 kstep

WORD TrAcEs::kstep

Definition at line 793 of file [structs.h](#).

3.147.2.21 mdum

WORD TrAcEs::mdum

Definition at line 793 of file [structs.h](#).

3.147.2.22 gamm

WORD TrAcEs::gamm

Definition at line 794 of file [structs.h](#).

3.147.2.23 ad

WORD TrAcEs::ad

Definition at line 794 of file [structs.h](#).

3.147.2.24 a3

WORD TrAcEs::a3

Definition at line 794 of file [structs.h](#).

3.147.2.25 a4

WORD TrAcEs::a4

Definition at line 794 of file [structs.h](#).

3.147.2.26 lc3

WORD TrAcEs::lc3

Definition at line 794 of file [structs.h](#).

3.147.2.27 lc4

WORD TrAcEs::lc4

Definition at line 794 of file [structs.h](#).

3.147.2.28 sign1

WORD TrAcEs::sign1

Definition at line 795 of file [structs.h](#).

3.147.2.29 sign2

WORD TrAcEs::sign2

Definition at line 795 of file [structs.h](#).

3.147.2.30 gamma5

WORD TrAcEs::gamma5

Definition at line 795 of file [structs.h](#).

3.147.2.31 num

WORD TrAcEs::num

Definition at line 795 of file [structs.h](#).

3.147.2.32 level

WORD TrAcEs::level

Definition at line 795 of file [structs.h](#).

3.147.2.33 factor

WORD TrAcEs::factor

Definition at line 795 of file [structs.h](#).

3.147.2.34 allsign

WORD TrAcEs::allsign

Definition at line 795 of file [structs.h](#).

3.147.2.35 finalstep

WORD TrAcEs::finalstep

Definition at line 796 of file [structs.h](#).

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

- [structs.h](#)

3.148 tree Struct Reference

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

Data Fields

- int [parent](#)
- int [left](#)
- int [right](#)
- int [value](#)
- int [blnce](#)
- int [usage](#)

3.148.1 Detailed Description

The subexpressions in the compiler are kept track of in a (balanced) tree to reduce the need for subexpressions and hence save much space in large rhs expressions (like when we have xxxxxx occurrences of objects like $f(x+1, x+1)$ in which each $x+1$ becomes a subexpression. The struct that controls this tree is COMPTREE.

Definition at line 288 of file [structs.h](#).

3.148.2 Field Documentation

3.148.2.1 parent

```
int tree::parent
```

Index of parent

Definition at line 289 of file [structs.h](#).

Referenced by [IniFbuffer\(\)](#).

3.148.2.2 left

```
int tree::left
```

Left child (if not -1)

Definition at line 290 of file [structs.h](#).

Referenced by [IniFbuffer\(\)](#).

3.148.2.3 right

```
int tree::right
```

Right child (if not -1)

Definition at line 291 of file [structs.h](#).

Referenced by [IniFbuffer\(\)](#).

3.148.2.4 value

```
int tree::value
```

The object to be sorted and searched

Definition at line 292 of file [structs.h](#).

Referenced by [IniFbuffer\(\)](#).

3.148.2.5 blnce

```
int tree::blnce
```

Balance factor

Definition at line 293 of file [structs.h](#).

Referenced by [IniFbuffer\(\)](#).

3.148.2.6 usage

```
int tree::usage
```

Number of uses in some types of trees

Definition at line 294 of file [structs.h](#).

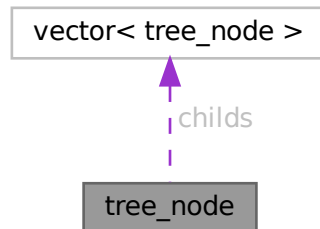
Referenced by [CleanupArgCache\(\)](#), and [IniFbuffer\(\)](#).

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

- [structs.h](#)

3.149 tree_node Class Reference

Collaboration diagram for tree_node:



Public Member Functions

- [tree_node](#) (int _var=0)
- [tree_node](#) (const [tree_node](#) &other)
- [tree_node](#) & [operator=](#) (const [tree_node](#) &other)

Data Fields

- vector< [tree_node](#) > [childs](#)
- double [sum_results](#)
- int [num_visits](#)
- WORD [var](#)
- bool [finished](#)

3.149.1 Detailed Description

Definition at line 125 of file [optimize.cc](#).

3.149.2 Constructor & Destructor Documentation

3.149.2.1 tree_node() [1/2]

```
tree_node::tree_node (
    int _var = 0 ) [inline]
```

Definition at line 133 of file [optimize.cc](#).

3.149.2.2 tree_node() [2/2]

```
tree_node::tree_node (
    const tree_node & other ) [inline]
```

Definition at line 136 of file [optimize.cc](#).

3.149.3 Member Function Documentation

3.149.3.1 operator=()

```
tree_node & tree_node::operator= (
    const tree_node & other ) [inline]
```

Definition at line 143 of file [optimize.cc](#).

3.149.4 Field Documentation

3.149.4.1 childs

```
vector<tree_node> tree_node::childs
```

Definition at line 127 of file [optimize.cc](#).

3.149.4.2 sum_results

```
double tree_node::sum_results
```

Definition at line 128 of file [optimize.cc](#).

3.149.4.3 num_visits

```
int tree_node::num_visits
```

Definition at line 129 of file [optimize.cc](#).

3.149.4.4 var

```
WORD tree_node::var
```

Definition at line 130 of file [optimize.cc](#).

3.149.4.5 finished

```
bool tree_node::finished
```

Definition at line 131 of file [optimize.cc](#).

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

- [optimize.cc](#)

3.150 VaRrEnUm Struct Reference

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

Data Fields

- WORD * [start](#)
- WORD * [lo](#)
- WORD * [hi](#)

3.150.1 Detailed Description

Contains the pointers to an array in which a binary search will be performed.

Definition at line 164 of file [structs.h](#).

3.150.2 Field Documentation

3.150.2.1 start

```
WORD* VaRrEnUm::start
```

Start point for search. Points inbetween lo and hi

Definition at line 165 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#).

3.150.2.2 lo

```
WORD* VaRrEnUm::lo
```

Start of memory area

Definition at line 166 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#), [Generator\(\)](#), and [GetFirstTerm\(\)](#).

3.150.2.3 hi

```
WORD* VaRrEnUm::hi
```

End of memory area

Definition at line 167 of file [structs.h](#).

Referenced by [DoRecovery\(\)](#).

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

- [structs.h](#)

3.151 VeCtOr Struct Reference

Data Fields

- LONG [name](#)
- WORD [complex](#)
- WORD [number](#)
- WORD [flags](#)
- WORD [node](#)
- WORD [namesize](#)
- WORD [dimension](#)

3.151.1 Detailed Description

Definition at line 462 of file [structs.h](#).

3.151.2 Field Documentation

3.151.2.1 name

```
LONG VeCtOr::name
```

Definition at line 463 of file [structs.h](#).

3.151.2.2 complex

```
WORD VeCtOr::complex
```

Definition at line 464 of file [structs.h](#).

3.151.2.3 number

WORD VeCtOr::number

Definition at line 465 of file [structs.h](#).

3.151.2.4 flags

WORD VeCtOr::flags

Definition at line 466 of file [structs.h](#).

3.151.2.5 node

WORD VeCtOr::node

Definition at line 467 of file [structs.h](#).

3.151.2.6 namesize

WORD VeCtOr::namesize

Definition at line 468 of file [structs.h](#).

3.151.2.7 dimension

WORD VeCtOr::dimension

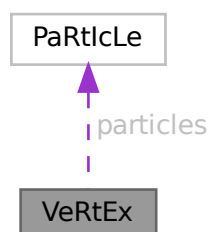
Definition at line 469 of file [structs.h](#).

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

- [structs.h](#)

3.152 VeRtEx Struct Reference

Collaboration diagram for VeRtEx:



Data Fields

- [PARTICLE particles](#) [MAXPARTICLES]
- WORD [couplings](#) [2 *MAXCOUPLINGS]
- WORD [nparticles](#)
- WORD [ncouplings](#)
- WORD [type](#)
- WORD [error](#)
- WORD [externonly](#)
- WORD [spare](#)

3.152.1 Detailed Description

Definition at line [626](#) of file [structs.h](#).

3.152.2 Field Documentation

3.152.2.1 particles

[PARTICLE](#) VeRtEx::particles [MAXPARTICLES]

Definition at line [627](#) of file [structs.h](#).

3.152.2.2 couplings

WORD VeRtEx::couplings [2 *MAXCOUPLINGS]

Definition at line [628](#) of file [structs.h](#).

3.152.2.3 nparticles

WORD VeRtEx::nparticles

Definition at line [629](#) of file [structs.h](#).

3.152.2.4 ncouplings

WORD VeRtEx::ncouplings

Definition at line [630](#) of file [structs.h](#).

3.152.2.5 type

WORD VeRtEx::type

Definition at line [631](#) of file [structs.h](#).

3.152.2.6 error

```
WORD VeRtEx::error
```

Definition at line 632 of file [structs.h](#).

3.152.2.7 externonly

```
WORD VeRtEx::externonly
```

Definition at line 633 of file [structs.h](#).

3.152.2.8 spare

```
WORD VeRtEx::spare
```

Definition at line 634 of file [structs.h](#).

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

- [structs.h](#)

3.153 X_const Struct Reference

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

Data Fields

- UBYTE * [currentPrompt](#)
- UBYTE * [shellname](#)
- UBYTE * [stderrname](#)
- int [timeout](#)
- int [killSignal](#)
- int [killWholeGroup](#)
- int [daemonize](#)
- int [currentExternalChannel](#)

3.153.1 Detailed Description

The [X_const](#) struct is part of the global data and resides in the ALLGLOBALS struct A under the name X We see it used with the macro AX as in AX.timeout It contains variables that involve communication with external programs

Definition at line 2410 of file [structs.h](#).

3.153.2 Field Documentation

3.153.2.1 currentPrompt

```
UBYTE* X_const::currentPrompt
```

Definition at line 2411 of file [structs.h](#).

3.153.2.2 shellname

```
UBYTE* X_const::shellname
```

Definition at line 2412 of file [structs.h](#).

3.153.2.3 stderrname

```
UBYTE* X_const::stderrname
```

Definition at line 2414 of file [structs.h](#).

3.153.2.4 timeout

```
int X_const::timeout
```

Definition at line 2416 of file [structs.h](#).

3.153.2.5 killSignal

```
int X_const::killSignal
```

Definition at line 2419 of file [structs.h](#).

3.153.2.6 killWholeGroup

```
int X_const::killWholeGroup
```

Definition at line 2420 of file [structs.h](#).

3.153.2.7 daemonize

```
int X_const::daemonize
```

Definition at line 2422 of file [structs.h](#).

3.153.2.8 currentExternalChannel

```
int X_const::currentExternalChannel
```

Definition at line 2423 of file [structs.h](#).

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

- [structs.h](#)

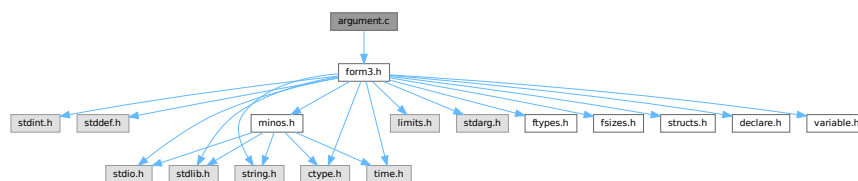
Chapter 4

File Documentation

4.1 argument.c File Reference

```
#include "form3.h"
```

Include dependency graph for argument.c:



Macros

- `#define` [NEWORDER](#)

Functions

- int [execarg](#) (PHEAD WORD *term, WORD level)
- int [execterm](#) (PHEAD WORD *term, WORD level)
- int [ArgumentImplode](#) (PHEAD WORD *term, WORD *thelist)
- int [ArgumentExplode](#) (PHEAD WORD *term, WORD *thelist)
- int [ArgFactorize](#) (PHEAD WORD *argin, WORD *argout)
- WORD [FindArg](#) (PHEAD WORD *a)
- int [InsertArg](#) (PHEAD WORD *argin, WORD *argout, int par)
- int [CleanupArgCache](#) (PHEAD WORD bufnum)
- int [ArgSymbolMerge](#) (WORD *t1, WORD *t2)
- int [ArgDotproductMerge](#) (WORD *t1, WORD *t2)
- WORD * [TakeArgContent](#) (PHEAD WORD *argin, WORD *argout)
- WORD * [MakeInteger](#) (PHEAD WORD *argin, WORD *argout, WORD *argfree)
- WORD * [MakeMod](#) (PHEAD WORD *argin, WORD *argout, WORD *argfree)
- void [SortWeights](#) (LONG *weights, LONG *extraspace, WORD number)

4.1.1 Detailed Description

Contains the routines that deal with the execution phase of the argument and related statements (like term)

Definition in file [argument.c](#).

4.1.2 Macro Definition Documentation

4.1.2.1 NEWORDER

```
#define NEWORDER
```

Factorizes an argument in general notation (meaning that the first word of the argument is a positive size indicator)
Input (argin): pointer to the complete argument [Output](#) (argout): Pointer to where the output should be written. This is in the WorkSpace Return value should be negative if anything goes wrong.

The notation of the output should be a string of arguments terminated by the number zero.

Originally we sorted in a way that the constants came last. This gave conflicts with the dollar and expression factorizations (in the expressions we wanted the zero first and then followed by the constants).

Definition at line [2064](#) of file [argument.c](#).

4.1.3 Function Documentation

4.1.3.1 `execarg()`

```
int execarg (
    PHEAD WORD * term,
    WORD level )
```

Definition at line [56](#) of file [argument.c](#).

4.1.3.2 `execterm()`

```
int execterm (
    PHEAD WORD * term,
    WORD level )
```

Definition at line [1775](#) of file [argument.c](#).

4.1.3.3 `ArgumentImplode()`

```
int ArgumentImplode (
    PHEAD WORD * term,
    WORD * thelist )
```

Definition at line [1851](#) of file [argument.c](#).

4.1.3.4 ArgumentExplode()

```
int ArgumentExplode (
    PHEAD WORD * term,
    WORD * thelist )
```

Definition at line 1955 of file [argument.c](#).

4.1.3.5 ArgFactorize()

```
int ArgFactorize (
    PHEAD WORD * argin,
    WORD * argout )
```

Definition at line 2066 of file [argument.c](#).

4.1.3.6 FindArg()

```
WORD FindArg (
    PHEAD WORD * a )
```

Looks the argument up in the (workers) table. If it is found the number in the table is returned (plus one to make it positive). If it is not found we look in the compiler provided table. If it is found - the number in the table is returned (minus one to make it negative). If in neither table we return zero.

Definition at line 2519 of file [argument.c](#).

4.1.3.7 InsertArg()

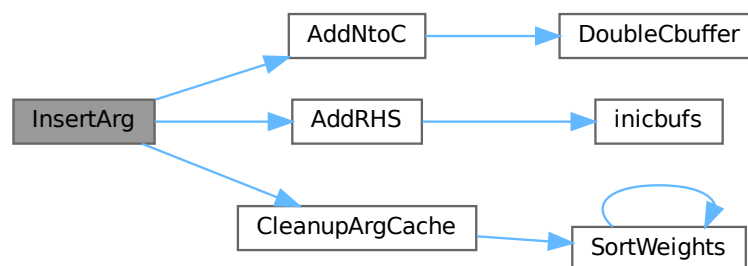
```
int InsertArg (
    PHEAD WORD * argin,
    WORD * argout,
    int par )
```

Inserts the argument into the (workers) table. If the table is too full we eliminate half of it. The eliminated elements are the ones that have not been used most recently, weighted by their total use and age(?). If par == 0 it inserts in the regular factorization cache If par == 1 it inserts in the cache defined with the FactorCache statement

Definition at line 2543 of file [argument.c](#).

References [AddNtoC\(\)](#), [AddRHS\(\)](#), and [CleanupArgCache\(\)](#).

Here is the call graph for this function:



4.1.3.8 CleanupArgCache()

```
int CleanupArgCache (
    PHEAD WORD bufnum )
```

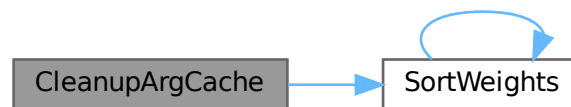
Cleans up the argument factorization cache. We throw half the elements. For a weight of what we want to keep we use the product of usage and the number in the buffer.

Definition at line 2578 of file [argument.c](#).

References [CbUf::boomlijst](#), [CbUf::Buffer](#), [CbUf::Pointer](#), [CbUf::rhs](#), [SortWeights\(\)](#), and [tree::usage](#).

Referenced by [InsertArg\(\)](#).

Here is the call graph for this function:



4.1.3.9 ArgSymbolMerge()

```
int ArgSymbolMerge (
    WORD * t1,
    WORD * t2 )
```

Definition at line 2649 of file [argument.c](#).

4.1.3.10 ArgDotproductMerge()

```
int ArgDotproductMerge (
    WORD * t1,
    WORD * t2 )
```

Definition at line 2704 of file [argument.c](#).

4.1.3.11 TakeArgContent()

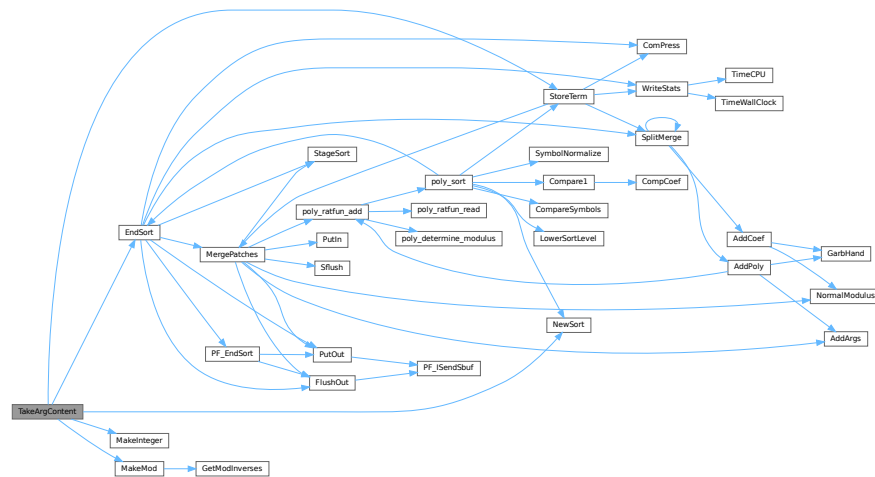
```
WORD * TakeArgContent (
    PHEAD WORD * argin,
    WORD * argout )
```

Implements part of the old ExecArg in which we take common factors from arguments with more than one term. The common pieces are put in argout as a sequence of arguments. The part with the multiple terms that are now relative prime is put in argfree which is allocated via TermMalloc and is given as the return value. The difference with the old code is that negative powers are always removed. Hence it is as in MakeInteger in which only numerators will be left: now only zero or positive powers will be remaining.

Definition at line 2772 of file [argument.c](#).

References [EndSort\(\)](#), [MakeInteger\(\)](#), [MakeMod\(\)](#), [NewSort\(\)](#), and [StoreTerm\(\)](#).

Here is the call graph for this function:



4.1.3.12 MakeInteger()

```
WORD * MakeInteger (
    PHEAD WORD * argin,
    WORD * argout,
    WORD * argfree )
```

For normalizing everything to integers we have to determine for all elements of this argument the LCM of the denominators and the GCD of the numerators. The input argument is in argin. The number that comes out should go to argout. The new pointer in the argout buffer is the return value. The normalized argument is in argfree.

Definition at line 3318 of file [argument.c](#).

Referenced by [TakeArgContent\(\)](#).

4.1.3.13 MakeMod()

```
WORD * MakeMod (
    PHEAD WORD * argin,
    WORD * argout,
    WORD * argfree )
```

Similar to MakeInteger but now with modulus arithmetic using only a one WORD 'prime'. We make the coefficient of the first term in the argument equal to one. Already the coefficients are taken modulus AN.cmod and AN.ncmod == 1

Definition at line 3489 of file [argument.c](#).

References [GetModInverses\(\)](#).

Referenced by [TakeArgContent\(\)](#).

Here is the call graph for this function:



4.1.3.14 SortWeights()

```
void SortWeights (
    LONG * weights,
    LONG * extraspace,
    WORD number )
```

Sorts an array of LONGS in the same way SplitMerge (in [sort.c](#)) works We use gradual division in two.

Definition at line 3534 of file [argument.c](#).

References [SortWeights\(\)](#).

Referenced by [CleanupArgCache\(\)](#), and [SortWeights\(\)](#).

Here is the call graph for this function:



4.2 argument.c

[Go to the documentation of this file.](#)

```

00001
00007 /* #[ License : */
00008 /*
00009  * Copyright (C) 1984-2026 J.A.M. Vermaseren
00010  * When using this file you are requested to refer to the publication
00011  * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00012  * This is considered a matter of courtesy as the development was paid
00013  * for by FOM the Dutch physics granting agency and we would like to
00014  * be able to track its scientific use to convince FOM of its value
00015  * for the community.
00016  *
00017  * This file is part of FORM.
00018  *
00019  * FORM is free software: you can redistribute it and/or modify it under the
00020  * terms of the GNU General Public License as published by the Free Software
00021  * Foundation, either version 3 of the License, or (at your option) any later
00022  * version.
00023  *
00024  * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00025  * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00026  * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00027  * details.
00028  *
00029  * You should have received a copy of the GNU General Public License along
00030  * with FORM. If not, see <http://www.gnu.org/licenses/>.
00031  */
00032 /* #[ License : */
00033
00034 /*
00035  *#[ include : argument.c
00036  */
00037
00038 #include "form3.h"
00039
00040 /*
00041  *#[ include :
00042  *#[ execarg :
00043
00044  Executes the subset of statements in an argument environment.
00045  The calling routine should be of the type
00046  if ( C->lhs[level][0] == TYPEARG ) {
00047      if ( execarg(term,level) ) goto GenCall;
00048      level = C->lhs[level][2];
00049      goto SkipCount;
00050  }
00051  Note that there will be cases in which extra space is needed.
00052  In addition the compare with C->numlhs isn't very fine, because we
00053  need to insert a different value (C->lhs[level][2]).
00054  */
00055
00056 int execarg(PHEAD WORD *term, WORD level)
00057 {
00058     GETBIDENTITY
00059     WORD *t, *r, *m, *v;
00060     WORD *start, *stop, *rstop, *r1, *r2 = 0, *r3 = 0, *r4, *r5, *r6, *r7, *r8, *r9;
00061     WORD *mm, *mstop, *rnext, *rrr, *factor, type, ngcd, nq;
00062     CBUF *C = cbuf+AM.rbufnum, *CC = cbuf+AT.ebufnum;
00063     WORD i, j, k, oldnumlhs = AR.Cnumlhs, count, olddefer = AR.DeferFlag;
00064     int action = 0;
00065     WORD oldnumrhs = CC->numrhs, size, pow, jj;
00066     LONG oldcpointer = CC->Pointer - CC->Buffer, oldppointer = AT.pWorkPointer, lp;
00067     WORD *oldwork = AT.WorkPointer, *oldwork2, scale, renorm;
00068     WORD kLCM = 0, kGCD = 0, kGCD2, kkLCM = 0, jLCM = 0, jGCD, sign = 1;
00069     int ii, didpolyratfun;
00070     UWORD *EAscrat, *GCDBuffer = 0, *GCDBuffer2 = 0, *LCMBuffer = 0, *LCMb = 0, *LCMc = 0;
00071     AT.WorkPointer += *term;
00072     start = C->lhs[level];
00073     AR.Cnumlhs = start[2];
00074     stop = start + start[1];
00075     type = *start;
00076     scale = start[4];
00077     renorm = start[5];
00078     start += TYPEARGHEADSIZE;
00079     /*
00080     *#[ Dollars :
00081     */
00082     if ( renorm && start[1] != 0 ) { /* We have to evaluate $ symbols inside () */
00083         t = start+1; factor = oldwork2 = v = AT.WorkPointer;
00084         i = *t; t++;
00085         *v++ = i+3; i--; NCOPY(v,t,i);
00086         *v++ = 1; *v++ = 1; *v++ = 3;
00087         AT.WorkPointer = v;

```

```

00088     start = t; AR.Eside = LHSIDEX;
00089     NewSort(BHEAD0);
00090     if ( Generator(BHEAD factor,AR.Cnumlhs) ) {
00091         LowerSortLevel();
00092         AT.WorkPointer = oldwork;
00093         return(-1);
00094     }
00095     AT.WorkPointer = v;
00096     if ( EndSort(BHEAD factor,0) < 0 ) {}
00097     if ( *factor && *(factor+*factor) != 0 ) {
00098         MLOCK(ErrorMessageLock);
00099         MesPrint("%$ in () does not evaluate into a single term");
00100         MUNLOCK(ErrorMessageLock);
00101         return(-1);
00102     }
00103     AR.Eside = RHSIDE;
00104     if ( *factor > 0 ) {
00105         v = factor+*factor;
00106         v -= ABS(v[-1]);
00107         *factor = v-factor;
00108     }
00109     AT.WorkPointer = v;
00110 }
00111 else {
00112     if ( *start < 0 ) {
00113         factor = start + 1;
00114         start += -*start;
00115     }
00116     else factor = 0;
00117 }
00118 /*
00119 #] Dollars :
00120 */
00121 t = term;
00122 r = t + *t;
00123 rstop = r - ABS(r[-1]);
00124 t++;
00125 /*
00126 #[ Argument detection : + argument statement
00127 */
00128 /*
00129 Allocate Numbers for MakeInteger here, for re-use in case multiple
00130 functions are treated in the same term.
00131 */
00132 if ( type == TYPENORM4 ) {
00133     GCDBuffer = NumberMalloc("execarg");
00134     GCDBuffer2 = NumberMalloc("execarg");
00135     LCMbuffer = NumberMalloc("execarg");
00136     LCMb = NumberMalloc("execarg"); LCMc = NumberMalloc("execarg");
00137 }
00138 didpolyratfun = 0;
00139 while ( t < rstop ) {
00140     if ( *t >= FUNCTION && functions[*t-FUNCTION].spec <= 0 ) {
00141 /*
00142         We have a function. First count the number of arguments.
00143         Tensors are excluded.
00144 */
00145         count = 0;
00146         v = t;
00147         m = t + FUNHEAD;
00148         r = t + t[1];
00149         while ( m < r ) {
00150             count++;
00151             NEXTARG(m)
00152         }
00153         if ( count <= 0 ) { t += t[1]; continue; }
00154 /*
00155         Now we take the arguments one by one and test for a match
00156 */
00157         for ( i = 1; i <= count; i++ ) {
00158             m = start;
00159             while ( m < stop ) {
00160                 r = m + m[1];
00161                 j = *r++;
00162                 if ( j > 1 ) {
00163                     while ( --j > 0 ) {
00164                         if ( *r == i ) goto RightNum;
00165                         r++;
00166                     }
00167                     m = r;
00168                     continue;
00169                 }
00170 RightNum:
00171                 if ( m[1] == 2 ) {
00172 #ifdef WITHFLOAT
00173                     if ( *t != FLOATFUN || TestFloat(t) == 0 )
00174 #endif

```

```

00175         {
00176             m += 2;
00177             m += *m;
00178             goto HaveToDo;
00179         }
00180 #ifdef WITHFLOAT
00181         else {
00182             m += 2;
00183         }
00184 #endif
00185     }
00186     else {
00187         r = m + m[1];
00188         m += 2;
00189         while ( m < r ) {
00190             if ( *m == CSET ) {
00191                 r1 = SetElements + Sets[m[1]].first;
00192                 r2 = SetElements + Sets[m[1]].last;
00193                 while ( r1 < r2 ) {
00194                     if ( *r1++ == *t ) goto HaveToDo;
00195                 }
00196             }
00197             else if ( m[1] == *t ) goto HaveToDo;
00198             m += 2;
00199         }
00200     }
00201     m += *m;
00202 }
00203 continue;
00204 HaveToDo:
00205 /*
00206     If we come here we have to do the argument i (first is 1).
00207 */
00208     sign = 1;
00209     action = 1;
00210     if ( *t == AR.PolyFun ) didpolyratfun = 1;
00211     v[2] |= DIRTYFLAG;
00212     r = t + FUNHEAD;
00213     j = i;
00214     while ( --j > 0 ) { NEXTARG(r) }
00215     if ( ( type == TYPESPLITARG ) || ( type == TYPESPLITFIRSTARG )
00216         || ( type == TYPESPLITLASTARG ) ) {
00217         if ( *t > FUNCTION && *r > 0 ) {
00218             WantAddPointers(2);
00219             AT.pWorkSpace[AT.pWorkPointer++] = t;
00220             AT.pWorkSpace[AT.pWorkPointer++] = r;
00221         }
00222         continue;
00223     }
00224     else if ( type == TYPESPLITARG2 ) {
00225         if ( *t > FUNCTION && *r > 0 ) {
00226             WantAddPointers(2);
00227             AT.pWorkSpace[AT.pWorkPointer++] = t;
00228             AT.pWorkSpace[AT.pWorkPointer++] = r;
00229         }
00230         continue;
00231     }
00232     else if ( type == TYPEFACTARG || type == TYPEFACTARG2 ) {
00233         if ( *t > FUNCTION || *t == DENOMINATOR ) {
00234             if ( *r > 0 ) {
00235                 mm = r + ARGHEAD; mstop = r + *r;
00236                 if ( mm + *mm < mstop ) {
00237                     WantAddPointers(2);
00238                     AT.pWorkSpace[AT.pWorkPointer++] = t;
00239                     AT.pWorkSpace[AT.pWorkPointer++] = r;
00240                     continue;
00241                 }
00242                 if ( *mm == 1+ABS(mstop[-1]) ) continue;
00243                 if ( mstop[-3] != 1 || mstop[-2] != 1
00244                     || mstop[-1] != 3 ) {
00245                     WantAddPointers(2);
00246                     AT.pWorkSpace[AT.pWorkPointer++] = t;
00247                     AT.pWorkSpace[AT.pWorkPointer++] = r;
00248                     continue;
00249                 }
00250             }
00251             GETSTOP(mm,mstop); mm++;
00252             if ( mm + mm[1] < mstop ) {
00253                 WantAddPointers(2);
00254                 AT.pWorkSpace[AT.pWorkPointer++] = t;
00255                 AT.pWorkSpace[AT.pWorkPointer++] = r;
00256                 continue;
00257             }
00258             if ( *mm == SYMBOL && ( mm[1] > 4 ||
00259                 ( mm[3] != 1 && mm[3] != -1 ) ) ) {
00260                 WantAddPointers(2);
00261                 AT.pWorkSpace[AT.pWorkPointer++] = t;
00262                 AT.pWorkSpace[AT.pWorkPointer++] = r;

```

```

00262         continue;
00263     }
00264     else if ( *mm == DOTPRODUCT && ( mm[1] > 5 ||
00265         ( mm[4] != 1 && mm[4] != -1 ) ) ) {
00266         WantAddPointers(2);
00267         AT.pWorkSpace[AT.pWorkPointer++] = t;
00268         AT.pWorkSpace[AT.pWorkPointer++] = r;
00269         continue;
00270     }
00271     else if ( ( *mm == DELTA || *mm == VECTOR )
00272         && mm[1] > 4 ) {
00273         WantAddPointers(2);
00274         AT.pWorkSpace[AT.pWorkPointer++] = t;
00275         AT.pWorkSpace[AT.pWorkPointer++] = r;
00276         continue;
00277     }
00278     }
00279     else if ( factor && *factor == 4 && factor[2] == 1 ) {
00280         WantAddPointers(2);
00281         AT.pWorkSpace[AT.pWorkPointer++] = t;
00282         AT.pWorkSpace[AT.pWorkPointer++] = r;
00283         continue;
00284     }
00285     else if ( factor && *factor == 0
00286         && ( *r == -SNUMBER && r[1] != 1 ) ) {
00287         WantAddPointers(2);
00288         AT.pWorkSpace[AT.pWorkPointer++] = t;
00289         AT.pWorkSpace[AT.pWorkPointer++] = r;
00290         continue;
00291     }
00292     else if ( *r == -MINVECTOR ) {
00293         WantAddPointers(2);
00294         AT.pWorkSpace[AT.pWorkPointer++] = t;
00295         AT.pWorkSpace[AT.pWorkPointer++] = r;
00296         continue;
00297     }
00298     }
00299     continue;
00300 }
00301 else if ( type == TYPENORM || type == TYPENORM2 || type == TYPENORM3 || type ==
TYPENORM4 ) {
00302     if ( *r < 0 ) {
00303         WORD rone;
00304         if ( *r == -MINVECTOR ) { rone = -1; *r = -INDEX; }
00305         else if ( *r != -SNUMBER || r[1] == 1 || r[1] == 0 ) continue;
00306         else { rone = r[1]; r[1] = 1; }
00307     /*
00308     Now we must multiply the general coefficient by r[1]
00309     */
00310     if ( scale && ( factor == 0 || *factor ) ) {
00311         action = 1;
00312         v[2] |= DIRTYFLAG;
00313         if ( rone < 0 ) {
00314             if ( type == TYPENORM3 ) k = 1;
00315             else k = -1;
00316             rone = -rone;
00317         }
00318         else k = 1;
00319         r1 = term + *term;
00320         size = r1[-1];
00321         size = REDLENG(size);
00322         if ( scale > 0 ) {
00323             for ( jj = 0; jj < scale; jj++ ) {
00324                 if ( Mully(BHEAD (UWORD *)rstop,&size,(UWORD *)(&rone),k) )
00325                     goto execargerr;
00326             }
00327         }
00328         else {
00329             for ( jj = 0; jj > scale; jj-- ) {
00330                 if ( Divvy(BHEAD (UWORD *)rstop,&size,(UWORD *)(&rone),k) )
00331                     goto execargerr;
00332             }
00333         }
00334         size = INCLENG(size);
00335         k = size < 0 ? -size: size;
00336         rstop[k-1] = size;
00337         *term = (WORD)(rstop - term) + k;
00338     }
00339     continue;
00340 }
00341 /*
00342 Now we have to find a reference term.
00343 If factor is defined and *factor != 0 we have to
00344 look for the first term that matches the pattern exactly
00345 Otherwise the first term plays this role
00346 If its coefficient is not one,
00347 we must set up a division of the whole argument by

```

```

00348      this coefficient, and a multiplication of the term
00349      when the type is not equal to TYPENORM2.
00350      We first multiply the coefficient of the term.
00351      Then we set up the division.
00352
00353      First find the magic term
00354  */
00355      if ( type == TYPENORM4 ) {
00356  /*
00357          For normalizing everything to integers we have to
00358          determine for all elements of this argument the LCM of
00359          the denominators and the GCD of the numerators.
00360          The buffers have been allocated already.
00361  */
00362          r4 = r + *r;
00363          r1 = r + ARGHEAD;
00364  /*
00365          First take the first term to load up the LCM and the GCD
00366  */
00367          r2 = r1 + *r1;
00368          j = r2[-1];
00369          if ( j < 0 ) sign = -1;
00370          r3 = r2 - ABS(j);
00371          k = REDLENG(j);
00372          if ( k < 0 ) k = -k;
00373          while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00374          for ( kGCD = 0; kGCD < k; kGCD++ ) GCDbuffer[kGCD] = r3[kGCD];
00375          k = REDLENG(j);
00376          if ( k < 0 ) k = -k;
00377          r3 += k;
00378          while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00379          for ( kLCM = 0; kLCM < k; kLCM++ ) LCMbuffer[kLCM] = r3[kLCM];
00380          r1 = r2;
00381  /*
00382          Now go through the rest of the terms in this argument.
00383  */
00384          while ( r1 < r4 ) {
00385              r2 = r1 + *r1;
00386              j = r2[-1];
00387              r3 = r2 - ABS(j);
00388              k = REDLENG(j);
00389              if ( k < 0 ) k = -k;
00390              while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00391              if ( ( ( GCDbuffer[0] == 1 ) && ( kGCD == 1 ) ) ) {
00392  /*
00393                  GCD is already 1
00394  */
00395              }
00396              else if ( ( ( k != 1 ) || ( r3[0] != 1 ) ) ) {
00397                  if ( GcdLong(BHEAD GCDbuffer,kGCD, (UWORD *)r3,k,GCDbuffer2,&kGCD2) ) {
00398                      NumberFree(GCDbuffer,"execarg");
00399                      NumberFree(GCDbuffer2,"execarg");
00400                      NumberFree(LCMbuffer,"execarg");
00401                      NumberFree(LCMB,"execarg"); NumberFree(LCMc,"execarg");
00402                      goto execargerr;
00403                  }
00404                  kGCD = kGCD2;
00405                  for ( ii = 0; ii < kGCD; ii++ ) GCDbuffer[ii] = GCDbuffer2[ii];
00406              }
00407              else {
00408                  kGCD = 1; GCDbuffer[0] = 1;
00409              }
00410              k = REDLENG(j);
00411              if ( k < 0 ) k = -k;
00412              r3 += k;
00413              while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00414              if ( ( ( LCMbuffer[0] == 1 ) && ( kLCM == 1 ) ) ) {
00415                  for ( kLCM = 0; kLCM < k; kLCM++ )
00416                      LCMbuffer[kLCM] = r3[kLCM];
00417              }
00418              else if ( ( k != 1 ) || ( r3[0] != 1 ) ) {
00419                  if ( GcdLong(BHEAD LCMbuffer,kLCM, (UWORD *)r3,k,LCMb,&kLCM) ) {
00420                      NumberFree(GCDbuffer,"execarg"); NumberFree(GCDbuffer2,"execarg");
00421                      NumberFree(LCMbuffer,"execarg"); NumberFree(LCMB,"execarg");
00422                      NumberFree(LCMc,"execarg");
00423                      goto execargerr;
00424                  }
00425                  DivLong((UWORD *)r3,k,LCMb,kLCM,LCMb,&kLCM,LCMc,&jLCM);
00426                  MulLong(LCMbuffer,kLCM,LCMb,kLCM,LCMc,&jLCM);
00427                  for ( kLCM = 0; kLCM < jLCM; kLCM++ )
00428                      LCMbuffer[kLCM] = LCMc[kLCM];
00429              }
00430              else {} /* LCM doesn't change */
00431              r1 = r2;
00432  /*
00433          Now put the factor together: GCD/LCM

```

```

00434 */
00435
00436     r3 = (WORD *) (GCDbuffer);
00437     if ( kGCD == kLCM ) {
00438         for ( jGCD = 0; jGCD < kGCD; jGCD++ )
00439             r3[jGCD+kGCD] = LCMbuffer[jGCD];
00440         k = kGCD;
00441     }
00442     else if ( kGCD > kLCM ) {
00443         for ( jGCD = 0; jGCD < kLCM; jGCD++ )
00444             r3[jGCD+kGCD] = LCMbuffer[jGCD];
00445         for ( jGCD = kLCM; jGCD < kGCD; jGCD++ )
00446             r3[jGCD+kGCD] = 0;
00447         k = kGCD;
00448     }
00449     else {
00450         for ( jGCD = kGCD; jGCD < kLCM; jGCD++ )
00451             r3[jGCD] = 0;
00452         for ( jGCD = 0; jGCD < kLCM; jGCD++ )
00453             r3[jGCD+kLCM] = LCMbuffer[jGCD];
00454         k = kLCM;
00455     }
00456     j = 2*k+1;
00457 /*
00458     Now we have to correct the overall factor
00459 */
00460     if ( scale && ( factor == 0 || *factor > 0 ) )
00461         goto ScaledVariety;
00462 /*
00463     The if was added 28-nov-2012 to give MakeInteger also
00464     the (0) option.
00465 */
00466     if ( scale && ( factor == 0 || *factor ) ) {
00467         size = term[*term-1];
00468         size = REDLENG(size);
00469         if ( MulRat(BHEAD (UWORD *)rstop,size,(UWORD *)r3,k,
00470             (UWORD *)rstop,&size) ) goto execargerr;
00471         size = INCLENG(size);
00472         k = size < 0 ? -size: size;
00473         rstop[k-1] = size*sign;
00474         *term = (WORD)(rstop - term) + k;
00475     }
00476 }
00477 else {
00478     if ( factor && *factor >= 1 ) {
00479         r4 = r + *r;
00480         r1 = r + ARGHEAD;
00481         while ( r1 < r4 ) {
00482             r2 = r1 + *r1;
00483             r3 = r2 - ABS(r2[-1]);
00484             j = r3 - r1;
00485             r5 = factor;
00486             if ( j != *r5 ) { r1 = r2; continue; }
00487             r5++; r6 = r1+1;
00488             while ( --j > 0 ) {
00489                 if ( *r5 != *r6 ) break;
00490                 r5++; r6++;
00491             }
00492             if ( j > 0 ) { r1 = r2; continue; }
00493             break;
00494         }
00495         if ( r1 >= r4 ) continue;
00496     }
00497     else {
00498         r1 = r + ARGHEAD;
00499         r2 = r1 + *r1;
00500         r3 = r2 - ABS(r2[-1]);
00501     }
00502     if ( *r3 == 1 && r3[1] == 1 ) {
00503         if ( r2[-1] == 3 ) continue;
00504         if ( r2[-1] == -3 && type == TYPENORM3 ) continue;
00505     }
00506     action = 1;
00507     v[2] |= DIRTYFLAG;
00508     j = r2[-1];
00509     k = REDLENG(j);
00510     if ( j < 0 ) j = -j;
00511     if ( type == TYPENORM && scale && ( factor == 0 || *factor ) ) {
00512 /*
00513         Now we correct the overall factor
00514 */
00515         ScaledVariety;;
00516         size = term[*term-1];
00517         size = REDLENG(size);
00518         if ( scale > 0 ) {
00519             for ( jj = 0; jj < scale; jj++ ) {
00520                 if ( MulRat(BHEAD (UWORD *)rstop,size,(UWORD *)r3,k,

```



```

00521             (UWORD *)rstop,&size) ) goto execargerr;
00522         }
00523     }
00524     else {
00525         for ( jj = 0; jj > scale; jj-- ) {
00526             if ( DivRat(BHEAD (UWORD *)rstop,size,(UWORD *)r3,k,
00527                 (UWORD *)rstop,&size) ) goto execargerr;
00528         }
00529     }
00530     size = INCLENG(size);
00531     k = size < 0 ? -size: size;
00532     rstop[k-1] = size*sign;
00533     *term = (WORD)(rstop - term) + k;
00534 }
00535 }
00536 /*
00537 We generate a statement for adapting all terms in the
00538 argument successively
00539 */
00540 r4 = AddRHS(AT.ebufnum,1);
00541 while ( (r4+j+12) > CC->Top ) r4 = DoubleCbuffer(AT.ebufnum,r4,3);
00542 *r4++ = j+1;
00543 i = (j-1)/2; /* was (j-1)*2 ????? 17-oct-2017 */
00544 for ( k = 0; k < i; k++ ) *r4++ = r3[i+k];
00545 for ( k = 0; k < i; k++ ) *r4++ = r3[k];
00546 if ( ( type == TYPENORM3 ) || ( type == TYPENORM4 ) ) *r4++ = j*sign;
00547 else *r4++ = r3[j-1];
00548 *r4++ = 0;
00549 CC->rhs[CC->numrhs+1] = r4;
00550 CC->Pointer = r4;
00551 AT.mulpat[5] = CC->numrhs;
00552 AT.mulpat[7] = AT.ebufnum;
00553 }
00554 else if ( type == TYPEARGTOEXTRASymbol ) {
00555     WORD n;
00556     if ( r[0] < 0 ) {
00557         /* The argument is in the fast notation. */
00558         WORD tmp[Max(9,FUNHEAD+5)];
00559         switch ( r[0] ) {
00560             case -SNUMBER:
00561                 if ( r[1] == 0 ) {
00562                     tmp[0] = 0;
00563                 }
00564                 else {
00565                     tmp[0] = 4;
00566                     tmp[1] = ABS(r[1]);
00567                     tmp[2] = 1;
00568                     tmp[3] = r[1] > 0 ? 3 : -3;
00569                     tmp[4] = 0;
00570                 }
00571                 break;
00572             case -SYMBOL:
00573                 tmp[0] = 8;
00574                 tmp[1] = SYMBOL;
00575                 tmp[2] = 4;
00576                 tmp[3] = r[1];
00577                 tmp[4] = 1;
00578                 tmp[5] = 1;
00579                 tmp[6] = 1;
00580                 tmp[7] = 3;
00581                 tmp[8] = 0;
00582                 break;
00583             case -INDEX:
00584             case -VECTOR:
00585             case -MINVECTOR:
00586                 tmp[0] = 7;
00587                 tmp[1] = INDEX;
00588                 tmp[2] = 3;
00589                 tmp[3] = r[1];
00590                 tmp[4] = 1;
00591                 tmp[5] = 1;
00592                 tmp[6] = r[0] != -MINVECTOR ? 3 : -3;
00593                 tmp[7] = 0;
00594                 break;
00595             default:
00596                 if ( r[0] <= -FUNCTION ) {
00597                     tmp[0] = FUNHEAD+4;
00598                     tmp[1] = -r[0];
00599                     tmp[2] = FUNHEAD;
00600                     ZeroFillRange(tmp,3,1+FUNHEAD);
00601                     tmp[FUNHEAD+1] = 1;
00602                     tmp[FUNHEAD+2] = 1;
00603                     tmp[FUNHEAD+3] = 3;
00604                     tmp[FUNHEAD+4] = 0;
00605                     break;
00606                 }
00607                 else {

```

```

00608                                     MLOCK(ErrorMessageLock);
00609                                     MesPrint("Unknown fast notation found (TYPEARGTOEXTRASymbol)");
00610                                     MUNLOCK(ErrorMessageLock);
00611                                     return(-1);
00612                                     }
00613                                     }
00614                                     n = FindSubexpression(tmp);
00615                                     }
00616                                     else {
00617                                         /*
00618                                         * NOTE: writing to r[r[0]] is legal. As long as we work
00619                                         * in a part of the term, at least the coefficient of
00620                                         * the term must follow.
00621                                         */
00622                                         WORD old_rr0 = r[r[0]];
00623                                         r[r[0]] = 0; /* zero-terminated */
00624                                         n = FindSubexpression(r+ARGHEAD);
00625                                         r[r[0]] = old_rr0;
00626                                     }
00627                                     /* Put the new argument in the work space. */
00628                                     if ( AT.WorkPointer+2 > AT.WorkTop ) {
00629                                         MLOCK(ErrorMessageLock);
00630                                         MesWork();
00631                                         MUNLOCK(ErrorMessageLock);
00632                                         return(-1);
00633                                     }
00634                                     r1 = AT.WorkPointer;
00635                                     if ( scale ) { /* means "tonumber" */
00636                                         r1[0] = -SNUMBER;
00637                                         r1[1] = n;
00638                                     }
00639                                     else {
00640                                         r1[0] = -SYMBOL;
00641                                         r1[1] = MAXVARIABLES-n;
00642                                     }
00643                                     /* We need r2, r3, m and k to shift the data. */
00644                                     r2 = r + (r[0] > 0 ? r[0] : r[0] <= -FUNCTION ? 1 : 2);
00645                                     r3 = r;
00646                                     m = r1+ARGHEAD+2;
00647                                     k = 2;
00648                                     goto do_shift;
00649                                     }
00650                                     r3 = r;
00651                                     AR.DeferFlag = 0;
00652                                     if ( *r > 0 ) {
00653                                         NewSort(BHEAD0);
00654                                         action = 1;
00655                                         r2 = r + *r;
00656                                         r += ARGHEAD;
00657                                         while ( r < r2 ) { /* Sum over the terms */
00658                                             m = AT.WorkPointer;
00659                                             j = *r;
00660                                             while ( --j >= 0 ) *m++ = *r++;
00661                                             r1 = AT.WorkPointer;
00662                                             AT.WorkPointer = m;
00663                                         } /*
00664                                         What to do with dummy indices?
00665                                         */
00666                                         if ( type == TYPENORM || type == TYPENORM2 || type == TYPENORM3 || type ==
TYPENORM4 ) {
00667                                             if ( MultDo(BHEAD r1,AT.mulp) ) goto execargerr;
00668                                             AT.WorkPointer = r1 + *r1;
00669                                         }
00670                                         if ( Generator(BHEAD r1,level) ) goto execargerr;
00671                                         AT.WorkPointer = r1;
00672                                     }
00673                                     }
00674                                     else {
00675                                         r2 = r + (( *r <= -FUNCTION ) ? 1:2);
00676                                         r1 = AT.WorkPointer;
00677                                         ToGeneral(r,r1,0);
00678                                         m = r1 + ARGHEAD;
00679                                         AT.WorkPointer = r1 + *r1;
00680                                         NewSort(BHEAD0);
00681                                         action = 1;
00682                                     } /*
00683                                     What to do with dummy indices?
00684                                     */
00685                                     if ( type == TYPENORM || type == TYPENORM2 || type == TYPENORM3 || type ==
TYPENORM4 ) {
00686                                         if ( MultDo(BHEAD m,AT.mulp) ) goto execargerr;
00687                                         AT.WorkPointer = m + *m;
00688                                     }
00689                                     if ( (*m != 0) && Generator(BHEAD m,level) ) goto execargerr;
00690                                     AT.WorkPointer = r1;
00691                                     }
00692                                     if ( EndSort(BHEAD AT.WorkPointer+ARGHEAD,1) < 0 ) goto execargerr;

```

```

00693         AR.DeferFlag = olddefer;
00694  /*
00695         Now shift the sorted entity over the old argument.
00696  */
00697         m = AT.WorkPointer+ARGHEAD;
00698         while ( *m ) m += *m;
00699         k = WORDDIF(m,AT.WorkPointer);
00700         *AT.WorkPointer = k;
00701         AT.WorkPointer[1] = 0;
00702         if ( ToFast(AT.WorkPointer,AT.WorkPointer) ) {
00703             if ( *AT.WorkPointer <= -FUNCTION ) k = 1;
00704             else k = 2;
00705         }
00706 do_shift:
00707         if ( *r3 > 0 ) j = k - *r3;
00708         else if ( *r3 <= -FUNCTION ) j = k - 1;
00709         else j = k - 2;
00710
00711         t[1] += j;
00712         action = 1;
00713         v[2] |= DIRTYFLAG;
00714         if ( j > 0 ) {
00715             r = m + j;
00716             while ( m > AT.WorkPointer ) *--r = *--m;
00717             AT.WorkPointer = r;
00718             m = term + *term;
00719             r = m + j;
00720             while ( m > r2 ) *--r = *--m;
00721         }
00722         else if ( j < 0 ) {
00723             r = r2 + j;
00724             r1 = term + *term;
00725             while ( r2 < r1 ) *r++ = *r2++;
00726         }
00727         r = r3;
00728         m = AT.WorkPointer;
00729         NCOPY(r,m,k);
00730         *term += j;
00731         rstop += j;
00732         CC->numrhs = oldnumrhs;
00733         CC->Pointer = CC->Buffer + oldcpointer;
00734     }
00735 }
00736 t += t[1];
00737 }
00738 /*
00739     If TYPENORM4, we allocated Number buffers before the above while loop. Free them.
00740 */
00741 if ( type == TYPENORM4 ) {
00742     NumberFree(GCDBuffer,"execarg");
00743     NumberFree(GCDBuffer2,"execarg");
00744     NumberFree(LCMBuffer,"execarg");
00745     NumberFree(LCMB,"execarg"); NumberFree(LCMc,"execarg");
00746 }
00747 if ( didpolyratfun ) {
00748     PolyFunDirty(BHEAD term);
00749     didpolyratfun = 0;
00750 }
00751 /*
00752 #] Argument detection :
00753 #[ SplitArg : + varieties
00754 */
00755 if ( ( type == TYPESPLITARG || type == TYPESPLITARG2
00756     || type == TYPESPLITFIRSTARG || type == TYPESPLITLASTARG ) &&
00757     AT.pWorkPointer > oldppointer ) {
00758     t = term+1;
00759     r1 = AT.WorkPointer + 1;
00760     lp = oldppointer;
00761     while ( t < rstop ) {
00762         if ( lp < AT.pWorkPointer && t == AT.pWorkSpace[lp] ) {
00763             v = t;
00764             m = t + FUNHEAD;
00765             r = t + t[1];
00766             r2 = r1; while ( t < m ) *r1++ = *t++;
00767             while ( m < r ) {
00768                 t = m;
00769                 NEXTARG(m)
00770                 if ( lp >= AT.pWorkPointer || t != AT.pWorkSpace[lp+1] ) {
00771                     if ( *t > 0 ) t[1] = 0;
00772                     while ( t < m ) *r1++ = *t++;
00773                     continue;
00774                 }
00775             }
00776             /*
00777             Now we have a nontrivial argument that should be done.
00778             */
00779             lp += 2;
00780             action = 1;

```

```

00780         v[2] |= DIRTYFLAG;
00781         r3 = t + *t;
00782         t += ARGHEAD;
00783         if ( type == TYPESPLITFIRSTARG ) {
00784             r4 = r1; r5 = t; r7 = oldwork;
00785             *r1++ = *t + ARGHEAD;
00786             for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
00787             j = 0;
00788             while ( t < r3 ) {
00789                 i = *t;
00790                 if ( j == 0 ) {
00791                     NCOPY(r7,t,i)
00792                     j++;
00793                 }
00794                 else {
00795                     NCOPY(r1,t,i)
00796                 }
00797             }
00798             *r4 = r1 - r4;
00799             if ( j ) {
00800                 if ( ToFast(r4,r4) ) {
00801                     r1 = r4;
00802                     if ( *r1 > -FUNCTION ) r1++;
00803                     r1++;
00804                 }
00805                 r7 = oldwork;
00806                 while ( --j >= 0 ) {
00807                     r4 = r1; i = *r7;
00808                     *r1++ = i+ARGHEAD; *r1++ = 0;
00809                     FILLARG(r1);
00810                     NCOPY(r1,r7,i)
00811                     if ( ToFast(r4,r4) ) {
00812                         r1 = r4;
00813                         if ( *r1 > -FUNCTION ) r1++;
00814                         r1++;
00815                     }
00816                 }
00817             }
00818             t = r3;
00819         }
00820         else if ( type == TYPESPLITLASTARG ) {
00821             r4 = r1; r5 = t; r7 = oldwork;
00822             *r1++ = *t + ARGHEAD;
00823             for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
00824             j = 0;
00825             while ( t < r3 ) {
00826                 i = *t;
00827                 if ( t+i >= r3 ) {
00828                     NCOPY(r7,t,i)
00829                     j++;
00830                 }
00831                 else {
00832                     NCOPY(r1,t,i)
00833                 }
00834             }
00835             *r4 = r1 - r4;
00836             if ( j ) {
00837                 if ( ToFast(r4,r4) ) {
00838                     r1 = r4;
00839                     if ( *r1 > -FUNCTION ) r1++;
00840                     r1++;
00841                 }
00842                 r7 = oldwork;
00843                 while ( --j >= 0 ) {
00844                     r4 = r1; i = *r7;
00845                     *r1++ = i+ARGHEAD; *r1++ = 0;
00846                     FILLARG(r1);
00847                     NCOPY(r1,r7,i)
00848                     if ( ToFast(r4,r4) ) {
00849                         r1 = r4;
00850                         if ( *r1 > -FUNCTION ) r1++;
00851                         r1++;
00852                     }
00853                 }
00854             }
00855             t = r3;
00856         }
00857         else if ( factor == 0 || ( type == TYPESPLITARG2 && *factor == 0 ) ) {
00858             while ( t < r3 ) {
00859                 r4 = r1;
00860                 *r1++ = *t + ARGHEAD;
00861                 for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
00862                 i = *t;
00863                 while ( --i >= 0 ) *r1++ = *t++;
00864                 if ( ToFast(r4,r4) ) {
00865                     r1 = r4;
00866                     if ( *r1 > -FUNCTION ) r1++;

```

```

00867             r1++;
00868         }
00869     }
00870 }
00871 else if ( type == TYPESPLITARG2 ) {
00872 /*
00873     Here we better put the pattern matcher at work?
00874     Remember: there are no wildcards.
00875 */
00876     WORD *oRepFunList = AN.RepFunList;
00877     WORD *oWildMask = AT.WildMask, *oWildValue = AN.WildValue;
00878     AN.WildValue = AT.locwildvalue; AT.WildMask = AT.locwildvalue+2;
00879     AN.NumWild = 0;
00880     r4 = r1; r5 = t; r7 = oldwork;
00881     *r1++ = *t + ARGHEAD;
00882     for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
00883     j = 0;
00884     while ( t < r3 ) {
00885         AN.UseFindOnly = 0; oldwork2 = AT.WorkPointer;
00886         AN.RepFunList = r1;
00887         AT.WorkPointer = r1+AN.RepFunNum+2;
00888         i = *t;
00889         if ( FindRest(BHEAD t,factor) &&
00890             ( AN.UsedOtherFind || FindOnce(BHEAD t,factor) ) ) {
00891             NCOPY(r7,t,i)
00892             j++;
00893         }
00894         else if ( factor[0] == FUNHEAD+1 && factor[1] >= FUNCTION ) {
00895             WORD *rr1 = t+1, *rr2 = t+i;
00896             rr2 -= ABS(rr2[-1]);
00897             while ( rr1 < rr2 ) {
00898                 if ( *rr1 == factor[1] ) break;
00899                 rr1 += rr1[1];
00900             }
00901             if ( rr1 < rr2 ) {
00902                 NCOPY(r7,t,i)
00903                 j++;
00904             }
00905             else {
00906                 NCOPY(r1,t,i)
00907             }
00908         }
00909         else {
00910             NCOPY(r1,t,i)
00911         }
00912         AT.WorkPointer = oldwork2;
00913     }
00914     AN.RepFunList = oRepFunList;
00915     *r4 = r1 - r4;
00916     if ( j ) {
00917         if ( ToFast(r4,r4) ) {
00918             r1 = r4;
00919             if ( *r1 > -FUNCTION ) r1++;
00920             r1++;
00921         }
00922         r7 = oldwork;
00923         while ( --j >= 0 ) {
00924             r4 = r1; i = *r7;
00925             *r1++ = i+ARGHEAD; *r1++ = 0;
00926             FILLARG(r1);
00927             NCOPY(r1,r7,i)
00928             if ( ToFast(r4,r4) ) {
00929                 r1 = r4;
00930                 if ( *r1 > -FUNCTION ) r1++;
00931                 r1++;
00932             }
00933         }
00934     }
00935     t = r3;
00936     AT.WildMask = oWildMask; AN.WildValue = oWildValue;
00937 }
00938 else {
00939 /*
00940     This code deals with splitting off a single term
00941 */
00942     r4 = r1; r5 = t;
00943     *r1++ = *t + ARGHEAD;
00944     for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
00945     j = 0;
00946     while ( t < r3 ) {
00947         r6 = t + *t; r6 -= ABS(r6[-1]);
00948         if ( (r6 - t) == *factor ) {
00949             k = *factor - 1;
00950             for ( ; k > 0; k-- ) {
00951                 if ( t[k] != factor[k] ) break;
00952             }
00953             if ( k <= 0 ) {

```

```

00954             j = r3 - t; t += *t; continue;
00955         }
00956     }
00957     else if ( (r6 - t) == 1 && *factor == 0 ) {
00958         j = r3 - t; t += *t; continue;
00959     }
00960     i = *t;
00961     NCOPY(r1,t,i)
00962 }
00963 *r4 = r1 - r4;
00964 if ( j ) {
00965     if ( ToFast(r4,r4) ) {
00966         r1 = r4;
00967         if ( *r1 > -FUNCTION ) r1++;
00968         r1++;
00969     }
00970     t = r3 - j;
00971     r4 = r1;
00972     *r1++ = *t + ARGHEAD;
00973     for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
00974     i = *t;
00975     while ( --i >= 0 ) *r1++ = *t++;
00976     if ( ToFast(r4,r4) ) {
00977         r1 = r4;
00978         if ( *r1 > -FUNCTION ) r1++;
00979         r1++;
00980     }
00981 }
00982 t = r3;
00983 }
00984 }
00985 r2[1] = r1 - r2;
00986 }
00987 else {
00988     r = t + t[1];
00989     while ( t < r ) *r1++ = *t++;
00990 }
00991 }
00992 r = term + *term;
00993 while ( t < r ) *r1++ = *t++;
00994 m = AT.WorkPointer;
00995 i = m[0] = r1 - m;
00996 t = term;
00997 while ( --i >= 0 ) *t++ = *m++;
00998 if ( AT.WorkPointer < m ) AT.WorkPointer = m;
00999 }
01000 /*
01001 #] SplitArg :
01002 #[ FACTARG :
01003 */
01004 if ( ( type == TYPEFACTARG || type == TYPEFACTARG2 ) &&
01005 AT.pWorkPointer > oldppointer ) {
01006     t = term+1;
01007     r1 = AT.WorkPointer + 1;
01008     lp = oldppointer;
01009     while ( t < rstop ) {
01010         if ( lp < AT.pWorkPointer && AT.pWorkspace[lp] == t ) {
01011             v = t;
01012             m = t + FUNHEAD;
01013             r = t + t[1];
01014             r2 = r1; while ( t < m ) *r1++ = *t++;
01015             while ( m < r ) {
01016                 rr = t = m;
01017                 NEXTARG(m)
01018                 if ( lp >= AT.pWorkPointer || AT.pWorkspace[lp+1] != t ) {
01019                     if ( *t > 0 ) t[1] = 0;
01020                     while ( t < m ) *r1++ = *t++;
01021                     continue;
01022                 }
01023             }
01024             /*
01025             Now we have a nontrivial argument that should be studied.
01026             Try to find common factors.
01027             */
01028             lp += 2;
01029             if ( *t < 0 ) {
01030                 if ( factor && ( *factor == 0 && *t == -SNUMBER ) ) {
01031                     *r1++ = *t++;
01032                     if ( *t == 0 ) *r1++ = *t++;
01033                     else { *r1++ = 1; t++; }
01034                     continue;
01035                 }
01036                 else if ( factor && *factor == 4 && factor[2] == 1 ) {
01037                     if ( *t == -SNUMBER ) {
01038                         if ( factor[3] < 0 || t[1] >= 0 ) {
01039                             while ( t < m ) *r1++ = *t++;
01040                         }
01041                     }
01042                     else {

```

```

01041             *r1++ = -SNUMBER; *r1++ = -1;
01042             *r1++ = *t++; *r1++ = -*t++;
01043         }
01044     }
01045     else {
01046         while ( t < m ) *r1++ = *t++;
01047         *r1++ = -SNUMBER; *r1++ = 1;
01048     }
01049     continue;
01050 }
01051 else if ( *t == -MINVECTOR ) {
01052     if ( AC.OldFactArgFlag == NEWFACTARG ) {
01053         *r1++ = -SNUMBER; *r1++ = -1;
01054         *r1++ = -VECTOR; t++; *r1++ = *t++;
01055     }
01056     else {
01057         *r1++ = -VECTOR; t++; *r1++ = *t++;
01058         *r1++ = -SNUMBER; *r1++ = -1;
01059         *r1++ = -SNUMBER; *r1++ = 1;
01060     }
01061     continue;
01062 }
01063 }
01064 /*
01065 Now we have a nontrivial argument
01066 */
01067 r3 = t + *t;
01068 t += ARGHEAD; r5 = t; /* Store starting point */
01069 /* We have terms from r5 to r3 */
01070 if ( r5+*r5 == r3 && factor ) { /* One term only */
01071     if ( *factor == 0 ) {
01072         GETSTOP(t,r6);
01073         r9 = r1; *r1++ = 0; *r1++ = 1;
01074         FILLARG(r1);
01075         *r1++ = (r6-t)+3; t++;
01076         while ( t < r6 ) *r1++ = *t++;
01077         *r1++ = 1; *r1++ = 1; *r1++ = 3;
01078         *r9 = r1-r9;
01079         if ( ToFast(r9,r9) ) {
01080             if ( *r9 <= -FUNCTION ) r1 = r9+1;
01081             else r1 = r9+2;
01082         }
01083         t = r3; continue;
01084     }
01085     if ( factor[0] == 4 && factor[2] == 1 ) {
01086         GETSTOP(t,r6);
01087         r7 = r1; *r1++ = (r6-t)+3+ARGHEAD; *r1++ = 0;
01088         FILLARG(r1);
01089         *r1++ = (r6-t)+3; t++;
01090         while ( t < r6 ) *r1++ = *t++;
01091         *r1++ = 1; *r1++ = 1; *r1++ = 3;
01092         if ( ToFast(r7,r7) ) {
01093             if ( *r7 <= -FUNCTION ) r1 = r7+1;
01094             else r1 = r7+2;
01095         }
01096         if ( r3[-1] < 0 && factor[3] > 0 ) {
01097             *r1++ = -SNUMBER; *r1++ = -1;
01098             if ( r3[-1] == -3 && r3[-2] == 1
01099                 && ( r3[-3] & MAXPOSITIVE ) == r3[-3] ) {
01100                 *r1++ = -SNUMBER; *r1++ = r3[-3];
01101             }
01102             else {
01103                 *r1++ = (r3-r6)+1+ARGHEAD;
01104                 *r1++ = 0;
01105                 FILLARG(r1);
01106                 *r1++ = (r3-r6+1);
01107                 while ( t < r3 ) *r1++ = *t++;
01108                 r1[-1] = -r1[-1];
01109             }
01110         }
01111         else {
01112             if ( ( r3[-1] == -3 || r3[-1] == 3 )
01113                 && r3[-2] == 1
01114                 && ( r3[-3] & MAXPOSITIVE ) == r3[-3] ) {
01115                 *r1++ = -SNUMBER; *r1++ = r3[-3];
01116                 if ( r3[-1] < 0 ) r1[-1] = - r1[-1];
01117             }
01118             else {
01119                 *r1++ = (r3-r6)+1+ARGHEAD;
01120                 *r1++ = 0;
01121                 FILLARG(r1);
01122                 *r1++ = (r3-r6+1);
01123                 while ( t < r3 ) *r1++ = *t++;
01124             }
01125         }
01126         t = r3; continue;
01127     }

```

```

01128     }
01129  /*
01130     Now we take the first term and look for its pieces
01131     inside the other terms.
01132
01133     It is at this point that a more general factorization
01134     routine could take over (allowing for writing the output
01135     properly of course).
01136  */
01137     if ( AC.OldFactArgFlag == NEWFACTARG ) {
01138     if ( factor == 0 ) {
01139         WORD *oldworkpointer2 = AT.WorkPointer;
01140         AT.WorkPointer = r1 + AM.MaxTer+FUNHEAD;
01141         if ( ArgFactorize(BHEAD t-ARGHEAD,r1) < 0 ) {
01142             MesCall("ExecArg");
01143             return(-1);
01144         }
01145         AT.WorkPointer = oldworkpointer2;
01146         t = r3;
01147         while ( *r1 ) { NEXTARG(r1) }
01148     }
01149     else {
01150         rnext = t + *t;
01151         GETSTOP(t,r6);
01152         t++;
01153         t = r5; pow = 1;
01154         while ( t < r3 ) {
01155             t += *t; if ( t[-1] > 0 ) { pow = 0; break; }
01156         }
01157     /*
01158         We have to add here the code for computing the GCD
01159         and to divide it out.
01160
01161     #[ Numerical factor :
01162  */
01163         t = r5;
01164         EAscrat = (UWORD *) (TermMalloc("execarg"));
01165         if ( t + *t == r3 ) {
01166             if ( factor == 0 || *factor > 2 ) {
01167                 if ( pow > 0 ) {
01168                     *r1++ = -SNUMBER; *r1++ = -1;
01169                     t = r5;
01170                     while ( t < r3 ) {
01171                         t += *t; t[-1] = -t[-1];
01172                     }
01173                 }
01174                 t = rr; *r1++ = *t++; *r1++ = 1; t++;
01175                 COPYARG(r1,t);
01176                 while ( t < m ) *r1++ = *t++;
01177             }
01178         }
01179         else {
01180             GETSTOP(t,r6);
01181             ngcd = t[t[0]-1];
01182             i = abs(ngcd)-1;
01183             while ( --i >= 0 ) EAscrat[i] = r6[i];
01184             t += *t;
01185             while ( t < r3 ) {
01186                 GETSTOP(t,r6);
01187                 i = t[t[0]-1];
01188                 if ( AccumGCD(BHEAD EAscrat,&ngcd, (UWORD *)r6,i) ) goto execargerr;
01189                 if ( ngcd == 3 && EAscrat[0] == 1 && EAscrat[1] == 1 ) break;
01190                 t += *t;
01191             }
01192         /*
01193             if ( ngcd != 3 || EAscrat[0] != 1 || EAscrat[1] != 1 )
01194         */
01195         {
01196             if ( pow ) ngcd = -ngcd;
01197             t = r5; r9 = r1; *r1++ = t[-ARGHEAD]; *r1++ = 1;
01198             FILLARG(r1); ngcd = REDLENG(ngcd);
01199             while ( t < r3 ) {
01200                 GETSTOP(t,r6);
01201                 r7 = t; r8 = r1;
01202                 while ( r7 < r6 ) *r1++ = *r7++;
01203                 t += *t;
01204                 i = REDLENG(t[-1]);
01205                 if ( DivRat(BHEAD (UWORD *)r6,i,EAscrat,ngcd, (UWORD *)r1,&nq) ) goto
execargerr;
01206                 nq = INCLENG(nq);
01207                 i = ABS(nq)-1;
01208                 r1 += i; *r1++ = nq; *r8 = r1-r8;
01209             }
01210             *r9 = r1-r9;
01211             ngcd = INCLENG(ngcd);
01212             i = ABS(ngcd)-1;
01213             if ( factor && *factor == 0 ) {}

```



```

01214         else if ( ( factor && factor[0] == 4 && factor[2] == 1
01215 && factor[3] == -3 ) || pow == 0 ) {
01216             r9 = r1; *r1++ = ARGHEAD+2+i; *r1++ = 0;
01217             FILLARG(r1); *r1++ = i+2;
01218             for ( j = 0; j < i; j++ ) *r1++ = EAscrat[j];
01219             *r1++ = ngcd;
01220             if ( ToFast(r9,r9) ) r1 = r9+2;
01221         }
01222     else if ( factor && factor[0] == 4 && factor[2] == 1
01223 && factor[3] > 0 && pow ) {
01224         if ( ngcd < 0 ) ngcd = -ngcd;
01225         *r1++ = -SNUMBER; *r1++ = -1;
01226         r9 = r1; *r1++ = ARGHEAD+2+i; *r1++ = 0;
01227         FILLARG(r1); *r1++ = i+2;
01228         for ( j = 0; j < i; j++ ) *r1++ = EAscrat[j];
01229         *r1++ = ngcd;
01230         if ( ToFast(r9,r9) ) r1 = r9+2;
01231     }
01232     else {
01233         if ( ngcd < 0 ) ngcd = -ngcd;
01234         if ( pow ) { *r1++ = -SNUMBER; *r1++ = -1; }
01235         if ( ngcd != 3 || EAscrat[0] != 1 || EAscrat[1] != 1 ) {
01236             r9 = r1; *r1++ = ARGHEAD+2+i; *r1++ = 0;
01237             FILLARG(r1); *r1++ = i+2;
01238             for ( j = 0; j < i; j++ ) *r1++ = EAscrat[j];
01239             *r1++ = ngcd;
01240             if ( ToFast(r9,r9) ) r1 = r9+2;
01241         }
01242     }
01243 }
01244 /*
01245     #] Numerical factor :
01246     else {
01247 onetermnew;;
01248
01249         if ( factor == 0 || *factor > 2 ) {
01250             if ( pow > 0 ) {
01251                 *r1++ = -SNUMBER; *r1++ = -1;
01252                 t = r5;
01253                 while ( t < r3 ) {
01254                     t += *t; t[-1] = -t[-1];
01255                 }
01256             }
01257             t = rr; *r1++ = *t++; *r1++ = 1; t++;
01258             COPYARG(r1,t);
01259             while ( t < m ) *r1++ = *t++;
01260         }
01261     }
01262 onetermnew;;
01263 */
01264     }
01265     TermFree(EAscrat,"execarg");
01266 }
01267 }
01268 else { /* AC.OldFactArgFlag is ON */
01269 {
01270     WORD *mnnext, ncom;
01271     rnext = t + *t;
01272     GETSTOP(t,r6);
01273     t++;
01274     if ( factor == 0 ) {
01275         while ( t < r6 ) {
01276             /*
01277             #] SYMBOL :
01278             */
01279             if ( *t == SYMBOL ) {
01280                 r7 = t; r8 = t + t[1]; t += 2;
01281                 while ( t < r8 ) {
01282                     pow = t[1];
01283                     mm = rnext;
01284                     while ( mm < r3 ) {
01285                         mnnext = mm + *mm;
01286                         GETSTOP(mm,mstop); mm++;
01287                         while ( mm < mstop ) {
01288                             if ( *mm != SYMBOL ) mm += mm[1];
01289                             else break;
01290                         }
01291                     }
01292                     if ( *mm == SYMBOL ) {
01293                         mstop = mm + mm[1]; mm += 2;
01294                         while ( *mm != *t && mm < mstop ) mm += 2;
01295                         if ( mm >= mstop ) pow = 0;
01296                         else if ( pow > 0 && mm[1] > 0 ) {
01297                             if ( mm[1] < pow ) pow = mm[1];
01298                         }
01299                         else if ( pow < 0 && mm[1] < 0 ) {
01300                             if ( mm[1] > pow ) pow = mm[1];
01301                         }
01302                     }
01303                 }
01304             }

```

```

01301         else pow = 0;
01302     }
01303     else pow = 0;
01304     if ( pow == 0 ) break;
01305     mm = mnext;
01306 }
01307 if ( pow == 0 ) { t += 2; continue; }
01308 /*
01309 We have a factor
01310 */
01311 action = 1; i = pow;
01312 if ( i > 0 ) {
01313     while ( --i >= 0 ) {
01314         *r1++ = -SYMBOL;
01315         *r1++ = *t;
01316     }
01317 }
01318 else {
01319     while ( i++ < 0 ) {
01320         *r1++ = 8 + ARGHEAD;
01321         for ( j = 1; j < ARGHEAD; j++ ) *r1++ = 0;
01322         *r1++ = 8; *r1++ = SYMBOL;
01323         *r1++ = 4; *r1++ = *t; *r1++ = -1;
01324         *r1++ = 1; *r1++ = 1; *r1++ = 3;
01325     }
01326 }
01327 /*
01328 Now we have to remove the symbols
01329 */
01330 t[1] -= pow;
01331 mm = rnext;
01332 while ( mm < r3 ) {
01333     mnext = mm + *mm;
01334     GETSTOP(mm,mstop); mm++;
01335     while ( mm < mstop ) {
01336         if ( *mm != SYMBOL ) mm += mm[1];
01337         else break;
01338     }
01339     mstop = mm + mm[1]; mm += 2;
01340     while ( mm < mstop && *mm != *t ) mm += 2;
01341     mm[1] -= pow;
01342     mm = mnext;
01343 }
01344 t += 2;
01345 }
01346 }
01347 /*
01348 #] SYMBOL :
01349 #[ DOTPRODUCT :
01350 */
01351 else if ( *t == DOTPRODUCT ) {
01352     r7 = t; r8 = t + t[1]; t += 2;
01353     while ( t < r8 ) {
01354         pow = t[2];
01355         mm = rnext;
01356         while ( mm < r3 ) {
01357             mnext = mm + *mm;
01358             GETSTOP(mm,mstop); mm++;
01359             while ( mm < mstop ) {
01360                 if ( *mm != DOTPRODUCT ) mm += mm[1];
01361                 else break;
01362             }
01363             if ( *mm == DOTPRODUCT ) {
01364                 mstop = mm + mm[1]; mm += 2;
01365                 while ( ( *mm != *t || mm[1] != t[1] )
01366                     && mm < mstop ) mm += 3;
01367                 if ( mm >= mstop ) pow = 0;
01368                 else if ( pow > 0 && mm[2] > 0 ) {
01369                     if ( mm[2] < pow ) pow = mm[2];
01370                 }
01371                 else if ( pow < 0 && mm[2] < 0 ) {
01372                     if ( mm[2] > pow ) pow = mm[2];
01373                 }
01374                 else pow = 0;
01375             }
01376             else pow = 0;
01377             if ( pow == 0 ) break;
01378             mm = mnext;
01379         }
01380     if ( pow == 0 ) { t += 3; continue; }
01381 /*
01382 We have a factor
01383 */
01384 action = 1; i = pow;
01385 if ( i > 0 ) {
01386     while ( --i >= 0 ) {
01387         *r1++ = 9 + ARGHEAD;

```

```

01388         for ( j = 1; j < ARGHEAD; j++ ) *r1++ = 0;
01389         *r1++ = 9; *r1++ = DOTPRODUCT;
01390         *r1++ = 5; *r1++ = *t; *r1++ = t[1]; *r1++ = 1;
01391         *r1++ = 1; *r1++ = 1; *r1++ = 3;
01392     }
01393 }
01394 else {
01395     while ( i++ < 0 ) {
01396         *r1++ = 9 + ARGHEAD;
01397         for ( j = 1; j < ARGHEAD; j++ ) *r1++ = 0;
01398         *r1++ = 9; *r1++ = DOTPRODUCT;
01399         *r1++ = 5; *r1++ = *t; *r1++ = t[1]; *r1++ = -1;
01400         *r1++ = 1; *r1++ = 1; *r1++ = 3;
01401     }
01402 }
01403 /*
01404 Now we have to remove the dotproducts
01405 */
01406 t[2] -= pow;
01407 mm = rnext;
01408 while ( mm < r3 ) {
01409     mnext = mm + *mm;
01410     GETSTOP(mm,mstop); mm++;
01411     while ( mm < mstop ) {
01412         if ( *mm != DOTPRODUCT ) mm += mm[1];
01413         else break;
01414     }
01415     mstop = mm + mm[1]; mm += 2;
01416     while ( mm < mstop && ( *mm != *t
01417         || mm[1] != t[1] ) ) mm += 3;
01418     mm[2] -= pow;
01419     mm = mnext;
01420 }
01421 t += 3;
01422 }
01423 }
01424 /*
01425 #] DOTPRODUCT :
01426 #[ DELTA/VECTOR :
01427 */
01428 else if ( *t == DELTA || *t == VECTOR ) {
01429     r7 = t; r8 = t + t[1]; t += 2;
01430     while ( t < r8 ) {
01431         mm = rnext;
01432         pow = 1;
01433         while ( mm < r3 ) {
01434             mnext = mm + *mm;
01435             GETSTOP(mm,mstop); mm++;
01436             while ( mm < mstop ) {
01437                 if ( *mm != *r7 ) mm += mm[1];
01438                 else break;
01439             }
01440             if ( *mm == *r7 ) {
01441                 mstop = mm + mm[1]; mm += 2;
01442                 while ( ( *mm != *t || mm[1] != t[1] )
01443                     && mm < mstop ) mm += 2;
01444                 if ( mm >= mstop ) pow = 0;
01445             }
01446             else pow = 0;
01447             if ( pow == 0 ) break;
01448             mm = mnext;
01449         }
01450         if( pow == 0 ) { t += 2; continue; }
01451     }
01452     /*
01453 We have a factor
01454 */
01455 action = 1;
01456 *r1++ = 8 + ARGHEAD;
01457 for ( j = 1; j < ARGHEAD; j++ ) *r1++ = 0;
01458 *r1++ = 8; *r1++ = *r7;
01459 *r1++ = 4; *r1++ = *t; *r1++ = t[1];
01460 *r1++ = 1; *r1++ = 1; *r1++ = 3;
01461 /*
01462 Now we have to remove the delta's/vectors
01463 */
01464 mm = rnext;
01465 while ( mm < r3 ) {
01466     mnext = mm + *mm;
01467     GETSTOP(mm,mstop); mm++;
01468     while ( mm < mstop ) {
01469         if ( *mm != *r7 ) mm += mm[1];
01470         else break;
01471     }
01472     mstop = mm + mm[1]; mm += 2;
01473     while ( mm < mstop && (
01474         *mm != *t || mm[1] != t[1] ) ) mm += 2;
01475     *mm = mm[1] = NOINDEX;

```

```

01475             mm = mnext;
01476         }
01477         *t = t[1] = NOINDEX;
01478         t += 2;
01479     }
01480 }
01481 /*
01482 #] DELTA/VECTOR :
01483 #[ INDEX :
01484 */
01485     else if ( *t == INDEX ) {
01486         r7 = t; r8 = t + t[1]; t += 2;
01487         while ( t < r8 ) {
01488             mm = rnext;
01489             pow = 1;
01490             while ( mm < r3 ) {
01491                 mnext = mm + *mm;
01492                 GETSTOP(mm,mstop); mm++;
01493                 while ( mm < mstop ) {
01494                     if ( *mm != *r7 ) mm += mm[1];
01495                     else break;
01496                 }
01497                 if ( *mm == *r7 ) {
01498                     mstop = mm + mm[1]; mm += 2;
01499                     while ( *mm != *t
01500                         && mm < mstop ) mm++;
01501                     if ( mm >= mstop ) pow = 0;
01502                 }
01503                 else pow = 0;
01504                 if ( pow == 0 ) break;
01505                 mm = mnext;
01506             }
01507             if ( pow == 0 ) { t++; continue; }
01508 /*
01509 We have a factor
01510 */
01511 action = 1;
01512 /*
01513 The next looks like an error.
01514 We should have here a VECTOR or INDEX like object
01515
01516 *r1++ = 7 + ARGHEAD;
01517 for ( j = 1; j < ARGHEAD; j++ ) *r1++ = 0;
01518 *r1++ = 7; *r1++ = *r7;
01519 *r1++ = 3; *r1++ = *t;
01520 *r1++ = 1; *r1++ = 1; *r1++ = 3;
01521
01522 Replace this by: (11-apr-2007)
01523 */
01524 if ( *t < 0 ) { *r1++ = -VECTOR; }
01525 else { *r1++ = -INDEX; }
01526 *r1++ = *t;
01527 /*
01528 Now we have to remove the index
01529 */
01530 *t = NOINDEX;
01531 mm = rnext;
01532 while ( mm < r3 ) {
01533     mnext = mm + *mm;
01534     GETSTOP(mm,mstop); mm++;
01535     while ( mm < mstop ) {
01536         if ( *mm != *r7 ) mm += mm[1];
01537         else break;
01538     }
01539     mstop = mm + mm[1]; mm += 2;
01540     while ( mm < mstop &&
01541         *mm != *t ) mm += 1;
01542     *mm = NOINDEX;
01543     mm = mnext;
01544 }
01545 t += 1;
01546 }
01547 }
01548 /*
01549 #] INDEX :
01550 #[ FUNCTION :
01551 */
01552     else if ( *t >= FUNCTION ) {
01553 /*
01554 In the next code we should actually look inside
01555 the DENOMINATOR or EXPONENT for noncommuting objects
01556 */
01557 if ( *t >= FUNCTION &&
01558     functions[*t-FUNCTION].commute == 0 ) ncom = 0;
01559 else ncom = 1;
01560 if ( ncom ) {
01561     mm = r5 + 1;

```

```

01562         while ( mm < t && ( *mm == DUMMYFUN
01563         || *mm == DUMMYTEN ) ) mm += mm[1];
01564         if ( mm < t ) { t += t[1]; continue; }
01565     }
01566     mm = rnext; pow = 1;
01567     while ( mm < r3 ) {
01568         mnext = mm + *mm;
01569         GETSTOP(mm,mstop); mm++;
01570         while ( mm < mstop ) {
01571             if ( *mm == *t && mm[1] == t[1] ) {
01572                 for ( i = 2; i < t[1]; i++ ) {
01573                     if ( mm[i] != t[i] ) break;
01574                 }
01575                 if ( i >= t[1] )
01576                     { mm += mm[1]; goto nextmterm; }
01577             }
01578             if ( ncom && *mm != DUMMYFUN && *mm != DUMMYTEN )
01579                 { pow = 0; break; }
01580             mm += mm[1];
01581         }
01582         if ( mm >= mstop ) pow = 0;
01583         if ( pow == 0 ) break;
01584     nextmterm:
01585     }
01586     if ( pow == 0 ) { t += t[1]; continue; }
01587 /*
01588 Copy the function
01589 */
01590     action = 1;
01591     *r1++ = t[1] + 4 + ARGHEAD;
01592     for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
01593     *r1++ = t[1] + 4;
01594     for ( i = 0; i < t[1]; i++ ) *r1++ = t[i];
01595     *r1++ = 1; *r1++ = 1; *r1++ = 3;
01596 /*
01597 Now we have to take out the functions
01598 */
01599     mm = rnext;
01600     while ( mm < r3 ) {
01601         mnext = mm + *mm;
01602         GETSTOP(mm,mstop); mm++;
01603         while ( mm < mstop ) {
01604             if ( *mm == *t && mm[1] == t[1] ) {
01605                 for ( i = 2; i < t[1]; i++ ) {
01606                     if ( mm[i] != t[i] ) break;
01607                 }
01608                 if ( i >= t[1] ) {
01609                     if ( functions[*t-FUNCTION].spec > 0 )
01610                         *mm = DUMMYTEN;
01611                     else
01612                         *mm = DUMMYFUN;
01613                     mm += mm[1];
01614                     goto nextterm;
01615                 }
01616             }
01617             mm += mm[1];
01618         }
01619     nextterm:
01620     }
01621     if ( functions[*t-FUNCTION].spec > 0 )
01622         *t = DUMMYTEN;
01623     else
01624         *t = DUMMYFUN;
01625     action = 1;
01626     v[2] = DIRTYFLAG;
01627     t += t[1];
01628 }
01629 /*
01630 #] FUNCTION :
01631 */
01632     else {
01633         t += t[1];
01634     }
01635 }
01636 }
01637 t = r5; pow = 1;
01638 while ( t < r3 ) {
01639     t += *t; if ( t[-1] > 0 ) { pow = 0; break; }
01640 }
01641 /*
01642 We have to add here the code for computing the GCD
01643 and to divide it out.
01644 */
01645 /*
01646 #[ Numerical factor :
01647 */
01648     t = r5;

```

```

01649      EAscrat = (UWORD *) (TermMalloc("execarg"));
01650      if ( t + *t == r3 ) goto oneterm;
01651      GETSTOP(t,r6);
01652      ngcd = t[t[0]-1];
01653      i = abs(ngcd)-1;
01654      while ( --i >= 0 ) EAscrat[i] = r6[i];
01655      t += *t;
01656      while ( t < r3 ) {
01657          GETSTOP(t,r6);
01658          i = t[t[0]-1];
01659          if ( AccumGCD(BHEAD EAscrat,&ngcd,(UWORD *)r6,i) ) goto execargerr;
01660          if ( ngcd == 3 && EAscrat[0] == 1 && EAscrat[1] == 1 ) break;
01661          t += *t;
01662      }
01663      if ( ngcd != 3 || EAscrat[0] != 1 || EAscrat[1] != 1 ) {
01664          if ( pow ) ngcd = -ngcd;
01665          t = r5; r9 = r1; *r1++ = t[-ARGHEAD]; *r1++ = 1;
01666          FILLARG(r1); ngcd = REDLENG(ngcd);
01667          while ( t < r3 ) {
01668              GETSTOP(t,r6);
01669              r7 = t; r8 = r1;
01670              while ( r7 < r6 ) *r1++ = *r7++;
01671              t += *t;
01672              i = REDLENG(t[-1]);
01673              if ( DivRat(BHEAD (UWORD *)r6,i,EAscrat,ngcd,(UWORD *)r1,&nq) ) goto
execargerr;
01674              nq = INCLENG(nq);
01675              i = ABS(nq)-1;
01676              r1 += i; *r1++ = nq; *r8 = r1-r8;
01677          }
01678          *r9 = r1-r9;
01679          ngcd = INCLENG(ngcd);
01680          i = ABS(ngcd)-1;
01681          if ( factor && *factor == 0 ) {}
01682          else if ( ( factor && factor[0] == 4 && factor[2] == 1
01683          && factor[3] == -3 ) || pow == 0 ) {
01684              r9 = r1; *r1++ = ARGHEAD+2+i; *r1++ = 0;
01685              FILLARG(r1); *r1++ = i+2;
01686              for ( j = 0; j < i; j++ ) *r1++ = EAscrat[j];
01687              *r1++ = ngcd;
01688              if ( ToFast(r9,r9) ) r1 = r9+2;
01689          }
01690          else if ( factor && factor[0] == 4 && factor[2] == 1
01691          && factor[3] > 0 && pow ) {
01692              if ( ngcd < 0 ) ngcd = -ngcd;
01693              *r1++ = -SNUMBER; *r1++ = -1;
01694              r9 = r1; *r1++ = ARGHEAD+2+i; *r1++ = 0;
01695              FILLARG(r1); *r1++ = i+2;
01696              for ( j = 0; j < i; j++ ) *r1++ = EAscrat[j];
01697              *r1++ = ngcd;
01698              if ( ToFast(r9,r9) ) r1 = r9+2;
01699          }
01700          else {
01701              if ( ngcd < 0 ) ngcd = -ngcd;
01702              if ( pow ) { *r1++ = -SNUMBER; *r1++ = -1; }
01703              if ( ngcd != 3 || EAscrat[0] != 1 || EAscrat[1] != 1 ) {
01704                  r9 = r1; *r1++ = ARGHEAD+2+i; *r1++ = 0;
01705                  FILLARG(r1); *r1++ = i+2;
01706                  for ( j = 0; j < i; j++ ) *r1++ = EAscrat[j];
01707                  *r1++ = ngcd;
01708                  if ( ToFast(r9,r9) ) r1 = r9+2;
01709              }
01710          }
01711      }
01712      /*
01713      #] Numerical factor :
01714      */
01715      else {
01716      oneterm;;
01717          if ( factor == 0 || *factor > 2 ) {
01718              if ( pow > 0 ) {
01719                  *r1++ = -SNUMBER; *r1++ = -1;
01720                  t = r5;
01721                  while ( t < r3 ) {
01722                      t += *t; t[-1] = -t[-1];
01723                  }
01724              }
01725              t = rr; *r1++ = *t++; *r1++ = 1; t++;
01726              COPYARG(r1,t);
01727              while ( t < m ) *r1++ = *t++;
01728          }
01729      }
01730      TermFree(EAscrat,"execarg");
01731      }
01732      } /* AC.OldFactArgFlag */
01733      }
01734      /* r1 is fout in ons voorbeeld. */

```

```

01735         r2[1] = r1 - r2;
01736         action = 1;
01737         v[2] = DIRTYFLAG;
01738     }
01739     else {
01740         r = t + t[1];
01741         while ( t < r ) *r1++ = *t++;
01742     }
01743 }
01744 r = term + *term;
01745 while ( t < r ) *r1++ = *t++;
01746 m = AT.WorkPointer;
01747 i = m[0] = r1 - m;
01748 t = term;
01749 while ( --i >= 0 ) *t++ = *m++;
01750 if ( AT.WorkPointer < t ) AT.WorkPointer = t;
01751 }
01752 /*
01753 #] FACTARG :
01754 */
01755 AR.Cnumlhs = oldnumlhs;
01756 if ( action && Normalize(BHEAD term) ) goto execargerr;
01757 AT.WorkPointer = oldwork;
01758 if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
01759 AT.pWorkPointer = oldppointer;
01760 return(action);
01761 execargerr:
01762 AT.WorkPointer = oldwork;
01763 AT.pWorkPointer = oldppointer;
01764 MLOCK(ErrorMessageLock);
01765 MesCall("execarg");
01766 MUNLOCK(ErrorMessageLock);
01767 return(-1);
01768 }
01769
01770 /*
01771 #] execarg :
01772 #[ execterm :
01773 */
01774
01775 int execterm(PHEAD WORD *term, WORD level)
01776 {
01777     GETBIDENTITY
01778     CBUF *C = cbuf+AM.rbufnum;
01779     WORD oldnumlhs = AR.Cnumlhs;
01780     WORD maxisat = C->lhs[level][2];
01781     WORD *buffer1 = 0;
01782     WORD *oldworkpointer = AT.WorkPointer;
01783     WORD *t1, i;
01784     WORD olddeferflag = AR.DeferFlag, tryterm = 0;
01785     AR.DeferFlag = 0;
01786     do {
01787         AR.Cnumlhs = C->lhs[level][3];
01788         NewSort(BHEAD0);
01789     /*
01790         Normally for function arguments we do not use PolyFun/PolyRatFun.
01791         Hence NewSort sets the corresponding variables to zero.
01792         Here we overwrite that.
01793     */
01794     AN.FunSorts[AR.sLevel]->PolyFlag = ( AR.PolyFun != 0 ) ? AR.PolyFunType: 0;
01795     if ( AR.PolyFun == 0 ) { AN.FunSorts[AR.sLevel]->PolyFlag = 0; }
01796     else if ( AR.PolyFunType == 1 ) { AN.FunSorts[AR.sLevel]->PolyFlag = 1; }
01797     else if ( AR.PolyFunType == 2 ) {
01798         if ( AR.PolyFunExp == 2 ) AN.FunSorts[AR.sLevel]->PolyFlag = 1;
01799         else AN.FunSorts[AR.sLevel]->PolyFlag = 2;
01800     }
01801     if ( buffer1 ) {
01802         term = buffer1;
01803         while ( *term ) {
01804             t1 = oldworkpointer;
01805             i = *term; while ( --i >= 0 ) *t1++ = *term++;
01806             AT.WorkPointer = t1;
01807             if ( Generator(BHEAD oldworkpointer,level) ) goto exectermerr;
01808         }
01809     }
01810     else {
01811         if ( Generator(BHEAD term,level) ) goto exectermerr;
01812     }
01813     if ( buffer1 ) {
01814         if ( tryterm ) { TermFree(buffer1,"buffer in sort statement"); tryterm = 0; }
01815         else { M_free((void *)buffer1,"buffer in sort statement"); }
01816         buffer1 = 0;
01817     }
01818     AN.tryterm = 1;
01819     if ( EndSort(BHEAD (WORD *)((void *)&buffer1),2) < 0 ) goto exectermerr;
01820     tryterm = AN.tryterm; AN.tryterm = 0;
01821     level = AR.Cnumlhs;

```

```

01822     } while ( AR.Cnumlhs < maxisat );
01823     AR.Cnumlhs = oldnumlhs;
01824     AR.DeferFlag = olddeferflag;
01825     term = buffer1;
01826     while ( *term ) {
01827         t1 = oldworkpointer;
01828         i = *term; while ( --i >= 0 ) *t1++ = *term++;
01829         AT.WorkPointer = t1;
01830         if ( Generator(BHEAD oldworkpointer,level) ) goto exectermerr;
01831     }
01832     if ( tryterm ) { TermFree(buffer1,"buffer in term statement"); tryterm = 0; }
01833     else { M_free(buffer1,"buffer in term statement"); }
01834     buffer1 = 0;
01835     AT.WorkPointer = oldworkpointer;
01836     return(0);
01837 exectermerr:
01838     AT.WorkPointer = oldworkpointer;
01839     AR.DeferFlag = olddeferflag;
01840     MLOCK(ErrorMessageLock);
01841     MesCall("execterm");
01842     MUNLOCK(ErrorMessageLock);
01843     return(-1);
01844 }
01845
01846 /*
01847  #] execterm :
01848  #[ ArgumentImplode :
01849  */
01850
01851 int ArgumentImplode(PHEAD WORD *term, WORD *thelist)
01852 {
01853     GETBIDENTITY
01854     WORD *liststart, *liststop, *inlist;
01855     WORD *w, *t, *tend, *tstop, *tt, *ttstop, *ttt, ncount, i;
01856     int action = 0;
01857     liststop = thelist + thelist[1];
01858     liststart = thelist + 2;
01859     t = term;
01860     tend = t + *t;
01861     tstop = tend - ABS(tend[-1]);
01862     t++;
01863     while ( t < tstop ) {
01864         if ( *t >= FUNCTION ) {
01865             inlist = liststart;
01866             while ( inlist < liststop && *inlist != *t ) inlist += inlist[1];
01867             if ( inlist < liststop ) {
01868                 tt = t; ttstop = t + t[1]; w = AT.WorkPointer;
01869                 for ( i = 0; i < FUNHEAD; i++ ) *w++ = *tt++;
01870                 while ( tt < ttstop ) {
01871                     ncount = 0;
01872                     if ( *tt == -SNUMBER && tt[1] == 0 ) {
01873                         ncount = 1; ttt = tt; tt += 2;
01874                         while ( tt < ttstop && *tt == -SNUMBER && tt[1] == 0 ) {
01875                             ncount++; tt += 2;
01876                         }
01877                     }
01878                     if ( ncount > 0 ) {
01879                         if ( tt < ttstop && *tt == -SNUMBER && ( tt[1] == 1 || tt[1] == -1 ) ) {
01880                             *w++ = -SNUMBER;
01881                             *w++ = (ncount+1) * tt[1];
01882                             tt += 2;
01883                             action = 1;
01884                         }
01885                         else if ( ( tt[0] == tt[ARGHEAD] + ARGHEAD )
01886                             && ( ABS(tt[tt[0]-1]) == 3 )
01887                             && ( tt[tt[0]-2] == 1 )
01888                             && ( tt[tt[0]-3] == 1 ) ) { /* Single term with coef +/- 1 */
01889                             i = *tt; NCOPY(w,tt,i)
01890                             w[-3] = ncount+1;
01891                             action = 1;
01892                         }
01893                     }
01894                     else if ( *tt == -SYMBOL ) {
01895                         *w++ = ARGHEAD+8;
01896                         *w++ = 0;
01897                         FILLARG(w)
01898                         *w++ = 8;
01899                         *w++ = SYMBOL;
01900                         *w++ = tt[1];
01901                         *w++ = 1;
01902                         *w++ = ncount+1; *w++ = 1; *w++ = 3;
01903                         tt += 2;
01904                         action = 1;
01905                     }
01906                     else if ( *tt <= -FUNCTION ) {
01907                         *w++ = ARGHEAD+FUNHEAD+4;
01908                         *w++ = 0;
01909                         FILLARG(w)

```



```

01909             *w++ = -*tt++;
01910             *w++ = FUNHEAD+4;
01911             FILLFUN(w)
01912             *w++ = ncount+1; *w++ = 1; *w++ = 3;
01913             action = 1;
01914         }
01915         else {
01916             while ( ttt < tt ) *w++ = *ttt++;
01917             if ( tt < ttstop && *tt == -SNUMBER ) {
01918                 *w++ = *tt++; *w++ = *tt++;
01919             }
01920         }
01921     }
01922     else if ( *tt <= -FUNCTION ) {
01923         *w++ = *ttt++;
01924     }
01925     else if ( *tt < 0 ) {
01926         *w++ = *ttt++;
01927         *w++ = *ttt++;
01928     }
01929     else {
01930         i = *tt; NCOPY(w,tt,i)
01931     }
01932 }
01933 AT.WorkPointer[1] = w - AT.WorkPointer;
01934 while ( tt < tend ) *w++ = *tt++;
01935 ttt = AT.WorkPointer; tt = t;
01936 while ( ttt < w ) *ttt++ = *ttt++;
01937 term[0] = tt - term;
01938 AT.WorkPointer = tt;
01939 tend = tt; tstop = tt - ABS(tt[-1]);
01940 }
01941 }
01942 t += t[1];
01943 }
01944 if ( action ) {
01945     if ( Normalize(BHEAD term) ) return(-1);
01946 }
01947 return(0);
01948 }
01949
01950 /*
01951 #] ArgumentImplode :
01952 #[ ArgumentExplode :
01953 */
01954
01955 int ArgumentExplode(PHEAD WORD *term, WORD *thelist)
01956 {
01957     GETBIDENTITY
01958     WORD *liststart, *liststop, *inlist, *old;
01959     WORD *w, *t, *tend, *tstop, *tt, *ttstop, *ttt, ncount, i;
01960     int action = 0;
01961     LONG x;
01962     liststop = thelist + thelist[1];
01963     liststart = thelist + 2;
01964     t = term;
01965     tend = t + *t;
01966     tstop = tend - ABS(tend[-1]);
01967     t++;
01968     while ( t < tstop ) {
01969         if ( *t >= FUNCTION ) {
01970             inlist = liststart;
01971             while ( inlist < liststop && *inlist != *t ) inlist += inlist[1];
01972             if ( inlist < liststop ) {
01973                 tt = t; ttstop = t + t[1]; w = AT.WorkPointer;
01974                 for ( i = 0; i < FUNHEAD; i++ ) *w++ = *tt++;
01975                 while ( tt < ttstop ) {
01976                     if ( *tt == -SNUMBER && tt[1] != 0 ) {
01977                         if ( tt[1] < AM.MaxTer/((WORD)sizeof(WORD)*4)
01978                             && tt[1] > -(AM.MaxTer/((WORD)sizeof(WORD)*4))
01979                             && ( tt[1] > 1 || tt[1] < -1 ) ) {
01980                             ncount = ABS(tt[1]);
01981                             while ( ncount > 1 ) {
01982                                 *w++ = -SNUMBER; *w++ = 0; ncount--;
01983                             }
01984                             *w++ = -SNUMBER;
01985                             if ( tt[1] < 0 ) *w++ = -1;
01986                             else *w++ = 1;
01987                             tt += 2;
01988                             action = 1;
01989                         }
01990                     }
01991                     else {
01992                         *w++ = *tt++; *w++ = *tt++;
01993                     }
01994                 }
01995             }
01996             else if ( *tt <= -FUNCTION ) {
01997                 *w++ = *ttt++;

```

```

01996         }
01997         else if ( *tt < 0 ) {
01998             *w++ = *t++;
01999             *w++ = *t++;
02000         }
02001         else if ( tt[0] == tt[ARGHEAD]+ARGHEAD ) {
02002             ttt = tt + tt[0] - 1;
02003             i = (ABS(ttt[0])-1)/2;
02004             if ( i > 1 ) {
02005 TooMany:
02006                 old = AN.currentTerm;
02007                 AN.currentTerm = term;
02008                 MesPrint("Too many arguments in output of ArgExplode");
02009                 MesPrint("Term = %t");
02010                 AN.currentTerm = old;
02011                 return(-1);
02012             }
02013             if ( ttt[-1] != 1 ) goto NoExplode;
02014             x = ttt[-2];
02015             if ( 2*x > (AT.WorkTop-w)-*term ) goto TooMany;
02016             ncount = x - 1;
02017             while ( ncount > 0 ) {
02018                 *w++ = -SNUMBER; *w++ = 0; ncount--;
02019             }
02020             ttt[-2] = 1;
02021             i = *tt; NCOPY(w,tt,i)
02022             action = 1;
02023         }
02024         else {
02025 NoExplode:
02026             i = *tt; NCOPY(w,tt,i)
02027         }
02028         AT.WorkPointer[1] = w - AT.WorkPointer;
02029         while ( tt < tend ) *w++ = *t++;
02030         ttt = AT.WorkPointer; tt = t;
02031         while ( ttt < w ) *t++ = *tt++;
02032         term[0] = tt - term;
02033         AT.WorkPointer = tt;
02034         tend = tt; tstop = tt - ABS(tt[-1]);
02035     }
02036 }
02037 t += t[1];
02038 }
02039 if ( action ) {
02040     if ( Normalize(BHEAD term) ) return(-1);
02041 }
02042 return(0);
02043 }
02044
02045 /*
02046 #] ArgumentExplode :
02047 #[ ArgFactorize :
02048 */
02049 #define NEWORDER
02050 int ArgFactorize(PHEAD WORD *argin, WORD *argout)
02051 {
02052     /*
02053     #[ step 0 : Declarations and initializations
02054     */
02055     WORD *argfree, *argextra, *argcopy, *t, *tstop, *a, *a1, *a2;
02056 #ifndef NEWORDER
02057     WORD *tt;
02058 #endif
02059     WORD startebuf = cbuf[AT.ebufnum].numrhs,oldword;
02060     WORD oldsorttype = AR.SortType, numargs;
02061     int error = 0, action = 0, i, ii, number, sign = 1;
02062
02063     *argout = 0;
02064     /*
02065     #] step 0 :
02066     #[ step 1 : Take care of ordering
02067     */
02068     AR.SortType = SORTHIGHFIRST;
02069     if ( oldsorttype != AR.SortType ) {
02070         NewSort(BHEAD0);
02071         oldword = argin[*argin]; argin[*argin] = 0;
02072         t = argin+ARGHEAD;
02073         while ( *t ) {
02074             tstop = t + *t;
02075             if ( AN.ncmod != 0 ) {
02076                 if ( AN.ncmod != 1 || ( (WORD)AN.cmod[0] < 0 ) ) {
02077                     MLOCK(ErrorMessageLock);
02078                     MesPrint("Factorization modulus a number, greater than a WORD not implemented.");
02079                     MUNLOCK(ErrorMessageLock);
02080                     Terminate(-1);
02081                 }
02082             }
02083         }
02084     }

```

```

02098         if ( Modulus(t) ) {
02099             MLOCK(ErrorMessageLock);
02100             MesCall("ArgFactorize");
02101             MUNLOCK(ErrorMessageLock);
02102             Terminate(-1);
02103         }
02104         if ( !*t ) { t = tstop; continue; }
02105     }
02106     StoreTerm(BHEAD t);
02107     t = tstop;
02108 }
02109 /* par = 1, in case the arg has more than SubTermsInSmall terms */
02110 EndSort(BHEAD argin+ARGHEAD,1);
02111 argin[*argin] = oldword;
02112 }
02113 /*
02114 #] step 1 :
02115 #[ step 2 : take out the 'content'.
02116 */
02117 argfree = TakeArgContent(BHEAD argin,argout);
02118 {
02119     al = argout;
02120     while ( *al ) {
02121         if ( al[0] == -SNUMBER && ( al[1] == 1 || al[1] == -1 ) ) {
02122             if ( al[1] == -1 ) { sign = -sign; al[1] = 1; }
02123             if ( al[2] ) {
02124                 a = t = al+2; while ( *t ) NEXTARG(t);
02125                 i = t - al-2;
02126                 t = al; NCOPY(t,a,i);
02127                 *t = 0;
02128                 continue;
02129             }
02130             else {
02131                 al[0] = 0;
02132             }
02133             break;
02134         }
02135         else if ( al[0] == FUNHEAD+ARGHEAD+4 && al[ARGHEAD] == FUNHEAD+4
02136             && al[*al-1] == 3 && al[*al-2] == 1 && al[*al-3] == 1
02137             && al[ARGHEAD+1] >= FUNCTION ) {
02138             a = t = al+*al; while ( *t ) NEXTARG(t);
02139             i = t - a;
02140             *al = -al[ARGHEAD+1]; t = al+1; NCOPY(t,a,i);
02141             *t = 0;
02142         }
02143         NEXTARG(al);
02144     }
02145 }
02146 if ( argfree == 0 ) {
02147     argfree = argin;
02148 }
02149 else if ( argfree[0] == ( argfree[ARGHEAD]+ARGHEAD ) ) {
02150     Normalize(BHEAD argfree+ARGHEAD);
02151     argfree[0] = argfree[ARGHEAD]+ARGHEAD;
02152     argfree[1] = 0;
02153     if ( ( argfree[0] == ARGHEAD+4 ) && ( argfree[ARGHEAD+3] == 3 )
02154         && ( argfree[ARGHEAD+1] == 1 ) && ( argfree[ARGHEAD+2] == 1 ) ) {
02155         goto return0;
02156     }
02157 }
02158 else {
02159     /*
02160         The way we took out objects is rather brutish. We have to
02161         normalize
02162     */
02163     NewSort(BHEAD0);
02164     t = argfree+ARGHEAD;
02165     while ( *t ) {
02166         tstop = t + *t;
02167         Normalize(BHEAD t);
02168         StoreTerm(BHEAD t);
02169         t = tstop;
02170     }
02171     /* par = 1, in case the arg has more than SubTermsInSmall terms */
02172     EndSort(BHEAD argfree+ARGHEAD,1);
02173     t = argfree+ARGHEAD;
02174     while ( *t ) t += *t;
02175     *argfree = t - argfree;
02176 }
02177 /*
02178 #] step 2 :
02179 #[ step 3 : look whether we have done this one already.
02180 */
02181 if ( ( number = FindArg(BHEAD argfree) ) != 0 ) {
02182     if ( number > 0 ) t = cbuf[AT.fbufnum].rhs[number-1];
02183     else t = cbuf[AC.ffbufnum].rhs[-number-1];
02184 }

```

```

02185         Now position on the result. Remember we have in the cache:
02186             inputarg,0,outputargs,0
02187         t is currently at inputarg. *inputarg is always positive.
02188         in principle this holds also for the arguments in the output
02189         but we take no risks here (in case of future developments).
02190     */
02191     t += *t; t++;
02192     tstop = t;
02193     ii = 0;
02194     while ( *tstop ) {
02195         if ( *tstop == -SNUMBER && tstop[1] == -1 ) {
02196             sign = -sign; ii += 2;
02197         }
02198         NEXTARG(tstop);
02199     }
02200     a = argout; while ( *a ) NEXTARG(a);
02201     #ifndef NEWORDER
02202     if ( sign == -1 ) { *a++ = -SNUMBER; *a++ = -1; *a = 0; sign = 1; }
02203     #endif
02204     i = tstop - t - ii;
02205     ii = a - argout;
02206     a2 = a; a1 = a + i;
02207     *a1 = 0;
02208     while ( ii > 0 ) { *--a1 = *--a2; ii--; }
02209     a = argout;
02210     while ( *t ) {
02211         if ( *t == -SNUMBER && t[1] == -1 ) { t += 2; }
02212         else { COPY1ARG(a,t) }
02213     }
02214     goto return0;
02215 }
02216 /*
02217 #] step 3 :
02218 #[ step 4 : invoke ConvertToPoly
02219
02220         We make a copy first in case there are no factors
02221 */
02222 argcopy = TermMalloc("argcopy");
02223 for ( i = 0; i <= *argfree; i++ ) argcopy[i] = argfree[i];
02224
02225 tstop = argfree + *argfree;
02226 {
02227     WORD sumcommu = 0;
02228     t = argfree + ARGHEAD;
02229     while ( t < tstop ) {
02230         sumcommu += DoesCommu(t);
02231         t += *t;
02232     }
02233     if ( sumcommu > 1 ) {
02234         MesPrint("ERROR: Cannot factorize an argument with more than one noncommuting object");
02235         Terminate(-1);
02236     }
02237 }
02238 t = argfree + ARGHEAD;
02239
02240 while ( t < tstop ) {
02241     if ( ( t[1] != SYMBOL ) && ( *t != (ABS(t[*t-1])+1) ) ) {
02242         action = 1; break;
02243     }
02244     t += *t;
02245 }
02246 if ( action ) {
02247     t = argfree + ARGHEAD;
02248     argextra = AT.WorkPointer;
02249     NewSort(BHEAD0);
02250     while ( t < tstop ) {
02251         if ( LocalConvertToPoly(BHEAD t,argextra,startebuf,0) < 0 ) {
02252             error = -1;
02253             goto:
02254             AR.SortType = oldsorttype;
02255             TermFree(argcopy,"argcopy");
02256             if ( argfree != argin ) TermFree(argfree,"argfree");
02257             MesCall("ArgFactorize");
02258             Terminate(-1);
02259             return(-1);
02260         }
02261         StoreTerm(BHEAD argextra);
02262         t += *t; argextra += *argextra;
02263     }
02264     /* par = 1, in case the arg has more than SubTermsInSmall terms */
02265     if ( EndSort(BHEAD argfree+ARGHEAD,1) < 0 ) { error = -2; goto: }
02266     t = argfree + ARGHEAD;
02267     while ( *t > 0 ) t += *t;
02268     *argfree = t - argfree;
02269 }
02270 /*
02271 #] step 4 :

```

```

02272     #[ step 5 : If not in the tables, we have to do this by hard work.
02273 */
02274
02275     a = argout;
02276     while ( *a ) NEXTARG(a);
02277     if ( poly_factorize_argument(BHEAD argfree,a) < 0 ) {
02278         MesCall("ArgFactorize");
02279         error = -1;
02280     }
02281 */
02282     #[ step 5 :
02283     #[ step 6 : use now ConvertFromPoly
02284
02285         Be careful: there should be more than one argument now.
02286 */
02287     if ( error == 0 && action ) {
02288         a1 = a; NEXTARG(a1);
02289         if ( *a1 != 0 ) {
02290             CBUF *C = cbuf+AC.cbunum;
02291             CBUF *CC = cbuf+AT.ebunum;
02292             WORD *oldworkpointer = AT.WorkPointer;
02293             WORD *argcopy2 = TermMalloc("argcopy2"), *a1, *a2;
02294             a1 = a; a2 = argcopy2;
02295             while ( *a1 ) {
02296                 if ( *a1 < 0 ) {
02297                     if ( *a1 > -FUNCTION ) *a2++ = *a1++;
02298                     *a2++ = *a1++; *a2 = 0;
02299                     continue;
02300                 }
02301                 t = a1 + ARGHEAD;
02302                 tstop = a1 + *a1;
02303                 argextra = AT.WorkPointer;
02304                 NewSort(BHEAD0);
02305                 while ( t < tstop ) {
02306                     if ( ConvertFromPoly(BHEAD t,argextra,numxsymbol,CC->numrhs-startebuf+numxsymbol
02307 ,startebuf-numxsymbol,1) <= 0 ) {
02308                         TermFree(argcopy2,"argcopy2");
02309                         LowerSortLevel();
02310                         error = -3;
02311                         goto getout;
02312                     }
02313                     t += *t;
02314                     AT.WorkPointer = argextra + *argextra;
02315 */
02316                     ConvertFromPoly leaves terms with subexpressions. Hence:
02317 */
02318                     if ( Generator(BHEAD argextra,C->numlhs) ) {
02319                         TermFree(argcopy2,"argcopy2");
02320                         LowerSortLevel();
02321                         error = -4;
02322                         goto getout;
02323                     }
02324                 }
02325                 AT.WorkPointer = oldworkpointer;
02326                 /* par = 1, in case the factor has more than SubTermsInSmall terms */
02327                 if ( EndSort(BHEAD a2+ARGHEAD,1) < 0 ) { error = -5; goto getout; }
02328                 t = a2+ARGHEAD; while ( *t ) t += *t;
02329                 *a2 = t - a2; a2[1] = 0; ZEROARG(a2);
02330                 ToFast(a2,a2); NEXTARG(a2);
02331                 a1 = tstop;
02332             }
02333             i = a2 - argcopy2;
02334             a2 = argcopy2; a1 = a;
02335             NCOPY(a1,a2,i);
02336             *a1 = 0;
02337             TermFree(argcopy2,"argcopy2");
02338 */
02339             Erase the entries we made temporarily in cbuf[AT.ebunum]
02340 */
02341             CC->numrhs = startebuf;
02342         }
02343         else { /* no factorization. recover the argument from before step 3. */
02344             for ( i = 0; i <= *argcopy; i++ ) a[i] = argcopy[i];
02345         }
02346     }
02347 */
02348     #[ step 6 :
02349     #[ step 7 : Add this one to the tables.
02350
02351         Possibly drop some elements in the tables
02352         when they become too full.
02353 */
02354     if ( error == 0 && AN.ncmod == 0 ) {
02355         if ( InsertArg(BHEAD argcopy,a,0) < 0 ) { error = -1; }
02356     }
02357 */
02358     #[ step 7 :

```

```

02359     #[ step 8 : Clean up and return.
02360
02361         Change the order of the arguments in argout and a.
02362         Use argcopy as spare space.
02363 */
02364     ii = a - argout;
02365     for ( i = 0; i < ii; i++ ) argcopy[i] = argout[i];
02366     a1 = a;
02367     while ( *a1 ) {
02368         if ( *a1 == -SNUMBER && a1[1] < 0 ) {
02369             sign = -sign; a1[1] = -a1[1];
02370             if ( a1[1] == 1 ) {
02371                 a2 = a1+2; while ( *a2 ) NEXTARG(a2);
02372                 i = a2-a1-2; a2 = a1+2;
02373                 NCOPY(a1,a2,i);
02374                 *a1 = 0;
02375             }
02376             while ( *a1 ) NEXTARG(a1);
02377             break;
02378         }
02379         else {
02380             if ( *a1 > 0 && *a1 == a1[ARGHEAD]+ARGHEAD && a1[*a1-1] < 0 ) {
02381                 a1[*a1-1] = -a1[*a1-1]; sign = -sign;
02382             }
02383             if ( *a1 == ARGHEAD+4 && a1[ARGHEAD+1] == 1 && a1[ARGHEAD+2] == 1 ) {
02384                 a2 = a1+ARGHEAD+4; while ( *a2 ) NEXTARG(a2);
02385                 i = a2-a1-ARGHEAD-4; a2 = a1+ARGHEAD+4;
02386                 NCOPY(a1,a2,i);
02387                 *a1 = 0;
02388                 break;
02389             }
02390             while ( *a1 ) NEXTARG(a1);
02391             break;
02392         }
02393         NEXTARG(a1);
02394     }
02395     i = a1 - a;
02396     a2 = argout;
02397     NCOPY(a2,a,i);
02398     for ( i = 0; i < ii; i++ ) *a2++ = argcopy[i];
02399 #ifndef NEWORDER
02400     if ( sign == -1 ) { *a2++ = -SNUMBER; *a2++ = -1; sign = 1; }
02401 #endif
02402     *a2 = 0;
02403     TermFree(argcopy,"argcopy");
02404     return 0;
02405     if ( argfree != argin ) TermFree(argfree,"argfree");
02406     if ( oldsorttype != AR.SortType ) {
02407         AR.SortType = oldsorttype;
02408         a = argout;
02409         while ( *a ) {
02410             if ( *a > 0 ) {
02411                 NewSort(BHEAD0);
02412                 oldword = a[*a]; a[*a] = 0;
02413                 t = a+ARGHEAD;
02414                 while ( *t ) {
02415                     tstop = t + *t;
02416                     StoreTerm(BHEAD t);
02417                     t = tstop;
02418                 }
02419                 /* par = 1, in case the factor has more than SubTermsInSmall terms */
02420                 EndSort(BHEAD a+ARGHEAD,1);
02421                 a[*a] = oldword;
02422                 a += *a;
02423             }
02424             else { NEXTARG(a); }
02425         }
02426     }
02427 #ifdef NEWORDER
02428     t = argout; numargs = 0;
02429     while ( *t ) {
02430         tt = t;
02431         NEXTARG(t);
02432         if ( *tt == ABS(t[-1])+1+ARGHEAD && sign == -1 ) { t[-1] = -t[-1]; sign = 1; }
02433         else if ( *tt == -SNUMBER && sign == -1 ) { tt[1] = -tt[1]; sign = 1; }
02434         numargs++;
02435     }
02436     if ( sign == -1 ) {
02437         *t++ = -SNUMBER; *t++ = -1; *t = 0; sign = 1; numargs++;
02438     }
02439 #else
02440 /*
02441     Now we have to sort the arguments
02442     First have the number of 'nontrivial/nonnumerical' arguments
02443     Then make a piece of code like in FullSymmetrize with that number
02444     of arguments to be symmetrized.
02445     Put a function in front

```

```

02446     Call the Symmetrize routine
02447  */
02448     t = argout; numargs = 0;
02449     while ( *t && *t != -SNUMBER && ( *t < 0 || ( ABS(t[*t-1]) != *t-1 ) ) ) {
02450         NEXTARG(t);
02451         numargs++;
02452     }
02453 #endif
02454     if ( numargs > 1 ) {
02455         WORD *Lijst;
02456         WORD x[3];
02457         x[0] = argout[-FUNHEAD];
02458         x[1] = argout[-FUNHEAD+1];
02459         x[2] = argout[-FUNHEAD+2];
02460         while ( *t ) { NEXTARG(t); }
02461         argout[-FUNHEAD] = SQRTFUNCTION;
02462         argout[-FUNHEAD+1] = t-argout+FUNHEAD;
02463         argout[-FUNHEAD+2] = 0;
02464         AT.WorkPointer = t+1;
02465         Lijst = AT.WorkPointer;
02466         for ( i = 0; i < numargs; i++ ) Lijst[i] = i;
02467         AT.WorkPointer += numargs;
02468         error = Symmetrize(BHEAD argout-FUNHEAD,Lijst,numargs,1,SYMMETRIC);
02469         AT.WorkPointer = Lijst;
02470         argout[-FUNHEAD] = x[0];
02471         argout[-FUNHEAD+1] = x[1];
02472         argout[-FUNHEAD+2] = x[2];
02473 #ifdef NEWORDER
02474  /*
02475         Now we have to get a potential numerical argument to the first position
02476  */
02477         tstop = argout; while ( *tstop ) { NEXTARG(tstop); }
02478         t = argout; number = 0;
02479         while ( *t ) {
02480             tt = t; NEXTARG(tt);
02481             if ( *tt == -SNUMBER ) {
02482                 if ( number == 0 ) break;
02483                 x[0] = tt[1];
02484                 while ( tt > argout ) { ---t = ---tt; }
02485                 argout[0] = -SNUMBER; argout[1] = x[0];
02486                 break;
02487             }
02488             else if ( *tt == ABS(t[-1])+1+ARGHEAD ) {
02489                 if ( number == 0 ) break;
02490                 ii = t - tt;
02491                 for ( i = 0; i < ii; i++ ) tstop[i] = tt[i];
02492                 while ( tt > argout ) { ---t = ---tt; }
02493                 for ( i = 0; i < ii; i++ ) argout[i] = tstop[i];
02494                 *tstop = 0;
02495                 break;
02496             }
02497             number++;
02498         }
02499 #endif
02500     }
02501  /*
02502     #] step 8 :
02503  */
02504     return(error);
02505 }
02506
02507  /*
02508     #] ArgFactorize :
02509     #[ FindArg :
02510  */
02519 WORD FindArg(PHEAD WORD *a)
02520 {
02521     int number;
02522     if ( AN.ncmod != 0 ) return(0); /* no room for mod stuff */
02523     number = FindTree(AT.fbufnum,a);
02524     if ( number >= 0 ) return(number+1);
02525     number = FindTree(AC.ffbufnum,a);
02526     if ( number >= 0 ) return(-number-1);
02527     return(0);
02528 }
02529
02530  /*
02531     #] FindArg :
02532     #[ InsertArg :
02533  */
02543 int InsertArg(PHEAD WORD *argin, WORD *argout,int par)
02544 {
02545     CBUF *C;
02546     WORD *a, i, bufnum;
02547     if ( par == 0 ) {
02548         bufnum = AT.fbufnum;
02549         C = cbuf+bufnum;

```

```

02550         if ( C->numrhs >= (C->maxrhs-2) ) CleanupArgCache(BHEAD AT.fbufnum);
02551     }
02552     else if ( par == 1 ) {
02553         bufnum = AC.fbufnum;
02554         C = cbuf+bufnum;
02555     }
02556     else { return(-1); }
02557     AddRHS(bufnum,1);
02558     AddNtoC(bufnum,*argin,argin,1);
02559     AddToCB(C,0)
02560     a = argout; while ( *a ) NEXTARG(a);
02561     i = a - argout;
02562     AddNtoC(bufnum,i,argout,2);
02563     AddToCB(C,0)
02564     return(InsTree(bufnum,C->numrhs));
02565 }
02566
02567 /*
02568 #] InsertArg :
02569 #[ CleanupArgCache :
02570 */
02571 int CleanupArgCache(PHEAD WORD bufnum)
02572 {
02573     CBUF *C = cbuf + bufnum;
02574     COMPTREE *boomlijst = C->boomlijst;
02575     LONG *weights = (LONG *)Malloca(2*(C->numrhs+1)*sizeof(LONG),"CleanupArgCache");
02576     LONG w, whalf, *extraweights;
02577     WORD *a, *to, *from;
02578     int i,j,k;
02579     for ( i = 1; i <= C->numrhs; i++ ) {
02580         weights[i] = ((LONG)i) * (boomlijst[i].usage);
02581     }
02582     /*
02583     Now sort the weights and determine the halfway weight
02584     */
02585     extraweights = weights+C->numrhs+1;
02586     SortWeights(weights+1,extraweights,C->numrhs);
02587     whalf = weights[C->numrhs/2+1];
02588     /*
02589     We should drop everybody with a weight < whalf.
02590     */
02591     to = C->Buffer;
02592     k = 1;
02593     for ( i = 1; i <= C->numrhs; i++ ) {
02594         from = C->rhs[i]; w = ((LONG)i) * (boomlijst[i].usage);
02595         if ( w >= whalf ) {
02596             if ( i < C->numrhs-1 ) {
02597                 if ( to == from ) {
02598                     to = C->rhs[i+1];
02599                 }
02600                 else {
02601                     j = C->rhs[i+1] - from;
02602                     C->rhs[k] = to;
02603                     NCOPY(to,from,j)
02604                 }
02605             }
02606             else if ( to == from ) {
02607                 to += *to + 1; while ( *to ) NEXTARG(to); to++;
02608             }
02609             else {
02610                 a = from; a += *a+1; while ( *a ) NEXTARG(a); a++;
02611                 j = a - from;
02612                 C->rhs[k] = to;
02613                 NCOPY(to,from,j)
02614             }
02615             weights[k++] = boomlijst[i].usage;
02616         }
02617     }
02618     C->numrhs = --k;
02619     C->Pointer = to;
02620     /*
02621     Next we need to rebuild the tree.
02622     Note that this can probably be done much faster by using the
02623     remains of the old tree !!!!!!!!!!!!!!!
02624     */
02625     ClearTree(AT.fbufnum);
02626     for ( i = 1; i <= k; i++ ) {
02627         InsTree(AT.fbufnum,i);
02628         boomlijst[i].usage = weights[i];
02629     }
02630     /*
02631     And cleanup
02632     */
02633     M_free(weights,"CleanupArgCache");
02634     return(0);
02635 }
02636
02637
02638

```



```

02644 /*
02645 #] CleanupArgCache :
02646 #[ ArgSymbolMerge :
02647 */
02648
02649 int ArgSymbolMerge(WORD *t1, WORD *t2)
02650 {
02651     WORD *t1e = t1+t1[1];
02652     WORD *t2e = t2+t2[1];
02653     WORD *t1a = t1+2;
02654     WORD *t2a = t2+2;
02655     WORD *t3;
02656     while ( t1a < t1e && t2a < t2e ) {
02657         if ( *t1a < *t2a ) {
02658             if ( t1a[1] >= 0 ) {
02659                 t3 = t1a+2;
02660                 while ( t3 < t1e ) { t3[-2] = *t3; t3[-1] = t3[1]; t3 += 2; }
02661                 t1e -= 2;
02662             }
02663             else t1a += 2;
02664         }
02665         else if ( *t1a > *t2a ) {
02666             if ( t2a[1] >= 0 ) t2a += 2;
02667             else {
02668                 t3 = t1e;
02669                 while ( t3 > t1a ) { *t3 = t3[-2]; t3[1] = t3[-1]; t3 -= 2; }
02670                 *t1a++ = *t2a++;
02671                 *t1a++ = *t2a++;
02672                 t1e += 2;
02673             }
02674         }
02675         else {
02676             if ( t2a[1] < t1a[1] ) t1a[1] = t2a[1];
02677             t1a += 2; t2a += 2;
02678         }
02679     }
02680     while ( t2a < t2e ) {
02681         if ( t2a[1] < 0 ) {
02682             *t1a++ = *t2a++;
02683             *t1a++ = *t2a++;
02684         }
02685         else t2a += 2;
02686     }
02687     while ( t1a < t1e ) {
02688         if ( t1a[1] >= 0 ) {
02689             t3 = t1a+2;
02690             while ( t3 < t1e ) { t3[-2] = *t3; t3[-1] = t3[1]; t3 += 2; }
02691             t1e -= 2;
02692         }
02693         else t1a += 2;
02694     }
02695     t1[1] = t1a - t1;
02696     return(0);
02697 }
02698
02699 /*
02700 #] ArgSymbolMerge :
02701 #[ ArgDotproductMerge :
02702 */
02703
02704 int ArgDotproductMerge(WORD *t1, WORD *t2)
02705 {
02706     WORD *t1e = t1+t1[1];
02707     WORD *t2e = t2+t2[1];
02708     WORD *t1a = t1+2;
02709     WORD *t2a = t2+2;
02710     WORD *t3;
02711     while ( t1a < t1e && t2a < t2e ) {
02712         if ( *t1a < *t2a || ( *t1a == *t2a && t1a[1] < t2a[1] ) ) {
02713             if ( t1a[2] >= 0 ) {
02714                 t3 = t1a+3;
02715                 while ( t3 < t1e ) { t3[-3] = *t3; t3[-2] = t3[1]; t3[-1] = t3[2]; t3 += 3; }
02716                 t1e -= 3;
02717             }
02718             else t1a += 3;
02719         }
02720         else if ( *t1a > *t2a || ( *t1a == *t2a && t1a[1] > t2a[1] ) ) {
02721             if ( t2a[2] >= 0 ) t2a += 3;
02722             else {
02723                 t3 = t1e;
02724                 while ( t3 > t1a ) { *t3 = t3[-3]; t3[1] = t3[-2]; t3[2] = t3[-1]; t3 -= 3; }
02725                 *t1a++ = *t2a++;
02726                 *t1a++ = *t2a++;
02727                 *t1a++ = *t2a++;
02728                 t1e += 3;
02729             }
02730         }
02731     }

```

```

02731         else {
02732             if ( t2a[2] < t1a[2] ) t1a[2] = t2a[2];
02733             t1a += 3; t2a += 3;
02734         }
02735     }
02736     while ( t2a < t2e ) {
02737         if ( t2a[2] < 0 ) {
02738             *t1a++ = *t2a++;
02739             *t1a++ = *t2a++;
02740             *t1a++ = *t2a++;
02741         }
02742         else t2a += 3;
02743     }
02744     while ( t1a < t1e ) {
02745         if ( t1a[2] >= 0 ) {
02746             t3 = t1a+3;
02747             while ( t3 < t1e ) { t3[-3] = *t3; t3[-2] = t3[1]; t3[-1] = t3[2]; t3 += 3; }
02748             t1e -= 3;
02749         }
02750         else t1a += 2;
02751     }
02752     t1[1] = t1a - t1;
02753     return(0);
02754 }
02755
02756 /*
02757 #] ArgDotproductMerge :
02758 #[ TakeArgContent :
02759 */
02772 WORD *TakeArgContent(PHEAD WORD *argin, WORD *argout)
02773 {
02774     GETBIDENTITY
02775     WORD *t, *rnext, *r1, *r2, *r3, *r5, *r6, *r7, *r8, *r9;
02776     WORD pow, *mm, *mnext, *mstop, *argin2 = argin, *argin3 = argin, *argfree;
02777     WORD ncom;
02778     int j, i, act;
02779     r5 = t = argin + ARGHEAD;
02780     r3 = argin + *argin;
02781     rnext = t + *t;
02782     GETSTOP(t,r6);
02783     r1 = argout;
02784     t++;
02785 /*
02786     First pass: arrange everything but the symbols and dotproducts.
02787     They need separate treatment because we have to take out negative
02788     powers.
02789 */
02790     while ( t < r6 ) {
02791 /*
02792         #[ DELTA/VECTOR :
02793 */
02794         if ( *t == DELTA || *t == VECTOR ) {
02795             r7 = t; r8 = t + t[1]; t += 2;
02796             while ( t < r8 ) {
02797                 mm = rnext;
02798                 pow = 1;
02799                 while ( mm < r3 ) {
02800                     mnext = mm + *mm;
02801                     GETSTOP(mm,mstop); mm++;
02802                     while ( mm < mstop ) {
02803                         if ( *mm != *r7 ) mm += mm[1];
02804                         else break;
02805                     }
02806                     if ( *mm == *r7 ) {
02807                         mstop = mm + mm[1]; mm += 2;
02808                         while ( ( *mm != *t || mm[1] != t[1] )
02809                             && mm < mstop ) mm += 2;
02810                         if ( mm >= mstop ) pow = 0;
02811                     }
02812                     else pow = 0;
02813                     if ( pow == 0 ) break;
02814                     mm = mnext;
02815                 }
02816                 if ( pow == 0 ) { t += 2; continue; }
02817 /*
02818                 We have a factor
02819 */
02820                 *r1++ = 8 + ARGHEAD;
02821                 for ( j = 1; j < ARGHEAD; j++ ) *r1++ = 0;
02822                 *r1++ = 8; *r1++ = *r7;
02823                 *r1++ = 4; *r1++ = *t; *r1++ = t[1];
02824                 *r1++ = 1; *r1++ = 1; *r1++ = 3;
02825                 argout = r1;
02826 /*
02827                 Now we have to remove the delta's/vectors
02828 */
02829                 mm = rnext;

```

```

02830         while ( mm < r3 ) {
02831             mnext = mm + *mm;
02832             GETSTOP(mm,mstop); mm++;
02833             while ( mm < mstop ) {
02834                 if ( *mm != *r7 ) mm += mm[1];
02835                 else break;
02836             }
02837             mstop = mm + mm[1]; mm += 2;
02838             while ( mm < mstop && (
02839                 *mm != *t || mm[1] != t[1] ) ) mm += 2;
02840             *mm = mm[1] = NOINDEX;
02841             mm = mnext;
02842         }
02843         *t = t[1] = NOINDEX;
02844         t += 2;
02845     }
02846 }
02847 /*
02848     #] DELTA/VECTOR :
02849     #[ INDEX :
02850 */
02851 else if ( *t == INDEX ) {
02852     r7 = t; r8 = t + t[1]; t += 2;
02853     while ( t < r8 ) {
02854         mm = rnext;
02855         pow = 1;
02856         while ( mm < r3 ) {
02857             mnext = mm + *mm;
02858             GETSTOP(mm,mstop); mm++;
02859             while ( mm < mstop ) {
02860                 if ( *mm != *r7 ) mm += mm[1];
02861                 else break;
02862             }
02863             if ( *mm == *r7 ) {
02864                 mstop = mm + mm[1]; mm += 2;
02865                 while ( *mm != *t
02866                     && mm < mstop ) mm++;
02867                 if ( mm >= mstop ) pow = 0;
02868             }
02869             else pow = 0;
02870             if ( pow == 0 ) break;
02871             mm = mnext;
02872         }
02873         if ( pow == 0 ) { t++; continue; }
02874     }
02875     /*
02876     We have a factor
02877     */
02878     if ( *t < 0 ) { *r1++ = -VECTOR; }
02879     else { *r1++ = -INDEX; }
02880     *r1++ = *t;
02881     argout = r1;
02882     /*
02883     Now we have to remove the index
02884     */
02885     *t = NOINDEX;
02886     mm = rnext;
02887     while ( mm < r3 ) {
02888         mnext = mm + *mm;
02889         GETSTOP(mm,mstop); mm++;
02890         while ( mm < mstop ) {
02891             if ( *mm != *r7 ) mm += mm[1];
02892             else break;
02893         }
02894         mstop = mm + mm[1]; mm += 2;
02895         while ( mm < mstop &&
02896             *mm != *t ) mm += 1;
02897         *mm = NOINDEX;
02898         mm = mnext;
02899     }
02900     t += 1;
02901 }
02902 /*
02903     #] INDEX :
02904     #[ FUNCTION :
02905 */
02906 else if ( *t >= FUNCTION ) {
02907     /*
02908     In the next code we should actually look inside
02909     the DENOMINATOR or EXPONENT for noncommuting objects
02910     */
02911     if ( *t >= FUNCTION &&
02912         functions[*t-FUNCTION].commute == 0 ) ncom = 0;
02913     else ncom = 1;
02914     if ( ncom ) {
02915         mm = r5 + 1;
02916         while ( mm < t && ( *mm == DUMMYFUN

```

```

02917         || *mm == DUMMYTEN ) ) mm += mm[1];
02918         if ( mm < t ) { t += t[1]; continue; }
02919     }
02920     mm = rnext; pow = 1;
02921     while ( mm < r3 ) {
02922         mnext = mm + *mm;
02923         GETSTOP(mm,mstop); mm++;
02924         while ( mm < mstop ) {
02925             if ( *mm == *t && mm[1] == t[1] ) {
02926                 for ( i = 2; i < t[1]; i++ ) {
02927                     if ( mm[i] != t[i] ) break;
02928                 }
02929                 if ( i >= t[1] )
02930                     { mm += mm[1]; goto nexttterm; }
02931             }
02932             if ( ncom && *mm != DUMMYFUN && *mm != DUMMYTEN )
02933                 { pow = 0; break; }
02934             mm += mm[1];
02935         }
02936         if ( mm >= mstop ) pow = 0;
02937         if ( pow == 0 ) break;
02938     nexttterm:    mm = mnext;
02939     }
02940     if ( pow == 0 ) { t += t[1]; continue; }
02941 /*
02942     Copy the function
02943 */
02944     *r1++ = t[1] + 4 + ARGHEAD;
02945     for ( i = 1; i < ARGHEAD; i++ ) *r1++ = 0;
02946     *r1++ = t[1] + 4;
02947     for ( i = 0; i < t[1]; i++ ) *r1++ = t[i];
02948     *r1++ = 1; *r1++ = 1; *r1++ = 3;
02949     argout = r1;
02950 /*
02951     Now we have to take out the functions
02952 */
02953     mm = rnext;
02954     while ( mm < r3 ) {
02955         mnext = mm + *mm;
02956         GETSTOP(mm,mstop); mm++;
02957         while ( mm < mstop ) {
02958             if ( *mm == *t && mm[1] == t[1] ) {
02959                 for ( i = 2; i < t[1]; i++ ) {
02960                     if ( mm[i] != t[i] ) break;
02961                 }
02962                 if ( i >= t[1] ) {
02963                     if ( functions[*t-FUNCTION].spec > 0 )
02964                         *mm = DUMMYTEN;
02965                     else
02966                         *mm = DUMMYFUN;
02967                     mm += mm[1];
02968                     goto nexttterm;
02969                 }
02970             }
02971             mm += mm[1];
02972         }
02973     nexttterm:    mm = mnext;
02974     }
02975     if ( functions[*t-FUNCTION].spec > 0 )
02976         *t = DUMMYTEN;
02977     else
02978         *t = DUMMYFUN;
02979     t += t[1];
02980 }
02981 /*
02982     #] FUNCTION :
02983 */
02984     else {
02985         t += t[1];
02986     }
02987 }
02988 /*
02989     #[ SYMBOL :
02990
02991     Now collect all symbols. We can use the space after r1 as storage
02992 */
02993     t = argin+ARGHEAD;
02994     rnext = t + *t;
02995     r2 = r1;
02996     while ( t < r3 ) {
02997         GETSTOP(t,r6);
02998         t++;
02999         act = 0;
03000         while ( t < r6 ) {
03001             if ( *t == SYMBOL ) {
03002                 act = 1;
03003                 i = t[1];

```

```

03004         NCOPY(r2,t,i)
03005     }
03006     else { t += t[1]; }
03007 }
03008 if ( act == 0 ) {
03009     *r2++ = SYMBOL; *r2++ = 2;
03010 }
03011 t = rnext; rnext = rnext + *rnext;
03012 }
03013 *r2 = 0;
03014 argin2 = argin;
03015 /*
03016     Now we have a list of all symbols as a sequence of SYMBOL subterms.
03017     Any symbol that is absent in a subterm has power zero.
03018     We now need a list of all minimum powers.
03019     This can be done by subsequent merges.
03020 */
03021 r7 = r1;          /* The first object into which we merge. */
03022 r8 = r7 + r7[1]; /* The object that gets merged into r7. */
03023 while ( *r8 ) {
03024     r2 = r8 + r8[1]; /* Next object */
03025     ArgSymbolMerge(r7,r8);
03026     r8 = r2;
03027 }
03028 /*
03029     Now we have to divide by the object in r7 and take it apart as factors.
03030     The division can be simple if there are no negative powers.
03031 */
03032 if ( r7[1] > 2 ) {
03033     r8 = r7+2;
03034     r2 = r7 + r7[1];
03035     act = 0;
03036     pow = 0;
03037     while ( r8 < r2 ) {
03038         if ( r8[1] < 0 ) { act = 1; pow += -r8[1]*(ARGHEAD+8); }
03039         else { pow += 2*r8[1]; }
03040         r8 += 2;
03041     }
03042 /*
03043     The amount of space we need to move r7 is given in pow
03044 */
03045     if ( act == 0 ) { /* this can be done 'in situ' */
03046         t = argin + ARGHEAD;
03047         while ( t < r3 ) {
03048             rnext = t + *t;
03049             GETSTOP(t,r6);
03050             t++;
03051             while ( t < r6 ) {
03052                 if ( *t != SYMBOL ) { t += t[1]; continue; }
03053                 r8 = r7+2; r9 = t + t[1]; t += 2;
03054                 while ( ( t < r9 ) && ( r8 < r2 ) ) {
03055                     if ( *t == *r8 ) {
03056                         t[1] -= r8[1]; t += 2; r8 += 2;
03057                     }
03058                     else { /* *t must be < than *r8 !!! */
03059                         t += 2;
03060                     }
03061                 }
03062                 t = r9;
03063             }
03064             t = rnext;
03065         }
03066 /*
03067         And now the factors that go to argout.
03068         First we have to move r7 out of the way.
03069 */
03070         r8 = r7+pow; i = r7[1];
03071         while ( --i >= 0 ) r8[i] = r7[i];
03072         r2 += pow;
03073         r8 += 2;
03074         while ( r8 < r2 ) {
03075             for ( i = 0; i < r8[1]; i++ ) { *r1++ = -SYMBOL; *r1++ = *r8; }
03076             r8 += 2;
03077         }
03078     }
03079     else { /* this needs a new location */
03080         argin2 = TermMalloc("TakeArgContent2");
03081 /*
03082         We have to multiply the inverse of r7 into argin
03083         The answer should go to argin2.
03084 */
03085         r5 = argin2; *r5++ = 0; *r5++ = 0; FILLARG(r5);
03086         t = argin+ARGHEAD;
03087         while ( t < r3 ) {
03088             rnext = t + *t;
03089             GETSTOP(t,r6);
03090             r9 = r5;

```

```

03091         *r5++ = *t++ + r7[1];
03092         while ( t < r6 ) *r5++ = *t++;
03093         i = r7[1] - 2; r8 = r7+2;
03094         *r5++ = r7[0]; *r5++ = r7[1];
03095         while ( i > 0 ) { *r5++ = *r8++; *r5++ = -*r8++; i -= 2; }
03096         while ( t < rnext ) *r5++ = *t++;
03097         Normalize(BHEAD r9);
03098         r5 = r9 + *r9;
03099     }
03100     *r5 = 0;
03101     *argin2 = r5-argin2;
03102 /*
03103     We may have to sort the terms in argin2.
03104 */
03105     NewSort(BHEAD0);
03106     t = argin2+ARGHEAD;
03107     while ( *t ) {
03108         StoreTerm(BHEAD t);
03109         t += *t;
03110     }
03111     t = argin2+ARGHEAD;
03112     if ( EndSort(BHEAD t,0) < 0 ) goto Irreg;
03113     while ( *t ) t += *t;
03114     *argin2 = t - argin2;
03115     r3 = t;
03116 /*
03117     And now the factors that go to argout.
03118     First we have to move r7 out of the way.
03119 */
03120     r8 = r7+pow; i = r7[1];
03121     while ( --i >= 0 ) r8[i] = r7[i];
03122     r2 += pow;
03123     r8 += 2;
03124     while ( r8 < r2 ) {
03125         if ( r8[1] >= 0 ) {
03126             for ( i = 0; i < r8[1]; i++ ) { *r1++ = -SYMBOL; *r1++ = *r8; }
03127         }
03128         else {
03129             for ( i = 0; i < -r8[1]; i++ ) {
03130                 *r1++ = ARGHEAD+8; *r1++ = 0;
03131                 FILLARG(r1);
03132                 *r1++ = 8; *r1++ = SYMBOL; *r1++ = 4; *r1++ = *r8;
03133                 *r1++ = -1; *r1++ = 1; *r1++ = 1; *r1++ = 3;
03134             }
03135             r8 += 2;
03136         }
03137     }
03138     argout = r1;
03139 }
03140 }
03141 /*
03142     #] SYMBOL :
03143     #[ DOTPRODUCT :
03144
03145     Now collect all dotproducts. We can use the space after r1 as storage
03146 */
03147     t = argin2+ARGHEAD;
03148     rnext = t + *t;
03149     r2 = r1;
03150     while ( t < r3 ) {
03151         GETSTOP(t,r6);
03152         t++;
03153         act = 0;
03154         while ( t < r6 ) {
03155             if ( *t == DOTPRODUCT ) {
03156                 act = 1;
03157                 i = t[1];
03158                 NCOPY(r2,t,i)
03159             }
03160             else { t += t[1]; }
03161         }
03162         if ( act == 0 ) {
03163             *r2++ = DOTPRODUCT; *r2++ = 2;
03164         }
03165         t = rnext; rnext = rnext + *rnext;
03166     }
03167     *r2 = 0;
03168     argin3 = argin2;
03169 /*
03170     Now we have a list of all dotproducts as a sequence of DOTPRODUCT
03171     subterms. Any dotproduct that is absent in a subterm has power zero.
03172     We now need a list of all minimum powers.
03173     This can be done by subsequent merges.
03174 */
03175     r7 = r1; /* The first object into which we merge. */
03176     r8 = r7 + r7[1]; /* The object that gets merged into r7. */
03177     while ( *r8 ) {

```

```

03178         r2 = r8 + r8[1]; /* Next object */
03179         ArgDotproductMerge(r7,r8);
03180         r8 = r2;
03181     }
03182     /*
03183     Now we have to divide by the object in r7 and take it apart as factors.
03184     The division can be simple if there are no negative powers.
03185     */
03186     if ( r7[1] > 2 ) {
03187         r8 = r7+2;
03188         r2 = r7 + r7[1];
03189         act = 0;
03190         pow = 0;
03191         while ( r8 < r2 ) {
03192             if ( r8[2] < 0 ) { pow += -r8[2]*(ARGHEAD+9); }
03193             else { pow += r8[2]*(ARGHEAD+9); }
03194             r8 += 3;
03195         }
03196         /*
03197         The amount of space we need to move r7 is given in pow
03198         For dotproducts we always need a new location
03199         */
03200         {
03201             argin3 = TermMalloc("TakeArgContent3");
03202             /*
03203             We have to multiply the inverse of r7 into argin
03204             The answer should go to argin2.
03205             */
03206             r5 = argin3; *r5++ = 0; *r5++ = 0; FILLARG(r5);
03207             t = argin2+ARGHEAD;
03208             while ( t < r3 ) {
03209                 rnext = t + *t;
03210                 GETSTOP(t,r6);
03211                 r9 = r5;
03212                 *r5++ = *t++ + r7[1];
03213                 while ( t < r6 ) *r5++ = *t++;
03214                 i = r7[1] - 2; r8 = r7+2;
03215                 *r5++ = r7[0]; *r5++ = r7[1];
03216                 while ( i > 0 ) { *r5++ = *r8++; *r5++ = *r8++; *r5++ = -*r8++; i -= 3; }
03217                 while ( t < rnext ) *r5++ = *t++;
03218                 Normalize(BHEAD r9);
03219                 r5 = r9 + *r9;
03220             }
03221             *r5 = 0;
03222             *argin3 = r5-argin3;
03223             /*
03224             We may have to sort the terms in argin3.
03225             */
03226             NewSort(BHEAD0);
03227             t = argin3+ARGHEAD;
03228             while ( *t ) {
03229                 StoreTerm(BHEAD t);
03230                 t += *t;
03231             }
03232             t = argin3+ARGHEAD;
03233             if ( EndSort(BHEAD t,0) < 0 ) goto Irreg;
03234             while ( *t ) t += *t;
03235             *argin3 = t - argin3;
03236             r3 = t;
03237             /*
03238             And now the factors that go to argout.
03239             First we have to move r7 out of the way.
03240             */
03241             r8 = r7+pow; i = r7[1];
03242             while ( --i >= 0 ) r8[i] = r7[i];
03243             r2 += pow;
03244             r8 += 2;
03245             while ( r8 < r2 ) {
03246                 for ( i = ABS(r8[2]); i > 0; i-- ) {
03247                     *r1++ = ARGHEAD+9; *r1++ = 0; FILLARG(r1);
03248                     *r1++ = 9; *r1++ = DOTPRODUCT; *r1++ = 5; *r1++ = *r8;
03249                     *r1++ = r8[1]; *r1++ = r8[2] < 0 ? -1: 1;
03250                     *r1++ = 1; *r1++ = 1; *r1++ = 3;
03251                 }
03252                 r8 += 3;
03253             }
03254             argout = r1;
03255         }
03256     }
03257     /*
03258     #] DOTPRODUCT :
03259
03260     We have now in argin3 the argument stripped of negative powers and
03261     common factors. The only thing left to deal with is to make the
03262     coefficients integer. For that we have to find the LCM of the denominators
03263     and the GCD of the numerators. And to start with, the sign.
03264     We force the sign of the first term to be positive.

```

```

03265 */
03266     t = argin3 + ARGHEAD; pow = 1;
03267     t += *t;
03268     if ( t[-1] < 0 ) {
03269         pow = 0;
03270         t[-1] = -t[-1];
03271         while ( t < r3 ) {
03272             t += *t; t[-1] = -t[-1];
03273         }
03274     }
03275 /*
03276     Now the GCD of the numerators and the LCM of the denominators:
03277 */
03278     argfree = TermMalloc("TakeArgContent1");
03279     if ( AN.cmod != 0 ) {
03280         r1 = MakeMod(BHEAD argin3,r1,argfree);
03281     }
03282     else {
03283         r1 = MakeInteger(BHEAD argin3,r1,argfree);
03284     }
03285     if ( pow == 0 ) {
03286         *r1++ = -SNUMBER;
03287         *r1++ = -1;
03288     }
03289     *r1 = 0;
03290 /*
03291     Cleanup
03292 */
03293     if ( argin3 != argin2 ) TermFree(argin3,"TakeArgContent3");
03294     if ( argin2 != argin ) TermFree(argin2,"TakeArgContent2");
03295     return(argfree);
03296 Irreg:
03297     MesPrint("Irregularity while sorting argument in TakeArgContent");
03298     if ( argin3 != argin2 ) TermFree(argin3,"TakeArgContent3");
03299     if ( argin2 != argin ) TermFree(argin2,"TakeArgContent2");
03300     Terminate(-1);
03301     return(0);
03302 }
03303
03304 /*
03305     #] TakeArgContent :
03306     #[ MakeInteger :
03307 */
03318 WORD *MakeInteger(PHEAD WORD *argin,WORD *argout,WORD *argfree)
03319 {
03320     UWORD *GCDbuffer, *GCDbuffer2, *LCMbuffer, *LCMb, *LCMc;
03321     WORD *r, *r1, *r2, *r3, *r4, *r5, *rnext, i, k, j;
03322     WORD kGCD, kLCM, kGCD2, kkLCM, jLCM, jGCD;
03323     GCDbuffer = NumberMalloc("MakeInteger");
03324     GCDbuffer2 = NumberMalloc("MakeInteger");
03325     LCMbuffer = NumberMalloc("MakeInteger");
03326     LCMb = NumberMalloc("MakeInteger");
03327     LCMc = NumberMalloc("MakeInteger");
03328     r4 = argin + *argin;
03329     r = argin + ARGHEAD;
03330 /*
03331     First take the first term to load up the LCM and the GCD
03332 */
03333     r2 = r + *r;
03334     j = r2[-1];
03335     r3 = r2 - ABS(j);
03336     k = REDLENG(j);
03337     if ( k < 0 ) k = -k;
03338     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
03339     for ( kGCD = 0; kGCD < k; kGCD++ ) GCDbuffer[kGCD] = r3[kGCD];
03340     k = REDLENG(j);
03341     if ( k < 0 ) k = -k;
03342     r3 += k;
03343     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
03344     for ( kLCM = 0; kLCM < k; kLCM++ ) LCMbuffer[kLCM] = r3[kLCM];
03345     r1 = r2;
03346 /*
03347     Now go through the rest of the terms in this argument.
03348 */
03349     while ( r1 < r4 ) {
03350         r2 = r1 + *r1;
03351         j = r2[-1];
03352         r3 = r2 - ABS(j);
03353         k = REDLENG(j);
03354         if ( k < 0 ) k = -k;
03355         while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
03356         if ( ( ( GCDbuffer[0] == 1 ) && ( kGCD == 1 ) ) ) {
03357 /*
03358             GCD is already 1
03359 */
03360         }
03361         else if ( ( ( k != 1 ) || ( r3[0] != 1 ) ) ) {

```



```

03362         if ( GcdLong(BHEAD GCDbuffer,kGCD, (UWORD *)r3,k,GCDbuffer2,&kGCD2) ) {
03363             NumberFree(GCDbuffer,"MakeInteger");
03364             NumberFree(GCDbuffer2,"MakeInteger");
03365             NumberFree(LCMbuffer,"MakeInteger");
03366             NumberFree(LCMb,"MakeInteger"); NumberFree(LCMc,"MakeInteger");
03367             goto MakeIntegerErr;
03368         }
03369         kGCD = kGCD2;
03370         for ( i = 0; i < kGCD; i++ ) GCDbuffer[i] = GCDbuffer2[i];
03371     }
03372     else {
03373         kGCD = 1; GCDbuffer[0] = 1;
03374     }
03375     k = REDLENG(j);
03376     if ( k < 0 ) k = -k;
03377     r3 += k;
03378     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
03379     if ( ( ( LCMbuffer[0] == 1 ) && ( kLCM == 1 ) ) ) {
03380         for ( kLCM = 0; kLCM < k; kLCM++ )
03381             LCMbuffer[kLCM] = r3[kLCM];
03382     }
03383     else if ( ( k != 1 ) || ( r3[0] != 1 ) ) {
03384         if ( GcdLong(BHEAD LCMbuffer,kLCM, (UWORD *)r3,k,LCMb,&kLCM) ) {
03385             NumberFree(GCDbuffer,"MakeInteger"); NumberFree(GCDbuffer2,"MakeInteger");
03386             NumberFree(LCMbuffer,"MakeInteger"); NumberFree(LCMb,"MakeInteger");
03387             NumberFree(LCMc,"MakeInteger");
03388             goto MakeIntegerErr;
03389         }
03390         DivLong((UWORD *)r3,k,LCMb,kLCM,LCMb,&kLCM,LCMc,&jLCM);
03391         MullLong(LCMbuffer,kLCM,LCMb,kLCM,LCMc,&jLCM);
03392         for ( kLCM = 0; kLCM < jLCM; kLCM++ )
03393             LCMbuffer[kLCM] = LCMc[kLCM];
03394     }
03395     else {} /* LCM doesn't change */
03396     r1 = r2;
03397 }
03398 /*
03399 Now put the factor together: GCD/LCM
03400 */
03401 r3 = (WORD *) (GCDbuffer);
03402 if ( kGCD == kLCM ) {
03403     for ( jGCD = 0; jGCD < kGCD; jGCD++ )
03404         r3[jGCD+kGCD] = LCMbuffer[jGCD];
03405     k = kGCD;
03406 }
03407 else if ( kGCD > kLCM ) {
03408     for ( jGCD = 0; jGCD < kLCM; jGCD++ )
03409         r3[jGCD+kGCD] = LCMbuffer[jGCD];
03410     for ( jGCD = kLCM; jGCD < kGCD; jGCD++ )
03411         r3[jGCD+kGCD] = 0;
03412     k = kGCD;
03413 }
03414 else {
03415     for ( jGCD = kGCD; jGCD < kLCM; jGCD++ )
03416         r3[jGCD] = 0;
03417     for ( jGCD = 0; jGCD < kLCM; jGCD++ )
03418         r3[jGCD+kLCM] = LCMbuffer[jGCD];
03419     k = kLCM;
03420 }
03421 j = 2*k+1;
03422 /*
03423 Now we have to write this to argout
03424 */
03425 if ( ( j == 3 ) && ( r3[1] == 1 ) && ( (WORD)(r3[0]) > 0 ) ) {
03426     *argout = -SNUMBER;
03427     argout[1] = r3[0];
03428     r1 = argout+2;
03429 }
03430 else {
03431     r1 = argout;
03432     *r1++ = j+1+ARGHEAD; *r1++ = 0; FILLARG(r1);
03433     *r1++ = j+1; r2 = r3;
03434     for ( i = 0; i < k; i++ ) { *r1++ = *r2++; *r1++ = *r2++; }
03435     *r1++ = j;
03436 }
03437 /*
03438 Next we have to take the factor out from the argument.
03439 This cannot be done in location, because the denominator stuff can make
03440 coefficients longer.
03441 */
03442 r2 = argfree + 2; FILLARG(r2)
03443 while ( r < r4 ) {
03444     rnext = r + *r;
03445     j = ABS(rnext[-1]);
03446     r5 = rnext - j;
03447     r3 = r2;
03448     while ( r < r5 ) *r2++ = *r++;

```

```

03448         j = (j-1)/2;      /* reduced length. Remember, k is the other red length */
03449         if ( DivRat(BHEAD (UWORD *)r5,j,GCDbuffer,k,(UWORD *)r2,&i) ) {
03450             goto MakeIntegerErr;
03451         }
03452         i = 2*i+1;
03453         r2 = r2 + i;
03454         if ( rnext[-1] < 0 ) r2[-1] = -i;
03455         else                 r2[-1] = i;
03456         *r3 = r2-r3;
03457         r = rnext;
03458     }
03459     *r2 = 0;
03460     argfree[0] = r2-argfree;
03461     argfree[1] = 0;
03462 /*
03463 Cleanup
03464 */
03465     NumberFree(LCMc,"MakeInteger");
03466     NumberFree(LCmb,"MakeInteger");
03467     NumberFree(LCmbuffer,"MakeInteger");
03468     NumberFree(GCDbuffer2,"MakeInteger");
03469     NumberFree(GCDbuffer,"MakeInteger");
03470     return(r1);
03471
03472 MakeIntegerErr:
03473     MesCall("MakeInteger");
03474     Terminate(-1);
03475     return(0);
03476 }
03477
03478 /*
03479 #] MakeInteger :
03480 #[ MakeMod :
03481 */
03489 WORD *MakeMod(PHEAD WORD *argin,WORD *argout,WORD *argfree)
03490 {
03491     WORD *r, *instop, *r1, *m, x, xx, ix, ip;
03492     int i;
03493     r = argin; instop = r + *r; r += ARGHEAD;
03494     x = r[*r-3];
03495     if ( r[*r-1] < 0 ) x += AN.cmod[0];
03496     if ( GetModInverses(x,(WORD)(AN.cmod[0]),&ix,&ip) ) {
03497         Terminate(-1);
03498     }
03499     argout[0] = -SNUMBER;
03500     argout[1] = x;
03501     argout[2] = 0;
03502     r1 = argout+2;
03503 /*
03504 Now we have to multiply all coefficients by ix.
03505 This does not make things longer, but we should keep to the conventions
03506 of MakeInteger.
03507 */
03508     m = argfree + ARGHEAD;
03509     while ( r < instop ) {
03510         xx = r[*r-3]; if ( r[*r-1] < 0 ) xx += AN.cmod[0];
03511         xx = (WORD) (((LONG)xx)*ix) % AN.cmod[0];
03512         if ( xx != 0 ) {
03513             i = *r; NCOPY(m,r,i);
03514             m[-3] = xx; m[-1] = 3;
03515         }
03516         else { r += *r; }
03517     }
03518     *m = 0;
03519     *argfree = m - argfree;
03520     argfree[1] = 0;
03521     argfree += 2; FILLARG(argfree);
03522     return(r1);
03523 }
03524
03525 /*
03526 #] MakeMod :
03527 #[ SortWeights :
03528 */
03534 void SortWeights(LONG *weights,LONG *extraspace,WORD number)
03535 {
03536     LONG w, *fill, *from1, *from2;
03537     int n1,n2,i;
03538     if ( number >= 4 ) {
03539         n1 = number/2; n2 = number - n1;
03540         SortWeights(weights,extraspace,n1);
03541         SortWeights(weights+n1,extraspace,n2);
03542 /*
03543 We copy the first patch to the extra space. Then we merge
03544 Note that a potential remaining n2 objects are already in place.
03545 */
03546         for ( i = 0; i < n1; i++ ) extraspace[i] = weights[i];

```

```

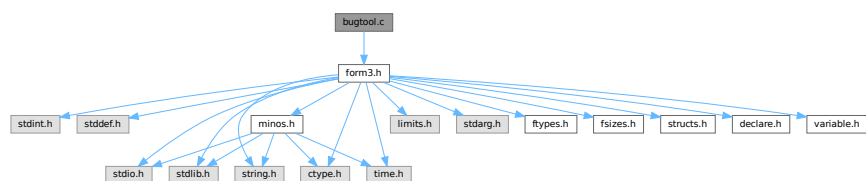
03547         fill = weights; from1 = extraspace; from2 = weights+n1;
03548         while ( n1 > 0 && n2 > 0 ) {
03549             if ( *from1 <= *from2 ) { *fill++ = *from1++; n1--; }
03550             else { *fill++ = *from2++; n2--; }
03551         }
03552         while ( n1 > 0 ) { *fill++ = *from1++; n1--; }
03553     }
03554     /*
03555     Special cases
03556     */
03557     else if ( number == 3 ) { /* 6 permutations of which one is trivial */
03558         if ( weights[0] > weights[1] ) {
03559             if ( weights[1] > weights[2] ) {
03560                 w = weights[0]; weights[0] = weights[2]; weights[2] = w;
03561             }
03562             else if ( weights[0] > weights[2] ) {
03563                 w = weights[0]; weights[0] = weights[1];
03564                 weights[1] = weights[2]; weights[2] = w;
03565             }
03566             else {
03567                 w = weights[0]; weights[0] = weights[1]; weights[1] = w;
03568             }
03569         }
03570         else if ( weights[0] > weights[2] ) {
03571             w = weights[0]; weights[0] = weights[2];
03572             weights[2] = weights[1]; weights[1] = w;
03573         }
03574         else if ( weights[1] > weights[2] ) {
03575             w = weights[1]; weights[1] = weights[2]; weights[2] = w;
03576         }
03577     }
03578     else if ( number == 2 ) {
03579         if ( weights[0] > weights[1] ) {
03580             w = weights[0]; weights[0] = weights[1]; weights[1] = w;
03581         }
03582     }
03583     return;
03584 }
03585
03586 /*
03587 #] SortWeights :
03588 */

```

4.3 bugtool.c File Reference

#include "form3.h"

Include dependency graph for bugtool.c:



Functions

- void [ExprStatus](#) (EXPRESSIONS e)

4.3.1 Detailed Description

Low level routines for debugging

Definition in file [bugtool.c](#).

4.3.2 Function Documentation

4.3.2.1 ExprStatus()

```
void ExprStatus (
    EXPRESSIONS e )
```

Definition at line 66 of file [bugtool.c](#).

4.4 bugtool.c

[Go to the documentation of this file.](#)

```
00001
00006 /* #[ License : */
00007 /*
00008 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009 *   When using this file you are requested to refer to the publication
00010 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011 *   This is considered a matter of courtesy as the development was paid
00012 *   for by FOM the Dutch physics granting agency and we would like to
00013 *   be able to track its scientific use to convince FOM of its value
00014 *   for the community.
00015 *
00016 *   This file is part of FORM.
00017 *
00018 *   FORM is free software: you can redistribute it and/or modify it under the
00019 *   terms of the GNU General Public License as published by the Free Software
00020 *   Foundation, either version 3 of the License, or (at your option) any later
00021 *   version.
00022 *
00023 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00026 *   details.
00027 *
00028 *   You should have received a copy of the GNU General Public License along
00029 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00030 */
00031 /* #[ License : */
00032
00033 /*
00034 *   #[ Includes :
00035 */
00036
00037 #include "form3.h"
00038
00039 /*
00040 *   #[ Includes :
00041 *   #[ ExprStatus :
00042 */
00043
00044 static UBYTE *statusexpr[] = {
00045     (UBYTE *) "LOCALEXPRESSION"
00046     , (UBYTE *) "SKIPLEXPRESSION"
00047     , (UBYTE *) "DROPLEXPRESSION"
00048     , (UBYTE *) "DROPPEDEXPRESSION"
00049     , (UBYTE *) "GLOBALEXPRESSION"
00050     , (UBYTE *) "SKIPGEXPRESSION"
00051     , (UBYTE *) "DROPGEXPRESSION"
00052     , (UBYTE *) "UNKNOWN"
00053     , (UBYTE *) "STOREDEXPRESSION"
00054     , (UBYTE *) "HIDDENLEXPRESSION"
00055     , (UBYTE *) "HIDELEXPRESSION"
00056     , (UBYTE *) "DROPHLEXPRESSION"
00057     , (UBYTE *) "UNHIDELEXPRESSION"
00058     , (UBYTE *) "HIDDENGEXPRESSION"
00059     , (UBYTE *) "HIDEGEXPRESSION"
00060     , (UBYTE *) "DROPHGEXPRESSION"
00061     , (UBYTE *) "UNHIDEGEXPRESSION"
00062     , (UBYTE *) "INTOHIDELEXPRESSION"
00063     , (UBYTE *) "INTOHIDEGEXPRESSION"
00064 };
00065
00066 void ExprStatus(EXPRESSIONS e)
00067 {
00068     MesPrint("Expression %s(%d) has status %s(%d,%d). Buffer: %d, Position: %15p",
```

```

00069      AC.exprnames->namebuffer+e->name, (WORD) (e->Expressions),
00070      statusexpr[e->status], e->status, e->hidelevel,
00071      e->whichbuffer, &(e->onfile));
00072  }
00073
00074  /*
00075   #] ExprStatus :
00076   */

```

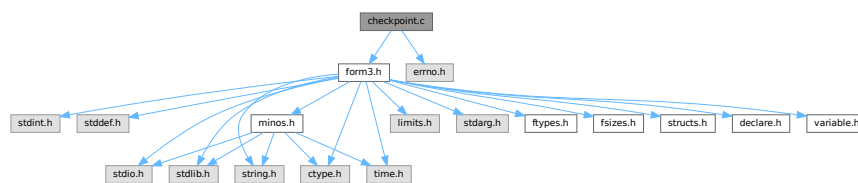
4.5 checkpoint.c File Reference

```

#include "form3.h"
#include <errno.h>

```

Include dependency graph for checkpoint.c:



Macros

- #define [CACHED_SNAPSHOT](#)
- #define [CACHE_SIZE](#) 4096
- #define [R_FREE](#)(ARG) if (ARG) M_free(ARG, #ARG);
- #define [R_FREE_NAMETREE](#)(ARG)
- #define [R_FREE_STREAM](#)(ARG)
- #define [R_SET](#)(VAR, TYPE) VAR = *((TYPE*)p); p = (unsigned char*)p + sizeof(TYPE);
- #define [R_COPY_B](#)(VAR, SIZE, CAST)
- #define [S_WRITE_B](#)(BUF, LEN) if (fwrite_cached(BUF, 1, LEN, fd) != (size_t)(LEN)) return(__LINE__);
- #define [S_FLUSH_B](#) if (flush_cache(fd) != 1) return(__LINE__);
- #define [R_COPY_S](#)(VAR, CAST)
- #define [S_WRITE_S](#)(STR)
- #define [R_COPY_LIST](#)(ARG)
- #define [S_WRITE_LIST](#)(LST)
- #define [R_COPY_NAMETREE](#)(ARG)
- #define [S_WRITE_NAMETREE](#)(ARG)
- #define [S_WRITE_DOLLAR](#)(ARG)
- #define [ANNOUNCE](#)(str)

Functions

- int [CheckRecoveryFile](#) (void)
- void [DeleteRecoveryFile](#) (void)
- char * [RecoveryFilename](#) (void)
- void [InitRecovery](#) (void)
- size_t [fwrite_cached](#) (const void *ptr, size_t size, size_t nmemb, FILE *fd)
- size_t [flush_cache](#) (FILE *fd)
- int [DoRecovery](#) (int *moduletype)
- void [DoCheckpoint](#) (int moduletype)

Variables

- unsigned char `cache_buffer` [CACHE_SIZE]
- size_t `cache_fill` = 0

4.5.1 Detailed Description

Contains all functions that deal with the recovery mechanism controlled and activated by the On Checkpoint switch.

The main function are DoCheckpoint, DoRecovery, and DoSnapshot. If the checkpoints are activated DoCheckpoint is called every time a module is finished executing. If the conditions for the creation of a recovery snapshot are met DoCheckpoint calls DoSnapshot. DoRecovery is called once when FORM starts up with the command line argument -R. Most of the other code contains debugging facilities that are only compiled if the macro PRINTDEBUG is defined.

The recovery mechanism is atomic, i.e. only if everything went well, the final recovery file is created (and the older one overwritten) in a single step (copying). If some errors occur, a warning is issued and the program continues without having created a new recovery file. The only situation in which the creation of the recovery data leads to a termination of the running program is if not enough disk or memory space is left.

For ParFORM each slave creates its own recovery file, sends it to the master and then it deletes the recovery file. The master stores all the recovery files and on recovery it feeds these files to the slaves. It is nearly impossible to recover after some MPI fault so ParFORM terminates on any recovery failure.

DoRecovery and DoSnapshot do the loading and saving of the recovery data, respectively. Every change in one functions needs to be accompanied by the appropriate change in the other function. The structure of both functions is quite similar. They handle the relevant global structs one after the other and then care about the copying of the hide and scratch files.

The names of the recovery, scratch and hide files are hard-coded in the variables in fold "filenames and system commands".

If the global structs AM,AP,AC,AR are changed, DoRecovery and DoSnapshot usually also have to be changed. Some structs are read/written as a whole (AP,AC), some are read/written only partly as a selection of their individual elements (AM,AR). If AM or AR have been changed by adding or removing an element that is important for the runtime status, then the reading/writing statements have to be added to or removed from DoRecovery and DoSnapshot. If AP or AC are changed, then for non-pointer variables (in the case of a struct it also means that none of its elements is a pointer) nothing has to be changed in the functions here. If pointers are involved, extra code has to be added (or removed). See the comments of DoRecovery and DoSnapshot.

Definition in file [checkpoint.c](#).

4.5.2 Macro Definition Documentation

4.5.2.1 CACHED_SNAPSHOT

```
#define CACHED_SNAPSHOT
```

Definition at line 1214 of file [checkpoint.c](#).

4.5.2.2 CACHE_SIZE

```
#define CACHE_SIZE 4096
```

Definition at line 1216 of file [checkpoint.c](#).

4.5.2.3 R_FREE

```
#define R_FREE(  
    ARG )    if ( ARG ) M_free(ARG, #ARG);
```

Definition at line 1281 of file [checkpoint.c](#).

4.5.2.4 R_FREE_NAMETREE

```
#define R_FREE_NAMETREE(  
    ARG )
```

Value:

```
R_FREE(ARG->namenode); \  
R_FREE(ARG->namebuffer); \  
R_FREE(ARG);
```

Definition at line 1284 of file [checkpoint.c](#).

4.5.2.5 R_FREE_STREAM

```
#define R_FREE_STREAM(  
    ARG )
```

Value:

```
R_FREE(ARG.buffer); \  
R_FREE(ARG.FoldName); \  
R_FREE(ARG.name);
```

Definition at line 1289 of file [checkpoint.c](#).

4.5.2.6 R_SET

```
#define R_SET(  
    VAR,  
    TYPE )    VAR = *((TYPE*)p); p = (unsigned char*)p + sizeof(TYPE);
```

Definition at line 1296 of file [checkpoint.c](#).

4.5.2.7 R_COPY_B

```
#define R_COPY_B(  
    VAR,  
    SIZE,  
    CAST )
```

Value:

```
VAR = (CAST)Malloc1(SIZE, #VAR); \  
memcpy(VAR, p, SIZE); p = (unsigned char*)p + SIZE;
```

Definition at line 1301 of file [checkpoint.c](#).

4.5.2.8 S_WRITE_B

```
#define S_WRITE_B(
    BUF,
    LEN ) if ( fwrite_cached(BUF, 1, LEN, fd) != (size_t) (LEN) ) return(__LINE__);
```

Definition at line 1305 of file [checkpoint.c](#).

4.5.2.9 S_FLUSH_B

```
#define S_FLUSH_B if ( flush_cache(fd) != 1 ) return(__LINE__);
```

Definition at line 1308 of file [checkpoint.c](#).

4.5.2.10 R_COPY_S

```
#define R_COPY_S(
    VAR,
    CAST )
```

Value:

```
if ( VAR ) { \
    VAR = (CAST)Malloc1(strlen(p)+1,"R_COPY_S"); \
    strcpy((char*)VAR, p); p = (unsigned char*)p + strlen(p) + 1; \
}
```

Definition at line 1313 of file [checkpoint.c](#).

4.5.2.11 S_WRITE_S

```
#define S_WRITE_S(
    STR )
```

Value:

```
if ( STR ) { \
    l = strlen((char*)STR) + 1; \
    if ( fwrite_cached(STR, 1, l, fd) != (size_t)l ) return(__LINE__); \
}
```

Definition at line 1319 of file [checkpoint.c](#).

4.5.2.12 R_COPY_LIST

```
#define R_COPY_LIST(
    ARG )
```

Value:

```
if ( ARG.maxnum ) { \
    R_COPY_B(ARG.lijst, ARG.size*ARG.maxnum, void*) \
}
```

Definition at line 1327 of file [checkpoint.c](#).

4.5.2.13 S_WRITE_LIST

```
#define S_WRITE_LIST(
    LST )
```

Value:

```
if ( LST.maxnum ) { \
    S_WRITE_B((char*)LST.lijst, LST.maxnum*LST.size) \
}
```

Definition at line 1332 of file [checkpoint.c](#).

4.5.2.14 R_COPY_NAMETREE

```
#define R_COPY_NAMETREE(
    ARG )
```

Value:

```
R_COPY_B(ARG, sizeof(NAMETREE), NAMETREE*); \
if ( ARG->namenode ) { \
    R_COPY_B(ARG->namenode, ARG->nodesize*sizeof(NAMENODE), NAMENODE*); \
} \
if ( ARG->namebuffer ) { \
    R_COPY_B(ARG->namebuffer, ARG->namesize, UBYTE*); \
}
```

Definition at line 1339 of file [checkpoint.c](#).

4.5.2.15 S_WRITE_NAMETREE

```
#define S_WRITE_NAMETREE(
    ARG )
```

Value:

```
S_WRITE_B(ARG, sizeof(NAMETREE)); \
if ( ARG->namenode ) { \
    S_WRITE_B(ARG->namenode, ARG->nodesize*sizeof(struct NaMeNode)); \
} \
if ( ARG->namebuffer ) { \
    S_WRITE_B(ARG->namebuffer, ARG->namesize); \
}
```

Definition at line 1348 of file [checkpoint.c](#).

4.5.2.16 S_WRITE_DOLLAR

```
#define S_WRITE_DOLLAR(
    ARG )
```

Value:

```
if ( ARG.size && ARG.where && ARG.where != &(AM.dollarzero) ) { \
    S_WRITE_B(ARG.where, ARG.size*sizeof(WORD)) \
}
```

Definition at line 1359 of file [checkpoint.c](#).

4.5.2.17 ANNOUNCE

```
#define ANNOUNCE(  
    str )
```

Definition at line 1370 of file [checkpoint.c](#).

4.5.3 Function Documentation

4.5.3.1 CheckRecoveryFile()

```
int CheckRecoveryFile (  
    void )
```

Checks whether a snapshot/recovery file exists. Returns 1 if it exists, 0 otherwise.

Definition at line 278 of file [checkpoint.c](#).

References [RecoveryFilename\(\)](#).

Here is the call graph for this function:



4.5.3.2 DeleteRecoveryFile()

```
void DeleteRecoveryFile (  
    void )
```

Deletes the recovery files. It is called by `CleanUp()` in the case of a successful completion.

Definition at line 333 of file [checkpoint.c](#).

4.5.3.3 RecoveryFilename()

```
char * RecoveryFilename (  
    void )
```

Returns pointer to recovery filename.

Definition at line 364 of file [checkpoint.c](#).

Referenced by [CheckRecoveryFile\(\)](#), and [DoRecovery\(\)](#).

4.5.3.4 InitRecovery()

```
void InitRecovery (
    void )
```

Sets up the strings for the filenames of the recovery files. This functions should only be called once to avoid memory leaks and after AM.TempDir has been initialized.

Definition at line 399 of file [checkpoint.c](#).

4.5.3.5 fwrite_cached()

```
size_t fwrite_cached (
    const void * ptr,
    size_t size,
    size_t nmemb,
    FILE * fd )
```

Definition at line 1222 of file [checkpoint.c](#).

4.5.3.6 flush_cache()

```
size_t flush_cache (
    FILE * fd )
```

Definition at line 1247 of file [checkpoint.c](#).

4.5.3.7 DoRecovery()

```
int DoRecovery (
    int * moduletype )
```

Reads from the recovery file and restores all necessary variables and states in FORM, so that the execution can recommence in preprocessor() as if no restart of FORM had occurred.

The recovery file is read into memory as a whole. The pointer p then points into this memory at the next non-processed data. The macros by which variables are restored, like R_SET, automatically increase p appropriately.

If something goes wrong, the function returns with a non-zero value.

Allocated memory that would be lost when overwriting the global structs with data from the file is freed first. A major part of the code deals with the restoration of pointers. The idiom we use is to memorize the original pointer value (org), allocate new memory and copy the data from the file into this memory, calculate the offset between the old pointer value and the new allocated memory position (ofs), and then correct all affected pointers (+=ofs).

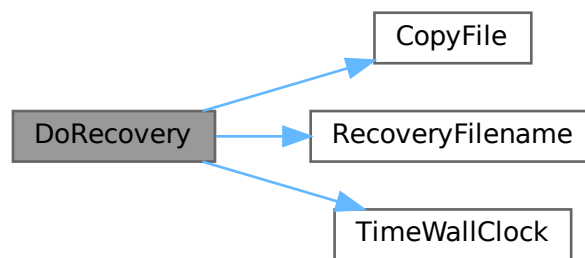
We rely on the fact that several variables (especially in AM) are already assigned the correct values by the startup functions. That means, in principle, that a change in the setup files between snapshot creation and recovery will be noticed.

Definition at line 1402 of file [checkpoint.c](#).

References [TaBIEs::argtail](#), [TaBIEs::boomlijst](#), [BrAcKeTiNfO::bracketbuffer](#), [TaBIEs::buffers](#), [TaBIEs::bufferssize](#), [CopyFile\(\)](#), [TaBIEs::flags](#), [ReNuMbEr::func](#), [ReNuMbEr::funnum](#), [VaRrEnUm::hi](#), [BrAcKeTiNfO::indexbuffer](#),

[ReNuMbEr::indi](#), [ReNuMbEr::indnum](#), [VaRrEnUm::lo](#), [TaBIes::MaxTreeSize](#), [TaBIes::mm](#), [TaBIes::numind](#), [TaBIes::pattern](#), [TaBIes::prototype](#), [TaBIes::prototypeSize](#), [RecoveryFilename\(\)](#), [ExPrEsSiOn::renumlists](#), [TaBIes::reserved](#), [TaBIes::spare](#), [TaBIes::sparse](#), [VaRrEnUm::start](#), [ReNuMbEr::symb](#), [ReNuMbEr::symnum](#), [TaBIes::tablepointers](#), [TimeWallClock\(\)](#), [TaBIes::totind](#), [ReNuMbEr::vecnum](#), and [ReNuMbEr::vect](#).

Here is the call graph for this function:



4.5.3.8 DoCheckpoint()

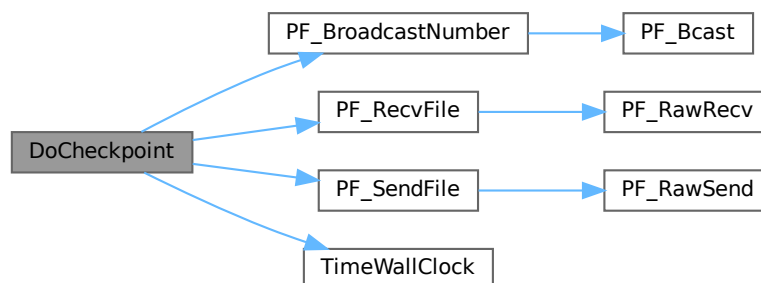
```
void DoCheckpoint (
    int moduletype )
```

Checks whether a snapshot should be done. Calls `DoSnapshot()` to create the snapshot.

Definition at line 3111 of file [checkpoint.c](#).

References [PF_BroadcastNumber\(\)](#), [PF_RecvFile\(\)](#), [PF_SendFile\(\)](#), and [TimeWallClock\(\)](#).

Here is the call graph for this function:



4.5.4 Variable Documentation

4.5.4.1 cache_buffer

```
unsigned char cache_buffer[CACHE_SIZE]
```

Definition at line 1219 of file [checkpoint.c](#).

4.5.4.2 cache_fill

```
size_t cache_fill = 0
```

Definition at line 1220 of file [checkpoint.c](#).

4.6 checkpoint.c

[Go to the documentation of this file.](#)

```
00001 /*
00002      #[ Explanations :
00003 */
00052 /*
00053      #[ Explanations :
00054      #[ License :
00055 *
00056 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00057 *   When using this file you are requested to refer to the publication
00058 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00059 *   This is considered a matter of courtesy as the development was paid
00060 *   for by FOM the Dutch physics granting agency and we would like to
00061 *   be able to track its scientific use to convince FOM of its value
00062 *   for the community.
00063 *
00064 *   This file is part of FORM.
00065 *
00066 *   FORM is free software: you can redistribute it and/or modify it under the
00067 *   terms of the GNU General Public License as published by the Free Software
00068 *   Foundation, either version 3 of the License, or (at your option) any later
00069 *   version.
00070 *
00071 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00072 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00073 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00074 *   details.
00075 *
00076 *   You should have received a copy of the GNU General Public License along
00077 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00078 */
00079 /*
00080      #[ License :
00081      #[ Includes :
00082 */
00083
00084 #include "form3.h"
00085
00086 #include <errno.h>
00087
00088 /*
00089 #define PRINTDEBUG
00090 */
00091
00092 /*
00093 #define PRINTTIMEMARKS
00094 */
00095
00096 /*
00097      #[ Includes :
00098      #[ filenames and system commands :
00099 */
00100
00104 #ifndef WITHMPI
00105 #define BASENAME_FMT "%c%04dFORMrecv"
```

```

00111 static char BaseName[] = BASENAME_FMT;
00112 #else
00113 static char *BaseName = "FORMrecv";
00114 #endif
00118 static char *recoveryfile = 0;
00124 static char *intermedfile = 0;
00128 static char *sortfile = 0;
00132 static char *hidefile = 0;
00136 static char *storefile = 0;
00137
00142 static int done_snapshot = 0;
00143
00144 #ifdef WITHMPI
00149 static int PF_fmt_pos;
00150
00155 static const char *PF_recoveryfile(char prefix, int id, int intermed)
00156 {
00157     /*
00158      * Assume that InitRecovery() has been already called, namely
00159      * recoveryfile, intermedfile and PF_fmt_pos are already initialized.
00160      */
00161     static char *tmp_recovery = NULL;
00162     static char *tmp_intermed = NULL;
00163     char *tmp, c;
00164     if ( tmp_recovery == NULL ) {
00165         if ( PF.numtasks > 9999 ) { /* see BASENAME_FMT */
00166             MesPrint("Checkpoint: too many number of processors.");
00167             Terminate(-1);
00168         }
00169         tmp_recovery = (char *)Malloc1(strlen(recoveryfile) + strlen(intermedfile) + 2,
"PF_recoveryfile");
00170         tmp_intermed = tmp_recovery + strlen(recoveryfile) + 1;
00171         strcpy(tmp_recovery, recoveryfile);
00172         strcpy(tmp_intermed, intermedfile);
00173     }
00174     tmp = intermed ? tmp_intermed : tmp_recovery;
00175     c = tmp[PF_fmt_pos + 13]; /* The magic number 13 comes from BASENAME_FMT. */
00176     sprintf(tmp + PF_fmt_pos, BASENAME_FMT, prefix, id);
00177     tmp[PF_fmt_pos + 13] = c;
00178     return tmp;
00179 }
00180 #endif
00181
00182 /*
00183  #] filenames and system commands :
00184  #[ CheckRecoveryFile :
00185  */
00186
00191 #ifdef WITHMPI
00192
00198 static int PF_CheckRecoveryFile()
00199 {
00200     int i,ret=0;
00201     FILE *fd;
00202     /* Check if the recovery file for the master exists. */
00203     if ( PF.me == MASTER ) {
00204         if ( (fd = fopen(recoveryfile, "r")) ) {
00205             fclose(fd);
00206             PF_BroadcastNumber(1);
00207         }
00208         else {
00209             PF_BroadcastNumber(0);
00210             return 0;
00211         }
00212     }
00213     else {
00214         if ( !PF_BroadcastNumber(0) )
00215             return 0;
00216     }
00217     /* Now the main part. */
00218     if (PF.me == MASTER){
00219         /*We have to have recovery files for the master and all the slaves:*/
00220         for(i=1; i<PF.numtasks;i++){
00221             const char *tmpnam = PF_recoveryfile('m', i, 0);
00222             if ( (fd = fopen(tmpnam, "r")) )
00223                 fclose(fd);
00224             else
00225                 break;
00226         }/*for(i=0; i<PF.numtasks;i++)*/
00227         if(i!=PF.numtasks){/*some files are absent*/
00228             int j;
00229             /*Send all slaves failure*/
00230             for(j=1; j<PF.numtasks;j++){
00231                 ret=PF_SendFile(j, NULL);
00232                 if(ret<0)
00233                     return(-1);
00234             }

```

```

00235         if(i==0)
00236             return(0);/*Ok, no recovery files at all.*/
00237         /*The master recovery file exists but some slave files are absent*/
00238         MesPrint("The file %s exists but some of the slave recovery files are absent.",
00239                 RecoveryFilename());
00240         return(-1);
00241     }/*if(i!=PF.numtasks)*/
00242     /*All the files are here.*/
00243     /*Send all slaves success and files:*/
00244     for(i=1; i<PF.numtasks;i++){
00245         const char *tmpnam = PF_recoveryfile('m', i, 0);
00246         fd = fopen(tmpnam, "r");
00247         ret=PF_SendFile(i, fd);/*if fd==NULL, PF_SendFile seds to a slave the failure tag*/
00248         if(fd == NULL)
00249             return(-1);
00250         else
00251             fclose(fd);
00252         if(ret<0)
00253             return(-1);
00254     }/*for(i=0; i<PF.numtasks;i++)*/
00255     return(1);
00256 }/*if (PF.me == MASTER)*/
00257 /*Slave:*/
00258 /*Get the answer from the master:*/
00259 fd=fopen(recoveryfile,"w");
00260 if(fd == NULL) {
00261     MesPrint("Failed to open %s in write mode in process %w", recoveryfile);
00262     return(-1);
00263 }
00264 ret=PF_RecvFile(MASTER,fd);
00265 if(ret<0)
00266     return(-1);
00267 fclose(fd);
00268 if(ret==0){
00269     /*Nothing is found by the master*/
00270     remove(recoveryfile);
00271     return(0);
00272 }
00273 /*Recovery file is successfully transferred*/
00274 return(1);
00275 }
00276 #endif
00277
00278 int CheckRecoveryFile(void)
00279 {
00280     int ret = 0;
00281     #ifdef WITHMPI
00282     ret = PF_CheckRecoveryFile();
00283     #else
00284     FILE *fd;
00285     if ( ( fd = fopen(recoveryfile, "r")) ) {
00286         fclose(fd);
00287         ret = 1;
00288     }
00289     #endif
00290     if ( ret < 0 ){/*In ParFORM CheckRecoveryFile() may return a fatal error.*/
00291         MesPrint("Fail checking recovery file");
00292         Terminate(-1);
00293     }
00294     else if ( ret > 0 ) {
00295         if ( AC.CheckpointFlag != -1 ) {
00296             /* recovery file exists but recovery option is not given */
00297             #ifdef WITHMPI
00298                 if ( PF.me == MASTER ) {
00299                     #endif
00300                     MesPrint("The recovery file %s exists, but the recovery option -R has not been given!",
00301                             RecoveryFilename());
00302                     MesPrint("FORM will be terminated to avoid unintentional loss of data.");
00303                     MesPrint("Delete the recovery file manually, if you want to start FORM without
00304 recovery.");
00305                     #ifdef WITHMPI
00306                     }
00307                     if(PF.me != MASTER)
00308                         remove(RecoveryFilename());
00309                     #endif
00310                     Terminate(-1);
00311                 }
00312             else {
00313                 if ( AC.CheckpointFlag == -1 ) {
00314                     /* recovery option given but recovery file does not exist */
00315                     #ifdef WITHMPI
00316                     if ( PF.me == MASTER )
00317                         #endif
00318                     MesPrint("Option -R for recovery has been given, but the recovery file %s does not
00319 exist!", RecoveryFilename());
00320                     Terminate(-1);

```

```

00319     }
00320 }
00321 return(ret);
00322 }
00323
00324 /*
00325  #] CheckRecoveryFile :
00326  #[ DeleteRecoveryFile :
00327 */
00328
00333 void DeleteRecoveryFile(void)
00334 {
00335     if ( done_snapshot ) {
00336         remove(recoveryfile);
00337 #ifdef WITHMPI
00338         if( PF.me == MASTER){
00339             int i;
00340             for(i=1; i<PF.numtasks;i++){
00341                 const char *tmpnam = PF_recoveryfile('m', i, 0);
00342                 remove(tmpnam);
00343             }/*for(i=1; i<PF.numtasks;i++)*/
00344             remove(storefile);
00345             remove(sortfile);
00346             remove(hidefile);
00347         }/*if( PF.me == MASTER)*/
00348 #else
00349         remove(storefile);
00350         remove(sortfile);
00351         remove(hidefile);
00352 #endif
00353     }
00354 }
00355
00356 /*
00357  #] DeleteRecoveryFile :
00358  #[ RecoveryFilename :
00359 */
00360
00364 char *RecoveryFilename(void)
00365 {
00366     return(recoveryfile);
00367 }
00368
00369 /*
00370  #] RecoveryFilename :
00371  #[ InitRecovery :
00372 */
00373
00377 static char *InitName(char *str, char *ext)
00378 {
00379     char *s, *d = str;
00380     if ( AM.TempDir ) {
00381         s = (char*)AM.TempDir;
00382         while ( *s ) { *d++ = *s++; }
00383         *d++ = SEPARATOR;
00384     }
00385     s = BaseName;
00386     while ( *s ) { *d++ = *s++; }
00387     *d++ = '.';
00388     s = ext;
00389     while ( *s ) { *d++ = *s++; }
00390     *d++ = 0;
00391     return d;
00392 }
00393
00399 void InitRecovery(void)
00400 {
00401     int lenpath = AM.TempDir ? strlen((char*)AM.TempDir)+1 : 0;
00402 #ifdef WITHMPI
00403     sprintf(BaseName,BASENAME_FMT,(PF.me == MASTER)?'m':'s',PF.me);
00404     /*Now BaseName has a form ?XXXXFORMrcv where ? == 'm' for master and 's' for slave,
00405        XXXX is a zero - padded PF.me*/
00406     PF_fmt_pos = lenpath;
00407 #endif
00408     recoveryfile = (char*)Malloc1(5*(lenpath+strlen(BaseName)+4+1),"InitRecovery");
00409     intermedfile = InitName(recoveryfile, "tmp");
00410     sortfile      = InitName(intermedfile, "XXX");
00411     hidefile      = InitName(sortfile, "out");
00412     storefile     = InitName(hidefile, "hid");
00413     InitName(storefile, "str");
00414 }
00415
00416 /*
00417  #] InitRecovery :
00418  #[ Debugging :
00419 */
00420

```



```

00421 #ifdef PRINTDEBUG
00422
00423 void print_BYTE(void *p)
00424 {
00425     UBYTE h = (UBYTE) (*(UBYTE*)p) >> 4;
00426     UBYTE l = (UBYTE) (*(UBYTE*)p) & 0x0F;
00427     if ( h > 9 ) h += 55; else h += 48;
00428     if ( l > 9 ) l += 55; else l += 48;
00429     printf("%c%c ", h, l);
00430 }
00431
00432 static void print_STR(UBYTE *p)
00433 {
00434     if ( p ) {
00435         MesPrint("%s", (char*)p);
00436     }
00437     else {
00438         MesPrint("NULL");
00439     }
00440 }
00441
00442 static void print_WORDB(WORD *buf, WORD *top)
00443 {
00444     LONG size = top-buf;
00445     int i;
00446     while ( size > 0 ) {
00447         if ( size > MAXPOSITIVE ) i = MAXPOSITIVE;
00448         else i = size;
00449         size -= i;
00450         MesPrint("%a",i,buf);
00451         buf += i;
00452     }
00453 }
00454
00455 static void print_VOIDP(void *p, size_t size)
00456 {
00457     int i;
00458     if ( p ) {
00459         while ( size > 0 ) {
00460             if ( size > MAXPOSITIVE ) i = MAXPOSITIVE;
00461             else i = size;
00462             size -= i;
00463             MesPrint("%b",i,(UBYTE *)p);
00464             p = ((UBYTE *)p)+i;
00465         }
00466     }
00467     else {
00468         MesPrint("NULL");
00469     }
00470 }
00471
00472 static void print_CHARS(UBYTE *p, size_t size)
00473 {
00474     int i;
00475     while ( size > 0 ) {
00476         if ( size > MAXPOSITIVE ) i = MAXPOSITIVE;
00477         else i = size;
00478         size -= i;
00479         MesPrint("%C",i,(char *)p);
00480         p += i;
00481     }
00482 }
00483
00484 static void print_WORDV(WORD *p, size_t size)
00485 {
00486     int i;
00487     if ( p ) {
00488         while ( size > 0 ) {
00489             if ( size > MAXPOSITIVE ) i = MAXPOSITIVE;
00490             else i = size;
00491             size -= i;
00492             MesPrint("%a",i,p);
00493             p += i;
00494         }
00495     }
00496     else {
00497         MesPrint("NULL");
00498     }
00499 }
00500
00501 static void print_INTV(int *p, size_t size)
00502 {
00503     int iarray[8];
00504     WORD i = 0;
00505     if ( p ) {
00506         while ( size > 0 ) {
00507             if ( i >= 8 ) {

```

```

00508         MesPrint("%I",i,iarray);
00509         i = 0;
00510     }
00511     iarray[i++] = *p++;
00512     size--;
00513 }
00514 if ( i > 0 ) MesPrint("%I",i,iarray);
00515 }
00516 else {
00517     MesPrint("NULL");
00518 }
00519 }
00520
00521 static void print_LONGV(LONG *p, size_t size)
00522 {
00523     LONG larray[8];
00524     WORD i = 0;
00525     if ( p ) {
00526         while ( size > 0 ) {
00527             if ( i >= 8 ) {
00528                 MesPrint("%I",i,larray);
00529                 i = 0;
00530             }
00531             larray[i++] = *p++;
00532             size--;
00533         }
00534         if ( i > 0 ) MesPrint("%I",i,larray);
00535     }
00536     else {
00537         MesPrint("NULL");
00538     }
00539 }
00540
00541 static void print_PRELOAD(PRELOAD *l)
00542 {
00543     if ( l->size ) {
00544         print_CHARS(l->buffer, l->size);
00545     }
00546     MesPrint("%ld", l->size);
00547 }
00548
00549 static void print_PREVAR(PREVAR *l)
00550 {
00551     MesPrint("%s", l->name);
00552     print_STR(l->value);
00553     if ( l->nargs ) print_STR(l->argnames);
00554     MesPrint("%d", l->nargs);
00555     MesPrint("%d", l->wildarg);
00556 }
00557
00558 static void print_DOLLARS(DOLLARS l)
00559 {
00560     print_VOIDP(l->where, l->size);
00561     MesPrint("%ld", l->size);
00562     MesPrint("%ld", l->name);
00563     MesPrint("%s", AC.dollarnames->namebuffer+l->name);
00564     MesPrint("%d", l->type);
00565     MesPrint("%d", l->node);
00566     MesPrint("%d", l->index);
00567     MesPrint("%d", l->zero);
00568     MesPrint("%d", l->numdummies);
00569     MesPrint("%d", l->nfactors);
00570 }
00571
00572 static void print_LIST(LIST *l)
00573 {
00574     print_VOIDP(l->lijst, l->size);
00575     MesPrint("%s", l->message);
00576     MesPrint("%d", l->num);
00577     MesPrint("%d", l->maxnum);
00578     MesPrint("%d", l->size);
00579     MesPrint("%d", l->numglobal);
00580     MesPrint("%d", l->numtemp);
00581     MesPrint("%d", l->numclear);
00582 }
00583
00584 static void print_DOLOOP(DOLOOP *l)
00585 {
00586     print_PRELOAD(&(l->p));
00587     print_STR(l->name);
00588     if ( l->type != NUMERICALLOOP ) {
00589         print_STR(l->vars);
00590     }
00591     print_STR(l->contents);
00592     if ( l->type != LISTEDLOOP && l->type != NUMERICALLOOP ) {
00593         print_STR(l->dollarname);
00594     }

```

```

00595     MesPrint("%l", l->startlinenumber);
00596     MesPrint("%l", l->firstnum);
00597     MesPrint("%l", l->lastnum);
00598     MesPrint("%l", l->incnum);
00599     MesPrint("%d", l->type);
00600     MesPrint("%d", l->NoShowInput);
00601     MesPrint("%d", l->errorsinloop);
00602     MesPrint("%d", l->firstloopcall);
00603 }
00604
00605 static void print_PROCEDURE(PROCEDURE *l)
00606 {
00607     if ( l->loadmode != 1 ) {
00608         print_PRELOAD(&(l->p));
00609     }
00610     print_STR(l->name);
00611     MesPrint("%d", l->loadmode);
00612 }
00613
00614 static void print_NAMETREE(NAMETREE *t)
00615 {
00616     int i;
00617     for ( i=0; i<t->nodefill; ++i ) {
00618         MesPrint("%l %d %d %d %d %d %d\n", t->namenode[i].name,
00619             t->namenode[i].parent, t->namenode[i].left, t->namenode[i].right,
00620             t->namenode[i].balance, t->namenode[i].type, t->namenode[i].number );
00621     }
00622     print_CHARS(t->namebuffer, t->namefill);
00623     MesPrint("%l", t->namesize);
00624     MesPrint("%l", t->namefill);
00625     MesPrint("%l", t->nodesize);
00626     MesPrint("%l", t->nodefill);
00627     MesPrint("%l", t->oldnamefill);
00628     MesPrint("%l", t->oldnodefill);
00629     MesPrint("%l", t->globalnamefill);
00630     MesPrint("%l", t->globalnodefill);
00631     MesPrint("%l", t->clearnamefill);
00632     MesPrint("%l", t->clearnodefill);
00633     MesPrint("%d", t->headnode);
00634 }
00635
00636 void print_CBUF(CBUF *c)
00637 {
00638     int i;
00639     print_WORDV(c->Buffer, c->BufferSize);
00640     /*
00641     MesPrint("%x", c->Buffer);
00642     MesPrint("%x", c->lhs);
00643     MesPrint("%x", c->rhs);
00644     */
00645     for ( i=0; i<c->numlhs; ++i ) {
00646         if ( c->lhs[i] ) MesPrint("%d", *(c->lhs[i]));
00647     }
00648     for ( i=0; i<c->numrhs; ++i ) {
00649         if ( c->rhs[i] ) MesPrint("%d", *(c->rhs[i]));
00650     }
00651     MesPrint("%l", *c->CanCommu);
00652     MesPrint("%l", *c->NumTerms);
00653     MesPrint("%d", *c->numdum);
00654     for ( i=0; i<c->MaxTreeSize; ++i ) {
00655         MesPrint("%d %d %d %d %d", c->boomlijst[i].parent, c->boomlijst[i].left,
c->boomlijst[i].right,
00656             c->boomlijst[i].value, c->boomlijst[i].blnce);
00657     }
00658 }
00659
00660 static void print_STREAM(STREAM *t)
00661 {
00662     print_CHARS(t->buffer, t->inbuffer);
00663     MesPrint("%l", (LONG)(t->pointer-t->buffer));
00664     print_STR(t->FoldName);
00665     print_STR(t->name);
00666     if ( t->type == PREVARSTREAM || t->type == DOLLARSTREAM ) {
00667         print_STR(t->pname);
00668     }
00669     MesPrint("%l", (LONG)t->fileposition);
00670     MesPrint("%l", (LONG)t->linenumber);
00671     MesPrint("%l", (LONG)t->prevline);
00672     MesPrint("%l", t->buffersize);
00673     MesPrint("%l", t->bufferposition);
00674     MesPrint("%l", t->inbuffer);
00675     MesPrint("%d", t->previous);
00676     MesPrint("%d", t->handle);
00677     switch ( t->type ) {
00678         case FILESTREAM: MesPrint("%d == FILESTREAM", t->type); break;
00679         case PREVARSTREAM: MesPrint("%d == PREVARSTREAM", t->type); break;
00680         case PREREADSTREAM: MesPrint("%d == PREREADSTREAM", t->type); break;

```

```

00681         case PIPESTREAM: MesPrint("%d == PIPESTREAM", t->type); break;
00682         case PRECALCSTREAM: MesPrint("%d == PRECALCSTREAM", t->type); break;
00683         case DOLLARSTREAM: MesPrint("%d == DOLLARSTREAM", t->type); break;
00684         case PREREADSTREAM2: MesPrint("%d == PREREADSTREAM2", t->type); break;
00685         case EXTERNALCHANNELSTREAM: MesPrint("%d == EXTERNALCHANNELSTREAM", t->type); break;
00686         case PREREADSTREAM3: MesPrint("%d == PREREADSTREAM3", t->type); break;
00687         default: MesPrint("%d == UNKNOWN", t->type);
00688     }
00689 }
00690
00691 static void print_M()
00692 {
00693     MesPrint("%%% M_const");
00694     MesPrint("%d", *AM.gcmmod);
00695     MesPrint("%d", *AM.gpowmod);
00696     print_STR(AM.TempDir);
00697     print_STR(AM.TempSortDir);
00698     print_STR(AM.IncDir);
00699     print_STR(AM.InputFileName);
00700     print_STR(AM.LogFileName);
00701     print_STR(AM.OutBuffer);
00702     print_STR(AM.Path);
00703     print_STR(AM.SetupDir);
00704     print_STR(AM.SetupFile);
00705     MesPrint("--MARK 1");
00706     MesPrint("%l", (LONG)BASEPOSITION(AM.zeropos));
00707 #ifdef WITHPTHREADS
00708     MesPrint("%l", AM.ThreadScratSize);
00709     MesPrint("%l", AM.ThreadScratOutSize);
00710 #endif
00711     MesPrint("%l", AM.MaxTer);
00712     MesPrint("%l", AM.CompressSize);
00713     MesPrint("%l", AM.ScratSize);
00714     MesPrint("%l", AM.SizeStoreCache);
00715     MesPrint("%l", AM.MaxStreamSize);
00716     MesPrint("%l", AM.SIOsize);
00717     MesPrint("%l", AM.SLargeSize);
00718     MesPrint("%l", AM.SSmallEsize);
00719     MesPrint("%l", AM.SSmallSize);
00720     MesPrint("--MARK 2");
00721     MesPrint("%l", AM.STermsInSmall);
00722     MesPrint("%l", AM.MaxBracketBufferSize);
00723     MesPrint("%l", AM.hProcessBucketSize);
00724     MesPrint("%l", AM.gProcessBucketSize);
00725     MesPrint("%l", AM.shmWinSize);
00726     MesPrint("%l", AM.OldChildTime);
00727     MesPrint("%l", AM.OldSecTime);
00728     MesPrint("%l", AM.OldMilliTime);
00729     MesPrint("%l", AM.WorkSize);
00730     MesPrint("%l", AM.gThreadBucketSize);
00731     MesPrint("--MARK 3");
00732     MesPrint("%l", AM.ggThreadBucketSize);
00733     MesPrint("%d", AM.FileOnlyFlag);
00734     MesPrint("%d", AM.Interact);
00735     MesPrint("%d", AM.MaxParLevel);
00736     MesPrint("%d", AM.OutBufSize);
00737     MesPrint("%d", AM.SMaxFpatches);
00738     MesPrint("%d", AM.SMaxPatches);
00739     MesPrint("%d", AM.StdOut);
00740     MesPrint("%d", AM.ginsidefirst);
00741     MesPrint("%d", AM.gDefDim);
00742     MesPrint("%d", AM.gDefDim4);
00743     MesPrint("--MARK 4");
00744     MesPrint("%d", AM.NumFixedSets);
00745     MesPrint("%d", AM.NumFixedFunctions);
00746     MesPrint("%d", AM.rbufnum);
00747     MesPrint("%d", AM.dbufnum);
00748     MesPrint("%d", AM.SkipClears);
00749     MesPrint("%d", AM.gfunpowers);
00750     MesPrint("%d", AM.gStatsFlag);
00751     MesPrint("%d", AM.gNamesFlag);
00752     MesPrint("%d", AM.gCodesFlag);
00753     MesPrint("%d", AM.gTokensWriteFlag);
00754     MesPrint("%d", AM.gSortType);
00755     MesPrint("%d", AM.gproperorderflag);
00756     MesPrint("--MARK 5");
00757     MesPrint("%d", AM.hparallelflag);
00758     MesPrint("%d", AM.gparallelflag);
00759     MesPrint("%d", AM.totalnumberofthreads);
00760     MesPrint("%d", AM.gSizeCommuteInSet);
00761     MesPrint("%d", AM.gThreadStats);
00762     MesPrint("%d", AM.ggThreadStats);
00763     MesPrint("%d", AM.gFinalStats);
00764     MesPrint("%d", AM.ggFinalStats);
00765     MesPrint("%d", AM.gThreadsFlag);
00766     MesPrint("%d", AM.ggThreadsFlag);
00767     MesPrint("%d", AM.gThreadBalancing);

```

```

00768     MesPrint("%d", AM.ggThreadBalancing);
00769     MesPrint("%d", AM.gThreadSortFileSynch);
00770     MesPrint("%d", AM.ggThreadSortFileSynch);
00771     MesPrint("%d", AM.gProcessStats);
00772     MesPrint("%d", AM.ggProcessStats);
00773     MesPrint("%d", AM.gOldParallelStats);
00774     MesPrint("%d", AM.ggOldParallelStats);
00775     MesPrint("%d", AM.gWTimeStatsFlag);
00776     MesPrint("%d", AM.ggWTimeStatsFlag);
00777     MesPrint("%d", AM.maxFlevels);
00778     MesPrint("--MARK 6");
00779     MesPrint("%d", AM.resetTimeOnClear);
00780     MesPrint("%d", AM.gcNumDollars);
00781     MesPrint("%d", AM.MultiRun);
00782     MesPrint("%d", AM.gNoSpacesInNumbers);
00783     MesPrint("%d", AM.ggNoSpacesInNumbers);
00784     MesPrint("%d", AM.MaxTal);
00785     MesPrint("%d", AM.IndDum);
00786     MesPrint("%d", AM.DumInd);
00787     MesPrint("%d", AM.WilInd);
00788     MesPrint("%d", AM.gncmod);
00789     MesPrint("%d", AM.gnpowmod);
00790     MesPrint("%d", AM.gmodmode);
00791     MesPrint("--MARK 7");
00792     MesPrint("%d", AM.gUnitTrace);
00793     MesPrint("%d", AM.gOutputMode);
00794     MesPrint("%d", AM.gCnumpows);
00795     MesPrint("%d", AM.gOutputSpaces);
00796     MesPrint("%d", AM.gOutNumberType);
00797     MesPrint("%d %d %d %d", AM.gUniTrace[0], AM.gUniTrace[1], AM.gUniTrace[2], AM.gUniTrace[3]);
00798     MesPrint("%d", AM.MaxWildcards);
00799     MesPrint("%d", AM.mTraceDum);
00800     MesPrint("%d", AM.OffsetIndex);
00801     MesPrint("%d", AM.OffsetVector);
00802     MesPrint("%d", AM.RepMax);
00803     MesPrint("%d", AM.LogType);
00804     MesPrint("%d", AM.ggStatsFlag);
00805     MesPrint("%d", AM.gLineLength);
00806     MesPrint("%d", AM.qError);
00807     MesPrint("--MARK 8");
00808     MesPrint("%d", AM.FortranCont);
00809     MesPrint("%d", AM.HoldFlag);
00810     MesPrint("%d %d %d %d %d", AM.Ordering[0], AM.Ordering[1], AM.Ordering[2], AM.Ordering[3],
AM.Ordering[4]);
00811     MesPrint("%d %d %d %d %d", AM.Ordering[5], AM.Ordering[6], AM.Ordering[7], AM.Ordering[8],
AM.Ordering[9]);
00812     MesPrint("%d %d %d %d %d", AM.Ordering[10], AM.Ordering[11], AM.Ordering[12], AM.Ordering[13],
AM.Ordering[14]);
00813     MesPrint("%d", AM.silent);
00814     MesPrint("%d", AM.tracebackflag);
00815     MesPrint("%d", AM.expnum);
00816     MesPrint("%d", AM.denomnum);
00817     MesPrint("%d", AM.facnum);
00818     MesPrint("%d", AM.invfacnum);
00819     MesPrint("%d", AM.sumnum);
00820     MesPrint("%d", AM.sumpnum);
00821     MesPrint("--MARK 9");
00822     MesPrint("%d", AM.OldOrderFlag);
00823     MesPrint("%d", AM.termfunnum);
00824     MesPrint("%d", AM.matchfunnum);
00825     MesPrint("%d", AM.countfunnum);
00826     MesPrint("%d", AM.gPolyFun);
00827     MesPrint("%d", AM.gPolyFunInv);
00828     MesPrint("%d", AM.gPolyFunType);
00829     MesPrint("%d", AM.gPolyFunExp);
00830     MesPrint("%d", AM.gPolyFunVar);
00831     MesPrint("%d", AM.gPolyFunPow);
00832     MesPrint("--MARK 10");
00833     MesPrint("%d", AM.dollarzero);
00834     MesPrint("%d", AM.atstartup);
00835     MesPrint("%d", AM.exitflag);
00836     MesPrint("%d", AM.NumStoreCaches);
00837     MesPrint("%d", AM.gIndentSpace);
00838     MesPrint("%d", AM.ggIndentSpace);
00839     MesPrint("%d", AM.gShortStatsMax);
00840     MesPrint("%d", AM.ggShortStatsMax);
00841     MesPrint("--MARK 11");
00842     MesPrint("%d", AM.FromStdin);
00843     MesPrint("%d", AM.IgnoreDeprecation);
00844     MesPrint("%%% END M_const");
00845     /* fflush(0); */
00846 }
00847
00848 static void print_P()
00849 {
00850     int i;
00851     MesPrint("%%% P_const");

```

```

00852     print_LIST(&AP.DollarList);
00853     for ( i=0; i<AP.DollarList.num; ++i ) {
00854         print_DOLLARS(&(Dollars[i]));
00855     }
00856     MesPrint("--MARK 1");
00857     print_LIST(&AP.PreVarList);
00858     for ( i=0; i<AP.PreVarList.num; ++i ) {
00859         print_PREVAR(&(PreVar[i]));
00860     }
00861     MesPrint("--MARK 2");
00862     print_LIST(&AP.LoopList);
00863     for ( i=0; i<AP.LoopList.num; ++i ) {
00864         print_DOLOOP(&(DoLoops[i]));
00865     }
00866     MesPrint("--MARK 3");
00867     print_LIST(&AP.ProcList);
00868     for ( i=0; i<AP.ProcList.num; ++i ) {
00869         print_PROCEDURE(&(Procedures[i]));
00870     }
00871     MesPrint("--MARK 4");
00872     for ( i=0; i<=AP.PreSwitchLevel; ++i ) {
00873         print_STR(AP.PreSwitchStrings[i]);
00874     }
00875     MesPrint("%l", AP.preStop-AP.preStart);
00876     if ( AP.preFill ) MesPrint("%l", AP.preFill-AP.preStart);
00877     print_CHARS(AP.preStart, AP.pSize);
00878     MesPrint("%s", AP.procedureExtension);
00879     MesPrint("%s", AP.cprocedureExtension);
00880     print_INTV(AP.PreIfStack, AP.MaxPreIfLevel);
00881     print_INTV(AP.PreSwitchModes, AP.NumPreSwitchStrings+1);
00882     print_INTV(AP.PreTypes, AP.NumPreTypes+1);
00883     MesPrint("%d", AP.PreAssignFlag);
00884     MesPrint("--MARK 5");
00885     MesPrint("%d", AP.PreContinuation);
00886     MesPrint("%l", AP.InOutBuf);
00887     MesPrint("%l", AP.pSize);
00888     MesPrint("%d", AP.PreproFlag);
00889     MesPrint("%d", AP.iBufError);
00890     MesPrint("%d", AP.PreOut);
00891     MesPrint("%d", AP.PreSwitchLevel);
00892     MesPrint("%d", AP.NumPreSwitchStrings);
00893     MesPrint("%d", AP.MaxPreTypes);
00894     MesPrint("--MARK 6");
00895     MesPrint("%d", AP.NumPreTypes);
00896     MesPrint("%d", AP.DelayPrevar);
00897     MesPrint("%d", AP.AllowDelay);
00898     MesPrint("%d", AP.lhdollarerror);
00899     MesPrint("%d", AP.eat);
00900     MesPrint("%d", AP.gNumPre);
00901     MesPrint("%d", AP.PreDebug);
00902     MesPrint("--MARK 7");
00903     MesPrint("%d", AP.DebugFlag);
00904     MesPrint("%d", AP.preError);
00905     MesPrint("%C", 1, &(AP.ComChar));
00906     MesPrint("%C", 1, &(AP.cComChar));
00907     MesPrint("%%%" END P_const");
00908     /* fflush(0); */
00909 }
00910
00911 static void print_C()
00912 {
00913     int i;
00914     UBYTE buf[40], *t;
00915     MesPrint("%%%" C_const");
00916     for ( i=0; i<32; ++i ) {
00917         t = buf;
00918         t = NumCopy((WORD) (AC.separators[i].bit_7), t);
00919         t = NumCopy((WORD) (AC.separators[i].bit_6), t);
00920         t = NumCopy((WORD) (AC.separators[i].bit_5), t);
00921         t = NumCopy((WORD) (AC.separators[i].bit_4), t);
00922         t = NumCopy((WORD) (AC.separators[i].bit_3), t);
00923         t = NumCopy((WORD) (AC.separators[i].bit_2), t);
00924         t = NumCopy((WORD) (AC.separators[i].bit_1), t);
00925         t = NumCopy((WORD) (AC.separators[i].bit_0), t);
00926         MesPrint("%s ", buf);
00927     }
00928     print_NAMETREE(AC.dollarnames);
00929     print_NAMETREE(AC.exprnames);
00930     print_NAMETREE(AC.varnames);
00931     MesPrint("--MARK 1");
00932     print_LIST(&AC.ChannelList);
00933     for ( i=0; i<AC.ChannelList.num; ++i ) {
00934         MesPrint("%s %d", channels[i].name, channels[i].handle);
00935     }
00936     MesPrint("--MARK 2");
00937     print_LIST(&AC.DubiousList);
00938     MesPrint("--MARK 3");

```

```

00939     print_LIST(&AC.FunctionList);
00940     for ( i=0; i<AC.FunctionList.num; ++i ) {
00941         if ( functions[i].tabl ) {
00942             }
00943             MesPrint("%1", functions[i].symminfo);
00944             MesPrint("%1", functions[i].name);
00945             MesPrint("%d", functions[i].namesize);
00946         }
00947     MesPrint("--MARK 4");
00948     print_LIST(&AC.ExpressionList);
00949     print_LIST(&AC.IndexList);
00950     print_LIST(&AC.SetElementList);
00951     print_LIST(&AC.SetList);
00952     MesPrint("--MARK 5");
00953     print_LIST(&AC.SymbolList);
00954     print_LIST(&AC.VectorList);
00955     print_LIST(&AC.PotModDolList);
00956     print_LIST(&AC.ModOptDolList);
00957     print_LIST(&AC.TableBaseList);
00958
00959
00960 /*
00961     print_LIST(&AC.cbufList);
00962     for ( i=0; i<AC.cbufList.num; ++i ) {
00963         MesPrint("cbufList.num == %d", i);
00964         print_CBUF(cbuf+i);
00965     }
00966     MesPrint("%d", AC.cbufnum);
00967 */
00968     MesPrint("--MARK 6");
00969
00970     print_LIST(&AC.AutoSymbolList);
00971     print_LIST(&AC.AutoIndexList);
00972     print_LIST(&AC.AutoVectorList);
00973     print_LIST(&AC.AutoFunctionList);
00974
00975     print_NAMETREE(AC.autonames);
00976     MesPrint("--MARK 7");
00977
00978     print_LIST(AC.Symbols);
00979     print_LIST(AC.Indices);
00980     print_LIST(AC.Vectors);
00981     print_LIST(AC.Functions);
00982     MesPrint("--MARK 8");
00983
00984     print_NAMETREE(*AC.activenames);
00985
00986     MesPrint("--MARK 9");
00987
00988     MesPrint("%d", AC.AutoDeclareFlag);
00989
00990     for ( i=0; i<AC.NumStreams; ++i ) {
00991         MesPrint("Stream %d\n", i);
00992         print_STREAM(AC.Streams+i);
00993     }
00994     print_STREAM(AC.CurrentStream);
00995     MesPrint("--MARK 10");
00996
00997     print_LONGV(AC.termstack, AC.maxtermlevel);
00998     print_LONGV(AC.termstack, AC.maxtermlevel);
00999     print_VOIDP(AC.cmod, AM.MaxTal*4*sizeof(UWORD));
01000     print_WORDV((WORD *) (AC.cmod), 1);
01001     print_WORDV((WORD *) (AC.powmod), 1);
01002     print_WORDV((WORD *) AC.modpowers, 1);
01003     print_WORDV((WORD *) AC.halfmod, 1);
01004     MesPrint("--MARK 10-2");
01005     /*
01006     print_WORDV(AC.ProtoType, AC.ProtoType[1]);
01007     print_WORDV(AC.WildC, 1);
01008     */
01009
01010     MesPrint("--MARK 11");
01011     /* IfHeap ... Labels */
01012
01013     print_CHARS((UBYTE *) AC.tokens, AC.toptokens-AC.tokens);
01014     MesPrint("%1", AC.endoftokens-AC.tokens);
01015     print_WORDV(AC.tokenarglevel, AM.MaxParLevel);
01016     print_WORDV((WORD *) AC.modinverses, ABS(AC.ncmod));
01017 #ifdef WITHPTHREADS
01018     print_LONGV(AC.inputnumbers, AC.sizefirstnum+AC.sizefirstnum*sizeof(WORD)/sizeof(LONG));
01019     print_WORDV(AC.pfirstnum, 1);
01020 #endif
01021     MesPrint("--MARK 12");
01022     print_LONGV(AC.argstack, MAXNEST);
01023     print_LONGV(AC.insidestack, MAXNEST);
01024     print_LONGV(AC.inexprstack, MAXNEST);
01025     MesPrint("%1", AC.iBufferSize);

```

```
01026     MesPrint("%l", AC.TransEname);
01027     MesPrint("%l", AC.ProcessBucketSize);
01028     MesPrint("%l", AC.mProcessBucketSize);
01029     MesPrint("%l", AC.CModule);
01030     MesPrint("%l", AC.ThreadBucketSize);
01031     MesPrint("%d", AC.NoShowInput);
01032     MesPrint("%d", AC.ShortStats);
01033     MesPrint("%d", AC.compiletype);
01034     MesPrint("%d", AC.firstconstindex);
01035     MesPrint("%d", AC.insidefirst);
01036     MesPrint("%d", AC.minsidefirst);
01037     MesPrint("%d", AC.wildflag);
01038     MesPrint("%d", AC.NumLabels);
01039     MesPrint("%d", AC.MaxLabels);
01040     MesPrint("--MARK 13");
01041     MesPrint("%d", AC.lDefDim);
01042     MesPrint("%d", AC.lDefDim4);
01043     MesPrint("%d", AC.NumWildcardNames);
01044     MesPrint("%d", AC.WildcardBufferSize);
01045     MesPrint("%d", AC.MaxIf);
01046     MesPrint("%d", AC.NumStreams);
01047     MesPrint("%d", AC.MaxNumStreams);
01048     MesPrint("%d", AC.firstctypemessage);
01049     MesPrint("%d", AC.tablecheck);
01050     MesPrint("%d", AC.idoption);
01051     MesPrint("%d", AC.BottomLevel);
01052     MesPrint("%d", AC.CompileLevel);
01053     MesPrint("%d", AC.TokensWriteFlag);
01054     MesPrint("%d", AC.UnsureDollarMode);
01055     MesPrint("%d", AC.outsidefun);
01056     MesPrint("%d", AC.funpowers);
01057     MesPrint("--MARK 14");
01058     MesPrint("%d", AC.WarnFlag);
01059     MesPrint("%d", AC.StatsFlag);
01060     MesPrint("%d", AC.NamesFlag);
01061     MesPrint("%d", AC.CodesFlag);
01062     MesPrint("%d", AC.TokensWriteFlag);
01063     MesPrint("%d", AC.SetupFlag);
01064     MesPrint("%d", AC.SortType);
01065     MesPrint("%d", AC.lSortType);
01066     MesPrint("%d", AC.ThreadStats);
01067     MesPrint("%d", AC.FinalStats);
01068     MesPrint("%d", AC.ThreadsFlag);
01069     MesPrint("%d", AC.ThreadBalancing);
01070     MesPrint("%d", AC.ThreadSortFileSynch);
01071     MesPrint("%d", AC.ProcessStats);
01072     MesPrint("%d", AC.OldParallelStats);
01073     MesPrint("%d", AC.WTimeStatsFlag);
01074     MesPrint("%d", AC.BraceNormalizer);
01075     MesPrint("%d", AC.maxtermlevel);
01076     MesPrint("%d", AC.dumnumflag);
01077     MesPrint("--MARK 15");
01078     MesPrint("%d", AC.bracketindexflag);
01079     MesPrint("%d", AC.parallelflag);
01080     MesPrint("%d", AC.mparallelflag);
01081     MesPrint("%d", AC.properorderflag);
01082     MesPrint("%d", AC.vetofilling);
01083     MesPrint("%d", AC.tablefilling);
01084     MesPrint("%d", AC.vetotablebasefill);
01085     MesPrint("%d", AC.exprfillwarning);
01086     MesPrint("%d", AC.lhdollarflag);
01087     MesPrint("%d", AC.NoCompress);
01088 #ifdef WITHPTHREADS
01089     MesPrint("%d", AC.numpfirstnum);
01090     MesPrint("%d", AC.sizepfirstnum);
01091 #endif
01092     MesPrint("%d", AC.RepLevel);
01093     MesPrint("%d", AC.arglevel);
01094     MesPrint("%d", AC.insidelevel);
01095     MesPrint("%d", AC.inexprlevel);
01096     MesPrint("%d", AC.termlevel);
01097     MesPrint("--MARK 16");
01098     print_WORDV(AC.argsumcheck, MAXNEST);
01099     print_WORDV(AC.insidesumcheck, MAXNEST);
01100     print_WORDV(AC.inexprsumcheck, MAXNEST);
01101     MesPrint("%d", AC.MustTestTable);
01102     MesPrint("%d", AC.DumNum);
01103     MesPrint("%d", AC.ncmod);
01104     MesPrint("%d", AC.npowmod);
01105     MesPrint("%d", AC.modmode);
01106     MesPrint("%d", AC.nhalfmod);
01107     MesPrint("%d", AC.DirtPow);
01108     MesPrint("%d", AC.lUnitTrace);
01109     MesPrint("%d", AC.NwildC);
01110     MesPrint("%d", AC.ComDefer);
01111     MesPrint("%d", AC.CollectFun);
01112     MesPrint("%d", AC.AltCollectFun);
```



```

01113     MesPrint("--MARK 17");
01114     MesPrint("%d", AC.OutputMode);
01115     MesPrint("%d", AC.Cnumpows);
01116     MesPrint("%d", AC.OutputSpaces);
01117     MesPrint("%d", AC.OutNumberType);
01118     print_WORDV(AC.lUniTrace, 4);
01119     print_WORDV(AC.RepSumCheck, MAXREPEAT);
01120     MesPrint("%d", AC.DidClean);
01121     MesPrint("%d", AC.IfLevel);
01122     MesPrint("%d", AC.WhileLevel);
01123     print_WORDV(AC.IfSumCheck, (AC.MaxIf+1));
01124     MesPrint("%d", AC.LogHandle);
01125     MesPrint("%d", AC.LineLength);
01126     MesPrint("%d", AC.StoreHandle);
01127     MesPrint("%d", AC.HideLevel);
01128     MesPrint("%d", AC.lPolyFun);
01129     MesPrint("%d", AC.lPolyFunInv);
01130     MesPrint("%d", AC.lPolyFunType);
01131     MesPrint("%d", AC.lPolyFunExp);
01132     MesPrint("%d", AC.lPolyFunVar);
01133     MesPrint("%d", AC.lPolyFunPow);
01134     MesPrint("%d", AC.SymChangeFlag);
01135     MesPrint("%d", AC.CollectPercentage);
01136     MesPrint("%d", AC.ShortStatsMax);
01137     MesPrint("--MARK 18");
01138
01139     print_CHARS(AC.Commercial, COMMERCIALSIZE+2);
01140
01141     MesPrint("%", AC.CheckpointFlag);
01142     MesPrint("%l", AC.CheckpointStamp);
01143     print_STR((unsigned char*)AC.CheckpointRunAfter);
01144     print_STR((unsigned char*)AC.CheckpointRunBefore);
01145     MesPrint("%l", AC.CheckpointInterval);
01146
01147     MesPrint("%%% END C_const");
01148     /* fflush(0); */
01149 }
01150
01151 static void print_R()
01152 {
01153     GETIDENTITY
01154     int i;
01155     MesPrint("%%% R_const");
01156     MesPrint("%l", (LONG)(AR.infile-AR.Fscr));
01157     MesPrint("%s", AR.infile->name);
01158     MesPrint("%l", (LONG)(AR.outfile-AR.Fscr));
01159     MesPrint("%s", AR.outfile->name);
01160     MesPrint("%l", AR.hidefile-AR.Fscr);
01161     MesPrint("%s", AR.hidefile->name);
01162     for ( i=0; i<3; ++i ) {
01163         MesPrint("FSCR %d", i);
01164         print_WORDB(AR.Fscr[i].PObuffer, AR.Fscr[i].POfull);
01165     }
01166     /* ... */
01167     MesPrint("%l", AR.OldTime);
01168     MesPrint("%l", AR.InInBuf);
01169     MesPrint("%l", AR.InHiBuf);
01170     MesPrint("%l", AR.pWorkSize);
01171     MesPrint("%l", AR.lWorkSize);
01172     MesPrint("%l", AR.posWorkSize);
01173     MesPrint("%d", AR.NoCompress);
01174     MesPrint("%d", AR.gzipCompress);
01175     MesPrint("%d", AR.Cnumlhs);
01176 #ifdef WITHPTHREADS
01177     MesPrint("%d", AR.exprtodo);
01178 #endif
01179     MesPrint("%d", AR.GetFile);
01180     MesPrint("%d", AR.KeptInHold);
01181     MesPrint("%d", AR.BraceOn);
01182     MesPrint("%d", AR.MaxBracket);
01183     MesPrint("%d", AR.CurDum);
01184     MesPrint("%d", AR.DeferFlag);
01185     MesPrint("%d", AR.TePos);
01186     MesPrint("%d", AR.sLevel);
01187     MesPrint("%d", AR.Stage4Name);
01188     MesPrint("%d", AR.GetOneFile);
01189     MesPrint("%d", AR.PolyFun);
01190     MesPrint("%d", AR.PolyFunInv);
01191     MesPrint("%d", AR.PolyFunType);
01192     MesPrint("%d", AR.PolyFunExp);
01193     MesPrint("%d", AR.PolyFunVar);
01194     MesPrint("%d", AR.PolyFunPow);
01195     MesPrint("%d", AR.Eside);
01196     MesPrint("%d", AR.MaxDum);
01197     MesPrint("%d", AR.level);
01198     MesPrint("%d", AR.expchanged);
01199     MesPrint("%d", AR.expflags);

```

```

01200     MesPrint("%d", AR.CurExpr);
01201     MesPrint("%d", AR.SortType);
01202     MesPrint("%d", AR.ShortSortCount);
01203     MesPrint("%%% END R_const");
01204 /*  fflush(0); */
01205 }
01206
01207 #endif /* ifdef PRINTDEBUG */
01208
01209 /*
01210     #] Debugging :
01211     #[ Cached file operation functions :
01212 */
01213
01214 #define CACHED_SNAPSHOT
01215
01216 #define CACHE_SIZE 4096
01217
01218 #ifndef CACHED_SNAPSHOT
01219 unsigned char cache_buffer[CACHE_SIZE];
01220 size_t cache_fill = 0;
01221
01222 size_t fwrite_cached(const void *ptr, size_t size, size_t nmemb, FILE *fd)
01223 {
01224     size_t fullsize = size*nmemb;
01225     if ( fullsize+cache_fill >= CACHE_SIZE ) {
01226         size_t overlap = CACHE_SIZE-cache_fill;
01227         memcpy(cache_buffer+cache_fill, (unsigned char*)ptr, overlap);
01228         if ( fwrite(cache_buffer, 1, CACHE_SIZE, fd) != CACHE_SIZE ) return 0;
01229         fullsize -= overlap;
01230         if ( fullsize >= CACHE_SIZE ) {
01231             cache_fill = fullsize % CACHE_SIZE;
01232             if ( cache_fill ) memcpy(cache_buffer, (unsigned char*)ptr+overlap+fullsize-cache_fill,
cache_fill);
01233             if ( fwrite((unsigned char*)ptr+overlap, 1, fullsize-cache_fill, fd) !=
fullsize-cache_fill ) return 0;
01234         }
01235         else {
01236             memcpy(cache_buffer, (unsigned char*)ptr+overlap, fullsize);
01237             cache_fill = fullsize;
01238         }
01239     }
01240     else {
01241         memcpy(cache_buffer+cache_fill, (unsigned char*)ptr, fullsize);
01242         cache_fill += fullsize;
01243     }
01244     return nmemb;
01245 }
01246
01247 size_t flush_cache(FILE *fd)
01248 {
01249     if ( cache_fill ) {
01250         size_t retval = fwrite(cache_buffer, 1, cache_fill, fd);
01251         if ( retval != cache_fill ) {
01252             cache_fill = 0;
01253             return 0;
01254         }
01255         cache_fill = 0;
01256     }
01257     return 1;
01258 }
01259 #else
01260 size_t fwrite_cached(const void *ptr, size_t size, size_t nmemb, FILE *fd)
01261 {
01262     return fwrite(ptr, size, nmemb, fd);
01263 }
01264
01265 size_t flush_cache(FILE *fd)
01266 {
01267     DUMMYUSE(fd)
01268     return 1;
01269 }
01270 #endif
01271
01272 /*
01273     #] Cached file operation functions :
01274     #[ Helper Macros :
01275 */
01276
01277 /* some helper macros to streamline the code in DoSnapshot() and DoRecovery() */
01278
01279 /* freeing memory */
01280
01281 #define R_FREE(ARG) \
01282     if ( ARG ) M_free(ARG, #ARG);
01283
01284 #define R_FREE_NAMETREE(ARG) \

```

```

01285     R_FREE(ARG->namenode); \
01286     R_FREE(ARG->namebuffer); \
01287     R_FREE(ARG);
01288
01289 #define R_FREE_STREAM(ARG) \
01290     R_FREE(ARG.buffer); \
01291     R_FREE(ARG.FoldName); \
01292     R_FREE(ARG.name);
01293
01294 /* reading a single variable */
01295
01296 #define R_SET(VAR,TYPE) \
01297     VAR = *((TYPE*)p); p = (unsigned char*)p + sizeof(TYPE);
01298
01299 /* general buffer */
01300
01301 #define R_COPY_B(VAR,SIZE,CAST) \
01302     VAR = (CAST)Malloc1(SIZE,#VAR); \
01303     memcpy(VAR, p, SIZE); p = (unsigned char*)p + SIZE;
01304
01305 #define S_WRITE_B(BUF,LEN) \
01306     if ( fwrite_cached(BUF, 1, LEN, fd) != (size_t)(LEN) ) return(__LINE__);
01307
01308 #define S_FLUSH_B \
01309     if ( flush_cache(fd) != 1 ) return(__LINE__);
01310
01311 /* character strings */
01312
01313 #define R_COPY_S(VAR,CAST) \
01314     if ( VAR ) { \
01315         VAR = (CAST)Malloc1(strlen(p)+1,"R_COPY_S"); \
01316         strcpy((char*)VAR, p); p = (unsigned char*)p + strlen(p) + 1; \
01317     }
01318
01319 #define S_WRITE_S(STR) \
01320     if ( STR ) { \
01321         l = strlen((char*)STR) + 1; \
01322         if ( fwrite_cached(STR, 1, l, fd) != (size_t)l ) return(__LINE__); \
01323     }
01324
01325 /* LIST */
01326
01327 #define R_COPY_LIST(ARG) \
01328     if ( ARG.maxnum ) { \
01329         R_COPY_B(ARG.lijst, ARG.size*ARG.maxnum, void*) \
01330     }
01331
01332 #define S_WRITE_LIST(LST) \
01333     if ( LST.maxnum ) { \
01334         S_WRITE_B((char*)LST.lijst, LST.maxnum*LST.size) \
01335     }
01336
01337 /* NAMETREE */
01338
01339 #define R_COPY_NAMETREE(ARG) \
01340     R_COPY_B(ARG, sizeof(NAMETREE), NAMETREE*); \
01341     if ( ARG->namenode ) { \
01342         R_COPY_B(ARG->namenode, ARG->nodesize*sizeof(NAMENODE), NAMENODE*); \
01343     } \
01344     if ( ARG->namebuffer ) { \
01345         R_COPY_B(ARG->namebuffer, ARG->namesize, UBYTE*); \
01346     }
01347
01348 #define S_WRITE_NAMETREE(ARG) \
01349     S_WRITE_B(ARG, sizeof(NAMETREE)); \
01350     if ( ARG->namenode ) { \
01351         S_WRITE_B(ARG->namenode, ARG->nodesize*sizeof(struct NaMeNode)); \
01352     } \
01353     if ( ARG->namebuffer ) { \
01354         S_WRITE_B(ARG->namebuffer, ARG->namesize); \
01355     }
01356
01357 /* DOLLAR */
01358
01359 #define S_WRITE_DOLLAR(ARG) \
01360     if ( ARG.size && ARG.where && ARG.where != &(AM.dollarzero) ) { \
01361         S_WRITE_B(ARG.where, ARG.size*sizeof(WORD)) \
01362     }
01363
01364 /* Printing time marks with ANNOUNCE macro */
01365
01366 #ifdef PRINTTIMEMARKS
01367     time_t announce_time;
01368     #define ANNOUNCE(str) time(&announce_time); MesPrint("TIMEMARK %s %s", ctime(&announce_time), #str);
01369     #else
01370     #define ANNOUNCE(str)
01371 #endif

```

```

01372
01373 /*
01374 #] Helper Macros :
01375 #[ DoRecovery :
01376 */
01377
01402 int DoRecovery(int *moduletype)
01403 {
01404     GETIDENTITY
01405     FILE *fd;
01406     POSITION pos;
01407     void *buf, *p;
01408     LONG size, l;
01409     int i, j;
01410     UBYTE *org;
01411     char *namebufout, *namebufhide;
01412     LONG ofs;
01413     void *oldAMdollarzero;
01414     LIST PotModDolListBackup;
01415     LIST ModOptDolListBackup;
01416     WORD oldLogHandle;
01417
01418     MesPrint("Recovering ... %"); fflush(0);
01419
01420     if ( !(fd = fopen(recoveryfile, "r")) ) return(__LINE__);
01421
01422     /* load the complete recovery file into a buffer */
01423     if ( fread(&pos, sizeof(POSITION), 1, fd) != 1 ) return(__LINE__);
01424     size = BASEPOSITION(pos) - sizeof(POSITION);
01425     buf = Malloc1(size, "recovery buffer");
01426     if ( fread(buf, size, 1, fd) != 1 ) return(__LINE__);
01427
01428     /* pointer p will go through the buffer in the following */
01429     p = buf;
01430
01431     /* read moduletype */
01432     R_SET(*moduletype, int);
01433
01434     /**[ AM : */
01435
01436     /* only certain elements will be restored. the rest of AM should have gotten
01437      * the correct values at startup. */
01438
01439     R_SET(AM.hparallelflag, int);
01440     R_SET(AM.gparallelflag, int);
01441     R_SET(AM.gCodesFlag, int);
01442     R_SET(AM.gNamesFlag, int);
01443     R_SET(AM.gStatsFlag, int);
01444     R_SET(AM.gTokensWriteFlag, int);
01445     R_SET(AM.gNoSpacesInNumbers, int);
01446     R_SET(AM.gIndentSpace, WORD);
01447     R_SET(AM.gUnitTrace, WORD);
01448     R_SET(AM.gDefDim, int);
01449     R_SET(AM.gDefDim4, int);
01450     R_SET(AM.gncmod, WORD);
01451     R_SET(AM.gnpowmod, WORD);
01452     R_SET(AM.gmodmode, WORD);
01453     R_SET(AM.gOutputMode, WORD);
01454     R_SET(AM.gCnumpows, WORD);
01455     R_SET(AM.gOutputSpaces, WORD);
01456     R_SET(AM.gOutNumberType, WORD);
01457     R_SET(AM.gfunpowers, int);
01458     R_SET(AM.gPolyFun, WORD);
01459     R_SET(AM.gPolyFunInv, WORD);
01460     R_SET(AM.gPolyFunType, WORD);
01461     R_SET(AM.gPolyFunExp, WORD);
01462     R_SET(AM.gPolyFunVar, WORD);
01463     R_SET(AM.gPolyFunPow, WORD);
01464     R_SET(AM.gProcessBucketSize, LONG);
01465     R_SET(AM.OldChildTime, LONG);
01466     R_SET(AM.OldSecTime, LONG);
01467     R_SET(AM.OldMilliTime, LONG);
01468     R_SET(AM.gproperorderflag, int);
01469     R_SET(AM.gThreadBucketSize, LONG);
01470     R_SET(AM.gSizeCommuteInSet, int);
01471     R_SET(AM.gThreadStats, int);
01472     R_SET(AM.gFinalStats, int);
01473     R_SET(AM.gThreadsFlag, int);
01474     R_SET(AM.gThreadBalancing, int);
01475     R_SET(AM.gThreadSortFileSynch, int);
01476     R_SET(AM.gProcessStats, int);
01477     R_SET(AM.gOldParallelStats, int);
01478     R_SET(AM.gSortType, int);
01479     R_SET(AM.gShortStatsMax, WORD);
01480     R_SET(AM.gIsFortran90, int);
01481     R_SET(oldAMdollarzero, void*);
01482     R_FREE(AM.gFortran90Kind);

```

```

01483     R_SET(AM.gFortran90Kind, UBYTE *);
01484     R_COPY_S(AM.gFortran90Kind, UBYTE *);
01485
01486     R_COPY_S(AM.gextrasy, UBYTE *);
01487     R_COPY_S(AM.ggextrasy, UBYTE *);
01488
01489     R_SET(AM.PrintTotalSize, int);
01490     R_SET(AM.fbufferSize, int);
01491     R_SET(AM.gOldFactArgFlag, int);
01492     R_SET(AM.ggOldFactArgFlag, int);
01493
01494     R_SET(AM.gnumextrasy, int);
01495     R_SET(AM.ggnumextrasy, int);
01496     R_SET(AM.NumSpectatorFiles, int);
01497     R_SET(AM.SizeForSpectatorFiles, int);
01498     R_SET(AM.gOldGCDflag, int);
01499     R_SET(AM.ggOldGCDflag, int);
01500     R_SET(AM.gWTimeStatsFlag, int);
01501
01502     R_FREE(AM.Path);
01503     R_SET(AM.Path, UBYTE *);
01504     R_COPY_S(AM.Path, UBYTE *);
01505
01506     R_SET(AM.FromStdin, BOOL);
01507     R_SET(AM.IgnoreDeprecation, BOOL);
01508
01509 #ifdef PRINTDEBUG
01510     print_M();
01511 #endif
01512
01513     /*#] AM : */
01514     /*#[ AC : */
01515
01516     /* #[ AC free pointers */
01517
01518     /* AC will be overwritten by data from the recovery file, therefore
01519      * dynamically allocated memory must be freed first. */
01520
01521     R_FREE_NAMETREE(AC.dollarnames);
01522     R_FREE_NAMETREE(AC.exprnames);
01523     R_FREE_NAMETREE(AC.varnames);
01524     for ( i=0; i<AC.ChannellList.num; ++i ) {
01525         R_FREE(channels[i].name);
01526     }
01527     R_FREE(AC.ChannellList.lijst);
01528     R_FREE(AC.DubiousList.lijst);
01529     for ( i=0; i<AC.FunctionList.num; ++i ) {
01530         TABLES T = functions[i].tabl;
01531         if ( T ) {
01532             R_FREE(T->buffers);
01533             R_FREE(T->mm);
01534             R_FREE(T->flags);
01535             R_FREE(T->prototype);
01536             R_FREE(T->tablepointers);
01537             if ( T->sparse ) {
01538                 R_FREE(T->boomlijst);
01539                 R_FREE(T->argtail);
01540             }
01541             if ( T->spare ) {
01542                 R_FREE(T->spare->buffers);
01543                 R_FREE(T->spare->mm);
01544                 R_FREE(T->spare->flags);
01545                 R_FREE(T->spare->tablepointers);
01546                 if ( T->spare->sparse ) {
01547                     R_FREE(T->spare->boomlijst);
01548                 }
01549                 R_FREE(T->spare);
01550             }
01551             R_FREE(T);
01552         }
01553     }
01554     R_FREE(AC.FunctionList.lijst);
01555     for ( i=0; i<AC.ExpressionList.num; ++i ) {
01556         if ( Expressions[i].renum ) {
01557             R_FREE(Expressions[i].renum->symb.lo);
01558             R_FREE(Expressions[i].renum);
01559         }
01560         if ( Expressions[i].bracketinfo ) {
01561             R_FREE(Expressions[i].bracketinfo->indexbuffer);
01562             R_FREE(Expressions[i].bracketinfo->bracketbuffer);
01563             R_FREE(Expressions[i].bracketinfo);
01564         }
01565         if ( Expressions[i].newbracketinfo ) {
01566             R_FREE(Expressions[i].newbracketinfo->indexbuffer);
01567             R_FREE(Expressions[i].newbracketinfo->bracketbuffer);
01568             R_FREE(Expressions[i].newbracketinfo);
01569         }

```

```

01570         if ( Expressions[i].renumlists != AN.dummyrenumlist ) {
01571             R_FREE(Expressions[i].renumlists);
01572         }
01573         R_FREE(Expressions[i].inmem);
01574     }
01575     R_FREE(AC.ExpressionList.lijst);
01576     R_FREE(AC.IndexList.lijst);
01577     R_FREE(AC.SetElementList.lijst);
01578     R_FREE(AC.SetList.lijst);
01579     R_FREE(AC.SymbolList.lijst);
01580     R_FREE(AC.VectorList.lijst);
01581     for ( i=0; i<AC.TableBaseList.num; ++i ) {
01582         R_FREE(tablebases[i].iblocks);
01583         R_FREE(tablebases[i].nblocks);
01584         R_FREE(tablebases[i].name);
01585         R_FREE(tablebases[i].fullname);
01586         R_FREE(tablebases[i].tablenames);
01587     }
01588     R_FREE(AC.TableBaseList.lijst);
01589     for ( i=0; i<AC.cbufList.num; ++i ) {
01590         R_FREE(cbuf[i].Buffer);
01591         R_FREE(cbuf[i].lhs);
01592         R_FREE(cbuf[i].rhs);
01593         R_FREE(cbuf[i].boomlijst);
01594     }
01595     R_FREE(AC.cbufList.lijst);
01596     R_FREE(AC.AutoSymbolList.lijst);
01597     R_FREE(AC.AutoIndexList.lijst);
01598     R_FREE(AC.AutoVectorList.lijst);
01599     /* Tables cannot be auto-declared, therefore no extra code here */
01600     R_FREE(AC.AutoFunctionList.lijst);
01601     R_FREE_NAMETREE(AC.autonames);
01602     for ( i=0; i<AC.NumStreams; ++i ) {
01603         R_FREE_STREAM(AC.Streams[i]);
01604     }
01605     R_FREE(AC.Streams);
01606     R_FREE(AC.termstack);
01607     R_FREE(AC.termstortstack);
01608     R_FREE(AC.cmod);
01609     R_FREE(AC.modpowers);
01610     R_FREE(AC.halfmod);
01611     R_FREE(AC.IfHeap);
01612     R_FREE(AC.IfCount);
01613     R_FREE(AC.iBuffer);
01614     for ( i=0; i<AC.NumLabels; ++i ) {
01615         R_FREE(AC.LabelNames[i]);
01616     }
01617     R_FREE(AC.LabelNames);
01618     R_FREE(AC.FixIndices);
01619     R_FREE(AC.termsumcheck);
01620     R_FREE(AC.WildcardNames);
01621     R_FREE(AC.tokens);
01622     R_FREE(AC.tokenarglevel);
01623     R_FREE(AC.modinverses);
01624     R_FREE(AC.Fortran90Kind);
01625 #ifdef WITHPTHREADS
01626     R_FREE(AC.inputnumbers);
01627 #endif
01628     R_FREE(AC.IfSumCheck);
01629     R_FREE(AC.CommuteInSet);
01630     R_FREE(AC.CheckpointRunAfter);
01631     R_FREE(AC.CheckpointRunBefore);
01632
01633     /* #] AC free pointers */
01634
01635     /* backup some lists in order to restore it to the initial setup */
01636     PotModDolListBackup = AC.PotModDolList;
01637     ModOptDolListBackup = AC.ModOptDolList;
01638     oldLogHandle = AC.LogHandle;
01639
01640     /* first we copy AC as a whole and then restore the pointer structures step
01641        by step. */
01642
01643     AC = *((struct C_const*)p); p = (unsigned char*)p + sizeof(struct C_const);
01644
01645     R_COPY_NAMETREE(AC.dollarnames);
01646     R_COPY_NAMETREE(AC.exprnames);
01647     R_COPY_NAMETREE(AC.varnames);
01648
01649     R_COPY_LIST(AC.ChannelList);
01650     for ( i=0; i<AC.ChannelList.num; ++i ) {
01651         R_COPY_S(channels[i].name, char*);
01652         channels[i].handle = ReOpenFile(channels[i].name);
01653     }
01654     AC.ChannelList.message = "channel buffer";
01655
01656     AC.DubiousList.lijst = 0;

```

```

01657     AC.DubiousList.message = "ambiguous variable";
01658     AC.DubiousList.num =
01659     AC.DubiousList.maxnum =
01660     AC.DubiousList.numglobal =
01661     AC.DubiousList.numtemp =
01662     AC.DubiousList.numclear = 0;
01663
01664     R_COPY_LIST(AC.FunctionList);
01665     for ( i=0; i<AC.FunctionList.num; ++i ) {
01666         if ( functions[i].tabl ) {
01667             TABLES tabl;
01668             R_COPY_B(tabl, sizeof(struct TableEs), TABLES);
01669             functions[i].tabl = tabl;
01670             if ( tabl->tablepointers ) {
01671                 if ( tabl->sparse ) {
01672                     R_COPY_B(tabl->tablepointers,
01673                             tabl->reserved*sizeof(WORD)*(tabl->numind+TABLEEXTENSION),
01674                             WORD*);
01675                 }
01676                 else {
01677                     R_COPY_B(tabl->tablepointers,
01678                             TABLEEXTENSION*sizeof(WORD)*(tabl->totind), WORD*);
01679                 }
01680             }
01681             org = (UBYTE*)tabl->prototype;
01682             #ifdef WITHPTHREADS
01683             R_COPY_B(tabl->prototype, tabl->prototypeSize, WORD**);
01684             ofs = (UBYTE*)tabl->prototype - org;
01685             for ( j=0; j<AM.totalnumberofthreads; ++j ) {
01686                 if ( tabl->prototype[j] ) {
01687                     tabl->prototype[j] = (WORD*)((UBYTE*)tabl->prototype[j] + ofs);
01688                 }
01689             }
01690             if ( tabl->pattern ) {
01691                 tabl->pattern = (WORD*)((UBYTE*)tabl->pattern + ofs);
01692                 for ( j=0; j<AM.totalnumberofthreads; ++j ) {
01693                     if ( tabl->pattern[j] ) {
01694                         tabl->pattern[j] = (WORD*)((UBYTE*)tabl->pattern[j] + ofs);
01695                     }
01696                 }
01697             }
01698             #else
01699             ofs = tabl->pattern - tabl->prototype;
01700             R_COPY_B(tabl->prototype, tabl->prototypeSize, WORD*);
01701             if ( tabl->pattern ) {
01702                 tabl->pattern = tabl->prototype + ofs;
01703             }
01704             #endif
01705             R_COPY_B(tabl->mm, tabl->numind*(LONG)sizeof(MINMAX), MINMAX*);
01706             R_COPY_B(tabl->flags, tabl->numind*(LONG)sizeof(WORD), WORD*);
01707             if ( tabl->sparse ) {
01708                 R_COPY_B(tabl->boomlijst, tabl->MaxTreeSize*(LONG)sizeof(COMPTREE), COMPTREE*);
01709                 R_COPY_S(tabl->argtail, UBYTE*);
01710             }
01711             R_COPY_B(tabl->buffers, tabl->buffersize*(LONG)sizeof(WORD), WORD*);
01712             if ( tabl->spare ) {
01713                 TABLES spare;
01714                 R_COPY_B(spare, sizeof(struct TableEs), TABLES);
01715                 tabl->spare = spare;
01716                 if ( spare->tablepointers ) {
01717                     if ( spare->sparse ) {
01718                         R_COPY_B(spare->tablepointers,
01719                                 spare->reserved*sizeof(WORD)*(spare->numind+TABLEEXTENSION),
01720                                 WORD*);
01721                     }
01722                     else {
01723                         R_COPY_B(spare->tablepointers,
01724                                 TABLEEXTENSION*sizeof(WORD)*(spare->totind), WORD*);
01725                     }
01726                 }
01727                 spare->prototype = tabl->prototype;
01728                 spare->pattern = tabl->pattern;
01729                 R_COPY_B(spare->mm, spare->numind*(LONG)sizeof(MINMAX), MINMAX*);
01730                 R_COPY_B(spare->flags, spare->numind*(LONG)sizeof(WORD), WORD*);
01731                 if ( tabl->sparse ) {
01732                     R_COPY_B(spare->boomlijst, spare->MaxTreeSize*(LONG)sizeof(COMPTREE), COMPTREE*);
01733                     spare->argtail = tabl->argtail;
01734                 }
01735                 spare->spare = tabl;
01736                 R_COPY_B(spare->buffers, spare->buffersize*(LONG)sizeof(WORD), WORD*);
01737             }
01738         }
01739     }
01740     AC.FunctionList.message = "function";
01741
01742     R_COPY_LIST(AC.ExpressionList);
01743     for ( i=0; i<AC.ExpressionList.num; ++i ) {

```

```

01744     EXPRESSIONS ex = Expressions + i;
01745     if ( ex->renum ) {
01746         R_COPY_B(ex->renum, sizeof(struct ReNuMbEr), RENUMBER);
01747         org = (UBYTE*)ex->renum->symb.lo;
01748         R_SET(size, size_t);
01749         R_COPY_B(ex->renum->symb.lo, size, WORD*);
01750         ofs = (UBYTE*)ex->renum->symb.lo - org;
01751         ex->renum->symb.start = (WORD*) ((UBYTE*)ex->renum->symb.start + ofs);
01752         ex->renum->symb.hi = (WORD*) ((UBYTE*)ex->renum->symb.hi + ofs);
01753         ex->renum->indi.lo = (WORD*) ((UBYTE*)ex->renum->indi.lo + ofs);
01754         ex->renum->indi.start = (WORD*) ((UBYTE*)ex->renum->indi.start + ofs);
01755         ex->renum->indi.hi = (WORD*) ((UBYTE*)ex->renum->indi.hi + ofs);
01756         ex->renum->vect.lo = (WORD*) ((UBYTE*)ex->renum->vect.lo + ofs);
01757         ex->renum->vect.start = (WORD*) ((UBYTE*)ex->renum->vect.start + ofs);
01758         ex->renum->vect.hi = (WORD*) ((UBYTE*)ex->renum->vect.hi + ofs);
01759         ex->renum->func.lo = (WORD*) ((UBYTE*)ex->renum->func.lo + ofs);
01760         ex->renum->func.start = (WORD*) ((UBYTE*)ex->renum->func.start + ofs);
01761         ex->renum->func.hi = (WORD*) ((UBYTE*)ex->renum->func.hi + ofs);
01762         ex->renum->symnum = (WORD*) ((UBYTE*)ex->renum->symnum + ofs);
01763         ex->renum->indnum = (WORD*) ((UBYTE*)ex->renum->indnum + ofs);
01764         ex->renum->vecnum = (WORD*) ((UBYTE*)ex->renum->vecnum + ofs);
01765         ex->renum->funnum = (WORD*) ((UBYTE*)ex->renum->funnum + ofs);
01766     }
01767     if ( ex->bracketinfo ) {
01768         R_COPY_B(ex->bracketinfo, sizeof(BRACKETINFO), BRACKETINFO*);
01769         R_COPY_B(ex->bracketinfo->indexbuffer,
01770 ex->bracketinfo->indexbuffersize*sizeof(BRACKETINDEX), BRACKETINDEX*);
01771         R_COPY_B(ex->bracketinfo->bracketbuffer, ex->bracketinfo->bracketbuffersize*sizeof(WORD),
01772 WORD*);
01773     }
01774     if ( ex->newbracketinfo ) {
01775         R_COPY_B(ex->newbracketinfo, sizeof(BRACKETINFO), BRACKETINFO*);
01776         R_COPY_B(ex->newbracketinfo->indexbuffer,
01777 ex->newbracketinfo->indexbuffersize*sizeof(BRACKETINDEX), BRACKETINDEX*);
01778         R_COPY_B(ex->newbracketinfo->bracketbuffer,
01779 ex->newbracketinfo->bracketbuffersize*sizeof(WORD), WORD*);
01780     }
01781 #ifdef WITHPTHREADS
01782     ex->renumlists = 0;
01783 #else
01784     ex->renumlists = AN.dummyrenumlist;
01785 #endif
01786     if ( ex->inmem ) {
01787         R_SET(size, size_t);
01788         R_COPY_B(ex->inmem, size, WORD*);
01789     }
01790     AC.ExpressionList.message = "expression";
01791     R_COPY_LIST(AC.IndexList);
01792     AC.IndexList.message = "index";
01793     R_COPY_LIST(AC.SetElementList);
01794     AC.SetElementList.message = "set element";
01795     R_COPY_LIST(AC.SetList);
01796     AC.SetList.message = "set";
01797     R_COPY_LIST(AC.SymbolList);
01798     AC.SymbolList.message = "symbol";
01799     R_COPY_LIST(AC.VectorList);
01800     AC.VectorList.message = "vector";
01801     AC.PotModDolList = PotModDolListBackup;
01802     AC.ModOptDolList = ModOptDolListBackup;
01803     R_COPY_LIST(AC.TableBaseList);
01804     for ( i=0; i<AC.TableBaseList.num; ++i ) {
01805         if ( tablebases[i].iblocks ) {
01806             R_COPY_B(tablebases[i].iblocks,
01807 tablebases[i].info.numberofindexblocks*sizeof(INDEXBLOCK*), INDEXBLOCK**);
01808             for ( j=0; j<tablebases[i].info.numberofindexblocks; ++j ) {
01809                 if ( tablebases[i].iblocks[j] ) {
01810                     R_COPY_B(tablebases[i].iblocks[j], sizeof(INDEXBLOCK), INDEXBLOCK*);
01811                 }
01812             }
01813             if ( tablebases[i].nblocks ) {
01814                 R_COPY_B(tablebases[i].nblocks,
01815 tablebases[i].info.numberofnamesblocks*sizeof(NAMESBLOCK*), NAMESBLOCK**);
01816                 for ( j=0; j<tablebases[i].info.numberofindexblocks; ++j ) {
01817                     if ( tablebases[i].nblocks[j] ) {
01818                         R_COPY_B(tablebases[i].nblocks[j], sizeof(NAMESBLOCK), NAMESBLOCK*);
01819                     }
01820                 }
01821             }
01822             /* reopen file */
01823             if ( ( tablebases[i].handle = fopen(tablebases[i].fullname, "r+b") ) == NULL ) {
01824                 MesPrint("ERROR: Could not reopen tablebase %s!", tablebases[i].name);
01825                 Terminate(-1);
01826             }
01827         }
01828     }

```



```

01825     }
01826     R_COPY_S(tablebases[i].name,char*);
01827     R_COPY_S(tablebases[i].fullname,char*);
01828     R_COPY_S(tablebases[i].tablenames,char*);
01829 }
01830 AC.TableBaseList.message = "list of tablebases";
01831
01832 R_COPY_LIST(AC.cbufList);
01833 for ( i=0; i<AC.cbufList.num; ++i ) {
01834     org = (UBYTE*)cbuf[i].Buffer;
01835     R_COPY_B(cbuf[i].Buffer, cbuf[i].BufferSize*sizeof(WORD), WORD*);
01836     ofs = (UBYTE*)cbuf[i].Buffer - org;
01837     cbuf[i].Top = (WORD*)((UBYTE*)cbuf[i].Top + ofs);
01838     cbuf[i].Pointer = (WORD*)((UBYTE*)cbuf[i].Pointer + ofs);
01839     R_COPY_B(cbuf[i].lhs, cbuf[i].maxlhs*(LONG)sizeof(WORD*), WORD**);
01840     for ( j=1; j<=cbuf[i].numlhs; ++j ) {
01841         if ( cbuf[i].lhs[j] ) cbuf[i].lhs[j] = (WORD*)((UBYTE*)cbuf[i].lhs[j] + ofs);
01842     }
01843     org = (UBYTE*)cbuf[i].rhs;
01844     R_COPY_B(cbuf[i].rhs, cbuf[i].maxrhs*(LONG)(sizeof(WORD*)+2*sizeof(LONG)+2*sizeof(WORD)),
WORD**);
01845     for ( j=1; j<=cbuf[i].numrhs; ++j ) {
01846         if ( cbuf[i].rhs[j] ) cbuf[i].rhs[j] = (WORD*)((UBYTE*)cbuf[i].rhs[j] + ofs);
01847     }
01848     ofs = (UBYTE*)cbuf[i].rhs - org;
01849     cbuf[i].CanCommu = (LONG*)((UBYTE*)cbuf[i].CanCommu + ofs);
01850     cbuf[i].NumTerms = (LONG*)((UBYTE*)cbuf[i].NumTerms + ofs);
01851     cbuf[i].numdum = (WORD*)((UBYTE*)cbuf[i].numdum + ofs);
01852     cbuf[i].dimension = (WORD*)((UBYTE*)cbuf[i].dimension + ofs);
01853     if ( cbuf[i].boomlijst ) {
01854         R_COPY_B(cbuf[i].boomlijst, cbuf[i].MaxTreeSize*sizeof(COMPTREE), COMPTREE*);
01855     }
01856 }
01857 AC.cbufList.message = "compiler buffer";
01858
01859 R_COPY_LIST(AC.AutoSymbolList);
01860 AC.AutoSymbolList.message = "autosymbol";
01861 R_COPY_LIST(AC.AutoIndexList);
01862 AC.AutoIndexList.message = "autoindex";
01863 R_COPY_LIST(AC.AutoVectorList);
01864 AC.AutoVectorList.message = "autovector";
01865 R_COPY_LIST(AC.AutoFunctionList);
01866 AC.AutoFunctionList.message = "autofunction";
01867
01868 R_COPY_NAMETREE(AC.autonames);
01869
01870 AC.Symbols = &(AC.SymbolList);
01871 AC.Indices = &(AC.IndexList);
01872 AC.Vectors = &(AC.VectorList);
01873 AC.Functions = &(AC.FunctionList);
01874 AC.activenames = &(AC.varnames);
01875
01876 org = (UBYTE*)AC.Streams;
01877 R_COPY_B(AC.Streams, AC.MaxNumStreams*(LONG)sizeof(STREAM), STREAM*);
01878 for ( i=0; i<AC.NumStreams; ++i ) {
01879     if ( AC.Streams[i].type != FILESTREAM ) {
01880         UBYTE *org2;
01881         org2 = AC.Streams[i].buffer;
01882         if ( AC.Streams[i].inbuffer ) {
01883             R_COPY_B(AC.Streams[i].buffer, AC.Streams[i].inbuffer, UBYTE*);
01884         }
01885         ofs = AC.Streams[i].buffer - org2;
01886         AC.Streams[i].pointer += ofs;
01887         AC.Streams[i].top += ofs;
01888     }
01889     else {
01890         p = (unsigned char*)p + AC.Streams[i].inbuffer;
01891     }
01892     AC.Streams[i].buffersize = AC.Streams[i].inbuffer;
01893     R_COPY_S(AC.Streams[i].FoldName,UBYTE*);
01894     R_COPY_S(AC.Streams[i].name,UBYTE*);
01895     if ( AC.Streams[i].type == PREVARSTREAM || AC.Streams[i].type == DOLLARSTREAM ) {
01896         AC.Streams[i].pname = AC.Streams[i].name;
01897     }
01898     else if ( AC.Streams[i].type == FILESTREAM ) {
01899         UBYTE *org2;
01900         org2 = AC.Streams[i].buffer;
01901         AC.Streams[i].buffer = (UBYTE*)Malloc1(AC.Streams[i].buffersize, "buffer");
01902         ofs = AC.Streams[i].buffer - org2;
01903         AC.Streams[i].pointer += ofs;
01904         AC.Streams[i].top += ofs;
01905
01906         /* open file except for already opened main input file */
01907         if ( i ) {
01908             AC.Streams[i].handle = OpenFile((char *) (AC.Streams[i].name));
01909             if ( AC.Streams[i].handle == -1 ) {
01910                 MesPrint("ERROR: Could not reopen stream %s!",AC.Streams[i].name);

```

```

01911         Terminate(-1);
01912     }
01913 }
01914
01915     PUTZERO(pos);
01916     ADDPOS(pos, AC.Streams[i].bufferposition);
01917     SeekFile(AC.Streams[i].handle, &pos, SEEK_SET);
01918
01919     AC.Streams[i].inbuffer = ReadFile(AC.Streams[i].handle, AC.Streams[i].buffer,
AC.Streams[i].inbuffer);
01920
01921     SETBASEPOSITION(pos, AC.Streams[i].fileposition);
01922     SeekFile(AC.Streams[i].handle, &pos, SEEK_SET);
01923 }
01924 /*
01925  * Ideally, we should check if we have a type PREREADSTREAM, PREREADSTREAM2, and
01926  * PRECALCSTREAM here. If so, we should free element name and point it
01927  * to the name element of the embracing stream's struct. In practice,
01928  * this is undoable without adding new data to STREAM. Since we create
01929  * only a small memory leak here (some few byte for each existing
01930  * stream of these types) and only once when we do a recovery, we
01931  * tolerate this leak and keep it STREAM as it is.
01932  */
01933 }
01934 ofs = (UBYTE*)AC.Streams - org;
01935 AC.CurrentStream = (STREAM*)((UBYTE*)AC.CurrentStream + ofs);
01936
01937 if ( AC.termstack ) {
01938     R_COPY_B(AC.termstack, AC.maxtermlevel*(LONG)sizeof(LONG), LONG*);
01939 }
01940
01941 if ( AC.termstack ) {
01942     R_COPY_B(AC.termstack, AC.maxtermlevel*(LONG)sizeof(LONG), LONG*);
01943 }
01944
01945 /* exception: here we also change values from struct AM */
01946 R_COPY_B(AC.cmod, AM.MaxTal*4*(LONG)sizeof(WORD), WORD*);
01947 AM.gcmmod = AC.cmod + AM.MaxTal;
01948 AC.powmod = AM.gcmmod + AM.MaxTal;
01949 AM.gpowmod = AC.powmod + AM.MaxTal;
01950
01951 AC.modpowers = 0;
01952 AC.halfmod = 0;
01953
01954 /* we don't care about AC.ProtoType/WildC */
01955
01956 if ( AC.IfHeap ) {
01957     ofs = AC.IfStack - AC.IfHeap;
01958     R_COPY_B(AC.IfHeap, (LONG)sizeof(LONG)*(AC.MaxIf+1), LONG*);
01959     AC.IfStack = AC.IfHeap + ofs;
01960     R_COPY_B(AC.IfCount, (LONG)sizeof(LONG)*(AC.MaxIf+1), LONG*);
01961 }
01962
01963 org = AC.iBuffer;
01964 size = AC.iStop - AC.iBuffer + 2;
01965 R_COPY_B(AC.iBuffer, size, UBYTE*);
01966 ofs = AC.iBuffer - org;
01967 AC.iPointer += ofs;
01968 AC.iStop += ofs;
01969
01970 if ( AC.LabelNames ) {
01971     org = (UBYTE*)AC.LabelNames;
01972     R_COPY_B(AC.LabelNames, AC.MaxLabels*(LONG)(sizeof(UBYTE*)+sizeof(WORD)), UBYTE*);
01973     for ( i=0; i<AC.NumLabels; ++i ) {
01974         R_COPY_S(AC.LabelNames[i], UBYTE*);
01975     }
01976     ofs = (UBYTE*)AC.LabelNames - org;
01977     AC.Labels = (int*)((UBYTE*)AC.Labels + ofs);
01978 }
01979
01980 R_COPY_B(AC.FixIndices, AM.OffsetIndex*(LONG)sizeof(WORD), WORD*);
01981
01982 if ( AC.termsumcheck ) {
01983     R_COPY_B(AC.termsumcheck, AC.maxtermlevel*(LONG)sizeof(WORD), WORD*);
01984 }
01985
01986 R_COPY_B(AC.WildcardNames, AC.WildcardBufferSize, UBYTE*);
01987
01988 if ( AC.tokens ) {
01989     size = AC.toptokens - AC.tokens;
01990     if ( size ) {
01991         org = (UBYTE*)AC.tokens;
01992         R_COPY_B(AC.tokens, size, SBYTE*);
01993         ofs = (UBYTE*)AC.tokens - org;
01994         AC.endoftokens += ofs;
01995         AC.toptokens += ofs;
01996     }

```

```

01997         else {
01998             AC.tokens = 0;
01999             AC.endoftokens = AC.tokens;
02000             AC.toptokens = AC.tokens;
02001         }
02002     }
02003
02004     R_COPY_B(AC.tokenarglevel, AM.MaxParLevel*(LONG)sizeof(WORD), WORD*);
02005
02006     AC.modinverses = 0;
02007
02008     R_COPY_S(AC.Fortran90Kind, UBYTE *);
02009
02010 #ifdef WITHPTHREADS
02011     if ( AC.inputnumbers ) {
02012         org = (UBYTE*)AC.inputnumbers;
02013         R_COPY_B(AC.inputnumbers, AC.sizepfirstnum*(LONG)(sizeof(WORD)+sizeof(LONG)), LONG*);
02014         ofs = (UBYTE*)AC.inputnumbers - org;
02015         AC.pfirstnum = (WORD*)((UBYTE*)AC.pfirstnum + ofs);
02016     }
02017     AC.halfmodlock = dummylock;
02018 #endif /* ifdef WITHPTHREADS */
02019
02020     if ( AC.IfSumCheck ) {
02021         R_COPY_B(AC.IfSumCheck, (LONG)sizeof(WORD)*(AC.MaxIf+1), WORD*);
02022     }
02023     if ( AC.CommuteInSet ) {
02024         R_COPY_B(AC.CommuteInSet, (LONG)sizeof(WORD)*(AC.SizeCommuteInSet+1), WORD*);
02025     }
02026
02027     AC.LogHandle = oldLogHandle;
02028
02029     R_COPY_S(AC.CheckpointRunAfter, char*);
02030     R_COPY_S(AC.CheckpointRunBefore, char*);
02031
02032     R_COPY_S(AC.extrasym, UBYTE *);
02033
02034 #ifdef PRINTDEBUG
02035     print_C();
02036 #endif
02037
02038     /*#] AC : */
02039     /*#[ AP : */
02040
02041     /* #[ AP free pointers */
02042
02043     /* AP will be overwritten by data from the recovery file, therefore
02044      * dynamically allocated memory must be freed first. */
02045
02046     for ( i=0; i<AP.DollarList.num; ++i ) {
02047         if ( Dollars[i].size ) {
02048             R_FREE(Dollars[i].where);
02049         }
02050         CleanDollarFactors(Dollars+i);
02051     }
02052     R_FREE(AP.DollarList.lijst);
02053
02054     for ( i=0; i<AP.PreVarList.num; ++i ) {
02055         R_FREE(PreVar[i].name);
02056     }
02057     R_FREE(AP.PreVarList.lijst);
02058
02059     for ( i=0; i<AP.LoopList.num; ++i ) {
02060         R_FREE(DoLoops[i].p.buffer);
02061         if ( DoLoops[i].dollarname ) {
02062             R_FREE(DoLoops[i].dollarname);
02063         }
02064     }
02065     R_FREE(AP.LoopList.lijst);
02066
02067     for ( i=0; i<AP.ProcList.num; ++i ) {
02068         R_FREE(Procedures[i].p.buffer);
02069         R_FREE(Procedures[i].name);
02070     }
02071     R_FREE(AP.ProcList.lijst);
02072
02073     for ( i=1; i<=AP.PreSwitchLevel; ++i ) {
02074         R_FREE(AP.PreSwitchStrings[i]);
02075     }
02076     R_FREE(AP.PreSwitchStrings);
02077     R_FREE(AP.preStart);
02078     R_FREE(AP.procedureExtension);
02079     R_FREE(AP.cprocedureExtension);
02080     R_FREE(AP.PreIfStack);
02081     R_FREE(AP.PreSwitchModes);
02082     R_FREE(AP.PreTypes);
02083

```

```

02084      /* #] AP free pointers */
02085
02086      /* first we copy AP as a whole and then restore the pointer structures step
02087      by step. */
02088
02089      AP = *((struct P_const*)p); p = (unsigned char*)p + sizeof(struct P_const);
02090 #ifdef WITHPTHREADS
02091      AP.PreVarLock = dummylock;
02092 #endif
02093
02094      R_COPY_LIST(AP.DollarList);
02095      for ( i=0; i<AP.DollarList.num; ++i ) {
02096          DOLLARS d = Dollars + i;
02097          size = d->size * sizeof(WORD);
02098          if ( size && d->where && d->where != oldAMDollarzero ) {
02099              R_COPY_B(d->where, size, void*);
02100          }
02101 #ifdef WITHPTHREADS
02102          d->pthreadlockread = dummylock;
02103          d->pthreadlockwrite = dummylock;
02104 #endif
02105          if ( d->nfactors > 1 ) {
02106              R_COPY_B(d->factors, sizeof(FACDOLLAR)*d->nfactors, FACDOLLAR*);
02107              for ( j = 0; j < d->nfactors; j++ ) {
02108                  if ( d->factors[j].size > 0 ) {
02109                      R_COPY_B(d->factors[j].where, sizeof(WORD)*(d->factors[j].size+1), WORD*);
02110                  }
02111              }
02112          }
02113      }
02114      AP.DollarList.message = "$-variable";
02115
02116      R_COPY_LIST(AP.PreVarList);
02117      for ( i=0; i<AP.PreVarList.num; ++i ) {
02118          R_SET(size, size_t);
02119          org = PreVar[i].name;
02120          R_COPY_B(PreVar[i].name, size, UBYTE*);
02121          ofs = PreVar[i].name - org;
02122          if ( PreVar[i].value ) PreVar[i].value += ofs;
02123          if ( PreVar[i].argnames ) PreVar[i].argnames += ofs;
02124      }
02125      AP.PreVarList.message = "PreVariable";
02126
02127      R_COPY_LIST(AP.LoopList);
02128      for ( i=0; i<AP.LoopList.num; ++i ) {
02129          org = DoLoops[i].p.buffer;
02130          R_COPY_B(DoLoops[i].p.buffer, DoLoops[i].p.size, UBYTE*);
02131          ofs = DoLoops[i].p.buffer - org;
02132          if ( DoLoops[i].name ) DoLoops[i].name += ofs;
02133          if ( DoLoops[i].vars ) DoLoops[i].vars += ofs;
02134          if ( DoLoops[i].contents ) DoLoops[i].contents += ofs;
02135          if ( DoLoops[i].type == ONEEXPRESSION ) {
02136              R_COPY_S(DoLoops[i].dollarname, UBYTE*);
02137          }
02138      }
02139      AP.LoopList.message = "doloop";
02140
02141      R_COPY_LIST(AP.ProcList);
02142      for ( i=0; i<AP.ProcList.num; ++i ) {
02143          if ( Procedures[i].p.size ) {
02144              if ( Procedures[i].loadmode != 1 ) {
02145                  R_COPY_B(Procedures[i].p.buffer, Procedures[i].p.size, UBYTE*);
02146              }
02147              else {
02148                  R_SET(j, int);
02149                  Procedures[i].p.buffer = Procedures[j].p.buffer;
02150              }
02151          }
02152          R_COPY_S(Procedures[i].name, UBYTE*);
02153      }
02154      AP.ProcList.message = "procedure";
02155
02156      size = (AP.NumPreSwitchStrings+1)*(LONG)sizeof(UBYTE*);
02157      R_COPY_B(AP.PreSwitchStrings, size, UBYTE*);
02158      for ( i=1; i<=AP.PreSwitchLevel; ++i ) {
02159          R_COPY_S(AP.PreSwitchStrings[i], UBYTE*);
02160      }
02161
02162      org = AP.preStart;
02163      R_COPY_B(AP.preStart, AP.pSize, UBYTE*);
02164      ofs = AP.preStart - org;
02165      if ( AP.preFill ) AP.preFill += ofs;
02166      if ( AP.preStop ) AP.preStop += ofs;
02167
02168      R_COPY_S(AP.procedureExtension, UBYTE*);
02169      R_COPY_S(AP.cprocedureExtension, UBYTE*);
02170

```

```

02171     R_COPY_B(AP.PreAssignStack, AP.MaxPreAssignLevel*(LONG)sizeof(LONG), LONG*);
02172     R_COPY_B(AP.PreIfStack, AP.MaxPreIfLevel*(LONG)sizeof(int), int*);
02173     R_COPY_B(AP.PreSwitchModes, (AP.NumPreSwitchStrings+1)*(LONG)sizeof(int), int*);
02174     R_COPY_B(AP.PreTypes, (AP.MaxPreTypes+1)*(LONG)sizeof(int), int*);
02175
02176 #ifdef PRINTDEBUG
02177     print_P();
02178 #endif
02179
02180     /*#] AP : */
02181     /*#[ AR : */
02182
02183     R_SET(ofs, LONG);
02184     if ( ofs ) {
02185         AR.infile = AR.Fscr+1;
02186         AR.outfile = AR.Fscr;
02187         AR.hidefile = AR.Fscr+2;
02188     }
02189     else {
02190         AR.infile = AR.Fscr;
02191         AR.outfile = AR.Fscr+1;
02192         AR.hidefile = AR.Fscr+2;
02193     }
02194
02195     /* #[ AR free pointers */
02196
02197     /* Parts of AR will be overwritten by data from the recovery file, therefore
02198      * dynamically allocated memory must be freed first. */
02199
02200     namebufout = AR.outfile->name;
02201     namebufhide = AR.hidefile->name;
02202     R_FREE(AR.outfile->PObuffer);
02203 #ifdef WITHZLIB
02204     R_FREE(AR.outfile->zsp);
02205     R_FREE(AR.outfile->ziobuffer);
02206 #endif
02207     namebufhide = AR.hidefile->name;
02208     R_FREE(AR.hidefile->PObuffer);
02209 #ifdef WITHZLIB
02210     R_FREE(AR.hidefile->zsp);
02211     R_FREE(AR.hidefile->ziobuffer);
02212 #endif
02213     /* no files should be opened -> nothing to do with handle */
02214
02215     /* #] AR free pointers */
02216
02217     /* outfile */
02218     R_SET(*AR.outfile, FILEHANDLE);
02219     org = (UBYTE*)AR.outfile->PObuffer;
02220     size = AR.outfile->POfull - AR.outfile->PObuffer;
02221     AR.outfile->PObuffer = (WORD*)Malloc1(AR.outfile->POsize, "PObuffer");
02222     if ( size ) {
02223         memcpy(AR.outfile->PObuffer, p, size*sizeof(WORD));
02224         p = (unsigned char*)p + size*sizeof(WORD);
02225     }
02226     ofs = (UBYTE*)AR.outfile->PObuffer - org;
02227     AR.outfile->PPostop = (WORD*)((UBYTE*)AR.outfile->PPostop + ofs);
02228     AR.outfile->POfill = (WORD*)((UBYTE*)AR.outfile->POfill + ofs);
02229     AR.outfile->POfull = (WORD*)((UBYTE*)AR.outfile->POfull + ofs);
02230     AR.outfile->name = namebufout;
02231 #ifdef WITHPTHREADS
02232     AR.outfile->wPObuffer = AR.outfile->PObuffer;
02233     AR.outfile->wPPostop = AR.outfile->PPostop;
02234     AR.outfile->wPOfill = AR.outfile->POfill;
02235     AR.outfile->wPOfull = AR.outfile->POfull;
02236 #endif
02237 #ifdef WITHZLIB
02238     /* zsp and ziobuffer will be allocated when used */
02239     AR.outfile->zsp = 0;
02240     AR.outfile->ziobuffer = 0;
02241 #endif
02242     /* reopen old outfile */
02243 #ifdef WITHMPI
02244     if (PF.me==MASTER)
02245 #endif
02246     if ( AR.outfile->handle >= 0 ) {
02247         if ( CopyFile(sortfile, AR.outfile->name) ) {
02248             MesPrint("ERROR: Could not copy old output sort file %s!", sortfile);
02249             Terminate(-1);
02250         }
02251         AR.outfile->handle = ReOpenFile(AR.outfile->name);
02252         if ( AR.outfile->handle == -1 ) {
02253             MesPrint("ERROR: Could not reopen output sort file %s!", AR.outfile->name);
02254             Terminate(-1);
02255         }
02256         SeekFile(AR.outfile->handle, &AR.outfile->POposition, SEEK_SET);
02257     }

```

```

02258
02259  /* hidefile */
02260  R_SET(*AR.hidefile, FILEHANDLE);
02261  AR.hidefile->name = namebufhide;
02262  if ( AR.hidefile->PObuffer ) {
02263      org = (UBYTE*)AR.hidefile->PObuffer;
02264      size = AR.hidefile->POfull - AR.hidefile->PObuffer;
02265      AR.hidefile->PObuffer = (WORD*)Malloc1(AR.hidefile->POsize, "PObuffer");
02266      if ( size ) {
02267          memcpy(AR.hidefile->PObuffer, p, size*sizeof(WORD));
02268          p = (unsigned char*)p + size*sizeof(WORD);
02269      }
02270      ofs = (UBYTE*)AR.hidefile->PObuffer - org;
02271      AR.hidefile->POstop = (WORD*)((UBYTE*)AR.hidefile->POstop + ofs);
02272      AR.hidefile->POfill = (WORD*)((UBYTE*)AR.hidefile->POfill + ofs);
02273      AR.hidefile->POfull = (WORD*)((UBYTE*)AR.hidefile->POfull + ofs);
02274  #ifdef WITHPTHREADS
02275      AR.hidefile->wPObuffer = AR.hidefile->PObuffer;
02276      AR.hidefile->wPOstop = AR.hidefile->POstop;
02277      AR.hidefile->wPOfill = AR.hidefile->POfill;
02278      AR.hidefile->wPOfull = AR.hidefile->POfull;
02279  #endif
02280  }
02281  #ifdef WITHZLIB
02282      /* zsp and ziobuffer will be allocated when used */
02283      AR.hidefile->zsp = 0;
02284      AR.hidefile->ziobuffer = 0;
02285  #endif
02286  /* reopen old hidefile */
02287  if ( AR.hidefile->handle >= 0 ) {
02288      if ( CopyFile(hidefile, AR.hidefile->name) ) {
02289          MesPrint("ERROR: Could not copy old hide file %s!",hidefile);
02290          Terminate(-1);
02291      }
02292      AR.hidefile->handle = ReOpenFile(AR.hidefile->name);
02293      if ( AR.hidefile->handle == -1 ) {
02294          MesPrint("ERROR: Could not reopen hide file %s!",AR.hidefile->name);
02295          Terminate(-1);
02296      }
02297      SeekFile(AR.hidefile->handle, &AR.hidefile->POposition, SEEK_SET);
02298  }
02299
02300  /* store file */
02301  R_SET(pos, POSITION);
02302  if ( ISNOTZEROPOS(pos) ) {
02303      CloseFile(AR.StoreData.Handle);
02304      R_SET(AR.StoreData, FILEDATA);
02305      if ( CopyFile(storefile, FG.fname) ) {
02306          MesPrint("ERROR: Could not copy old store file %s!",storefile);
02307          Terminate(-1);
02308      }
02309      AR.StoreData.Handle = (WORD)ReOpenFile(FG.fname);
02310      SeekFile(AR.StoreData.Handle, &AR.StoreData.Position, SEEK_SET);
02311  }
02312
02313  R_SET(AR.DefPosition, POSITION);
02314  R_SET(AR.OldTime, LONG);
02315  R_SET(AR.InInBuf, LONG);
02316  R_SET(AR.InHiBuf, LONG);
02317
02318  R_SET(AR.NoCompress, int);
02319  R_SET(AR.gzipCompress, int);
02320
02321  R_SET(AR.outtohide, int);
02322
02323  R_SET(AR.GetFile, WORD);
02324  R_SET(AR.KeptInHold, WORD);
02325  R_SET(AR.BracketOn, WORD);
02326  R_SET(AR.MaxBracket, WORD);
02327  R_SET(AR.CurDum, WORD);
02328  R_SET(AR.DeferFlag, WORD);
02329  R_SET(AR.TePos, WORD);
02330  R_SET(AR.sLevel, WORD);
02331  R_SET(AR.Stage4Name, WORD);
02332  R_SET(AR.GetOneFile, WORD);
02333  R_SET(AR.PolyFun, WORD);
02334  R_SET(AR.PolyFunInv, WORD);
02335  R_SET(AR.PolyFunType, WORD);
02336  R_SET(AR.PolyFunExp, WORD);
02337  R_SET(AR.PolyFunVar, WORD);
02338  R_SET(AR.PolyFunPow, WORD);
02339  R_SET(AR.Eside, WORD);
02340  R_SET(AR.MaxDum, WORD);
02341  R_SET(AR.level, WORD);
02342  R_SET(AR.expchanged, WORD);
02343  R_SET(AR.expflags, WORD);
02344  R_SET(AR.CurExpr, WORD);

```

```

02345     R_SET(AR.SortType, WORD);
02346     R_SET(AR.ShortSortCount, WORD);
02347
02348     /* this is usually done in Process(), but sometimes FORM does not
02349        end up executing Process() before it uses the AR.CompressPointer,
02350        so we need to explicitly set it here. */
02351     AR.CompressPointer = AR.CompressBuffer;
02352
02353 #ifdef WITHPTHREADS
02354     for ( j = 0; j < AM.totalnumberofthreads; j++ ) {
02355         R_SET(AB[j]->R.wranfnpair1, int);
02356         R_SET(AB[j]->R.wranfnpair2, int);
02357         R_SET(AB[j]->R.wranfcall, int);
02358         R_SET(AB[j]->R.wranfseed, ULONG);
02359         R_SET(AB[j]->R.wranfia, ULONG*);
02360         if ( AB[j]->R.wranfia ) {
02361             R_COPY_B(AB[j]->R.wranfia, sizeof(ULONG)*AB[j]->R.wranfnpair2, ULONG*);
02362         }
02363     }
02364 #else
02365     R_SET(AR.wranfnpair1, int);
02366     R_SET(AR.wranfnpair2, int);
02367     R_SET(AR.wranfcall, int);
02368     R_SET(AR.wranfseed, ULONG);
02369     R_SET(AR.wranfia, ULONG*);
02370     if ( AR.wranfia ) {
02371         R_COPY_B(AR.wranfia, sizeof(ULONG)*AR.wranfnpair2, ULONG*);
02372     }
02373 #endif
02374
02375 #ifdef PRINTDEBUG
02376     print_R();
02377 #endif
02378
02379     /*#] AR : */
02380     /*#[ AO :*/
02381     /*
02382     We copy all non-pointer variables.
02383     */
02384     l = sizeof(A.O) - ((UBYTE *)(&(A.O.NumInBrack))-(UBYTE *)(&A.O));
02385     memcpy(&(A.O.NumInBrack), p, l); p = (unsigned char*)p + l;
02386     /*
02387     Now the variables in OptimizeResult
02388     */
02389     memcpy(&(A.O.OptimizeResult), p, sizeof(OPTIMIZERESULT));
02390     p = (unsigned char*)p + sizeof(OPTIMIZERESULT);
02391
02392     if ( A.O.OptimizeResult.codesize > 0 ) {
02393         R_COPY_B(A.O.OptimizeResult.code, A.O.OptimizeResult.codesize*sizeof(WORD), WORD *);
02394     }
02395     R_COPY_S(A.O.OptimizeResult.nameofexpr, UBYTE *);
02396     /*
02397     And now the dictionaries. We know how many there are. We also know
02398     how many elements the array AO.Dictionaries should have.
02399     */
02400     if ( AO.SizeDictionaries > 0 ) {
02401         AO.Dictionaries = (DICTIONARY **)Malloc1(AO.SizeDictionaries*sizeof(DICTIONARY *),
02402            "Dictionaries");
02403         for ( i = 0; i < AO.NumDictionaries; i++ ) {
02404             R_SET(i, LONG)
02405             AO.Dictionaries[i] = DictFromBytes(p);
02406             p = (char *)p + l;
02407         }
02408     }
02409     /*#] AO :*/
02410 #ifdef WITHMPI
02411     /*#[ PF : */
02412     /*Block*/
02413     int numtasks;
02414     R_SET(numtasks, int);
02415     if(numtasks!=PF.numtasks){
02416         MesPrint("%d number of tasks expected instead of %d; use mpirun -np %d",
02417             numtasks, PF.numtasks, numtasks);
02418         if(PF.me!=MASTER)
02419             remove(RecoveryFilename());
02420         Terminate(-1);
02421     }
02422     /*Block*/
02423     R_SET(PF.rhsInParallel, int);
02424     R_SET(PF.exprbufsize, int);
02425     R_SET(PF.log, int);
02426     /*#] PF : */
02427 #endif
02428
02429 #ifdef WITHPTHREADS
02430     /* read timing information of individual threads */
02431     R_SET(i, int);

```

```

02432     for ( j=1; j<AM.totalnumberofthreads; ++j ) {
02433         /* ... and correcting OldTime */
02434         AB[j]->R.OldTime = -(*(LONG*)p+j));
02435     }
02436     WriteTimerInfo((LONG*)p, (LONG *) ((unsigned char*)p + i*(LONG)sizeof(LONG)));
02437     p = (unsigned char*)p + 2*i*(LONG)sizeof(LONG);
02438 #endif /* ifdef WITHPTHREADS */
02439
02440     if ( fclose(fd) ) return(__LINE__);
02441
02442     M_free(buf, "recovery buffer");
02443
02444     /* cares about data in S_const */
02445     UpdatePositions();
02446     AT.SS = AT.S0;
02447 /*
02448     Set the checkpoint parameter right for the next checkpoint.
02449 */
02450     AC.CheckpointStamp = TimeWallClock(1);
02451
02452     done_snapshot = 1;
02453     MesPrint("done."); fflush(0);
02454
02455     return(0);
02456 }
02457
02458 /*
02459     #] DoRecovery :
02460     #[ DoSnapshot :
02461 */
02462
02463 static int DoSnapshot(int moduletype)
02464 {
02465     GETIDENTITY
02466     FILE *fd;
02467     POSITION pos;
02468     int i, j;
02469     LONG l;
02470     WORD *w;
02471     void *adr;
02472 #ifdef WITHPTHREADS
02473     LONG *longp, *longpp;
02474 #endif /* ifdef WITHPTHREADS */
02475
02476     MesPrint("Saving recovery point ... %"); fflush(0);
02477 #ifdef PRINTTIMEMARKS
02478     MesPrint("\n");
02479 #endif
02480
02481     if ( !(fd = fopen(intermedfile, "wb")) ) return(__LINE__);
02482
02483     /* reserve space in the file for a length field */
02484     if ( fwrite(&pos, 1, sizeof(POSITION), fd) != sizeof(POSITION) ) return(__LINE__);
02485
02486     /* write moduletype */
02487     if ( fwrite(&moduletype, 1, sizeof(int), fd) != sizeof(int) ) return(__LINE__);
02488
02489     /*#[ AM :*/
02490
02491     /* since most values don't change during execution, AM doesn't need to be
02492      * written as a whole. all values will be correctly set when starting up
02493      * anyway. only the exceptions need to be taken care of. see MakeGlobal()
02494      * and PopVariables() in execute.c. */
02495
02496     ANNOUNCE(AM)
02497     S_WRITE_B(&AM.hparallelflag, sizeof(int));
02498     S_WRITE_B(&AM.gparallelflag, sizeof(int));
02499     S_WRITE_B(&AM.gCodesFlag, sizeof(int));
02500     S_WRITE_B(&AM.gNamesFlag, sizeof(int));
02501     S_WRITE_B(&AM.gStatsFlag, sizeof(int));
02502     S_WRITE_B(&AM.gTokensWriteFlag, sizeof(int));
02503     S_WRITE_B(&AM.gNoSpacesInNumbers, sizeof(int));
02504     S_WRITE_B(&AM.gIndentSpace, sizeof(WORD));
02505     S_WRITE_B(&AM.gUnitTrace, sizeof(WORD));
02506     S_WRITE_B(&AM.gDefDim, sizeof(int));
02507     S_WRITE_B(&AM.gDefDim4, sizeof(int));
02508     S_WRITE_B(&AM.gncmod, sizeof(WORD));
02509     S_WRITE_B(&AM.gnpowmod, sizeof(WORD));
02510     S_WRITE_B(&AM.gmodmode, sizeof(WORD));
02511     S_WRITE_B(&AM.gOutputMode, sizeof(WORD));
02512     S_WRITE_B(&AM.gCnumpows, sizeof(WORD));
02513     S_WRITE_B(&AM.gOutputSpaces, sizeof(WORD));
02514     S_WRITE_B(&AM.gOutNumberType, sizeof(WORD));
02515     S_WRITE_B(&AM.gfunpowers, sizeof(int));
02516     S_WRITE_B(&AM.gPolyFun, sizeof(WORD));
02517     S_WRITE_B(&AM.gPolyFunInv, sizeof(WORD));
02518     S_WRITE_B(&AM.gPolyFunType, sizeof(WORD));

```



```

02534     S_WRITE_B(&AM.gPolyFunExp, sizeof(WORD));
02535     S_WRITE_B(&AM.gPolyFunVar, sizeof(WORD));
02536     S_WRITE_B(&AM.gPolyFunPow, sizeof(WORD));
02537     S_WRITE_B(&AM.gProcessBucketSize, sizeof(LONG));
02538     S_WRITE_B(&AM.OldChildTime, sizeof(LONG));
02539     S_WRITE_B(&AM.OldSecTime, sizeof(LONG));
02540     S_WRITE_B(&AM.OldMilliTime, sizeof(LONG));
02541     S_WRITE_B(&AM.gproperorderflag, sizeof(int));
02542     S_WRITE_B(&AM.gThreadBucketSize, sizeof(LONG));
02543     S_WRITE_B(&AM.gSizeCommuteInSet, sizeof(int));
02544     S_WRITE_B(&AM.gThreadStats, sizeof(int));
02545     S_WRITE_B(&AM.gFinalStats, sizeof(int));
02546     S_WRITE_B(&AM.gThreadsFlag, sizeof(int));
02547     S_WRITE_B(&AM.gThreadBalancing, sizeof(int));
02548     S_WRITE_B(&AM.gThreadSortFileSynch, sizeof(int));
02549     S_WRITE_B(&AM.gProcessStats, sizeof(int));
02550     S_WRITE_B(&AM.gOldParallelStats, sizeof(int));
02551     S_WRITE_B(&AM.gSortType, sizeof(int));
02552     S_WRITE_B(&AM.gShortStatsMax, sizeof(WORD));
02553     S_WRITE_B(&AM.gIsFortran90, sizeof(int));
02554     adr = &AM.dollarzero;
02555     S_WRITE_B(adr, sizeof(void*));
02556     S_WRITE_B(&AM.gFortran90Kind, sizeof(UBYTE *));
02557     S_WRITE_S(AM.gFortran90Kind);
02558
02559     S_WRITE_S(AM.gextrasymp);
02560     S_WRITE_S(AM.ggextrasymp);
02561
02562     S_WRITE_B(&AM.PrintTotalSize, sizeof(int));
02563     S_WRITE_B(&AM.fbuffersize, sizeof(int));
02564     S_WRITE_B(&AM.gOldFactArgFlag, sizeof(int));
02565     S_WRITE_B(&AM.ggOldFactArgFlag, sizeof(int));
02566
02567     S_WRITE_B(&AM.gnumextrasymp, sizeof(int));
02568     S_WRITE_B(&AM.ggnumextrasymp, sizeof(int));
02569     S_WRITE_B(&AM.NumSpectatorFiles, sizeof(int));
02570     S_WRITE_B(&AM.SizeForSpectatorFiles, sizeof(int));
02571     S_WRITE_B(&AM.gOldGCDflag, sizeof(int));
02572     S_WRITE_B(&AM.ggOldGCDflag, sizeof(int));
02573     S_WRITE_B(&AM.gWTimeStatsFlag, sizeof(int));
02574
02575     S_WRITE_B(&AM.Path, sizeof(UBYTE *));
02576     S_WRITE_S(AM.Path);
02577
02578     S_WRITE_B(&AM.FromStdin, sizeof(BOOL));
02579     S_WRITE_B(&AM.IgnoreDeprecation, sizeof(BOOL));
02580
02581     /*#] AM :*/
02582     /*#[ AC :*/
02583
02584     /* we write AC as a whole and then write all additional data step by step.
02585      * AC.DubiousList doesn't need to be treated, because it should be empty. */
02586
02587     ANNOUNCE(AC)
02588     S_WRITE_B(&AC, sizeof(struct C_const));
02589
02590     S_WRITE_NAMETREE(AC.dollarnames);
02591     S_WRITE_NAMETREE(AC.exprnames);
02592     S_WRITE_NAMETREE(AC.varnames);
02593
02594     S_WRITE_LIST(AC.ChannelList);
02595     for ( i=0; i<AC.ChannelList.num; ++i ) {
02596         S_WRITE_S(channels[i].name);
02597     }
02598
02599     ANNOUNCE(AC.FunctionList)
02600     S_WRITE_LIST(AC.FunctionList);
02601     for ( i=0; i<AC.FunctionList.num; ++i ) {
02602         /* if the function is a table */
02603         if ( functions[i].tabl ) {
02604             TABLES tabl = functions[i].tabl;
02605             S_WRITE_B(tabl, sizeof(struct TABEs));
02606             if ( tabl->tablepointers ) {
02607                 if ( tabl->sparse ) {
02608                     /* sparse tables. reserved holds number of allocated
02609                      * elements. the size of an element is numind plus
02610                      * TABLEEXTENSION times the size of WORD. */
02611                     S_WRITE_B(tabl->tablepointers,
02612                               tabl->reserved*sizeof(WORD)*(tabl->numind+TABLEEXTENSION));
02613                 }
02614                 else {
02615                     /* matrix like tables. */
02616                     S_WRITE_B(tabl->tablepointers,
02617                               TABLEEXTENSION*sizeof(WORD)*(tabl->totind));
02618                 }
02619             }
02620             S_WRITE_B(tabl->prototype, tabl->prototypeSize);

```

```

02621     S_WRITE_B(tabl->mm, tabl->numind*(LONG)sizeof(MINMAX));
02622     S_WRITE_B(tabl->flags, tabl->numind*(LONG)sizeof(WORD));
02623     if ( tabl->sparse ) {
02624         S_WRITE_B(tabl->boomlijst, tabl->MaxTreeSize*(LONG)sizeof(COMPTREE));
02625         S_WRITE_S(tabl->argtail);
02626     }
02627     S_WRITE_B(tabl->buffers, tabl->bufferssize*(LONG)sizeof(WORD));
02628     if ( tabl->sparse ) {
02629         TABLES spare = tabl->spare;
02630         S_WRITE_B(spare, sizeof(struct TaBLeS));
02631         if ( spare->tablepointers ) {
02632             if ( spare->sparse ) {
02633                 /* sparse tables */
02634                 S_WRITE_B(spare->tablepointers,
02635                     spare->reserved*sizeof(WORD)*(spare->numind+TABLEEXTENSION));
02636             }
02637             else {
02638                 /* matrix like tables */
02639                 S_WRITE_B(spare->tablepointers,
02640                     TABLEEXTENSION*sizeof(WORD)*(spare->totind));
02641             }
02642         }
02643         S_WRITE_B(spare->mm, spare->numind*(LONG)sizeof(MINMAX));
02644         S_WRITE_B(spare->flags, spare->numind*(LONG)sizeof(WORD));
02645         if ( spare->sparse ) {
02646             S_WRITE_B(spare->boomlijst, spare->MaxTreeSize*(LONG)sizeof(COMPTREE));
02647         }
02648         S_WRITE_B(spare->buffers, spare->bufferssize*(LONG)sizeof(WORD));
02649     }
02650 }
02651 }
02652
02653 ANNOUNCE(AC.ExpressionList)
02654 S_WRITE_LIST(AC.ExpressionList);
02655 for ( i=0; i<AC.ExpressionList.num; ++i ) {
02656     EXPRESSIONS ex = Expressions + i;
02657     if ( ex->renum ) {
02658         S_WRITE_B(ex->renum, sizeof(struct ReNuMbEr));
02659         /* there is one dynamically allocated buffer for struct ReNuMbEr and
02660          * symb.lo points to its beginning. the size of the buffer is not
02661          * stored anywhere but we know it is 2*sizeof(WORD)*N, where N is
02662          * the number of all vectors, indices, functions and symbols. since
02663          * funum points into the buffer at a distance 2N-[Number of
02664          * functions] from symb.lo (see GetTable() in store.c), we can
02665          * calculate the buffer size by some pointer arithmetic. the size is
02666          * then written to the file. */
02667         l = ex->renum->funnum - ex->renum->symb.lo;
02668         l += ex->renum->funnum - ex->renum->func.lo;
02669         S_WRITE_B(&l, sizeof(size_t));
02670         S_WRITE_B(ex->renum->symb.lo, l);
02671     }
02672     if ( ex->bracketinfo ) {
02673         S_WRITE_B(ex->bracketinfo, sizeof(BRACKETINFO));
02674         S_WRITE_B(ex->bracketinfo->indexbuffer,
02675             ex->bracketinfo->indexbuffersize*sizeof(BRACKETINDEX));
02676         S_WRITE_B(ex->bracketinfo->bracketbuffer,
02677             ex->bracketinfo->bracketbuffersize*sizeof(WORD));
02678     }
02679     if ( ex->newbracketinfo ) {
02680         S_WRITE_B(ex->newbracketinfo, sizeof(BRACKETINFO));
02681         S_WRITE_B(ex->newbracketinfo->indexbuffer,
02682             ex->newbracketinfo->indexbuffersize*sizeof(BRACKETINDEX));
02683         S_WRITE_B(ex->newbracketinfo->bracketbuffer,
02684             ex->newbracketinfo->bracketbuffersize*sizeof(WORD));
02685     }
02686     /* don't need to write ex->renumlists */
02687     if ( ex->inmem ) {
02688         /* size of the inmem buffer has to be determined. we use the fact
02689          * that the end of an expression is marked by a zero. */
02690         w = ex->inmem;
02691         while ( *w++ );
02692         l = w - ex->inmem;
02693         S_WRITE_B(&l, sizeof(size_t));
02694         S_WRITE_B(ex->inmem, l);
02695     }
02696 }
02697
02698 ANNOUNCE(AC.IndexList)
02699 S_WRITE_LIST(AC.IndexList);
02700 S_WRITE_LIST(AC.SetElementList);
02701 S_WRITE_LIST(AC.SetList);
02702 S_WRITE_LIST(AC.SymbolList);
02703 S_WRITE_LIST(AC.VectorList);
02704
02705 ANNOUNCE(AC.TableBaseList)
02706 S_WRITE_LIST(AC.TableBaseList);
02707 for ( i=0; i<AC.TableBaseList.num; ++i ) {

```

```

02704      /* see struct dbase in minos.h */
02705      if ( tablebases[i].iblocks ) {
02706          S_WRITE_B(tablebases[i].iblocks, tablebases[i].info.numberofindexblocks *
sizeof(INDEXBLOCK*));
02707          for ( j=0; j<tablebases[i].info.numberofindexblocks; ++j ) {
02708              if ( tablebases[i].iblocks[j] ) {
02709                  S_WRITE_B(tablebases[i].iblocks[j], sizeof(INDEXBLOCK));
02710              }
02711          }
02712      }
02713      if ( tablebases[i].nblocks ) {
02714          S_WRITE_B(tablebases[i].nblocks, tablebases[i].info.numberofnamesblocks *
sizeof(NAMESBLOCK*));
02715          for ( j=0; j<tablebases[i].info.numberofnamesblocks; ++j ) {
02716              if ( tablebases[i].nblocks[j] ) {
02717                  S_WRITE_B(tablebases[i].nblocks[j], sizeof(NAMESBLOCK));
02718              }
02719          }
02720      }
02721      S_WRITE_S(tablebases[i].name);
02722      S_WRITE_S(tablebases[i].fullname);
02723      S_WRITE_S(tablebases[i].tablenames);
02724  }
02725
02726  ANNOUNCE(AC.cbufList)
02727  S_WRITE_LIST(AC.cbufList);
02728  for ( i=0; i<AC.cbufList.num; ++i ) {
02729      S_WRITE_B(cbuf[i].Buffer, cbuf[i].BufferSize*sizeof(WORD));
02730      /* see inicbufs in comtool.c */
02731      S_WRITE_B(cbuf[i].lhs, cbuf[i].maxlhs*(LONG)sizeof(WORD*));
02732      S_WRITE_B(cbuf[i].rhs, cbuf[i].maxrhs*(LONG)(sizeof(WORD*)+2*sizeof(LONG)+2*sizeof(WORD)));
02733      if ( cbuf[i].boomlijst ) {
02734          S_WRITE_B(cbuf[i].boomlijst, cbuf[i].MaxTreeSize*(LONG)sizeof(COMPTREE));
02735      }
02736  }
02737
02738  S_WRITE_LIST(AC.AutoSymbolList);
02739  S_WRITE_LIST(AC.AutoIndexList);
02740  S_WRITE_LIST(AC.AutoVectorList);
02741  S_WRITE_LIST(AC.AutoFunctionList);
02742
02743  S_WRITE_NAMETREE(AC.autonames);
02744
02745  ANNOUNCE(AC.Streams)
02746  S_WRITE_B(AC.Streams, AC.MaxNumStreams*(LONG)sizeof(STREAM));
02747  for ( i=0; i<AC.NumStreams; ++i ) {
02748      if ( AC.Streams[i].inbuffer ) {
02749          S_WRITE_B(AC.Streams[i].buffer, AC.Streams[i].inbuffer);
02750      }
02751      S_WRITE_S(AC.Streams[i].FoldName);
02752      S_WRITE_S(AC.Streams[i].name);
02753  }
02754
02755  if ( AC.termstack ) {
02756      S_WRITE_B(AC.termstack, AC.maxtermlevel*(LONG)sizeof(LONG));
02757  }
02758
02759  if ( AC.termstortstack ) {
02760      S_WRITE_B(AC.termstortstack, AC.maxtermlevel*(LONG)sizeof(LONG));
02761  }
02762
02763  S_WRITE_B(AC.cmod, AM.MaxTal*4*(LONG)sizeof(UWORD));
02764
02765  if ( AC.IfHeap ) {
02766      S_WRITE_B(AC.IfHeap, (LONG)sizeof(LONG)*(AC.MaxIf+1));
02767      S_WRITE_B(AC.IfCount, (LONG)sizeof(LONG)*(AC.MaxIf+1));
02768  }
02769
02770  l = AC.iStop - AC.iBuffer + 2;
02771  S_WRITE_B(AC.iBuffer, l);
02772
02773  if ( AC.LabelNames ) {
02774      S_WRITE_B(AC.LabelNames, AC.MaxLabels*(LONG)(sizeof(UBYTE*)+sizeof(WORD)));
02775      for ( i=0; i<AC.NumLabels; ++i ) {
02776          S_WRITE_S(AC.LabelNames[i]);
02777      }
02778  }
02779
02780  S_WRITE_B(AC.FixIndices, AM.OffsetIndex*(LONG)sizeof(WORD));
02781
02782  if ( AC.termsumcheck ) {
02783      S_WRITE_B(AC.termsumcheck, AC.maxtermlevel*(LONG)sizeof(WORD));
02784  }
02785
02786  S_WRITE_B(AC.WildcardNames, AC.WildcardBufferSize);
02787
02788  if ( AC.tokens ) {

```

```

02789         l = AC.toptokens - AC.tokens;
02790         if ( l ) {
02791             S_WRITE_B(AC.tokens, l);
02792         }
02793     }
02794
02795     S_WRITE_B(AC.tokenarglevel, AM.MaxParLevel*(LONG)sizeof(WORD));
02796
02797     S_WRITE_S(AC.Fortran90Kind);
02798
02799 #ifdef WITHPTHREADS
02800     if ( AC.inputnumbers ) {
02801         S_WRITE_B(AC.inputnumbers, AC.sizepfirstnum*(LONG) (sizeof(WORD)+sizeof(LONG)));
02802     }
02803 #endif /* ifdef WITHPTHREADS */
02804
02805     if ( AC.IfSumCheck ) {
02806         S_WRITE_B(AC.IfSumCheck, (LONG)sizeof(WORD)*(AC.MaxIf+1));
02807     }
02808     if ( AC.CommuteInSet ) {
02809         S_WRITE_B(AC.CommuteInSet, (LONG)sizeof(WORD)*(AC.SizeCommuteInSet+1));
02810     }
02811
02812     S_WRITE_S(AC.CheckpointRunAfter);
02813     S_WRITE_S(AC.CheckpointRunBefore);
02814
02815     S_WRITE_S(AC.extrasym);
02816
02817     /*# AC :*/
02818     /*# AP :*/
02819
02820     /* we write AP as a whole and then write all additional data step by step. */
02821
02822     ANNOUNCE(AP)
02823     S_WRITE_B(&AP, sizeof(struct P_const));
02824
02825     S_WRITE_LIST(AP.DollarList);
02826     for ( i=0; i<AP.DollarList.num; ++i ) {
02827         DOLLARS d = Dollars + i;
02828         S_WRITE_DOLLAR(Dollars[i]);
02829         if ( d->nfactors > 1 ) {
02830             S_WRITE_B(&(d->factors),sizeof(FACDOLLAR)*d->nfactors);
02831             for ( j = 0; j < d->nfactors; j++ ) {
02832                 if ( d->factors[j].size > 0 ) {
02833                     S_WRITE_B(&(d->factors[i].where),sizeof(WORD)*(d->factors[j].size+1));
02834                 }
02835             }
02836         }
02837     }
02838
02839     S_WRITE_LIST(AP.PreVarList);
02840     for ( i=0; i<AP.PreVarList.num; ++i ) {
02841         /* there is one dynamically allocated buffer in struct pReVar holding
02842          * the strings name, value and several argnames. the size of the buffer
02843          * can be calculated by adding up their string lengths. */
02844         l = strlen((char*)PreVar[i].name) + 1;
02845         if ( PreVar[i].value ) {
02846             l += strlen((char*)(PreVar[i].name+1)) + 1;
02847         }
02848         for ( j=0; j<PreVar[i].nargs; ++j ) {
02849             l += strlen((char*)(PreVar[i].name+1)) + 1;
02850         }
02851         S_WRITE_B(&l, sizeof(size_t));
02852         S_WRITE_B(PreVar[i].name, l);
02853     }
02854
02855     ANNOUNCE(AP.LoopList)
02856     S_WRITE_LIST(AP.LoopList);
02857     for ( i=0; i<AP.LoopList.num; ++i ) {
02858         S_WRITE_B(DoLoops[i].p.buffer, DoLoops[i].p.size);
02859         if ( DoLoops[i].type == ONEEXPRESSION ) {
02860             /* do loops with an expression keep this expression in a dynamically
02861              * allocated buffer in dollarname */
02862             S_WRITE_S(DoLoops[i].dollarname);
02863         }
02864     }
02865
02866     S_WRITE_LIST(AP.ProcList);
02867     for ( i=0; i<AP.ProcList.num; ++i ) {
02868         if ( Procedures[i].p.size ) {
02869             if ( Procedures[i].loadmode != 1 ) {
02870                 S_WRITE_B(Procedures[i].p.buffer, Procedures[i].p.size);
02871             }
02872             else {
02873                 for ( j=0; j<AP.ProcList.num; ++j ) {
02874                     if ( Procedures[i].p.buffer == Procedures[j].p.buffer ) {
02875                         break;

```

```

02876         }
02877     }
02878     if ( j == AP.ProcList.num ) {
02879         MesPrint("Error writing procedures to recovery file!");
02880     }
02881     S_WRITE_B(&j, sizeof(int));
02882 }
02883 }
02884 S_WRITE_S(Procedures[i].name);
02885 }
02886
02887 S_WRITE_B(AP.PreSwitchStrings, (AP.NumPreSwitchStrings+1)*(LONG)sizeof(UBYTE*));
02888 for ( i=1; i<=AP.PreSwitchLevel; ++i ) {
02889     S_WRITE_S(AP.PreSwitchStrings[i]);
02890 }
02891
02892 S_WRITE_B(AP.preStart, AP.pSize);
02893
02894 S_WRITE_S(AP.procedureExtension);
02895 S_WRITE_S(AP.cprocedureExtension);
02896
02897 S_WRITE_B(AP.PreAssignStack, AP.MaxPreAssignLevel*(LONG)sizeof(LONG));
02898 S_WRITE_B(AP.PreIfStack, AP.MaxPreIfLevel*(LONG)sizeof(int));
02899 S_WRITE_B(AP.PreSwitchModes, (AP.NumPreSwitchStrings+1)*(LONG)sizeof(int));
02900 S_WRITE_B(AP.PreTypes, (AP.MaxPreTypes+1)*(LONG)sizeof(int));
02901
02902 /*#] AP :*/
02903 /*#[ AR :*/
02904
02905 ANNOUNCE(AR)
02906 /* to remember which entry in AR.Fscr corresponds to the infile */
02907 l = AR.infile - AR.Fscr;
02908 S_WRITE_B(&l, sizeof(LONG));
02909
02910 /* write the FILEHANDLES */
02911 S_WRITE_B(AR.outfile, sizeof(FILEHANDLE));
02912 l = AR.outfile->POfull - AR.outfile->PObuffer;
02913 if ( l ) {
02914     S_WRITE_B(AR.outfile->PObuffer, l*sizeof(WORD));
02915 }
02916 S_WRITE_B(AR.hidefile, sizeof(FILEHANDLE));
02917 l = AR.hidefile->POfull - AR.hidefile->PObuffer;
02918 if ( l ) {
02919     S_WRITE_B(AR.hidefile->PObuffer, l*sizeof(WORD));
02920 }
02921
02922 S_WRITE_B(&AR.StoreData.Fill, sizeof(POSITION));
02923 if ( ISNOTZEROPOS(AR.StoreData.Fill) ) {
02924     S_WRITE_B(&AR.StoreData, sizeof(FILEDATA));
02925 }
02926
02927 S_WRITE_B(&AR.DefPosition, sizeof(POSITION));
02928
02929 l = TimeCPU(1); l = -l;
02930 S_WRITE_B(&l, sizeof(LONG));
02931
02932 ANNOUNCE(AR.InInBuf)
02933 S_WRITE_B(&AR.InInBuf, sizeof(LONG));
02934 S_WRITE_B(&AR.InHiBuf, sizeof(LONG));
02935
02936 S_WRITE_B(&AR.NoCompress, sizeof(int));
02937 S_WRITE_B(&AR.zipCompress, sizeof(int));
02938
02939 S_WRITE_B(&AR.outtohide, sizeof(int));
02940
02941 S_WRITE_B(&AR.GetFile, sizeof(WORD));
02942 S_WRITE_B(&AR.KeptInHold, sizeof(WORD));
02943 S_WRITE_B(&AR.BraceOn, sizeof(WORD));
02944 S_WRITE_B(&AR.MaxBracket, sizeof(WORD));
02945 S_WRITE_B(&AR.CurDum, sizeof(WORD));
02946 S_WRITE_B(&AR.DeferFlag, sizeof(WORD));
02947 S_WRITE_B(&AR.TePos, sizeof(WORD));
02948 S_WRITE_B(&AR.sLevel, sizeof(WORD));
02949 S_WRITE_B(&AR.Stage4Name, sizeof(WORD));
02950 S_WRITE_B(&AR.GetOneFile, sizeof(WORD));
02951 S_WRITE_B(&AR.PolyFun, sizeof(WORD));
02952 S_WRITE_B(&AR.PolyFunInv, sizeof(WORD));
02953 S_WRITE_B(&AR.PolyFunType, sizeof(WORD));
02954 S_WRITE_B(&AR.PolyFunExp, sizeof(WORD));
02955 S_WRITE_B(&AR.PolyFunVar, sizeof(WORD));
02956 S_WRITE_B(&AR.PolyFunPow, sizeof(WORD));
02957 S_WRITE_B(&AR.Eside, sizeof(WORD));
02958 S_WRITE_B(&AR.MaxDum, sizeof(WORD));
02959 S_WRITE_B(&AR.level, sizeof(WORD));
02960 S_WRITE_B(&AR.expchanged, sizeof(WORD));
02961 S_WRITE_B(&AR.expflags, sizeof(WORD));
02962 S_WRITE_B(&AR.CurExpr, sizeof(WORD));

```

```

02963     S_WRITE_B(&AR.SortType, sizeof(WORD));
02964     S_WRITE_B(&AR.ShortSortCount, sizeof(WORD));
02965
02966 #ifdef WITHPTHREADS
02967     for ( j = 0; j < AM.totalnumberofthreads; j++ ) {
02968         S_WRITE_B(&(AB[j]->R.wranfnpair1), sizeof(int));
02969         S_WRITE_B(&(AB[j]->R.wranfnpair2), sizeof(int));
02970         S_WRITE_B(&(AB[j]->R.wranfcall), sizeof(int));
02971         S_WRITE_B(&(AB[j]->R.wranfseed), sizeof(ULONG));
02972         S_WRITE_B(&(AB[j]->R.wranfia), sizeof(ULONG *));
02973         if ( AB[j]->R.wranfia ) {
02974             S_WRITE_B(AB[j]->R.wranfia, sizeof(ULONG)*AB[j]->R.wranfnpair2);
02975         }
02976     }
02977 #else
02978     S_WRITE_B(&(AR.wranfnpair1), sizeof(int));
02979     S_WRITE_B(&(AR.wranfnpair2), sizeof(int));
02980     S_WRITE_B(&(AR.wranfcall), sizeof(int));
02981     S_WRITE_B(&(AR.wranfseed), sizeof(ULONG));
02982     S_WRITE_B(&(AR.wranfia), sizeof(ULONG *));
02983     if ( AR.wranfia ) {
02984         S_WRITE_B(AR.wranfia, sizeof(ULONG)*AR.wranfnpair2);
02985     }
02986 #endif
02987
02988     /*#] AR :*/
02989     /*#[ AO :*/
02990 /*
02991     We copy all non-pointer variables.
02992 */
02993     ANNOUNCE(AO)
02994     l = sizeof(A.O) - ((UBYTE *)(&(A.O.NumInBrack)) - (UBYTE *)(&(A.O)));
02995     S_WRITE_B(&(A.O.NumInBrack), l);
02996 /*
02997     Now the variables in OptimizeResult
02998 */
02999     S_WRITE_B(&(A.O.OptimizeResult), sizeof(OPTIMIZERESULT));
03000     if ( A.O.OptimizeResult.codesize > 0 ) {
03001         S_WRITE_B(A.O.OptimizeResult.code, A.O.OptimizeResult.codesize*sizeof(WORD));
03002     }
03003     S_WRITE_S(A.O.OptimizeResult.nameofexpr);
03004 /*
03005     And now the dictionaries.
03006     We write each dictionary to a buffer and get the size of that buffer.
03007     Then we write the size and the buffer.
03008 */
03009     for ( i = 0; i < AO.NumDictionaries; i++ ) {
03010         l = DictToBytes(AO.Dictionaries[i], (UBYTE *) (AT.WorkPointer));
03011         S_WRITE_B(&l, sizeof(LONG));
03012         S_WRITE_B(AT.WorkPointer, l);
03013     }
03014
03015     /*#] AO :*/
03016     /*#[ PF :*/
03017 #ifdef WITHMPI
03018     S_WRITE_B(&PF.numtasks, sizeof(int));
03019     S_WRITE_B(&PF.rhsInParallel, sizeof(int));
03020     S_WRITE_B(&PF.exprbufsize, sizeof(int));
03021     S_WRITE_B(&PF.log, sizeof(int));
03022 #endif
03023     /*#] PF :*/
03024
03025 #ifdef WITHPTHREADS
03026
03027     ANNOUNCE(GetTimerInfo)
03028 /*
03029     write timing information of individual threads
03030 */
03031     i = GetTimerInfo(&longp, &longpp);
03032     S_WRITE_B(&i, sizeof(int));
03033     S_WRITE_B(longp, i*(LONG)sizeof(LONG));
03034     S_WRITE_B(&l, sizeof(int));
03035     S_WRITE_B(longpp, i*(LONG)sizeof(LONG));
03036
03037 #endif
03038
03039     S_FLUSH_B /* because we will call fwrite() directly in the following code */
03040
03041     /* save length of data at the beginning of the file */
03042     ANNOUNCE(file length)
03043     SETBASEPOSITION(pos, (ftell(fd)));
03044     fseek(fd, 0, SEEK_SET);
03045     if ( fwrite(&pos, 1, sizeof(POSITION), fd) != sizeof(POSITION) ) return(__LINE__);
03046     fseek(fd, BASEPOSITION(pos), SEEK_SET);
03047
03048     ANNOUNCE(file close)
03049     if ( fclose(fd) ) return(__LINE__);

```

```

03050 #ifdef WITHMPI
03051     if ( PF.me == MASTER ) {
03052 #endif
03053 /*
03054     copy store file if necessary
03055 */
03056     ANNOUNCE(copy store file)
03057     if ( ISNOTZEROPOS(AR.StoreData.Fill) ) {
03058         if ( CopyFile(FG.fname, storefile) ) return(__LINE__);
03059     }
03060 /*
03061     copy sort file if necessary
03062 */
03063     ANNOUNCE(copy sort file)
03064     if ( AR.outfile->handle >= 0 ) {
03065         if ( CopyFile(AR.outfile->name, sortfile) ) return(__LINE__);
03066     }
03067 /*
03068     copy hide file if necessary
03069 */
03070     ANNOUNCE(copy hide file)
03071     if ( AR.hidefile->handle >= 0 ) {
03072         if ( CopyFile(AR.hidefile->name, hidefile) ) return(__LINE__);
03073     }
03074 #ifdef WITHMPI
03075     }
03076 /*
03077     * For ParFORM, the renaming will be performed after the master got
03078     * all recovery files from the slaves.
03079     */
03080 #else
03081 /*
03082     make the intermediate file the recovery file
03083 */
03084     ANNOUNCE(rename intermediate file)
03085     if ( rename(intermedfile, recoveryfile) ) return(__LINE__);
03086
03087     done_snapshot = 1;
03088
03089     MesPrint("done."); fflush(0);
03090 #endif
03091
03092 #ifdef PRINTDEBUG
03093     print_M();
03094     print_C();
03095     print_P();
03096     print_R();
03097 #endif
03098
03099     return(0);
03100 }
03101
03102 /*
03103     #] DoSnapshot :
03104     #[ DoCheckpoint :
03105 */
03106
03111 void DoCheckpoint(int moduletype)
03112 {
03113     int error;
03114     LONG timestamp = TimeWallClock(1);
03115 #ifdef WITHMPI
03116     if (PF.me == MASTER){
03117 #endif
03118         if ( timestamp - AC.CheckpointStamp >= AC.CheckpointInterval ) {
03119             char argbuf[20];
03120             int retvalue = 0;
03121             if ( AC.CheckpointRunBefore ) {
03122                 size_t l1, l2;
03123                 char *str;
03124                 l1 = strlen(AC.CheckpointRunBefore);
03125                 NumToStr((UBYTE*)argbuf, AC.CModule);
03126                 l2 = strlen(argbuf);
03127                 str = (char*)Malloc(1+l1+l2+2, "callbefore");
03128                 strcpy(str, AC.CheckpointRunBefore);
03129                 *(str+l1) = ' ';
03130                 strcpy(str+l1+1, argbuf);
03131                 retvalue = system(str);
03132                 M_free(str, "callbefore");
03133                 if ( retvalue ) {
03134                     MesPrint("Script returned error -> no recovery file will be created.");
03135                 }
03136             }
03137 #ifdef WITHMPI
03138             /* Confirm slaves to make snapshots. */
03139             PF_BroadcastNumber(retvalue == 0);
03140 #endif

```

```

03141         if ( retvalue == 0 ) {
03142             if ( (error = DoSnapshot(moduletype)) ) {
03143                 MesPrint("Error creating recovery files: %d", error);
03144             }
03145 #ifdef WITHMPI
03146         {
03147             int i;
03148             /*get recovery files from slaves:*/
03149             for(i=1; i<PF.numtasks;i++){
03150                 FILE *fd;
03151                 const char *tmpnam = PF_recoveryfile('m', i, 1);
03152                 fd = fopen(tmpnam, "w");
03153                 if(fd == NULL){
03154                     MesPrint("Error opening recovery file for slave %d",i);
03155                     Terminate(-1);
03156                 }/*if(fd == NULL)*/
03157                 retvalue=PF_RecvFile(i,fd);
03158                 if(retvalue<=0){
03159                     MesPrint("Error receiving recovery file from slave %d",i);
03160                     Terminate(-1);
03161                 }/*if(retvalue<=0)*/
03162                 fclose(fd);
03163             }/*for(i=0; i<PF.numtasks;i++)*/
03164             /*
03165              * Make the intermediate files the recovery files.
03166              */
03167             ANNOUNCE(rename intermediate file)
03168             for ( i = 0; i < PF.numtasks; i++ ) {
03169                 const char *src = PF_recoveryfile('m', i, 1);
03170                 const char *dst = PF_recoveryfile('m', i, 0);
03171                 if ( rename(src, dst) ) {
03172                     MesPrint("Error renaming recovery file %s -> %s", src, dst);
03173                 }
03174             }
03175             done_snapshot = 1;
03176             MesPrint("done."); fflush(0);
03177         }
03178 #endif
03179     }
03180     if ( AC.CheckpointRunAfter ) {
03181         size_t l1, l2;
03182         char *str;
03183         l1 = strlen(AC.CheckpointRunAfter);
03184         NumToStr((UBYTE*)argbuf, AC.CModule);
03185         l2 = strlen(argbuf);
03186         str = (char*)Malloc1(l1+l2+2, "callafter");
03187         strcpy(str, AC.CheckpointRunAfter);
03188         *(str+l1) = ' ';
03189         strcpy(str+l1+1, argbuf);
03190         retvalue = system(str);
03191         M_free(str, "callafter");
03192         if ( retvalue ) {
03193             MesPrint("Error calling script after recovery.");
03194         }
03195     }
03196     AC.CheckpointStamp = TimeWallClock(1);
03197 }
03198 #ifdef WITHMPI
03199 else/* timestamp - AC.CheckpointStamp < AC.CheckpointInterval*/
03200 /* The slaves don't need to make snapshots. */
03201 PF_BroadcastNumber(0);
03202 }
03203 }/*if(PF.me == MASTER)*/
03204 else{/*Slave*/
03205     int i;
03206     /* Check if the slave needs to make a snapshot. */
03207     if ( PF_BroadcastNumber(0) ) {
03208         error = DoSnapshot(moduletype);
03209         if(error == 0){
03210             FILE *fd;
03211             /*
03212              * Send the recovery file to the master. Note that no renaming
03213              * has been performed and what we have to send is actually sitting
03214              * in the intermediate file.
03215              */
03216             fd = fopen(intermedfile, "r");
03217             i=PF_SendFile(MASTER, fd);/*if fd=NULL, PF_SendFile seds to a slave the failure tag*/
03218             if(fd == NULL)
03219                 Terminate(-1);
03220             fclose(fd);
03221             if(i<=0)
03222                 Terminate(-1);
03223             /*Now the slave need not the recovery file so remove it:*/
03224             remove(intermedfile);
03225         }
03226         else{
03227             /*send the error tag to the master:*/

```



```

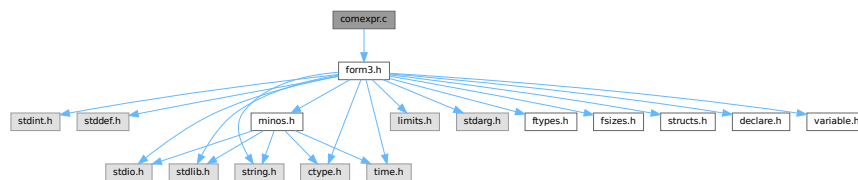
03228         PF_SendFile(MASTER, NULL); /*if fd==NULL, PF_SendFile seds to a slave the failure tag*/
03229         Terminate(-1);
03230     }
03231     done_snapshot = 1;
03232     } /*if (tag=PF_DATA_MSGTAG) */
03233     } /*if (PF.me != MASTER) */
03234 #endif
03235 }
03236
03237 /*
03238     #] DoCheckpoint :
03239 */

```

4.7 comexpr.c File Reference

```
#include "form3.h"
```

Include dependency graph for comexpr.c:



Data Structures

- struct `id_options`

Functions

- int `CoLocal` (UBYTE *inp)
- int `CoGlobal` (UBYTE *inp)
- int `CoLocalFactorized` (UBYTE *inp)
- int `CoGlobalFactorized` (UBYTE *inp)
- int `DoExpr` (UBYTE *inp, int type, int par)
- int `ColdOld` (UBYTE *inp)
- int `Cold` (UBYTE *inp)
- int `ColdNew` (UBYTE *inp)
- int `CoDisorder` (UBYTE *inp)
- int `CoMany` (UBYTE *inp)
- int `CoMulti` (UBYTE *inp)
- int `ColfMatch` (UBYTE *inp)
- int `ColfNoMatch` (UBYTE *inp)
- int `CoOnce` (UBYTE *inp)
- int `CoOnly` (UBYTE *inp)
- int `CoSelect` (UBYTE *inp)
- int `ColdExpression` (UBYTE *inp, int type)
- int `CoMultiply` (UBYTE *inp)
- int `CoFill` (UBYTE *inp)
- int `CoFillExpression` (UBYTE *inp)
- int `CoPrintTable` (UBYTE *inp)
- int `CoAssign` (UBYTE *inp)
- int `CoDeallocateTable` (UBYTE *inp)

4.7.1 Detailed Description

Compiler routines for statements that involve algebraic expressions. These involve definitions, id-statements, the multiply statement and the fill statement.

Definition in file [comexpr.c](#).

4.7.2 Function Documentation

4.7.2.1 CoLocal()

```
int CoLocal (
    UBYTE * inp )
```

Definition at line 66 of file [comexpr.c](#).

4.7.2.2 CoGlobal()

```
int CoGlobal (
    UBYTE * inp )
```

Definition at line 73 of file [comexpr.c](#).

4.7.2.3 CoLocalFactorized()

```
int CoLocalFactorized (
    UBYTE * inp )
```

Definition at line 80 of file [comexpr.c](#).

4.7.2.4 CoGlobalFactorized()

```
int CoGlobalFactorized (
    UBYTE * inp )
```

Definition at line 87 of file [comexpr.c](#).

4.7.2.5 DoExpr()

```
int DoExpr (
    UBYTE * inp,
    int type,
    int par )
```

Definition at line 96 of file [comexpr.c](#).

4.7.2.6 ColdOld()

```
int ColdOld (
    UBYTE * inp )
```

Definition at line 384 of file [comexpr.c](#).

4.7.2.7 Cold()

```
int Cold (
    UBYTE * inp )
```

Definition at line 395 of file [comexpr.c](#).

4.7.2.8 ColdNew()

```
int ColdNew (
    UBYTE * inp )
```

Definition at line 406 of file [comexpr.c](#).

4.7.2.9 CoDisorder()

```
int CoDisorder (
    UBYTE * inp )
```

Definition at line 417 of file [comexpr.c](#).

4.7.2.10 CoMany()

```
int CoMany (
    UBYTE * inp )
```

Definition at line 428 of file [comexpr.c](#).

4.7.2.11 CoMulti()

```
int CoMulti (
    UBYTE * inp )
```

Definition at line 439 of file [comexpr.c](#).

4.7.2.12 ColfMatch()

```
int ColfMatch (
    UBYTE * inp )
```

Definition at line 450 of file [comexpr.c](#).

4.7.2.13 ColfNoMatch()

```
int ColfNoMatch (
    UBYTE * inp )
```

Definition at line [461](#) of file [comexpr.c](#).

4.7.2.14 CoOnce()

```
int CoOnce (
    UBYTE * inp )
```

Definition at line [472](#) of file [comexpr.c](#).

4.7.2.15 CoOnly()

```
int CoOnly (
    UBYTE * inp )
```

Definition at line [483](#) of file [comexpr.c](#).

4.7.2.16 CoSelect()

```
int CoSelect (
    UBYTE * inp )
```

Definition at line [494](#) of file [comexpr.c](#).

4.7.2.17 ColdExpression()

```
int ColdExpression (
    UBYTE * inp,
    int type )
```

Definition at line [507](#) of file [comexpr.c](#).

4.7.2.18 CoMultiply()

```
int CoMultiply (
    UBYTE * inp )
```

Definition at line [1031](#) of file [comexpr.c](#).

4.7.2.19 CoFill()

```
int CoFill (
    UBYTE * inp )
```

Definition at line [1070](#) of file [comexpr.c](#).

4.7.2.20 CoFillExpression()

```
int CoFillExpression (
    UBYTE * inp )
```

Definition at line 1356 of file [comexpr.c](#).

4.7.2.21 CoPrintTable()

```
int CoPrintTable (
    UBYTE * inp )
```

Definition at line 1748 of file [comexpr.c](#).

4.7.2.22 CoAssign()

```
int CoAssign (
    UBYTE * inp )
```

Definition at line 1924 of file [comexpr.c](#).

4.7.2.23 CoDeallocateTable()

```
int CoDeallocateTable (
    UBYTE * inp )
```

Definition at line 1982 of file [comexpr.c](#).

4.8 comexpr.c

[Go to the documentation of this file.](#)

```
00001
00008 /* # [ License : */
00009 /*
00010 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00011 * When using this file you are requested to refer to the publication
00012 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013 * This is considered a matter of courtesy as the development was paid
00014 * for by FOM the Dutch physics granting agency and we would like to
00015 * be able to track its scientific use to convince FOM of its value
00016 * for the community.
00017 *
00018 * This file is part of FORM.
00019 *
00020 * FORM is free software: you can redistribute it and/or modify it under the
00021 * terms of the GNU General Public License as published by the Free Software
00022 * Foundation, either version 3 of the License, or (at your option) any later
00023 * version.
00024 *
00025 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00028 * details.
00029 *
00030 * You should have received a copy of the GNU General Public License along
00031 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00032 */
00033 /* # ] License : */
00034
00035 /*
```

```

00036     # [ Includes : compi2.c
00037
00038     File contains most of what has to do with compiling expressions.
00039     Main supporting file: token.c
00040 */
00041
00042 #include "form3.h"
00043
00044 static struct id_options {
00045     UBYTE *name;
00046     int code;
00047     int dummy;
00048 } IdOptions[] = {
00049     { (UBYTE *) "multi",      SUBMULTI      , 0 }
00050     , { (UBYTE *) "many",      SUBMANY       , 0 }
00051     , { (UBYTE *) "only",      SUBONLY       , 0 }
00052     , { (UBYTE *) "once",      SUBONCE       , 0 }
00053     , { (UBYTE *) "ifmatch",   SUBAFTER     , 0 }
00054     , { (UBYTE *) "ifnomatch", SUBAFTERNOT , 0 }
00055     , { (UBYTE *) "ifnotmatch", SUBAFTERNOT , 0 }
00056     , { (UBYTE *) "disorder",  SUBDISORDER , 0 }
00057     , { (UBYTE *) "select",    SUBSELECT    , 0 }
00058     , { (UBYTE *) "all",       SUBALL       , 0 }
00059 };
00060
00061 /*
00062 #] Includes :
00063 # [ CoLocal :
00064 */
00065
00066 int CoLocal(UBYTE *inp) { return (DoExpr(inp, LOCALEXPRESSION, 0)); }
00067
00068 /*
00069 #] CoLocal :
00070 # [ CoGlobal :
00071 */
00072
00073 int CoGlobal(UBYTE *inp) { return (DoExpr(inp, GLOBALEXPRESSION, 0)); }
00074
00075 /*
00076 #] CoGlobal :
00077 # [ CoLocalFactorized :
00078 */
00079
00080 int CoLocalFactorized(UBYTE *inp) { return (DoExpr(inp, LOCALEXPRESSION, 1)); }
00081
00082 /*
00083 #] CoLocalFactorized :
00084 # [ CoGlobalFactorized :
00085 */
00086
00087 int CoGlobalFactorized(UBYTE *inp) { return (DoExpr(inp, GLOBALEXPRESSION, 1)); }
00088
00089 /*
00090 #] CoGlobalFactorized :
00091 # [ DoExpr:
00092
00093
00094 */
00095
00096 int DoExpr(UBYTE *inp, int type, int par)
00097 {
00098     GETIDENTITY
00099     int error = 0;
00100     UBYTE *p, *q, c;
00101     WORD *w, i, j = 0, c1, c2, *OldWork = AT.WorkPointer, osize;
00102     WORD jold = 0;
00103     POSITION pos;
00104     while ( *inp == ',' ) inp++;
00105     if ( par ) AC.ToBeInFactors = 1;
00106     else AC.ToBeInFactors = 0;
00107     p = inp;
00108     while ( *p && *p != '=' ) {
00109         if ( *p == '(' ) SKIPBRA4(p)
00110         else if ( *p == '{' ) SKIPBRA5(p)
00111         else if ( *p == '[' ) SKIPBRA1(p)
00112         else p++;
00113     }
00114     if ( *p ) { /* Variety with the = sign */
00115         if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
00116             MesPrint("%Illegal name for expression");
00117             error = 1;
00118             if ( q[-1] == '_' ) {
00119                 while ( FG.cTable[*q] < 2 || *q == '_' ) q++;
00120             }
00121         }
00122         else {

```

```

00123     c = *q; *q = 0;
00124     if ( GetVar(inp,&c1,&c2,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
00125         if ( c1 == CEXPRESSION ) {
00126             if ( Expressions[c2].status == STOREDEXPRESSION ) {
00127                 MesPrint("&Illegal attempt to overwrite a stored expression");
00128                 error = 1;
00129             }
00130             else {
00131                 HighWarning("Expression is replaced by new definition");
00132                 if ( AO.OptimizeResult.nameofexpr != NULL &&
00133                     StrCmp(inp,AO.OptimizeResult.nameofexpr) == 0 ) {
00134                     ClearOptimize();
00135                 }
00136                 if ( Expressions[c2].status != DROPEDEXPRESSION ) {
00137                     w = &(Expressions[c2].status);
00138                     if ( *w == LOCALEXPRESSION || *w == SKIPLEXPRESSION )
00139                         *w = DROPEXPRESSION;
00140                     else if ( *w == GLOBALEXPRESSION || *w == SKIPGEXPRESSION )
00141                         *w = DROPGEXPRESSION;
00142                     else if ( *w == HIDDENLEXPRESSION )
00143                         *w = DROPHLEXPRESSION;
00144                     else if ( *w == HIDDENGEXPRESSION )
00145                         *w = DROPHGEXPRESSION;
00146                 }
00147                 AC.TransEname = Expressions[c2].name;
00148                 j = EntVar(CEXPRESSION,0,type,0,0,0);
00149                 Expressions[j].node = Expressions[c2].node;
00150                 Expressions[c2].replace = j;
00151             }
00152         }
00153         else {
00154             MesPrint("&name of expression is also name of a variable");
00155             error = 1;
00156             j = EntVar(CEXPRESSION,inp,type,0,0,0);
00157         }
00158         jold = c2;
00159     }
00160     else {
00161         /*
00162             Here we have to worry about reuse of the expression in the
00163             same module. That will need AS.Oldvflags but that may not
00164             be defined or have the proper value.
00165         */
00166         j = EntVar(CEXPRESSION,inp,type,0,0,0);
00167         jold = j;
00168     }
00169     *q = c;
00170     OldWork = w = AT.WorkPointer;
00171     *w++ = TYPEEXPRESSION;
00172     *w++ = 3+SUBEXPSIZE;
00173     *w++ = j;
00174     AC.ProtoType = w;
00175     AR.CurExpr = j; /* Block expression j */
00176     *w++ = SUBEXPRESSION;
00177     *w++ = SUBEXPSIZE;
00178     *w++ = j;
00179     *w++ = 1;
00180     *w++ = AC.cbunum;
00181     FILLSUB(w)
00182
00183     if ( c == '(' ) {
00184         while ( *q == ',' || *q == '(' ) {
00185             inp = q+1;
00186             if ( ( q = SkipAName(inp) ) == 0 ) {
00187                 MesPrint("&Illegal name for expression argument");
00188                 error = 1;
00189                 q = p - 1;
00190                 break;
00191             }
00192             c = *q; *q = 0;
00193             if ( GetVar(inp,&c1,&c2,ALLVARIABLES,WITHAUTO) < 0 ) c1 = -1;
00194             switch ( c1 ) {
00195                 case CSYMBOL :
00196                     *w++ = SYMTOSYM; *w++ = 4; *w++ = c2; *w++ = 0;
00197                     break;
00198                 case CINDEXT :
00199                     *w++ = INDTOIND; *w++ = 4;
00200                     *w++ = c2 + AM.OffsetIndex; *w++ = 0;
00201                     break;
00202                 case CVECTOR :
00203                     *w++ = VECTOVEC; *w++ = 4;
00204                     *w++ = c2 + AM.OffsetVector; *w++ = 0;
00205                     break;
00206                 case CFUNCTION :
00207                     *w++ = FUNTOFUN; *w++ = 4; *w++ = c2 + FUNCTION; *w++ = 0;
00208                     break;
00209                 default :

```

```

00210             MesPrint("&Illegal expression parameter: %s",inp);
00211             error = 1;
00212             break;
00213         }
00214         *q = c;
00215     }
00216     if ( *q != ')' || q+1 != p ) {
00217         MesPrint("&Illegal use of arguments for expression");
00218         error = 1;
00219     }
00220     AC.ProtoType[1] = w - AC.ProtoType;
00221 }
00222 else if ( c != '=' ) {
00223 /*
00224     The dummy accepted L F := RHS;
00225 */
00226     MesPrint("&Illegal LHS for expression definition");
00227     error = 1;
00228 }
00229 *w++ = 1;
00230 *w++ = 1;
00231 *w++ = 3;
00232 *w++ = 0;
00233 SeekScratch(AR.outfile,&pos);
00234 Expressions[j].counter = 1;
00235 Expressions[j].onfile = pos;
00236 Expressions[j].whichbuffer = 0;
00237 #ifdef PARALLELCODE
00238     Expressions[j].partodo = AC.inparallelfalg;
00239 #endif
00240 OldWork[2] = w - OldWork - 3;
00241 AT.WorkPointer = w;
00242 /*
00243     Writing the expression prototype to disk and to the compiler
00244     buffer is done only after the RHS has been compiled because
00245     we don't know the number of the main level RHS yet.
00246 */
00247 }
00248 inp = p+1;
00249 ClearWildcardNames();
00250 osize = AC.ProtoType[1]; AC.ProtoType[1] = SUBEXPSIZE;
00251 PutInVflags(jold);
00252 if ( ( i = CompileAlgebra(inp,RHSIDE,AC.ProtoType) ) < 0 ) {
00253     AC.ProtoType[1] = osize;
00254     error = 1;
00255 }
00256 else if ( error == 0 ) {
00257     AC.ProtoType[1] = osize;
00258     AC.ProtoType[2] = i;
00259     if ( PutOut(BHEAD OldWork+2,&pos,AR.outfile,0) < 0 ) {
00260         MesPrint("&Cannot create expression");
00261         error = -1;
00262     }
00263     else {
00264         Expressions[j].sizeprototype = OldWork[2];
00265         OldWork[2] = 4+SUBEXPSIZE;
00266         OldWork[4] = SUBEXPSIZE;
00267         OldWork[5] = i;
00268         OldWork[SUBEXPSIZE+3] = 1;
00269         OldWork[SUBEXPSIZE+4] = 1;
00270         OldWork[SUBEXPSIZE+5] = 3;
00271         OldWork[SUBEXPSIZE+6] = 0;
00272         if ( PutOut(BHEAD OldWork+2,&pos,AR.outfile,0) < 0 ) {
00273             || FlushOut(&pos,AR.outfile,0) ) {
00274                 MesPrint("&Cannot create expression");
00275                 error = -1;
00276             }
00277             AR.outfile->POfull = AR.outfile->POfill;
00278         }
00279         OldWork[2] = j;
00280 /*
00281     Seems unnecessary (13-feb-2018)
00282
00283     AddNtoL(OldWork[1],OldWork);
00284 */
00285     AT.WorkPointer = OldWork;
00286     if ( AC.dumnumflag ) Add2Com(TYPEDETCURDUM)
00287 }
00288 AC.ToBeInFactors = 0;
00289 }
00290 else { /* Variety in which expressions change property */
00291 /*
00292     This code got a major revision because it didn't
00293     take hidden expressions into account. (1-jun-2010 JV)
00294 */
00295     do {
00296         if ( ( q = SkipAName(inp) ) == 0 ) {

```



```

00297         MesPrint("&Illegal name(s) for expression(s)");
00298         return(1);
00299     }
00300     c = *q; *q = 0;
00301     if ( GetName(AC.exprnames,inp,&c2,NOAUTO) == NAMENOTFOUND ) {
00302         MesPrint("&%s is not a valid expression",inp);
00303         error = 1;
00304     }
00305     else {
00306         w = &(Expressions[c2].status);
00307         if ( type == LOCALEXPRESSION ) {
00308             switch ( *w ) {
00309                 case GLOBALEXPRESSION:
00310                     *w = LOCALEXPRESSION;
00311                     break;
00312                 case SKIPGEXPRESSION:
00313                     *w = SKIPLEXPRESSION;
00314                     break;
00315                 case DROPGEXPRESSION:
00316                     *w = DROPLEXPRESSION;
00317                     break;
00318                 case HIDDENGEXPRESSION:
00319                     *w = HIDDENLEXPRESSION;
00320                     break;
00321                 case HIDEGEXPRESSION:
00322                     *w = HIDELEXPRESSION;
00323                     break;
00324                 case UNHIDEGEXPRESSION:
00325                     *w = UNHIDELEXPRESSION;
00326                     break;
00327                 case INTOHIDEGEXPRESSION:
00328                     *w = INTOHIDELEXPRESSION;
00329                     break;
00330                 case DROPHGEXPRESSION:
00331                     *w = DROPHLEXPRESSION;
00332                     break;
00333             }
00334         }
00335         else if ( type == GLOBALEXPRESSION ) {
00336             switch ( *w ) {
00337                 case LOCALEXPRESSION:
00338                     *w = GLOBALEXPRESSION;
00339                     break;
00340                 case SKIPLEXPRESSION:
00341                     *w = SKIPGEXPRESSION;
00342                     break;
00343                 case DROPLEXPRESSION:
00344                     *w = DROPGEXPRESSION;
00345                     break;
00346                 case HIDDENLEXPRESSION:
00347                     *w = HIDDENGEXPRESSION;
00348                     break;
00349                 case HIDELEXPRESSION:
00350                     *w = HIDEGEXPRESSION;
00351                     break;
00352                 case UNHIDELEXPRESSION:
00353                     *w = UNHIDEGEXPRESSION;
00354                     break;
00355                 case INTOHIDELEXPRESSION:
00356                     *w = INTOHIDEGEXPRESSION;
00357                     break;
00358                 case DROPHLEXPRESSION:
00359                     *w = DROPHGEXPRESSION;
00360                     break;
00361             }
00362         }
00363         /*
00364         old code
00365         if ( type != LOCALEXPRESSION || *w != STOREDEXPRESSION )
00366             *w = type;
00367         */
00368     }
00369     *q = c; inp = q+1;
00370 } while ( c == ',' );
00371 if ( c ) {
00372     MesPrint("&Illegal object in local or global redefinition");
00373     error = 1;
00374 }
00375 }
00376 return(error);
00377 }
00378
00379 /*
00380 #] DoExpr:
00381 #[ CoIdOld :
00382 */
00383

```

```
00384 int CoIdOld(UBYTE *inp)
00385 {
00386     AC.idoption = 0;
00387     return(CoIdExpression(inp, TYPEIDOLD));
00388 }
00389
00390 /*
00391     #] CoIdOld :
00392     #[ CoId :
00393 */
00394
00395 int CoId(UBYTE *inp)
00396 {
00397     AC.idoption = 0;
00398     return(CoIdExpression(inp, TYPEIDNEW));
00399 }
00400
00401 /*
00402     #] CoId :
00403     #[ CoIdNew :
00404 */
00405
00406 int CoIdNew(UBYTE *inp)
00407 {
00408     AC.idoption = 0;
00409     return(CoIdExpression(inp, TYPEIDNEW));
00410 }
00411
00412 /*
00413     #] CoIdNew :
00414     #[ CoDisorder :
00415 */
00416
00417 int CoDisorder(UBYTE *inp)
00418 {
00419     AC.idoption = SUBDISORDER;
00420     return(CoIdExpression(inp, TYPEIDNEW));
00421 }
00422
00423 /*
00424     #] CoDisorder :
00425     #[ CoMany :
00426 */
00427
00428 int CoMany(UBYTE *inp)
00429 {
00430     AC.idoption = SUBMANY;
00431     return(CoIdExpression(inp, TYPEIDNEW));
00432 }
00433
00434 /*
00435     #] CoMany :
00436     #[ CoMulti :
00437 */
00438
00439 int CoMulti(UBYTE *inp)
00440 {
00441     AC.idoption = SUBMULTI;
00442     return(CoIdExpression(inp, TYPEIDNEW));
00443 }
00444
00445 /*
00446     #] CoMulti :
00447     #[ CoIfMatch :
00448 */
00449
00450 int CoIfMatch(UBYTE *inp)
00451 {
00452     AC.idoption = SUBAFTER;
00453     return(CoIdExpression(inp, TYPEIDNEW));
00454 }
00455
00456 /*
00457     #] CoIfMatch :
00458     #[ CoIfNoMatch :
00459 */
00460
00461 int CoIfNoMatch(UBYTE *inp)
00462 {
00463     AC.idoption = SUBAFTERNOT;
00464     return(CoIdExpression(inp, TYPEIDNEW));
00465 }
00466
00467 /*
00468     #] CoIfNoMatch :
00469     #[ CoOnce :
00470 */
```

```

00471
00472 int CoOnce(UBYTE *inp)
00473 {
00474     AC.idoption = SUBONCE;
00475     return(CoIdExpression(inp,TYPEIDNEW));
00476 }
00477
00478 /*
00479     #] CoOnce :
00480     #[ CoOnly :
00481 */
00482
00483 int CoOnly(UBYTE *inp)
00484 {
00485     AC.idoption = SUBONLY;
00486     return(CoIdExpression(inp,TYPEIDNEW));
00487 }
00488
00489 /*
00490     #] CoOnly :
00491     #[ CoSelect :
00492 */
00493
00494 int CoSelect(UBYTE *inp)
00495 {
00496     AC.idoption = SUBSELECT;
00497     return(CoIdExpression(inp,TYPEIDNEW));
00498 }
00499
00500 /*
00501     #] CoSelect :
00502     #[ CoIdExpression :
00503
00504     First finish dealing with secondary keywords
00505 */
00506
00507 int CoIdExpression(UBYTE *inp, int type)
00508 {
00509     GETIDENTITY
00510     int i, j, idhead, error = 0, MinusSign = 0, opt, retcode;
00511     WORD *w, *s, *m, *mm, *ww, *FirstWork, *OldWork, cl, numsets = 0,
00512         oldnumrhs, *ow, oldEside;
00513     UBYTE *p, *pp, c;
00514     CBUF *C = cbuf+AC.cbufnum;
00515     LONG oldcpointer, x;
00516     FirstWork = OldWork = AT.WorkPointer;
00517 /*
00518     Don't forget to change in StudyPattern if we change/add_to the
00519     following setup.
00520     if ( type == TYPEIF ) idhead = IDHEAD-1;
00521     else
00522 */
00523     idhead = IDHEAD;
00524     AR.CurExpr = -1;
00525     w = AT.WorkPointer;
00526     *w++ = type;
00527     *w++ = idhead + SUBEXPSIZE;
00528     w++;
00529     if ( idhead >= IDHEAD ) *w++ = -1;
00530 #if IDHEAD > 4
00531     for ( i = 4; i < idhead; i++ ) *w++ = 0;
00532 #endif
00533     while ( *inp == ',' ) inp++;
00534     p = inp;
00535     if ( AC.idoption == SUBSELECT ) {
00536         p--;
00537         goto findsets;
00538     }
00539     else if ( ( AC.idoption == SUBAFTER ) || ( AC.idoption == SUBAFTERNOT ) ) {
00540         while ( *p && *p != '=' && *p != ',' ) {
00541             if ( *p == '(' ) SKIPBRA4(p)
00542             else if ( *p == '{' ) SKIPBRA5(p)
00543             else if ( *p == '[' ) SKIPBRA1(p)
00544             else p++;
00545         }
00546         if ( *p == '=' || *inp != '-' || inp[1] != '>' ) {
00547             MesPrint("&Illegal use if if[no]match in id statement");
00548             error = 1; goto AllDone;
00549         }
00550         if ( *p == 0 ) {
00551             MesPrint("&id-statement without = sign");
00552             error = 1; goto AllDone;
00553         }
00554         inp += 2; pp = inp;
00555         goto readlabel;
00556     }
00557     for(;;) {

```

```

00558     while ( *p && *p != '=' && *p != ',' ) {
00559         if ( *p == '(' ) SKIPBRA4(p)
00560         else if ( *p == '{' ) SKIPBRA5(p)
00561         else if ( *p == '[' ) SKIPBRA1(p)
00562         else p++;
00563     }
00564     if ( *p == '=' ) break;
00565     if ( *p == 0 ) {
00566         MesPrint("&id-statement without = sign");
00567         error = 1; goto AllDone;
00568     }
00569     /*
00570     We have either a secondary option or a syntax error
00571     */
00572     pp = inp;
00573     while ( FG.cTable[*pp] == 0 ) pp++;
00574     c = *pp; *pp = 0;
00575     i = sizeof(IdOptions)/sizeof(struct id_options);
00576     while ( --i >= 0 ) {
00577         if ( StrICmp(inp,IdOptions[i].name) == 0 ) break;
00578     }
00579     if ( i < 0 ) {
00580         MesPrint("&Illegal option %s in id-statement",inp);
00581         *pp = c; error = 1; p++; inp = p; continue;
00582     }
00583     opt = IdOptions[i].code;
00584     *pp = c;
00585     inp = pp+1;
00586     switch ( opt ) {
00587     case SUBDISORDER:
00588         if ( pp != p ) goto IllField;
00589         AC.idoption |= SUBDISORDER;
00590         p++; inp = p;
00591         break;
00592     case SUBSELECT:
00593         if ( p != pp ) goto IllField;
00594         if ( ( AC.idoption & SUBMASK ) != 0 ) {
00595             if ( AC.idoption == SUBMULTI && type == TYPEIF ) {}
00596             else {
00597                 MesPrint("&Conflicting options in id-statement");
00598                 error = 1;
00599             }
00600         }
00601     findsets;;
00602     /*
00603     Now we read the sets
00604     */
00605     numsets = 0;
00606     for(;;) {
00607         inp = ++p;
00608         while ( *p && *p != '=' && *p != ',' ) {
00609             if ( *p == '(' ) SKIPBRA4(p)
00610             else if ( *p == '{' ) SKIPBRA5(p)
00611             else if ( *p == '[' ) SKIPBRA1(p)
00612             else p++;
00613         }
00614         if ( *p == '=' ) break;
00615         if ( *p == 0 ) {
00616             MesPrint("&id-statement without = sign");
00617             error = 1; goto AllDone;
00618         }
00619     /*
00620     We have a set at inp.
00621     */
00622     if ( *inp == '{' ) {
00623         if ( p[-1] != '}' ) {
00624             c = *p; *p = 0;
00625             MesPrint("&Illegal temporary set: %s",inp);
00626             error = 1; *p = c;
00627         }
00628         else {
00629             inp++;
00630             c = p[-1]; p[-1] = 0;
00631             c1 = DoTempSet(inp,p-1);
00632             *w++ = c1;
00633             p[-1] = c;
00634             numsets++;
00635             if ( w[-1] < 0 ) error = 1;
00636         }
00637     }
00638     else {
00639         c = *p; *p = 0;
00640         if ( GetName(AC.varnames,inp,&c1,NOAUTO) != CSET ) {
00641             MesPrint("&%s is not a set",inp);
00642             error = 1;
00643         }
00644     }

```

```

00645             if ( c1 < AM.NumFixedSets ) {
00646                 MesPrint("&Built in sets are not allowed in the select option");
00647                 error = 1;
00648             }
00649             else if ( Sets[c1].type == CRANGE ) {
00650                 MesPrint("&Ranged sets are not allowed in the select option");
00651                 error = 1;
00652             }
00653             numsets++;
00654             *w++ = c1;
00655         }
00656         *p = c;
00657     }
00658 }
00659 /*
00660     Now exchange the positions a bit.
00661     Regular stuff at OldWork, numsets sets at FirstWork[idhead]
00662 */
00663 OldWork = w;
00664 for ( i = 0; i < idhead; i++ ) *w++ = FirstWork[i];
00665 AC.idoption = SUBSELECT;
00666 break;
00667 case SUBAFTER:
00668 case SUBAFTERNOT:
00669     if ( type == TYPEIF ) {
00670         MesPrint("&The if[no]match->label option is not allowed in an if statement");
00671         error = 1; goto AllDone;
00672     }
00673     if ( pp[0] != '-' || pp[1] != '>' ) goto IllField;
00674     pp += 2; /* points now at the label */
00675     inp = pp;
00676     AC.idoption |= opt;
00677 readlabel:
00678     while ( FG.cTable[*pp] <= 1 ) pp++;
00679     if ( pp != p ) {
00680         c = *p; *p = 0;
00681         MesPrint("&Illegal label %s in if[no]match option of id-statement",inp);
00682         *p = c; error = 1; inp = p+1; continue;
00683     }
00684     c = *p; *p = 0;
00685     OldWork[3] = GetLabel(inp);
00686     *p++ = c; inp = p;
00687     break;
00688 case SUBALL:
00689     x = 0;
00690     if ( *pp == '(' ) {
00691         if ( FG.cTable[*inp] == 1 ) {
00692             while ( *inp >= '0' && *inp <= '9' ) x = 10*x+*inp++ - '0';
00693         }
00694         else {
00695             pp++;
00696             while ( FG.cTable[*inp] == 0 ) inp++;
00697             c = *inp; *inp = 0;
00698             if ( StrICont(pp,(UBYTE *)"normalize") != 0 ) goto IllOpt;
00699             *inp = c;
00700             OldWork[4] |= NORMALIZEFLAG;
00701         }
00702         if ( *inp != ')' || inp+1 != p ) {
00703             c = *inp; *inp = 0;
00704 IllOpt:
00705             MesPrint("&Illegal ALL option in id-statement: ",pp);
00706             *inp++ = c;
00707             error = 1;
00708             continue;
00709         }
00710         pp = inp;
00711         inp = pp+1;
00712     }
00713 /*
00714     Note that the following statement limits x to
00715 */
00716     if ( x > MAXPOSITIVE ) {
00717         MesPrint("&Requested maximum number of matches %1 in ALL option in id-statement is
greater than %1 ",x,MAXPOSITIVE);
00718         error = 1;
00719     }
00720     OldWork[5] = x;
00721     if ( type != TYPEIDNEW ) {
00722         if ( type == TYPEIDOLD ) {
00723             MesPrint("&Requested ALL option not allowed in idold/also statement.");
00724             error = 1;
00725         }
00726         else if ( type == TYPEIF ) {
00727             MesPrint("&Requested ALL option not allowed in if(match())");
00728             error = 1;
00729         }
00730         else {

```

```

00731             MesPrint("&ALL option only allowed in regular id-statement.");
00732             error = 1;
00733         }
00734     }
00735     p++; inp = p;
00736     AC.idoption = opt;
00737     break;
00738     default:
00739         if ( pp != p ) {
00740 IllField:
00741             c = *p; *p = 0;
00742             MesPrint("&Illegal optionfield %s in id-statement",inp);
00743             *p = c; error = 1; inp = p+1; continue;
00744         }
00745         i = AC.idoption & SUBMASK;
00746         if ( i && i != opt ) {
00747             MesPrint("&Conflicting options in id-statement");
00748             error = 1; continue;
00749         }
00750         else AC.idoption |= opt;
00751         while ( *p == ',' ) p++;
00752         inp = p;
00753         break;
00754     }
00755     if ( ( AC.idoption & SUBMASK ) == 0 ) AC.idoption |= SUBMULTI;
00756     OldWork[2] = AC.idoption;
00757 /*
00758     Now we have a field till the = sign
00759     Now the subexpression prototype
00760 */
00761     AC.ProtoType = w;
00762     *w++ = SUBEXPRESSION;
00763     *w++ = SUBEXPSIZE;
00764     *w++ = C->numrhs+1;
00765     *w++ = 1;
00766     *w++ = AC.cbufnum;
00767     FILLSUB(w)
00768     AC.WildC = w;
00769     AC.NwildC = 0;
00770     AT.WorkPointer = s = w + 4*AM.MaxWildcards + 8;
00771 /*
00772     Now read the LHS
00773 */
00774     ClearWildcardNames();
00775     oldcpointer = AddLHS(AC.cbufnum) - C->Buffer;
00776
00777     *p = 0;
00778     oldnumrhs = C->numrhs;
00779     if ( ( retcode = CompileAlgebra(inp,LHSIDE,AC.ProtoType) ) < 0 ) { error = 1; }
00780     else AC.ProtoType[2] = retcode;
00781     *p = '='; inp = p+1;
00782     AT.WorkPointer = s;
00783     if ( AC.NwildC && SortWild(w,AC.NwildC) ) error = 1;
00784
00785     /* Make the LHS pointers ready */
00786
00787     OldWork[1] = AC.WildC-OldWork;
00788     OldWork[idhead+1] = OldWork[1] - idhead;
00789     w = AC.WildC;
00790     AT.WorkPointer = w;
00791     s = C->rhs[C->numrhs];
00792 /*
00793     Now check whether wildcards get converted to dollars (for PARALLEL)
00794 */
00795     {
00796         WORD *tw, *twstop;
00797         tw = AC.ProtoType; twstop = tw + tw[1]; tw += SUBEXPSIZE;
00798         while ( tw < twstop ) {
00799             if ( *tw == LOADDOLLAR ) {
00800                 AddPotModdollar(tw[2]);
00801             }
00802             tw += tw[1];
00803         }
00804     }
00805 /*
00806     We have the expression in the compiler buffers.
00807     The main level is at lhs[numlhs]
00808     The partial lhs (including ProtoType) is in OldWork (in Workspace)
00809     We need to load the result at w after the prototype
00810     Because these sort routines don't use the Workspace
00811     there should not be a conflict
00812 */
00813     if ( !error && *s == 0 ) {
00814 IllLeft:MesPrint("&Illegal LHS");
00815         AC.lhdollarflag = 0;
00816         return(1);
00817     }

```

```

00818     if ( !error && *(s+s) != 0 ) {
00819         MesPrint("&LHS should be one term only");
00820         return(1);
00821     }
00822     if ( error == 0 ) {
00823         WORD oldpolyfun = AR.PolyFun;
00824         if ( NewSort(BHEAD0) || NewSort(BHEAD0) ) {
00825             if ( !error ) error = 1;
00826             return(error);
00827         }
00828         AN.RepPoint = AT.RepCount + 1;
00829         ow = (WORD *) ((UBYTE *) (AT.WorkPointer)) + AM.MaxTer;
00830         mm = s; ww = ow; i = *mm;
00831         while ( --i >= 0 ) { *ww++ = *mm++; } AT.WorkPointer = ww;
00832         AC.lhdollarflag = 0; oldEside = AR.Eside; AR.Eside = LHSIDE;
00833         AR.Cnumlhs = C->numlhs;
00834         AR.PolyFun = 0;
00835         if ( Generator(BHEAD ow,C->numlhs) ) {
00836             AR.Eside = oldEside;
00837             LowerSortLevel(); LowerSortLevel(); AR.PolyFun = oldpolyfun; goto IllLeft;
00838         }
00839         AR.Eside = oldEside;
00840         AT.WorkPointer = w;
00841         if ( EndSort(BHEAD w,0) < 0 ) { LowerSortLevel(); AR.PolyFun = oldpolyfun; goto IllLeft; }
00842         AR.PolyFun = oldpolyfun;
00843         if ( *w == 0 || *(w+w) != 0 ) {
00844             MesPrint("&LHS must be one term");
00845             AC.lhdollarflag = 0;
00846             return(1);
00847         }
00848         LowerSortLevel();
00849         if ( AC.lhdollarflag ) MarkDirty(w,DIRTYFLAG);
00850     }
00851     AT.WorkPointer = w + *w;
00852     AC.DumNum = 0;
00853 /*
00854     Everything is now after OldWork. We can pop the compilerbuffer.
00855     Next test for illegal things like a coefficient
00856     At this point we have:
00857     w = the term of the LHS
00858 */
00859     C->Pointer = C->Buffer + oldcpointer;
00860     C->numrhs = oldnumrhs;
00861     C->numlhs--;
00862
00863     m = w + *w - 3;
00864     AC.vectorlikeLHS = 0;
00865     if ( !error ) {
00866         if ( m[2] != 3 || m[1] != 1 || *m != 1 ) {
00867             if ( *m == 1 && m[1] == 1 && m[2] == -3 ) {
00868                 MinusSign = 1;
00869             }
00870             else {
00871                 MesPrint("&Coefficient in LHS");
00872                 error = 1;
00873                 AC.DumNum = 0;
00874                 *w -= ABS(m[2])-3;
00875             }
00876         }
00877         if ( *w == 7 && w[1] == INDEX && w[3] < 0 ) {
00878             if ( ( AC.idoption & SUBMASK ) != 0 && ( AC.idoption & SUBMASK ) !=
00879                 SUBMULTI ) {
00880                 MesPrint("&Illegal option for substitution of a vector");
00881                 error = 1;
00882             }
00883             AC.DumNum = AM.IndDum;
00884             OldWork[2] = ( OldWork[2] - ( OldWork[2] & SUBMASK ) ) | SUBVECTOR;
00885             c1 = w[3];
00886             /* We overwrite the LHS */
00887             *w++ = INDTOIND;
00888             *w++ = 4;
00889             *w++ = AC.DumNum + WILDOffset;
00890             *w++ = 0;
00891             w[0] = 5;
00892             w[1] = VECTOR;
00893             w[2] = 4;
00894             w[3] = c1;
00895             w[4] = AC.DumNum + WILDOffset;
00896             OldWork[idhead+1] = w - OldWork - idhead;
00897             AC.vectorlikeLHS = 1;
00898         }
00899         else {
00900             AC.DumNum = 0;
00901             *w -= 3;
00902             i = OldWork[2] & SUBMASK;
00903             m = w + *w;
00904             if ( i == 0 || i == SUBMULTI ) {

```

```

00905         s = w+1;
00906         while ( s < m ) {
00907             if ( *s == SYMBOL ) {
00908                 j = s[1]/2; s += 2;
00909                 while ( --j >= 0 ) {
00910                     if ( ABS(s[1]) > 2*MAXPOWER ) {
00911                         OldWork[2] = ( OldWork[2] - i ) | SUBONCE;
00912                         break;
00913                     }
00914                     s += 2;
00915                 }
00916                 if ( j >= 0 ) break;
00917             }
00918             else if ( *s == DOTPRODUCT ) {
00919                 j = s[1]/3; s += 2;
00920                 while ( --j >= 0 ) {
00921                     if ( ABS(s[2]) > 2*MAXPOWER ) {
00922                         OldWork[2] = ( OldWork[2] - i ) | SUBONCE;
00923                         break;
00924                     }
00925                     else if ( s[1] >= -(2*WILDOFFSET) || s[0] >= -(2*WILDOFFSET) ) {
00926                         OldWork[2] = ( OldWork[2] - i ) | SUBMANY;
00927                         i = SUBMANY;
00928                     }
00929                     s += 3;
00930                 }
00931                 if ( j >= 0 ) break;
00932             }
00933             else {
00934                 OldWork[2] = ( OldWork[2] - i ) | SUBMANY;
00935                 break;
00936             }
00937         }
00938     }
00939     if ( ( OldWork[2] & SUBMASK ) == 0 ) OldWork[2] |= SUBMULTI;
00940 }
00941 if ( ( OldWork[2] & SUBMASK ) == SUBSELECT ) {
00942     /*
00943         Paste the SETSET information after the pattern.
00944         Important note: We will still get function information for the
00945         smart patternmatching after it. To distinguish them we need to have
00946         that SETSET != m*n+1 in which m is the number of words per function
00947         and n the number of functions. Currently (29-may-1997) m = 4.
00948     */
00949     *m++ = SETSET;
00950     *m++ = numsets+2;
00951     s = FirstWork + idhead;
00952     while ( --numsets >= 0 ) *m++ = *s++;
00953 }
00954 else {
00955     m = w + *w;
00956 }
00957 }
00958 /*
00959     We keep the whole thing in OldWork for the moment.
00960     We still have to add the number of the RHS expression.
00961     There is also some opportunity now to be smart about the pattern.
00962     This is needed for complicated wilddcarding with symmetric functions.
00963     We do this in a special routine during compile time to make sure
00964     that we loose as little time as possible (during running) if there
00965     is no need to be smart.
00966 */
00967 *m++ = 0;
00968 OldWork[1] = m - OldWork;
00969 AC.ProtoType = OldWork+idhead;
00970 if ( !error ) {
00971     if ( StudyPattern(OldWork) ) error = 1;
00972 }
00973 AT.WorkPointer = OldWork + OldWork[1];
00974 if ( AC.lhdollarflag ) OldWork[4] |= DOLLARFLAG;
00975 AC.lhdollarflag = 0;
00976 /*
00977     Test whether the id/idold configuration is fine.
00978 */
00979 if ( type == TYPEIDOLD ) {
00980     WORD ci = C->numlhs;
00981     while ( ci >= 1 ) {
00982         if ( C->lhs[ci][0] == TYPEIDNEW ) {
00983             if ( (C->lhs[ci][2] & SUBMASK) == SUBALL ) {
00984                 MesPrint("&Idold/also cannot follow an id,all statement.");
00985                 error = 1;
00986             }
00987             break;
00988         }
00989         else if ( C->lhs[ci][0] == TYPEDETCURDUM ) { ci--; continue; }
00990         else if ( C->lhs[ci][0] == TYPEIDOLD ) { ci--; continue; }
00991         else ci = 0;

```



```

00992     }
00993     if ( ci < 1 ) {
00994         MesPrint("&Idold/also should follow an id/idnew statement.");
00995         error = 1;
00996     }
00997 }
00998 /*
00999 Now the right hand side.
01000 */
01001 if ( type != TYPEIF ) {
01002     if ( ( retcode = CompileAlgebra(inp,RHSIDE,AC.ProtoType) ) < 0 ) error = 1;
01003     else {
01004         AC.ProtoType[2] = retcode;
01005         AC.DumNum = 0;
01006         if ( MinusSign ) { /* Flip the sign of the RHS */
01007             w = C->rhs[retcode];
01008             while ( *w ) { w += *w; w[-1] = -w[-1]; }
01009         }
01010         if ( AC.dumnumflag ) Add2Com(TYPEDETCURDUM)
01011     }
01012 }
01013 /*
01014 Actual adding happens only now after numrhs insertion
01015 */
01016 if ( !error ) { AddNtoL(OldWork[1],OldWork); }
01017 AllDone:
01018 AC.lhdollarflag = 0;
01019 AT.WorkPointer = FirstWork;
01020 return(error);
01021 }
01022
01023 /*
01024 #] CoIdExpression :
01025 #[ CoMultiply :
01026 */
01027
01028 static WORD mularray[13] = { TYPEMULT, SUBEXPSIZE+3, 0, SUBEXPRESSION,
01029     SUBEXPSIZE, 0, 1, 0, 0, 0, 0, 0, 0, 0 };
01030
01031 int CoMultiply(UBYTE *inp)
01032 {
01033     UBYTE *p;
01034     int error = 0, RetCode;
01035     mularray[2] = 0; /* right multiply is default */
01036     while ( *inp == ',' ) inp++;
01037     /* if ( inp[-1] == '-' || inp[-1] == '+' ) inp--; */
01038     p = SkipField(inp,0);
01039     if ( *p ) {
01040         *p = 0;
01041         if ( StrICont(inp,(UBYTE *)"left") == 0 ) mularray[2] = 1;
01042         else if ( StrICont(inp,(UBYTE *)"right") == 0 ) mularray[2] = 0;
01043         else {
01044             MesPrint("&Illegal option in multiply statement or ; forgotten.");
01045             return(1);
01046         }
01047         *p = ',';
01048         inp = p + 1;
01049     }
01050     ClearWildcardNames();
01051     while ( *inp == ',' ) inp++;
01052     AC.ProtoType = mularray+3;
01053     mularray[7] = AC.cbufnum;
01054     if ( ( RetCode = CompileAlgebra(inp,RHSIDE,AC.ProtoType) ) < 0 ) error = 1;
01055     else {
01056         mularray[5] = RetCode;
01057         AddNtoL(SUBEXPSIZE+3,mularray);
01058         if ( AC.dumnumflag ) Add2Com(TYPEDETCURDUM)
01059     }
01060     return(error);
01061 }
01062
01063 /*
01064 #] CoMultiply :
01065 #[ CoFill :
01066
01067 Special additions for tablebase-like tables added 12-aug-2002
01068 */
01069 int CoFill(UBYTE *inp)
01070 {
01071     GETIDENTITY
01072     WORD error = 0, x, xx, funnum, type, *oldwp = AT.WorkPointer;
01073     int i, oldcbufnum = AC.cbufnum, nofill = 0, numover, redef = 0;
01074     WORD *w, *wold, *Tprototype;
01075     UBYTE *p = inp, c, *inpl;
01076     TABLES T = 0, oldT;
01077     LONG newreservation, sum = 0;

```

```

01079     UBYTE *p1, *p2, *p3, *p4, *fake = 0;
01080     int tablestub = 0;
01081     if ( AC.exprfillwarning == 1 ) AC.exprfillwarning = 0;
01082     /*
01083     Read the name of the function and test that it is in the table.
01084     */
01085     p1 = inp;
01086     if ( ( p = SkipAName(inp) ) == 0 ) return(1);
01087     p2 = p;
01088     c = *p; *p = 0;
01089     if ( ( GetVar(inp,&type,&funnum,CFUNCTION,WITHAUTO) == NAMENOTFOUND )
01090     || ( T = functions[funnum].tabl ) == 0 || ( T->numind > 0 && c != '(' ) ) {
01091         MesPrint("%s should be a table with argument(s)",inp);
01092         *p = c; return(1);
01093     }
01094     oldT = T;
01095     *p++ = c;
01096     if ( T->numind == 0 ) {
01097         if ( c == '(' ) {
01098             if ( *p != ')' ) {
01099                 c = *p; *p = 0;
01100                 MesPrint("%s should be a table without arguments",inp);
01101                 *p = c; return(1);
01102             }
01103             else { p++; }
01104         }
01105         else { p--; }
01106         sum = 0;
01107         p3 = p;
01108         goto andagain;
01109     }
01110     w = oldwp;
01111     if ( T->numind < 0 ) { /* Pick up the first index */
01112         ParseSignedNumber(xx,p);
01113         if ( FG.cTable[p[-1]] != 1 || *p != ',' || xx < 1 || ( xx > ( -T->numind - 1 ) ) ) {
01114             MesPrint("&No valid number of table indices in *-table fill statement.");
01115             return(1);
01116         }
01117         *w++ = xx;
01118         p++;
01119     }
01120     else { xx = T->numind; }
01121     for ( sum = 0, i = 0; i < xx; i++ ) {
01122         ParseSignedNumber(x,p);
01123         if ( FG.cTable[p[-1]] != 1 || ( *p != ',' && *p != ')' ) ) {
01124             MesPrint("&Table arguments in fill statement should be numbers");
01125             return(1);
01126         }
01127         if ( T->sparse ) *w++ = x;
01128         else if ( x < T->mm[i].mini || x > T->mm[i].maxi ) {
01129             MesPrint("&Value %d for argument %d of table out of bounds",x,i+1);
01130             error = 1; nofill = 1;
01131         }
01132         else sum += ( x - T->mm[i].mini ) * T->mm[i].size;
01133         if ( *p == ')' ) break;
01134         p++;
01135     }
01136     p3 = p;
01137     if ( T->numind < 0 ) {
01138         for ( ; i < ABS(T->numind)-1; i++ ) *w++ = 0;
01139         xx = -T->numind;
01140     }
01141     if ( *p != ')' || i < ( xx - 1 ) ) {
01142         MesPrint("&Incorrect number of table arguments in fill statement. Should be %d",
01143         ,T->numind);
01144         error = 1; nofill = 1;
01145     }
01146     AT.WorkPointer = w;
01147     if ( T->sparse == 0 ) sum *= TABLEEXTENSION;
01148     andagain:;
01149     AC.cbunum = T->bunum;
01150     if ( T->sparse ) {
01151         i = FindTableTree(T,oldwp,1);
01152         if ( i >= 0 ) {
01153             sum = i + ABS(T->numind);
01154             if ( tablestub == 0 && ( ( T->sparse & 2 ) == 2 ) && ( T->mode != 0 )
01155             && ( AC.vetotablebasefill == 0 ) ) {
01156                 /*
01157                 This redefinition does not need a new stub
01158                 */
01159                 functions[funnum].tabl = T = T->sparse;
01160                 tablestub = 1;
01161                 goto andagain;
01162             }
01163             redef = 1;
01164             goto redef;
01165         }

```

```

01166         if ( T->totind >= T->reserved ) {
01167             if ( T->reserved == 0 ) newreservation = 20;
01168             else newreservation = T->reserved;
01169             while ( T->totind >= newreservation && newreservation < MAXTABLECOMBUF )
01170                 newreservation = 2*newreservation;
01171             if ( newreservation > MAXTABLECOMBUF ) newreservation = MAXTABLECOMBUF;
01172             if ( T->totind >= newreservation ) {
01173                 MesPrint("@More than %ld elements in sparse table",MAXTABLECOMBUF);
01174                 AC.cbufnum = oldcbufnum;
01175                 Terminate(-1);
01176             }
01177             wold = (WORD *)Malloc1(newreservation*sizeof(WORD)*
01178                                   (ABS(T->numind)+TABLEEXTENSION),"tablepointers");
01179             for ( i = T->reserved*(ABS(T->numind)+TABLEEXTENSION)-1; i >= 0; i-- )
01180                 wold[i] = T->tablepointers[i];
01181             if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
01182             T->tablepointers = wold;
01183             T->reserved = newreservation;
01184         }
01185         w = oldwp;
01186         for ( sum = T->totind*(ABS(T->numind)+TABLEEXTENSION), i = 0; i < ABS(T->numind); i++ ) {
01187             T->tablepointers[sum++] = *w++;
01188         }
01189         InsTableTree(T,T->tablepointers+sum-ABS(T->numind));
01190 #if TABLEEXTENSION == 2
01191         T->tablepointers[sum+TABLEEXTENSION-1] = -1; /* New element! */
01192 #else
01193         T->tablepointers[sum+1] = T->bufnum;
01194         T->tablepointers[sum+2] = -1;
01195         T->tablepointers[sum+3] = -1;
01196         T->tablepointers[sum+4] = 0;
01197         T->tablepointers[sum+5] = 0;
01198 #endif
01199     }
01200     else {
01201         if ( !nofill && T->tablepointers[sum] >= 0 ) {
01202 redef::
01203             if ( AC.vetofilling ) nofill = 1;
01204             else {
01205                 Warning("Table element was already defined. New definition will be used");
01206             }
01207         }
01208 #if TABLEEXTENSION == 2
01209         T->tablepointers[sum+TABLEEXTENSION-1] = -1; /* New element! */
01210 #else
01211         T->tablepointers[sum+1] = T->bufnum;
01212         T->tablepointers[sum+2] = -1;
01213         T->tablepointers[sum+3] = -1;
01214         T->tablepointers[sum+4] = 0;
01215         T->tablepointers[sum+5] = 0;
01216 #endif
01217     }
01218     if ( T->numind ) { p++; }
01219     if ( *p != '=' ) {
01220         MesPrint("&Fill statement misses = sign after the table element");
01221         AC.cbufnum = oldcbufnum;
01222         AT.WorkPointer = oldwp;
01223         functions[funnum].tabl = oldT;
01224         return(1);
01225     }
01226     if ( tablestub == 0 && T->mode == 1 && AC.vetotablebasefill == 0 ) {
01227 /*
01228         Here we construct a righthandside from the indices and the wildcards
01229 */
01230         int numfake;
01231         tablestub = 1;
01232         p4 = T->argtail;
01233         while ( *p4 ) p4++;
01234         numfake = (p4-T->argtail)+(p3-p1)+10;
01235
01236         fake = (UBYTE *)Malloc1(numfake*sizeof(UBYTE),"Fill fake rhs");
01237         p = fake;
01238         *p++ = 't'; *p++ = 'b'; *p++ = 'l'; *p++ = '_'; *p++ = '(';
01239         p4 = p1; while ( p4 < p2 ) *p++ = *p4++; *p++ = ',';
01240         p4 = p2+1; while ( p4 < p3 ) *p++ = *p4++;
01241         if ( T->argtail ) {
01242             p4 = T->argtail + 1;
01243             while ( FG.cTable[*p4] == 1 ) p4++;
01244             while ( *p4 ) {
01245                 if ( *p4 == '?' && p[-1] != ',' ) {
01246                     p4++;
01247                     if ( FG.cTable[*p4] == 0 || *p4 == '$' || *p4 == '[' ) {
01248                         p4 = SkipAName(p4);
01249                         if ( *p4 == '[' ) {
01250                             SKIPBRA1(p4);
01251                         }
01252                     }
01253                 }

```

```

01253         else if ( *p4 == '{' ) {
01254             SKIPBRA2(p4);
01255         }
01256         else if ( *p4 ) { *p++ = *p4++; continue; }
01257     }
01258     else *p++ = *p4++;
01259 }
01260 }
01261 *p++ = '>';
01262 *p = 0;
01263 inpl = fake;
01264 }
01265 else {
01266     inpl = ++p;
01267 }
01268 c = 0;
01269 /*
01270  Now we have the indices and p points to the rhs.
01271 */
01272 numover = 0;
01273 AC.tablefilling = funnum;
01274 while ( *inpl ) {
01275     p = SkipField(inpl,0);
01276     c = *p; *p = 0;
01277 #ifdef WITHPTHREADS
01278     Tprototype = T->prototype[0];
01279 #else
01280     Tprototype = T->prototype;
01281 #endif
01282     if ( ( i = CompileAlgebra(inpl,RHSIDE,Tprototype) ) < 0 ) { error = 1; i = 0; }
01283     if ( !nofill ) {
01284         T->tablepointers[sum] = i;
01285         T->tablepointers[sum+1] = T->bufnum;
01286     }
01287     AC.DumNum = 0;
01288     *p = c;
01289     if ( T->sparse || c == 0 ) break;
01290     inpl = ++p;
01291 #if ( TABLEEXTENSION == 2 )
01292     sum++;
01293 #else
01294     sum += 2;
01295 #endif
01296     if ( !nofill && T->tablepointers[sum] >= 0 ) numover++;
01297 #if ( TABLEEXTENSION == 2 )
01298     sum++;
01299 #else
01300     sum += TABLEEXTENSION-2;
01301 #endif
01302 }
01303 if ( AC.exprfillwarning == 1 ) {
01304     AC.exprfillwarning = 2;
01305     Warning("Use of expressions and/or $variables in Fill statements is potentially very
dangerous.");
01306 }
01307 AC.tablefilling = 0;
01308 if ( T->sparse && c != 0 ) {
01309     MesPrint("&In sparse tables one can fill only one element at a time");
01310     error = 1;
01311 }
01312 else if ( numover ) {
01313     if ( numover == 1 )
01314         Warning("one element was overwritten. New definition will be used");
01315     else if ( AC.WarnFlag )
01316         MesPrint("&Warning: %d elements were overwritten. New definitions will be used",numover);
01317 }
01318 if ( T->sparse ) {
01319     if ( redef == 0 ) T->totind++;
01320 }
01321 else T->defined++;
01322 /*
01323 NumSets = AC.SetList.numtemp;
01324 NumSetElements = AC.SetElementList.numtemp;
01325 */
01326 if ( fake ) {
01327     M_free(fake,"Fill fake rhs");
01328     fake = 0;
01329     functions[funnum].tabl = T = T->sparse;
01330     p = p3;
01331     goto andagain;
01332 }
01333 AC.cbunum = oldcbunum;
01334 AC.SymChangeFlag = 1;
01335 AT.WorkPointer = oldwp;
01336 functions[funnum].tabl = oldT;
01337 return(error);
01338 }

```

```

01339
01340 /*
01341 #] CoFill :
01342 #[ CoFillExpression :
01343
01344 Syntax: FillExpression table = expression(x1,...,xn);
01345 The arguments should have been bracketed. Each corresponds to one
01346 of the dimensions of the table. Then the bracket with x1^2*x3^4
01347 will fill the (2,0,4) element of the table (if n=3 of course).
01348 Brackets that don't fit will be skipped. It just gives a warning.
01349
01350 New option (13-jul-2005)
01351 Syntax: FillExpression table = expression(f);
01352 The table indices are arguments of the function f which should
01353 have been bracketed before.
01354 */
01355
01356 int CoFillExpression(UBYTE *inp)
01357 {
01358     GETIDENTITY
01359     UBYTE *p, c;
01360     WORD type, funnum, expnum, symnum, numsym = 0, *oldwork = AT.WorkPointer;
01361     WORD *brackets, *term, brasize, *b, *m, *w, *pw, *tstop, zero = 0;
01362     WORD oldcbuf = AC.cbufnum, curelement = 0;
01363     int weneedit, i, j, numzero, pow, numfirst;
01364     TABLES T = 0;
01365     LONG newreservation, numcommu, sum;
01366     POSITION oldposition;
01367     FILEHANDLE *fi;
01368     CBUF *C;
01369     WORD numdummies;
01370
01371     AN.IndDum = AM.IndDum;
01372     if ( ( p = SkipAName(inp) ) == 0 ) return(1);
01373     c = *p; *p = 0;
01374     if ( ( GetVar(inp,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
01375     || ( T = functions[funnum].tabl ) == 0 ) {
01376         MesPrint("%s should be a previously declared table",inp);
01377         *p = c; return(1);
01378     }
01379     *p++ = c;
01380     if ( T->spare ) T = T->spare;
01381     C = cbuf + T->bufnum;
01382     if ( c != '=' ) {
01383         MesPrint("%sNo = sign in FillExpression statement");
01384         return(1);
01385     }
01386     inp = p;
01387     if ( ( p = SkipAName(inp) ) == 0 ) return(1);
01388     c = *p; *p = 0;
01389     if ( ( type = GetName(AC.exprnames,inp,&expnum,NOAUTO) ) == NAMENOTFOUND
01390     || c != '(' || (
01391         Expressions[expnum].status != LOCALEXPRESSION &&
01392         Expressions[expnum].status != SKIPLEXPRESSION &&
01393         Expressions[expnum].status != DROPLEXPRESSION &&
01394         Expressions[expnum].status != GLOBALEXPRESSION &&
01395         Expressions[expnum].status != SKIPGEXPRESSION &&
01396         Expressions[expnum].status != DROPGEXPRESSION ) ) {
01397         MesPrint("%s should be an active expression with arguments",inp);
01398         *p = c; return(1);
01399     }
01400     if ( Expressions[expnum].inmem ) {
01401         MesPrint("%s cannot be used in a FillExpression statement in the same %n\
01402 module that it has been redefined",inp);
01403         *p = c; return(1);
01404     }
01405     *p++ = c;
01406     while ( *p ) {
01407         inp = p;
01408         if ( ( p = SkipAName(inp) ) == 0 ) return(1);
01409         c = *p; *p = 0;
01410
01411         if ( GetVar(inp,&type,&symnum,-1,NOAUTO) == NAMENOTFOUND ) {
01412             MesPrint("%s should be a previously declared symbol or function",inp);
01413             *p = c; return(1);
01414         }
01415         else if ( type == CSYMBOL ) {
01416             *p++ = c;
01417             *AT.WorkPointer++ = symnum;
01418             numsym++;
01419         }
01420         else if ( type == CFUNCTION ) {
01421             numsym = -1;
01422             *p++ = c;
01423             if ( c != ')' ) {
01424                 MesPrint("%sArgument should be a single function or a list of symbols");
01425                 return(1);

```

```

01426         }
01427         symnum += FUNCTION;
01428         *AT.WorkPointer++ = symnum;
01429     }
01430     else {
01431         MesPrint("%&s should be a previously declared symbol or function",inp);
01432         *p = c; return(1);
01433     }
01434     if ( c == ' ' ) break;
01435     if ( c != ',' ) {
01436         MesPrint("%Illegal separator in FillExpression statement");
01437         goto noway;
01438     }
01439 }
01440 if ( *p ) {
01441     MesPrint("%Illegal end of FillExpression statement");
01442     goto noway;
01443 }
01444 /*
01445     We have the number of the table in funnum.
01446     The number of the expression in expnum, the table struct in T
01447     and either the numbers of the symbols in oldwork (there are numsym of them)
01448     or the number of the function in oldwork (just one and numsym = -1).
01449     We don't sort them!!!!
01450 */
01451 if ( ( numsym > 0 ) && ( ABS(T->numind) != numsym ) ) {
01452     MesPrint("%This table needs %d symbols for its array indices");
01453     goto noway;
01454 }
01455 EXCHINOUT
01456 #ifdef WITHMPI
01457 /*
01458     * The workers can't access to the data of the input expression. We need to
01459     * broadcast it to all the workers.
01460 */
01461 PF_BroadcastExpr(&Expressions[expnum], AR.infile);
01462 if ( PF.me == MASTER ) {
01463     /*
01464         * Restore the file position on the master.
01465     */
01466     POSITION pos;
01467     SetEndScratch(AR.infile, &pos);
01468 }
01469 #endif
01470 fi = AR.infile;
01471 if ( fi->handle >= 0 ) {
01472     PUTZERO(oldposition);
01473     SeekFile(fi->handle,&oldposition,SEEK_CUR);
01474     SetScratch(fi,&(Expressions[expnum].onfile));
01475     if ( ISNEGPOS(Expressions[expnum].onfile) ) {
01476         MesPrint("%File error in FillExpression");
01477         BACKINOUT
01478         goto noway;
01479     }
01480 }
01481 else {
01482 /*
01483     Note: Because everything fits inside memory we never get problems
01484     with excessive file sizes.
01485 */
01486     SETBASEPOSITION(oldposition,(UBYTE *) (fi->POfill)-(UBYTE *) (fi->PObuffer));
01487     fi->POfill = (WORD *) ((UBYTE *) (fi->PObuffer) + BASEPOSITION(Expressions[expnum].onfile));
01488 }
01489 pw = AT.WorkPointer;
01490 if ( numsym < 0 ) { brackets = pw + 1; }
01491 else { brackets = pw + numsym; }
01492 brasize = -1; wenedit = 0; /* stands for we need it */
01493 term = (WORD *) ((UBYTE *) (brackets)) + AM.MaxTer;
01494 AT.WorkPointer = (WORD *) ((UBYTE *) (term)) + AM.MaxTer;
01495 AC.cbunum = T->bunum;
01496 AC.tablefilling = funnum;
01497 if ( GetTerm(BHEAD term) > 0 ) { /* Skip prototype */
01498     while ( GetTerm(BHEAD term) > 0 ) {
01499         GETSTOP(term,tstop);
01500         w = m = term + 1;
01501         while ( m < tstop && *m != HAAKJE ) m += m[1];
01502         if ( *m != HAAKJE ) {
01503             MesPrint("%Illegal attempt to put an expression without brackets in a table");
01504             BACKINOUT
01505             goto noway;
01506         }
01507         if ( brasize == m - w ) {
01508             b = brackets;
01509             while ( *b == *w && w < m ) { b++; w++; }
01510             if ( w == m ) { /* Same as current bracket. Copy. */
01511                 if ( wenedit ) {
01512                     m += m[1] - 1;

```

```

01513         *m = *term - (m-term);
01514         AddNtoC(AC.cbufnum,*m,m,3);
01515         numdummies = DetCurDum(BHEAD term) - AM.IndDum;
01516         if ( numdummies > T->numdummies ) T->numdummies = numdummies;
01517     }
01518     continue; /* Next term */
01519 }
01520 }
01521 if ( weneedit ) {
01522     AddNtoC(AC.cbufnum,1,&zero,4); /* Terminate old bracket */
01523     numcommu = numcommute(C->rhs[curelement],&(C->NumTerms[curelement]));
01524     C->CanCommu[curelement] = numcommu;
01525 }
01526 b = brackets; w = term + 1;
01527 if ( numsym < 0 ) pw = oldwork + 1;
01528 else pw = oldwork + numsym;
01529 while ( w < m ) *b++ = *w++;
01530 brasize = b - brackets;
01531 /*
01532 Now compute the element. See whether we need it
01533 */
01534 if ( numsym < 0 ) {
01535     WORD *bb, bnum;
01536     if ( *brackets != symnum || brasize != brackets[1] ) {
01537         weneedit = 0; continue; /* Cannot work! */
01538     }
01539 /*
01540 Now count the number of arguments and whether they are numbers
01541 */
01542 b = brackets + FUNHEAD;
01543 bb = brackets+brackets[1];
01544 i = 0;
01545 if ( T->numind < 0 ) {
01546     bnum = b[1]+1;
01547     if ( bnum > -T->numind ) {
01548         weneedit = 0; continue; /* Cannot work! */
01549     }
01550 }
01551 else bnum = T->numind;
01552 while ( b < bb ) {
01553     if ( *b != -SNUMBER ) break;
01554     i++;
01555     b += 2;
01556 }
01557 if ( b < bb || i != bnum ) {
01558     weneedit = 0; continue; /* Cannot work! */
01559 }
01560 }
01561 else if ( brasize > 0 && ( *brackets != SYMBOL
01562 || brackets[1] < brasize || (brackets[1]-2) > numsym*2 ) ) {
01563     weneedit = 0; continue; /* Cannot work! */
01564 }
01565 numzero = 0; sum = 0;
01566 numfirst = 0;
01567 if ( numsym > 0 ) {
01568     for ( i = 0; i < numsym; i++ ) {
01569         if ( brasize > 0 ) {
01570             b = brackets + 2; j = brackets[1]-2;
01571             while ( j > 0 ) {
01572                 if ( *b == oldwork[i] ) break;
01573                 j -= 2; b += 2;
01574             }
01575             if ( j <= 0 ) { /* it was not there */
01576                 numzero++; pow = 0;
01577                 if ( 2*numzero+brackets[1]-2 > numsym*2 ) {
01578                     weneedit = 0; goto nextterm;
01579                 }
01580             }
01581             else pow = b[1];
01582         }
01583         else pow = 0;
01584         if ( T->sparse ) {
01585             if ( T->numind < 0 ) {
01586                 if ( i == 0 ) {
01587                     numfirst = pow;
01588                     if ( pow > -T->numind ) {
01589                         weneedit = 0; goto nextterm;
01590                     }
01591                 }
01592                 else if ( i > pow ) {
01593                     weneedit = 0; goto nextterm;
01594                 }
01595             }
01596             *pw++ = pow;
01597         }
01598         else if ( pow < T->mm[i].mini || pow > T->mm[i].maxi ) {
01599             weneedit = 0; goto nextterm;

```

```

01600         }
01601         else sum += ( pow - T->mm[i].mini ) * T->mm[i].size;
01602     }
01603 }
01604 else {
01605     WORD xx;
01606     b = brackets + FUNHEAD;
01607     sum = 0;
01608 /*
01609     Now scan the arguments of the function.
01610     We did check already the number and type of the arguments.
01611 */
01612     xx = (brackets[1]-FUNHEAD)/2;
01613     for ( i = 0; i < xx; i++ ) {
01614         pow = b[1];
01615         b += 2;
01616         if ( T->sparse ) {
01617             if ( T->numind < 0 ) {
01618                 if ( i == 0 ) {
01619                     numfirst = pow;
01620                     if ( pow >= -T->numind ) {
01621                         weneedit = 0; goto nextterm;
01622                     }
01623                 }
01624             }
01625             *pw++ = pow;
01626         }
01627         else if ( pow < T->mm[i].mini || pow > T->mm[i].maxi ) {
01628             weneedit = 0; goto nextterm;
01629         }
01630         else sum += ( pow - T->mm[i].mini ) * T->mm[i].size;
01631     }
01632 }
01633 if ( T->numind < 0 ) {
01634     for ( i = numfirst+1; i < -T->numind; i++ ) *pw++ = 0;
01635 }
01636 weneedit = 1;
01637 if ( T->sparse ) {
01638     if ( numsym < 0 ) pw = oldwork + 1;
01639     else pw = oldwork + ABS(T->numind);
01640     i = FindTableTree(T,pw,1);
01641     if ( i >= 0 ) {
01642         sum = i+ABS(T->numind);
01643 /*
01644 Wrong!!!!
01645 */
01646         C->rhs[T->tablepointers[sum]] = C->Pointer;
01647         C->Pointer--; /* Back up over the zero */
01648         goto newentry;
01649     }
01650     if ( T->totind >= T->reserved ) {
01651         if ( T->reserved == 0 ) newreservation = 20;
01652         else newreservation = T->reserved;
01653 /*---Copied from Fill-----*/
01654         while ( T->totind >= newreservation && newreservation < MAXTABLECOMBUF )
01655             newreservation = 2*newreservation;
01656         if ( newreservation > MAXTABLECOMBUF ) newreservation = MAXTABLECOMBUF;
01657         if ( T->totind >= newreservation ) {
01658             MesPrint("@More than %ld elements in sparse table",MAXTABLECOMBUF);
01659             AC.cbufnum = oldcbuf;
01660             AT.WorkPointer = oldwork;
01661             Terminate(-1);
01662         }
01663 /*---Copied from Fill-----*/
01664         if ( T->totind >= newreservation ) {
01665             MesPrint("@More than %ld elements in sparse table",MAXTABLECOMBUF);
01666             AC.cbufnum = oldcbuf;
01667             AT.WorkPointer = oldwork;
01668             Terminate(-1);
01669         }
01670         w = (WORD *)Malloca1(newreservation*sizeof(WORD)*
01671             (ABS(T->numind)+TABLEEXTENSION),"tablepointers");
01672         for ( i = T->reserved*(ABS(T->numind)+TABLEEXTENSION)-1; i >= 0; i-- )
01673             w[i] = T->tablepointers[i];
01674         if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
01675         T->tablepointers = w;
01676         T->reserved = newreservation;
01677     }
01678     if ( numsym < 0 ) pw = oldwork + 1;
01679     else pw = oldwork + numsym;
01680     for ( sum = T->totind*(ABS(T->numind)+TABLEEXTENSION), i = 0; i < ABS(T->numind); i++
01681 ) {
01682         T->tablepointers[sum++] = *pw++;
01683     }
01684     InsTableTree(T,T->tablepointers+sum-ABS(T->numind));
01685     (T->totind)++;
01686 }
01687 #if ( TABLEEXTENSION != 2 )

```



```

01686         else {
01687             sum *= TABLEEXTENSION;
01688         }
01689 #endif
01690 /*
01691     Start a new entry. Copy the element.
01692 */
01693     AddRHS(T->bufnum,0);
01694     T->tablepointers[sum] = C->numrhs;
01695 #if ( TABLEEXTENSION == 2 )
01696     T->tablepointers[sum+TABLEEXTENSION-1] = -1;
01697 #else
01698     T->tablepointers[sum+1] = T->bufnum;
01699     T->tablepointers[sum+2] = -1;
01700     T->tablepointers[sum+3] = -1;
01701     T->tablepointers[sum+4] = 0;
01702     T->tablepointers[sum+5] = 0;
01703 #endif
01704 newentry: if ( *m == HAAKJE ) { m += m[1] - 1; }
01705           else m--;
01706           *m = *term - (m-term);
01707           AddNtoC(AC.cbufnum,*m,m,5);
01708           curelement = T->tablepointers[sum];
01709 nextterm:
01710     }
01711     if ( weneedit ) {
01712         AddNtoC(AC.cbufnum,1,&zero,6); /* Terminate old bracket */
01713         numcommu = numcommute(C->rhs[curelement],&(C->NumTerms[curelement]));
01714         C->CanCommu[curelement] = numcommu;
01715     }
01716 }
01717 if ( fi->handle >= 0 ) {
01718     SetScratch(fi,&(oldposition));
01719 }
01720 else {
01721     fi->POfill = (WORD *) ((UBYTE *) (fi->PObuffer) + BASEPOSITION(oldposition));
01722 }
01723 BACKINOUT
01724 AC.cbufnum = oldcbuf;
01725 AC.tablefilling = 0;
01726 AT.WorkPointer = oldwork;
01727 return(0);
01728 noway:
01729 BACKINOUT
01730 AC.cbufnum = oldcbuf;
01731 AC.tablefilling = 0;
01732 AT.WorkPointer = oldwork;
01733 return(1);
01734 }
01735
01736 /*
01737 #] CoFillExpression :
01738 #[ CoPrintTable :
01739
01740 Syntax
01741     PrintTable [+f] [+s] tablename [>[>] file];
01742     All defined elements are written with individual Fill statements.
01743     If a file is specified, the result is written to file only.
01744     The flags of the print statement apply as much as possible.
01745     We make use of the regular write routines.
01746 */
01747
01748 int CoPrintTable(UBYTE *inp)
01749 {
01750     GETIDENTITY
01751     int fflag = 0, sflag = 0, addflag = 0, error = 0, sum, i, j;
01752     UBYTE *filename, *p, c, buffer[100], *s, *oldoutputline = AO.OutputLine;
01753     WORD type, funnum, *expr, *m, num;
01754     TABLES T = 0;
01755     WORD oldSkip = AO.OutSkip, oldMode = AC.OutputMode, oldHandle = AC.LogHandle;
01756     WORD oldType = AO.PrintType, *oldwork = AT.WorkPointer;
01757     UBYTE *oldFill = AO.OutFill, *oldLine = AO.OutputLine;
01758 #ifdef WITHMPI
01759     if ( PF.me != MASTER ) return 0;
01760 #endif
01761 /*
01762     First the flags
01763 */
01764     while ( *inp == '+' ) {
01765         inp++;
01766         if ( *inp == 'f' || *inp == 'F' ) { fflag = 1; inp++; }
01767         else if ( *inp == 's' || *inp == 'S' ) { sflag = PRINTONETERM; inp++; }
01768         else {
01769             MesPrint("%Illegal + option in PrintTable statement");
01770             error = 1; inp++;
01771         }
01772     } while ( *inp != ',' && *inp && *inp != '+' ) {

```

```

01773         if ( !error ) {
01774             if ( *inp ) {
01775                 MesPrint("&Illegal + option in PrintTable statement");
01776                 inp++;
01777             }
01778             else {
01779                 MesPrint("&Unfinished PrintTable statement");
01780                 return(1);
01781             }
01782             error = 1;
01783         }
01784         inp++;
01785     }
01786     if ( *inp == ',' ) inp++;
01787 }
01788 /*
01789 Now the name of the table
01790 */
01791 if ( ( p = SkipAName(inp) ) == 0 ) return(1);
01792 c = *p; *p = 0;
01793 if ( ( GetVar(inp,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
01794 || ( T = functions[funnum].tabl ) == 0 ) {
01795     MesPrint("&%s should be a previously declared table",inp);
01796     *p = c; return(1);
01797 }
01798 if ( T->sparse && T->mode == 1 ) T = T->sparse;
01799 *p++ = c;
01800 /*
01801 Check for a filename. Runs to the end of the statement.
01802 */
01803 filename = 0;
01804 if ( c == '>' ) {
01805     if ( *p == '>' ) { addflag = 1; p++; }
01806     filename = p;
01807 }
01808 else filename = 0;
01809
01810 if ( filename ) {
01811     if ( addflag ) AC.LogHandle = OpenAddFile((char *)filename);
01812     else AC.LogHandle = CreateFile((char *)filename);
01813     if ( AC.LogHandle < 0 ) {
01814         MesPrint("&Cannot open file '%s' properly",filename);
01815         error = 1; goto finally;
01816     }
01817     AO.PrintType = PRINTLFILE;
01818 }
01819 else if ( fflag && AC.LogHandle >= 0 ) {
01820     AO.PrintType = PRINTLFILE;
01821 }
01822 AO.OutFill = AO.OutputLine = (UBYTE *)AT.WorkPointer;
01823 AT.WorkPointer += 2*AC.LineLength;
01824
01825 AO.PrintType |= sflag;
01826 AC.OutputMode = 0;
01827 AO.IsBracket = 0;
01828 AO.OutSkip = 0;
01829 AR.DeferFlag = 0;
01830 AC.outsidefun = 1;
01831 if ( AC.LogHandle == oldHandle ) FiniLine();
01832 AO.OutputLine = AO.OutFill = (UBYTE *)Mallocl(AC.LineLength+20,"PrintTable");
01833 AO.OutStop = AO.OutFill + AC.LineLength;
01834 for ( i = 0; i < T->totind; i++ ) {
01835     if ( !T->sparse && T->tablepointers[i*TABLEEXTENSION] < 0 ) continue;
01836     TokenToLine((UBYTE *)"Fill ");
01837     TokenToLine((UBYTE *) (VARNAME(functions,funnum)));
01838     TokenToLine((UBYTE *) "(");
01839     AO.OutSkip = 3;
01840     if ( T->sparse ) {
01841         sum = i * ( T->numind + TABLEEXTENSION );
01842         for ( j = 0; j < T->numind; j++, sum++ ) {
01843             if ( j > 0 ) TokenToLine((UBYTE *)",");
01844             num = T->tablepointers[sum];
01845             s = buffer; s = NumCopy(num,s);
01846             TokenToLine(buffer);
01847         }
01848         expr = cbuf[T->bufnum].rhs[T->tablepointers[sum]];
01849     }
01850     else {
01851         for ( j = 0; j < T->numind; j++ ) {
01852             if ( j > 0 ) {
01853                 TokenToLine((UBYTE *)",");
01854                 num = T->mm[j].mini + ( i % T->mm[j-1].size ) / T->mm[j].size;
01855             }
01856             else {
01857                 num = T->mm[j].mini + i / T->mm[j].size;
01858             }
01859             s = buffer; s = NumCopy(num,s);

```

```

01860         TokenToLine(buffer);
01861     }
01862     expr = cbuf[T->bufnum].rhs[T->tablepointers[TABLEEXTENSION*i]];
01863 }
01864 TOKENTOLINE(" =", "=");
01865 if ( sflag ) {
01866     FiniLine();
01867     if ( AC.OutputSpaces != NOSPACEFORMAT ) TokenToLine((UBYTE *) " ");
01868 }
01869 m = expr;
01870 /*
01871     WORD lbrac, first;
01872     lbrac = 0; first = 1;
01873     while ( *m ) {
01874         if ( WriteTerm(m,&lbrac,first,1,0) ) {
01875             MesPrint("Error while writing table");
01876             error = 1;
01877             goto finally;
01878         }
01879         first = 0;
01880         m += *m;
01881     }
01882     if ( first ) { TOKENTOLINE(" 0","0") }
01883     else if ( lbrac ) { TOKENTOLINE(" ","") }
01884 */
01885     while ( *m ) m += *m;
01886     if ( m > expr ) {
01887         if ( WriteExpression(expr, (LONG) (m-expr)) ) { error = 1; goto finally; }
01888         AO.OutSkip = 0;
01889     }
01890     else {
01891         TokenToLine((UBYTE *) "0");
01892     }
01893     TokenToLine((UBYTE *) ";" );
01894     FiniLine();
01895 }
01896 M_free(AO.OutputLine, "PrintTable");
01897 AO.OutputLine = AO.OutFill = oldoutputline;
01898 /*
01899     Reset the file pointers and parameters if any. Close file if needed.
01900 */
01901 finally:
01902     AO.OutSkip = oldSkip;
01903     AC.OutputMode = oldMode;
01904     AC.LogHandle = oldHandle;
01905     AO.PrintType = oldType;
01906     AO.OutFill = oldFill;
01907     AO.OutputLine = oldLine;
01908     AT.WorkPointer = oldwork;
01909     AC.outsidefun = 0;
01910     return(error);
01911 }
01912
01913 /*
01914     #] CoPrintTable :
01915     #[ CoAssign :
01916
01917     This statement has an easy syntax:
01918         $name = expression
01919 */
01920
01921 static WORD AssignLHS[14] = { TYPEASSIGN, 3+SUBEXPSIZE, 0,
01922                             SUBEXPRESSION, SUBEXPSIZE, 0, 1, 0, 0,0,0,0,0 };
01923
01924 int CoAssign(UBYTE *inp)
01925 {
01926     int error = 0, retcode;
01927     UBYTE *name, c;
01928     WORD number;
01929     if ( *inp != '$' ) {
01930 nolhs: MesPrint("&assign statement should have a dollar variable in the LHS");
01931         return(1);
01932     }
01933     inp++; name = inp;
01934     if ( FG.cTable[*inp] != 0 ) goto nolhs;
01935     while ( FG.cTable[*inp] < 2 ) inp++;
01936     if ( AP.PreAssignFlag == 2 ) {
01937         if ( *inp == '_' ) inp++;
01938     }
01939     if ( ( *inp == ',' && inp[1] != '=' ) && ( *inp != '=' ) ) {
01940         MesPrint("&assign statement should have only a dollar variable in the LHS");
01941         return(1);
01942     }
01943     c = *inp;
01944     *inp = 0;
01945     if ( GetName(AC.dollarnames, name, &number, NOAUTO) == NAMENOTFOUND ) {
01946         number = AddDollar(name, DOLUNDEFINED, 0, 0);

```

```

01947     }
01948     *inp = c;
01949     if ( c == ',' ) inp++;
01950     *inp++ = '=';
01951     if ( *inp == ',' ) inp++;
01952 /*
01953     Fake a Prototype and read the RHS
01954 */
01955     AssignLHS[7] = AC.cbunum;
01956     retcode = CompileAlgebra(inp,RHSIDE,(AssignLHS+3));
01957     if ( retcode < 0 ) error = 1;
01958     AC.DumNum = 0;
01959 /*
01960     Now add the LHS
01961 */
01962     AssignLHS[2] = number;
01963     AssignLHS[5] = retcode;
01964     AddNtoL(AssignLHS[1],AssignLHS);
01965 /*
01966     Add to the list of potentially modified dollars (for PARALLEL)
01967 */
01968     AddPotModdollar(number);
01969     return(error);
01970 }
01971
01972 /*
01973     #] CoAssign :
01974     #[ CoDeallocateTable :
01975
01976     Syntax: DeallocateTable tablename(s);
01977     Should work only for sparse tables.
01978     Action: Cleans all definitions of elements of a table as if there have
01979            never been any fill statements.
01980 */
01981
01982 int CoDeallocateTable(UBYTE *inp)
01983 {
01984     UBYTE *p, c;
01985     TABLES T = 0;
01986     WORD type, funnum, i;
01987     c = *inp;
01988     while ( c ) {
01989         while ( *inp == ',' ) inp++;
01990         if ( *inp == 0 ) break;
01991         if ( ( p = SkipAName(inp) ) == 0 ) return(1);
01992         c = *p; *p = 0;
01993         if ( ( GetVar(inp,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
01994             || ( T = functions[funnum].tabl ) == 0 ) {
01995             MesPrint("%s should be a previously declared table",inp);
01996             *p = c; return(1);
01997         }
01998         if ( T->sparse == 0 ) {
01999             MesPrint("%s should be a sparse table",inp);
02000             *p = c; return(1);
02001         }
02002         if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
02003         ClearTableTree(T);
02004         for ( i = 0; i < T->buffersfill; i++ ) { /* was <= */
02005             finishcbuf(T->buffers[i]);
02006         }
02007         T->bufnum = inicbufs();
02008         T->buffersfill = 0;
02009         T->buffers[T->buffersfill++] = T->bufnum;
02010         T->tablepointers = 0;
02011         T->boomlijst = 0;
02012         T->totind = 0;
02013         T->reserved = 0;
02014
02015         if ( T->sparse ) {
02016             TABLES TT = T->sparse;
02017             if ( TT->tablepointers ) M_free(TT->tablepointers,"tablepointers");
02018             ClearTableTree(TT);
02019             for ( i = 0; i < TT->buffersfill; i++ ) { /* was <= */
02020                 finishcbuf(TT->buffers[i]);
02021             }
02022             TT->bufnum = inicbufs();
02023             TT->buffersfill = 0;
02024             TT->buffers[TT->buffersfill++] = T->bufnum;
02025             TT->tablepointers = 0;
02026             TT->boomlijst = 0;
02027             TT->totind = 0;
02028             TT->reserved = 0;
02029         }
02030         *p++ = c;
02031         inp = p;
02032     }
02033     return(0);

```

```

02034 }
02035
02036 /*
02037  #] CoDeallocateTable :
02038  #[ CoFactorCache :
02039 */
02040 /*
02050 int CoFactorCache(UBYTE *inp)
02051 {
02052     Code to be added in due time
02053     We need to read 'expression', get its terms through Generator and sort them.
02054     We store the result in the WorkSpace in argument notation.
02055     This will be argin.
02056     Then we do the same with the sequence of factors. They formargout.
02057     The whole is put in the buffer with the call
02058         InsertArg(BHEAD argin,argout,1)
02059     return(0);
02060 }
02061 */
02062
02063 /*
02064  #] CoFactorCache :
02065 */

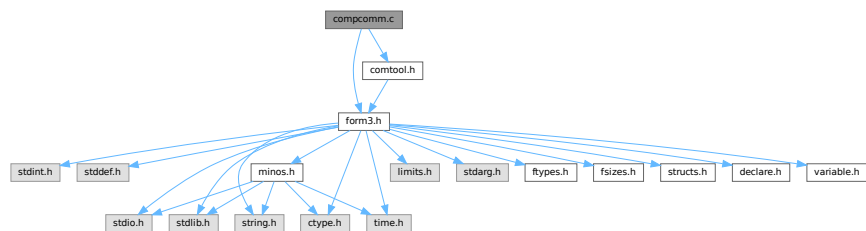
```

4.9 compcomm.c File Reference

```

#include "form3.h"
#include "comtool.h"
Include dependency graph for compcomm.c:

```



Functions

- int [CoFormat](#) (UBYTE *s)
- int [CoCollect](#) (UBYTE *s)
- int [setonoff](#) (UBYTE *s, int *flag, int onvalue, int offvalue)
- int [CoCompress](#) (UBYTE *s)
- int [CoFlags](#) (UBYTE *s, int value)
- int [CoOff](#) (UBYTE *s)
- int [CoOn](#) (UBYTE *s)
- int [CoInsideFirst](#) (UBYTE *s)
- int [CoProperCount](#) (UBYTE *s)
- int [CoDelete](#) (UBYTE *s)
- int [CoKeep](#) (UBYTE *s)
- int [CoFixIndex](#) (UBYTE *s)
- int [CoMetric](#) (UBYTE *s)
- int [DoPrint](#) (UBYTE *s, int par)
- int [CoPrint](#) (UBYTE *s)
- int [CoPrintB](#) (UBYTE *s)
- int [CoNPrint](#) (UBYTE *s)

- int [CoPushHide](#) (UBYTE *s)
- int [CoPopHide](#) (UBYTE *s)
- int [SetExprCases](#) (int par, int setunset, int val)
- int [SetExpr](#) (UBYTE *s, int setunset, int par)
- int [CoDrop](#) (UBYTE *s)
- int [CoNoDrop](#) (UBYTE *s)
- int [CoSkip](#) (UBYTE *s)
- int [CoNoSkip](#) (UBYTE *s)
- int [CoHide](#) (UBYTE *inp)
- int [CoIntoHide](#) (UBYTE *inp)
- int [CoNoIntoHide](#) (UBYTE *inp)
- int [CoNoHide](#) (UBYTE *inp)
- int [CoUnHide](#) (UBYTE *inp)
- int [CoNoUnHide](#) (UBYTE *inp)
- void [AddToCom](#) (int n, WORD *array)
- int [AddComString](#) (int n, WORD *array, UBYTE *thestring, int par)
- int [Add2ComStrings](#) (int n, WORD *array, UBYTE *string1, UBYTE *string2)
- int [CoDiscard](#) (UBYTE *s)
- int [CoContract](#) (UBYTE *s)
- int [CoGoTo](#) (UBYTE *inp)
- int [CoLabel](#) (UBYTE *inp)
- int [DoArgument](#) (UBYTE *s, int par)
- int [CoArgument](#) (UBYTE *s)
- int [CoEndArgument](#) (UBYTE *s)
- int [CoInside](#) (UBYTE *s)
- int [CoEndInside](#) (UBYTE *s)
- int [CoNormalize](#) (UBYTE *s)
- int [CoMakeInteger](#) (UBYTE *s)
- int [CoSplitArg](#) (UBYTE *s)
- int [CoSplitFirstArg](#) (UBYTE *s)
- int [CoSplitLastArg](#) (UBYTE *s)
- int [CoFactArg](#) (UBYTE *s)
- int [DoSymmetrize](#) (UBYTE *s, int par)
- int [CoSymmetrize](#) (UBYTE *s)
- int [CoAntiSymmetrize](#) (UBYTE *s)
- int [CoCycleSymmetrize](#) (UBYTE *s)
- int [CoRCycleSymmetrize](#) (UBYTE *s)
- int [CoWrite](#) (UBYTE *s)
- int [CoNWrite](#) (UBYTE *s)
- int [CoRatio](#) (UBYTE *s)
- int [CoRedefine](#) (UBYTE *s)
- int [CoRenumber](#) (UBYTE *s)
- int [CoSum](#) (UBYTE *s)
- int [CoToTensor](#) (UBYTE *s)
- int [CoToVector](#) (UBYTE *s)
- int [CoTrace4](#) (UBYTE *s)
- int [CoTraceN](#) (UBYTE *s)
- int [CoChisholm](#) (UBYTE *s)
- int [DoChain](#) (UBYTE *s, int option)
- int [CoChainin](#) (UBYTE *s)
- int [CoChainout](#) (UBYTE *s)
- int [CoExit](#) (UBYTE *s)
- int [CoInParallel](#) (UBYTE *s)
- int [CoNotInParallel](#) (UBYTE *s)
- int [DoInParallel](#) (UBYTE *s, int par)

- int [ColnExpression](#) (UBYTE *s)
- int [CoEndlnExpression](#) (UBYTE *s)
- int [CoSetExitFlag](#) (UBYTE *s)
- int [CoTryReplace](#) (UBYTE *p)
- int [CoModulus](#) (UBYTE *inp)
- int [CoRepeat](#) (UBYTE *inp)
- int [CoEndRepeat](#) (UBYTE *inp)
- int [DoBrackets](#) (UBYTE *inp, int par)
- int [CoBracket](#) (UBYTE *inp)
- int [CoAntiBracket](#) (UBYTE *inp)
- int [CoMultiBracket](#) (UBYTE *inp)
- WORD * [CountComp](#) (UBYTE *inp, WORD *to)
- int [Colf](#) (UBYTE *inp)
- int [CoElse](#) (UBYTE *p)
- int [CoElseif](#) (UBYTE *inp)
- int [CoEndIf](#) (UBYTE *inp)
- int [CoWhile](#) (UBYTE *inp)
- int [CoEndWhile](#) (UBYTE *inp)
- int [DoFindLoop](#) (UBYTE *inp, int mode)
- int [CoFindLoop](#) (UBYTE *inp)
- int [CoReplaceLoop](#) (UBYTE *inp)
- int [CoFunPowers](#) (UBYTE *inp)
- int [CoUnitTrace](#) (UBYTE *s)
- int [CoTerm](#) (UBYTE *s)
- int [CoEndTerm](#) (UBYTE *s)
- int [CoSort](#) (UBYTE *s)
- int [CoPolyFun](#) (UBYTE *s)
- int [CoPolyRatFun](#) (UBYTE *s)
- int [CoMerge](#) (UBYTE *inp)
- int [CoStuffle](#) (UBYTE *inp)
- int [CoProcessBucket](#) (UBYTE *s)
- int [CoThreadBucket](#) (UBYTE *s)
- int [DoArgPlode](#) (UBYTE *s, int par)
- int [CoArgExplode](#) (UBYTE *s)
- int [CoArgImplode](#) (UBYTE *s)
- int [CoClearTable](#) (UBYTE *s)
- int [CoDenominators](#) (UBYTE *s)
- int [CoDropCoefficient](#) (UBYTE *s)
- int [CoDropSymbols](#) (UBYTE *s)
- int [CoToPolynomial](#) (UBYTE *inp)
- int [CoFromPolynomial](#) (UBYTE *inp)
- int [CoArgToExtraSymbol](#) (UBYTE *s)
- int [CoExtraSymbols](#) (UBYTE *inp)
- WORD * [GetIfDollarFactor](#) (UBYTE **inp, WORD *w)
- UBYTE * [GetDoParam](#) (UBYTE *inp, WORD **wp, int par)
- int [CoDo](#) (UBYTE *inp)
- int [CoEndDo](#) (UBYTE *inp)
- int [CoFactDollar](#) (UBYTE *inp)
- int [CoFactorize](#) (UBYTE *s)
- int [CoNFactorize](#) (UBYTE *s)
- int [CoUnFactorize](#) (UBYTE *s)
- int [CoNUnFactorize](#) (UBYTE *s)
- int [DoFactorize](#) (UBYTE *s, int par)
- int [CoOptimizeOption](#) (UBYTE *s)
- int [CoPutInside](#) (UBYTE *inp)

- int [CoAntiPutInside](#) (UBYTE *inp)
- int [DoPutInside](#) (UBYTE *inp, int par)
- int [CoSwitch](#) (UBYTE *s)
- int [CoCase](#) (UBYTE *s)
- int [CoBreak](#) (UBYTE *s)
- int [CoDefault](#) (UBYTE *s)
- int [CoEndSwitch](#) (UBYTE *s)
- int [CoSetUserFlag](#) (UBYTE *s)
- int [CoClearUserFlag](#) (UBYTE *s)
- int [CoCreateAllLoops](#) (UBYTE *s)
- int [CoCreateAllPaths](#) (UBYTE *s)
- int [CoCreateAll](#) (UBYTE *s)

4.9.1 Detailed Description

Compiler routines for most statements that don't involve algebraic expressions. Exceptions: all routines involving declarations are in the file [names.c](#) When making new statements one can add the compiler routines here and have a look whether there is already a routine that is similar. In that case one can make a copy and modify it.

Definition in file [compcomm.c](#).

4.9.2 Function Documentation

4.9.2.1 CoFormat()

```
int CoFormat (  
    UBYTE * s )
```

Definition at line [154](#) of file [compcomm.c](#).

4.9.2.2 CoCollect()

```
int CoCollect (  
    UBYTE * s )
```

Definition at line [434](#) of file [compcomm.c](#).

4.9.2.3 setonoff()

```
int setonoff (  
    UBYTE * s,  
    int * flag,  
    int onvalue,  
    int offvalue )
```

Definition at line [496](#) of file [compcomm.c](#).

4.9.2.4 CoCompress()

```
int CoCompress (
    UBYTE * s )
```

Definition at line 512 of file [compcomm.c](#).

4.9.2.5 CoFlags()

```
int CoFlags (
    UBYTE * s,
    int value )
```

Definition at line 561 of file [compcomm.c](#).

4.9.2.6 CoOff()

```
int CoOff (
    UBYTE * s )
```

Definition at line 596 of file [compcomm.c](#).

4.9.2.7 CoOn()

```
int CoOn (
    UBYTE * s )
```

Definition at line 662 of file [compcomm.c](#).

4.9.2.8 CoInsideFirst()

```
int CoInsideFirst (
    UBYTE * s )
```

Definition at line 934 of file [compcomm.c](#).

4.9.2.9 CoProperCount()

```
int CoProperCount (
    UBYTE * s )
```

Definition at line 941 of file [compcomm.c](#).

4.9.2.10 CoDelete()

```
int CoDelete (
    UBYTE * s )
```

Definition at line 948 of file [compcomm.c](#).

4.9.2.11 CoKeep()

```
int CoKeep (
    UBYTE * s )
```

Definition at line 993 of file [compcomm.c](#).

4.9.2.12 CoFixIndex()

```
int CoFixIndex (
    UBYTE * s )
```

Definition at line 1005 of file [compcomm.c](#).

4.9.2.13 CoMetric()

```
int CoMetric (
    UBYTE * s )
```

Definition at line 1039 of file [compcomm.c](#).

4.9.2.14 DoPrint()

```
int DoPrint (
    UBYTE * s,
    int par )
```

Definition at line 1047 of file [compcomm.c](#).

4.9.2.15 CoPrint()

```
int CoPrint (
    UBYTE * s )
```

Definition at line 1263 of file [compcomm.c](#).

4.9.2.16 CoPrintB()

```
int CoPrintB (
    UBYTE * s )
```

Definition at line 1270 of file [compcomm.c](#).

4.9.2.17 CoNPrint()

```
int CoNPrint (
    UBYTE * s )
```

Definition at line 1277 of file [compcomm.c](#).

4.9.2.18 CoPushHide()

```
int CoPushHide (
    UBYTE * s )
```

Definition at line [1284](#) of file [compcomm.c](#).

4.9.2.19 CoPopHide()

```
int CoPopHide (
    UBYTE * s )
```

Definition at line [1328](#) of file [compcomm.c](#).

4.9.2.20 SetExprCases()

```
int SetExprCases (
    int par,
    int setunset,
    int val )
```

Definition at line [1363](#) of file [compcomm.c](#).

4.9.2.21 SetExpr()

```
int SetExpr (
    UBYTE * s,
    int setunset,
    int par )
```

Definition at line [1518](#) of file [compcomm.c](#).

4.9.2.22 CoDrop()

```
int CoDrop (
    UBYTE * s )
```

Definition at line [1563](#) of file [compcomm.c](#).

4.9.2.23 CoNoDrop()

```
int CoNoDrop (
    UBYTE * s )
```

Definition at line [1570](#) of file [compcomm.c](#).

4.9.2.24 CoSkip()

```
int CoSkip (
    UBYTE * s )
```

Definition at line 1577 of file [compcomm.c](#).

4.9.2.25 CoNoSkip()

```
int CoNoSkip (
    UBYTE * s )
```

Definition at line 1584 of file [compcomm.c](#).

4.9.2.26 CoHide()

```
int CoHide (
    UBYTE * inp )
```

Definition at line 1591 of file [compcomm.c](#).

4.9.2.27 CoIntoHide()

```
int CoIntoHide (
    UBYTE * inp )
```

Definition at line 1609 of file [compcomm.c](#).

4.9.2.28 CoNoIntoHide()

```
int CoNoIntoHide (
    UBYTE * inp )
```

Definition at line 1627 of file [compcomm.c](#).

4.9.2.29 CoNoHide()

```
int CoNoHide (
    UBYTE * inp )
```

Definition at line 1634 of file [compcomm.c](#).

4.9.2.30 CoUnHide()

```
int CoUnHide (
    UBYTE * inp )
```

Definition at line 1641 of file [compcomm.c](#).

4.9.2.31 CoNoUnHide()

```
int CoNoUnHide (
    UBYTE * inp )
```

Definition at line 1648 of file [compcomm.c](#).

4.9.2.32 AddToCom()

```
void AddToCom (
    int n,
    WORD * array )
```

Definition at line 1655 of file [compcomm.c](#).

4.9.2.33 AddComString()

```
int AddComString (
    int n,
    WORD * array,
    UBYTE * thestring,
    int par )
```

Definition at line 1670 of file [compcomm.c](#).

4.9.2.34 Add2ComStrings()

```
int Add2ComStrings (
    int n,
    WORD * array,
    UBYTE * string1,
    UBYTE * string2 )
```

Definition at line 1729 of file [compcomm.c](#).

4.9.2.35 CoDiscard()

```
int CoDiscard (
    UBYTE * s )
```

Definition at line 1770 of file [compcomm.c](#).

4.9.2.36 CoContract()

```
int CoContract (
    UBYTE * s )
```

Definition at line 1793 of file [compcomm.c](#).

4.9.2.37 CoGoTo()

```
int CoGoTo (
    UBYTE * inp )
```

Definition at line 1822 of file [compcomm.c](#).

4.9.2.38 CoLabel()

```
int CoLabel (
    UBYTE * inp )
```

Definition at line 1841 of file [compcomm.c](#).

4.9.2.39 DoArgument()

```
int DoArgument (
    UBYTE * s,
    int par )
```

Definition at line 1868 of file [compcomm.c](#).

4.9.2.40 CoArgument()

```
int CoArgument (
    UBYTE * s )
```

Definition at line 2119 of file [compcomm.c](#).

4.9.2.41 CoEndArgument()

```
int CoEndArgument (
    UBYTE * s )
```

Definition at line 2126 of file [compcomm.c](#).

4.9.2.42 CoInside()

```
int CoInside (
    UBYTE * s )
```

Definition at line 2152 of file [compcomm.c](#).

4.9.2.43 CoEndInside()

```
int CoEndInside (
    UBYTE * s )
```

Definition at line 2159 of file [compcomm.c](#).

4.9.2.44 CoNormalize()

```
int CoNormalize (
    UBYTE * s )
```

Definition at line 2185 of file [compcomm.c](#).

4.9.2.45 CoMakeInteger()

```
int CoMakeInteger (
    UBYTE * s )
```

Definition at line 2192 of file [compcomm.c](#).

4.9.2.46 CoSplitArg()

```
int CoSplitArg (
    UBYTE * s )
```

Definition at line 2199 of file [compcomm.c](#).

4.9.2.47 CoSplitFirstArg()

```
int CoSplitFirstArg (
    UBYTE * s )
```

Definition at line 2206 of file [compcomm.c](#).

4.9.2.48 CoSplitLastArg()

```
int CoSplitLastArg (
    UBYTE * s )
```

Definition at line 2213 of file [compcomm.c](#).

4.9.2.49 CoFactArg()

```
int CoFactArg (
    UBYTE * s )
```

Definition at line 2220 of file [compcomm.c](#).

4.9.2.50 DoSymmetrize()

```
int DoSymmetrize (
    UBYTE * s,
    int par )
```

Definition at line 2240 of file [compcomm.c](#).

4.9.2.51 CoSymmetrize()

```
int CoSymmetrize (  
    UBYTE * s )
```

Definition at line [2360](#) of file [compcomm.c](#).

4.9.2.52 CoAntiSymmetrize()

```
int CoAntiSymmetrize (  
    UBYTE * s )
```

Definition at line [2367](#) of file [compcomm.c](#).

4.9.2.53 CoCycleSymmetrize()

```
int CoCycleSymmetrize (  
    UBYTE * s )
```

Definition at line [2374](#) of file [compcomm.c](#).

4.9.2.54 CoRCycleSymmetrize()

```
int CoRCycleSymmetrize (  
    UBYTE * s )
```

Definition at line [2381](#) of file [compcomm.c](#).

4.9.2.55 CoWrite()

```
int CoWrite (  
    UBYTE * s )
```

Definition at line [2388](#) of file [compcomm.c](#).

4.9.2.56 CoNWrite()

```
int CoNWrite (  
    UBYTE * s )
```

Definition at line [2413](#) of file [compcomm.c](#).

4.9.2.57 CoRatio()

```
int CoRatio (  
    UBYTE * s )
```

Definition at line [2440](#) of file [compcomm.c](#).

4.9.2.58 CoRedefine()

```
int CoRedefine (
    UBYTE * s )
```

Definition at line [2480](#) of file [compcomm.c](#).

4.9.2.59 CoRenumber()

```
int CoRenumber (
    UBYTE * s )
```

Definition at line [2585](#) of file [compcomm.c](#).

4.9.2.60 CoSum()

```
int CoSum (
    UBYTE * s )
```

Definition at line [2606](#) of file [compcomm.c](#).

4.9.2.61 CoToTensor()

```
int CoToTensor (
    UBYTE * s )
```

Definition at line [2726](#) of file [compcomm.c](#).

4.9.2.62 CoToVector()

```
int CoToVector (
    UBYTE * s )
```

Definition at line [2894](#) of file [compcomm.c](#).

4.9.2.63 CoTrace4()

```
int CoTrace4 (
    UBYTE * s )
```

Definition at line [2964](#) of file [compcomm.c](#).

4.9.2.64 CoTraceN()

```
int CoTraceN (
    UBYTE * s )
```

Definition at line [3051](#) of file [compcomm.c](#).

4.9.2.65 CoChisholm()

```
int CoChisholm (
    UBYTE * s )
```

Definition at line 3111 of file [compcomm.c](#).

4.9.2.66 DoChain()

```
int DoChain (
    UBYTE * s,
    int option )
```

Definition at line 3193 of file [compcomm.c](#).

4.9.2.67 CoChainin()

```
int CoChainin (
    UBYTE * s )
```

Definition at line 3237 of file [compcomm.c](#).

4.9.2.68 CoChainout()

```
int CoChainout (
    UBYTE * s )
```

Definition at line 3249 of file [compcomm.c](#).

4.9.2.69 CoExit()

```
int CoExit (
    UBYTE * s )
```

Definition at line 3259 of file [compcomm.c](#).

4.9.2.70 CoInParallel()

```
int CoInParallel (
    UBYTE * s )
```

Definition at line 3286 of file [compcomm.c](#).

4.9.2.71 CoNotInParallel()

```
int CoNotInParallel (
    UBYTE * s )
```

Definition at line 3296 of file [compcomm.c](#).

4.9.2.72 DoInParallel()

```
int DoInParallel (
    UBYTE * s,
    int par )
```

Definition at line 3311 of file [compcomm.c](#).

4.9.2.73 CoInExpression()

```
int CoInExpression (
    UBYTE * s )
```

Definition at line 3380 of file [compcomm.c](#).

4.9.2.74 CoEndInExpression()

```
int CoEndInExpression (
    UBYTE * s )
```

Definition at line 3433 of file [compcomm.c](#).

4.9.2.75 CoSetExitFlag()

```
int CoSetExitFlag (
    UBYTE * s )
```

Definition at line 3459 of file [compcomm.c](#).

4.9.2.76 CoTryReplace()

```
int CoTryReplace (
    UBYTE * p )
```

Definition at line 3473 of file [compcomm.c](#).

4.9.2.77 CoModulus()

```
int CoModulus (
    UBYTE * inp )
```

Definition at line 3576 of file [compcomm.c](#).

4.9.2.78 CoRepeat()

```
int CoRepeat (
    UBYTE * inp )
```

Definition at line 3714 of file [compcomm.c](#).

4.9.2.79 CoEndRepeat()

```
int CoEndRepeat (
    UBYTE * inp )
```

Definition at line 3737 of file [compcomm.c](#).

4.9.2.80 DoBrackets()

```
int DoBrackets (
    UBYTE * inp,
    int par )
```

Definition at line 3775 of file [compcomm.c](#).

4.9.2.81 CoBracket()

```
int CoBracket (
    UBYTE * inp )
```

Definition at line 3902 of file [compcomm.c](#).

4.9.2.82 CoAntiBracket()

```
int CoAntiBracket (
    UBYTE * inp )
```

Definition at line 3910 of file [compcomm.c](#).

4.9.2.83 CoMultiBracket()

```
int CoMultiBracket (
    UBYTE * inp )
```

Definition at line 3921 of file [compcomm.c](#).

4.9.2.84 CountComp()

```
WORD * CountComp (
    UBYTE * inp,
    WORD * to )
```

Definition at line 4053 of file [compcomm.c](#).

4.9.2.85 Colf()

```
int CoIf (
    UBYTE * inp )
```

Definition at line 4243 of file [compcomm.c](#).

4.9.2.86 CoElse()

```
int CoElse (
    UBYTE * p )
```

Definition at line 4883 of file [compcomm.c](#).

4.9.2.87 CoElseIf()

```
int CoElseIf (
    UBYTE * inp )
```

Definition at line 4910 of file [compcomm.c](#).

4.9.2.88 CoEndIf()

```
int CoEndIf (
    UBYTE * inp )
```

Definition at line 4937 of file [compcomm.c](#).

4.9.2.89 CoWhile()

```
int CoWhile (
    UBYTE * inp )
```

Definition at line 4982 of file [compcomm.c](#).

4.9.2.90 CoEndWhile()

```
int CoEndWhile (
    UBYTE * inp )
```

Definition at line 5003 of file [compcomm.c](#).

4.9.2.91 DoFindLoop()

```
int DoFindLoop (
    UBYTE * inp,
    int mode )
```

Definition at line 5031 of file [compcomm.c](#).

4.9.2.92 CoFindLoop()

```
int CoFindLoop (
    UBYTE * inp )
```

Definition at line 5148 of file [compcomm.c](#).

4.9.2.93 CoReplaceLoop()

```
int CoReplaceLoop (
    UBYTE * inp )
```

Definition at line 5156 of file [compcomm.c](#).

4.9.2.94 CoFunPowers()

```
int CoFunPowers (
    UBYTE * inp )
```

Definition at line 5176 of file [compcomm.c](#).

4.9.2.95 CoUnitTrace()

```
int CoUnitTrace (
    UBYTE * s )
```

Definition at line 5203 of file [compcomm.c](#).

4.9.2.96 CoTerm()

```
int CoTerm (
    UBYTE * s )
```

Definition at line 5238 of file [compcomm.c](#).

4.9.2.97 CoEndTerm()

```
int CoEndTerm (
    UBYTE * s )
```

Definition at line 5286 of file [compcomm.c](#).

4.9.2.98 CoSort()

```
int CoSort (
    UBYTE * s )
```

Definition at line 5313 of file [compcomm.c](#).

4.9.2.99 CoPolyFun()

```
int CoPolyFun (
    UBYTE * s )
```

Definition at line 5352 of file [compcomm.c](#).

4.9.2.100 CoPolyRatFun()

```
int CoPolyRatFun (
    UBYTE * s )
```

Definition at line 5399 of file [compcomm.c](#).

4.9.2.101 CoMerge()

```
int CoMerge (
    UBYTE * inp )
```

Definition at line 5569 of file [compcomm.c](#).

4.9.2.102 CoStuffle()

```
int CoStuffle (
    UBYTE * inp )
```

Definition at line 5625 of file [compcomm.c](#).

4.9.2.103 CoProcessBucket()

```
int CoProcessBucket (
    UBYTE * s )
```

Definition at line 5680 of file [compcomm.c](#).

4.9.2.104 CoThreadBucket()

```
int CoThreadBucket (
    UBYTE * s )
```

Definition at line 5698 of file [compcomm.c](#).

4.9.2.105 DoArgPlode()

```
int DoArgPlode (
    UBYTE * s,
    int par )
```

Definition at line 5728 of file [compcomm.c](#).

4.9.2.106 CoArgExplode()

```
int CoArgExplode (
    UBYTE * s )
```

Definition at line 5784 of file [compcomm.c](#).

4.9.2.107 CoArgImplode()

```
int CoArgImplode (
    UBYTE * s )
```

Definition at line 5791 of file [compcomm.c](#).

4.9.2.108 CoClearTable()

```
int CoClearTable (
    UBYTE * s )
```

Definition at line 5798 of file [compcomm.c](#).

4.9.2.109 CoDenominators()

```
int CoDenominators (
    UBYTE * s )
```

Definition at line 5880 of file [compcomm.c](#).

4.9.2.110 CoDropCoefficient()

```
int CoDropCoefficient (
    UBYTE * s )
```

Definition at line 5909 of file [compcomm.c](#).

4.9.2.111 CoDropSymbols()

```
int CoDropSymbols (
    UBYTE * s )
```

Definition at line 5923 of file [compcomm.c](#).

4.9.2.112 CoToPolynomial()

```
int CoToPolynomial (
    UBYTE * inp )
```

Definition at line 5946 of file [compcomm.c](#).

4.9.2.113 CoFromPolynomial()

```
int CoFromPolynomial (
    UBYTE * inp )
```

Definition at line 6021 of file [compcomm.c](#).

4.9.2.114 CoArgToExtraSymbol()

```
int CoArgToExtraSymbol (
    UBYTE * s )
```

Definition at line [6047](#) of file [compcomm.c](#).

4.9.2.115 CoExtraSymbols()

```
int CoExtraSymbols (
    UBYTE * inp )
```

Definition at line [6097](#) of file [compcomm.c](#).

4.9.2.116 GetIfDollarFactor()

```
WORD * GetIfDollarFactor (
    UBYTE ** inp,
    WORD * w )
```

Definition at line [6170](#) of file [compcomm.c](#).

4.9.2.117 GetDoParam()

```
UBYTE * GetDoParam (
    UBYTE * inp,
    WORD ** wp,
    int par )
```

Definition at line [6228](#) of file [compcomm.c](#).

4.9.2.118 CoDo()

```
int CoDo (
    UBYTE * inp )
```

Definition at line [6305](#) of file [compcomm.c](#).

4.9.2.119 CoEndDo()

```
int CoEndDo (
    UBYTE * inp )
```

Definition at line [6408](#) of file [compcomm.c](#).

4.9.2.120 CoFactDollar()

```
int CoFactDollar (
    UBYTE * inp )
```

Definition at line [6439](#) of file [compcomm.c](#).

4.9.2.121 CoFactorize()

```
int CoFactorize (
    UBYTE * s )
```

Definition at line [6468](#) of file [compcomm.c](#).

4.9.2.122 CoNFactorize()

```
int CoNFactorize (
    UBYTE * s )
```

Definition at line [6475](#) of file [compcomm.c](#).

4.9.2.123 CoUnFactorize()

```
int CoUnFactorize (
    UBYTE * s )
```

Definition at line [6482](#) of file [compcomm.c](#).

4.9.2.124 CoNUnFactorize()

```
int CoNUnFactorize (
    UBYTE * s )
```

Definition at line [6489](#) of file [compcomm.c](#).

4.9.2.125 DoFactorize()

```
int DoFactorize (
    UBYTE * s,
    int par )
```

Definition at line [6496](#) of file [compcomm.c](#).

4.9.2.126 CoOptimizeOption()

```
int CoOptimizeOption (
    UBYTE * s )
```

Definition at line [6629](#) of file [compcomm.c](#).

4.9.2.127 CoPutInside()

```
int CoPutInside (
    UBYTE * inp )
```

Definition at line 7065 of file [compcomm.c](#).

4.9.2.128 CoAntiPutInside()

```
int CoAntiPutInside (
    UBYTE * inp )
```

Definition at line 7066 of file [compcomm.c](#).

4.9.2.129 DoPutInside()

```
int DoPutInside (
    UBYTE * inp,
    int par )
```

Definition at line 7068 of file [compcomm.c](#).

4.9.2.130 CoSwitch()

```
int CoSwitch (
    UBYTE * s )
```

Definition at line 7188 of file [compcomm.c](#).

4.9.2.131 CoCase()

```
int CoCase (
    UBYTE * s )
```

Definition at line 7234 of file [compcomm.c](#).

4.9.2.132 CoBreak()

```
int CoBreak (
    UBYTE * s )
```

Definition at line 7284 of file [compcomm.c](#).

4.9.2.133 CoDefault()

```
int CoDefault (
    UBYTE * s )
```

Definition at line 7310 of file [compcomm.c](#).

4.9.2.134 CoEndSwitch()

```
int CoEndSwitch (
    UBYTE * s )
```

Definition at line [7337](#) of file [compcomm.c](#).

4.9.2.135 CoSetUserFlag()

```
int CoSetUserFlag (
    UBYTE * s )
```

Definition at line [7405](#) of file [compcomm.c](#).

4.9.2.136 CoClearUserFlag()

```
int CoClearUserFlag (
    UBYTE * s )
```

Definition at line [7433](#) of file [compcomm.c](#).

4.9.2.137 CoCreateAllLoops()

```
int CoCreateAllLoops (
    UBYTE * s )
```

Definition at line [7470](#) of file [compcomm.c](#).

4.9.2.138 CoCreateAllPaths()

```
int CoCreateAllPaths (
    UBYTE * s )
```

Definition at line [7590](#) of file [compcomm.c](#).

4.9.2.139 CoCreateAll()

```
int CoCreateAll (
    UBYTE * s )
```

Definition at line [7718](#) of file [compcomm.c](#).

4.10 compcomm.c

[Go to the documentation of this file.](#)

```

00001
00010 /* #[] License : */
00011 /*
00012 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00013 *   When using this file you are requested to refer to the publication
00014 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00015 *   This is considered a matter of courtesy as the development was paid
00016 *   for by FOM the Dutch physics granting agency and we would like to
00017 *   be able to track its scientific use to convince FOM of its value
00018 *   for the community.
00019 *
00020 *   This file is part of FORM.
00021 *
00022 *   FORM is free software: you can redistribute it and/or modify it under the
00023 *   terms of the GNU General Public License as published by the Free Software
00024 *   Foundation, either version 3 of the License, or (at your option) any later
00025 *   version.
00026 *
00027 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00028 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00029 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00030 *   details.
00031 *
00032 *   You should have received a copy of the GNU General Public License along
00033 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00034 */
00035 /* #[] License : */
00036 /*
00037 *   #[] includes :
00038 */
00039
00040 #include "form3.h"
00041 #include "comtool.h"
00042 #ifdef WITHFLOAT
00043 #include <gmp.h>
00044 #include <math.h>
00045 #endif
00046
00047 static KEYWORD formatoptions[] = {
00048     {"allfloat",      (TFUN)0,    ALLINTEGERDOUBLE, 0}
00049     ,{"c",            (TFUN)0,    CMODE,            0}
00050     ,{"doublefortran", (TFUN)0,    DOUBLEFORTRANMODE, 0}
00051     ,{"float",         (TFUN)0,    0,                2}
00052 #ifdef WITHFLOAT
00053     ,{"floatprecision", (TFUN)0,    0,                5}
00054 #endif
00055     ,{"fortran",       (TFUN)0,    FORTRANMODE,      0}
00056     ,{"fortran90",     (TFUN)0,    FORTRANMODE,      4}
00057     ,{"maple",         (TFUN)0,    MAPLEMODE,        0}
00058     ,{"mathematica",   (TFUN)0,    MATHEMATICAMODE, 0}
00059     ,{"normal",        (TFUN)0,    NORMALFORMAT,    1}
00060     ,{"nospaces",      (TFUN)0,    NOSPACEFORMAT,    3}
00061     ,{"pfortran",      (TFUN)0,    PFORTRANMODE,    0}
00062     ,{"quadfortran",   (TFUN)0,    QUADRUPLEFORTRANMODE, 0}
00063     ,{"quadruplefortran", (TFUN)0,    QUADRUPLEFORTRANMODE, 0}
00064     ,{"rational",      (TFUN)0,    RATIONALMODE,    1}
00065     ,{"reduce",        (TFUN)0,    REDUCEMODE,      0}
00066     ,{"spaces",        (TFUN)0,    NORMALFORMAT,    3}
00067     ,{"vortran",       (TFUN)0,    VORTRANMODE,     0}
00068 };
00069
00070 static KEYWORD trace4options[] = {
00071     {"contract",      (TFUN)0,    CHISHOLM,        0 }
00072     ,{"nocontract",   (TFUN)0,    0,                CHISHOLM }
00073     ,{"nosymmetrize", (TFUN)0,    0,                ALSOREVERSE}
00074     ,{"notrick",      (TFUN)0,    NOTRICK,          0 }
00075     ,{"symmetrize",   (TFUN)0,    ALSOREVERSE,      0 }
00076     ,{"trick",        (TFUN)0,    0,                NOTRICK }
00077 };
00078
00079 static KEYWORD chisoptions[] = {
00080     {"nosymmetrize", (TFUN)0,    0,                ALSOREVERSE}
00081     ,{"symmetrize",  (TFUN)0,    ALSOREVERSE,      0 }
00082 };
00083
00084 static KEYWORDV writeoptions[] = {
00085     {"stats",         &(AC.StatsFlag), 1, 0}
00086     ,{"statistics",   &(AC.StatsFlag), 1, 0}
00087     ,{"shortstats",    &(AC.ShortStats), 1, 0}
00088     ,{"shortstatistics",&(AC.ShortStats), 1, 0}
00089     ,{"warnings",      &(AC.WarnFlag), 1, 0}
00090     ,{"allwarnings",   &(AC.WarnFlag), 2, 0}

```

```

00091     ,{"setup",          &(AC.SetupFlag),    1,      0}
00092     ,{"names",         &(AC.NamesFlag),     1,      0}
00093     ,{"allnames",      &(AC.NamesFlag),     2,      0}
00094     ,{"codes",         &(AC.CodesFlag),     1,      0}
00095     ,{"highfirst",     &(AC.SortType),    SORTHIGHFIRST,  SORTLOWFIRST}
00096     ,{"lowfirst",      &(AC.SortType),    SORTLOWFIRST,  SORTHIGHFIRST}
00097     ,{"powerfirst",    &(AC.SortType),    SORTPOWERFIRST, SORTHIGHFIRST}
00098     ,{"tokens",        &(AC.TokensWriteFlag),1,      0}
00099 };
00100
00101 static KEYWORDV onoffoptions[] = {
00102     {"compress",       &(AC.NoCompress),    0,      1}
00103     ,{"checkpoint",    &(AC.CheckpointFlag), 1,      0}
00104     ,{"insidefirst",   &(AC.insidefirst), 1,      0}
00105     ,{"propercount",   &(AC.BottomLevel), 1,      0}
00106     ,{"stats",         &(AC.StatsFlag),     1,      0}
00107     ,{"statistics",    &(AC.StatsFlag),     1,      0}
00108     ,{"shortstats",    &(AC.ShortStats),    1,      0}
00109     ,{"shortstatistics",&(AC.ShortStats),    1,      0}
00110     ,{"names",         &(AC.NamesFlag),     1,      0}
00111     ,{"allnames",      &(AC.NamesFlag),     2,      0}
00112     ,{"warnings",      &(AC.WarnFlag),    1,      0}
00113     ,{"allwarnings",   &(AC.WarnFlag),    2,      0}
00114     ,{"highfirst",     &(AC.SortType),    SORTHIGHFIRST,  SORTLOWFIRST}
00115     ,{"lowfirst",      &(AC.SortType),    SORTLOWFIRST,  SORTHIGHFIRST}
00116     ,{"powerfirst",    &(AC.SortType),    SORTPOWERFIRST, SORTHIGHFIRST}
00117     ,{"setup",         &(AC.SetupFlag),    1,      0}
00118     ,{"codes",         &(AC.CodesFlag),     1,      0}
00119     ,{"tokens",        &(AC.TokensWriteFlag),1,      0}
00120     ,{"properorder",   &(AC.properorderflag),1,0}
00121     ,{"threadloadbalancing",&(AC.ThreadBalancing),1, 0}
00122     ,{"threads",       &(AC.ThreadsFlag),1, 0}
00123     ,{"threadsortfilesynch",&(AC.ThreadSortFileSynch),1, 0}
00124     ,{"threadstats",   &(AC.ThreadStats),1, 0}
00125     ,{"finalstats",    &(AC.FinalStats),1, 0}
00126     ,{"fewerstats",    &(AC.ShortStatsMax), 10, 0}
00127     ,{"fewerstatistics",&(AC.ShortStatsMax), 10, 0}
00128     ,{"processstats",  &(AC.ProcessStats),1, 0}
00129     ,{"oldparallelstats",&(AC.OldParallelStats),1,0}
00130     ,{"parallel",      &(AC.parallelflag),PARALLELF,NOPARALLEL_USER}
00131     ,{"nospacesinnnumbers",&(AO.NoSpacesInNumbers),1,0}
00132     ,{"indentSPACE",   &(AO.IndentSpace),INDENTSPACE,0}
00133     ,{"totalsize",      &(AM.PrintTotalSize), 1, 0}
00134     ,{"flag",          (int *)&(AC.debugFlags), 1, 0}
00135     ,{"oldfactarg",    &(AC.OldFactArgFlag), 1, 0}
00136     ,{"memdebugflag",  &(AC.MemDebugFlag), 1, 0}
00137     ,{"oldgcd",        &(AC.OldGCDflag), 1, 0}
00138     ,{"innertest",     &(AC.InnerTest), 1, 0}
00139     ,{"wtimestats",    &(AC.WTimeStatsFlag), 1, 0}
00140     ,{"sortreallocate",&(AC.SortReallocateFlag), 1, 0}
00141     ,{"backtrace",     &(AC.PrintBacktraceFlag), 1, 0}
00142     ,{"flint",         &(AC.FlintPolyFlag), 1, 0}
00143     ,{"humanstats",    &(AC.HumanStatsFlag), 1, 0}
00144     ,{"humanstatistics",&(AC.HumanStatsFlag), 1, 0}
00145 };
00146
00147 static WORD one = 1;
00148
00149 /*
00150     #] includes :
00151     #[ CoFormat :
00152 */
00153
00154 int CoFormat(UBYTE *s)
00155 {
00156     int error = 0, x;
00157     KEYWORD *key;
00158     UBYTE *ss;
00159     while ( *s == ' ' || *s == ',' ) s++;
00160     if ( *s == 0 ) {
00161         AC.OutputMode = 72;
00162         AC.OutputSpaces = NORMALFORMAT;
00163         return(error);
00164     }
00165     /*
00166     First the optimization level
00167     */
00168     if ( *s == 'O' || *s == 'o' ) {
00169         if ( ( FG.cTable[s[1]] == 1 ) ||
00170              ( s[1] == '=' && FG.cTable[s[2]] == 1 ) ) {
00171             s++; if ( *s == '=' ) s++;
00172             x = 0;
00173             while ( *s >= '0' && *s <= '9' ) x = 10*x + *s++ - '0';
00174             while ( *s == ',' ) s++;
00175             AO.OptimizationLevel = x;
00176             AO.Optimize.greedytimelimit = 0;
00177             AO.Optimize.mctsttimelimit = 0;

```

```

00178         AO.Optimize.printstats = 0;
00179         AO.Optimize.debugflags = 0;
00180         AO.Optimize.schemeflags = 0;
00181         AO.Optimize.mctsdecaymode = 1; /* default is decreasing C_p with iteration number */
00182         if ( AO.inscheme ) {
00183             M_free(AO.inscheme,"Horner input scheme");
00184             AO.inscheme = 0; AO.schemenum = 0;
00185         }
00186         switch ( x ) {
00187             case 0:
00188                 break;
00189             case 1:
00190                 AO.Optimize.mctsconstant.fval = -1.0;
00191                 AO.Optimize.horner = O_OCCURRENCE;
00192                 AO.Optimize.hornerdirection = O_FORWARDORBACKWARD;
00193                 AO.Optimize.method = O_CSE;
00194                 break;
00195             case 2:
00196                 AO.Optimize.horner = O_OCCURRENCE;
00197                 AO.Optimize.hornerdirection = O_FORWARDORBACKWARD;
00198                 AO.Optimize.method = O_GREEDY;
00199                 AO.Optimize.greedyminnum = 10;
00200                 AO.Optimize.greedymaxperc = 5;
00201                 break;
00202             case 3:
00203                 AO.Optimize.mctsconstant.fval = 1.0;
00204                 AO.Optimize.horner = O_MCTS;
00205                 AO.Optimize.hornerdirection = O_FORWARDORBACKWARD;
00206                 AO.Optimize.method = O_GREEDY;
00207                 AO.Optimize.mctsnumexpand = 1000;
00208                 AO.Optimize.mctsnumkeep = 10;
00209                 AO.Optimize.mctsnumrepeat = 1;
00210                 AO.Optimize.greedyminnum = 10;
00211                 AO.Optimize.greedymaxperc = 5;
00212                 break;
00213             case 4:
00214                 AO.Optimize.horner = O_SIMULATED_ANNEALING;
00215                 AO.Optimize.saIter = 1000;
00216                 AO.Optimize.saMaxT.fval = 2000;
00217                 AO.Optimize.saMinT.fval = 1;
00218                 break;
00219             default:
00220                 error = 1;
00221                 MesPrint("&Illegal optimization specification in format statement");
00222                 break;
00223         }
00224         if ( error == 0 && *s != 0 && x > 0 ) return(CoOptimizeOption(s));
00225         return(error);
00226     }
00227 #ifdef EXPOPT
00228     { UBYTE c;
00229       ss = s;
00230       while ( FG.cTable[*s] == 0 ) s++;
00231       c = *s; *s = 0;
00232       if ( StrICont(ss, (UBYTE *) "optimize") == 0 ) {
00233           *s = c;
00234           while ( *s == ',' ) s++;
00235           if ( *s == '=' ) s++;
00236           AO.OptimizationLevel = 3;
00237           AO.Optimize.mctsconstant.fval = 1.0;
00238           AO.Optimize.horner = O_MCTS;
00239           AO.Optimize.hornerdirection = O_FORWARDORBACKWARD;
00240           AO.Optimize.method = O_GREEDY;
00241           AO.Optimize.mctstimelimit = 0;
00242           AO.Optimize.mctsnumexpand = 1000;
00243           AO.Optimize.mctsnumkeep = 10;
00244           AO.Optimize.mctsnumrepeat = 1;
00245           AO.Optimize.greedytimelimit = 0;
00246           AO.Optimize.greedyminnum = 10;
00247           AO.Optimize.greedymaxperc = 5;
00248           AO.Optimize.printstats = 0;
00249           AO.Optimize.debugflags = 0;
00250           AO.Optimize.schemeflags = 0;
00251           AO.Optimize.mctsdecaymode = 1;
00252           if ( AO.inscheme ) {
00253               M_free(AO.inscheme,"Horner input scheme");
00254               AO.inscheme = 0; AO.schemenum = 0;
00255           }
00256           return(CoOptimizeOption(s));
00257       }
00258       else {
00259           error = 1;
00260           MesPrint("&Illegal optimization specification in format statement");
00261           return(error);
00262       }
00263     }
00264 #endif

```

```

00265     }
00266     else if ( FG.cTable[*s] == 1 ) {
00267         x = 0;
00268         while ( FG.cTable[*s] == 1 ) x = 10*x + *s++ - '0';
00269         if ( x <= 0 || x >= MAXLINELENGTH ) {
00270             error = 1;
00271             MesPrint("&Illegal value for linesize: %d",x);
00272             x = 72;
00273         }
00274         if ( x < 39 ) {
00275             MesPrint(" ... Too small value for linesize corrected to 39");
00276             x = 39;
00277         }
00278         AO.DoubleFlag = 0;
00279     /*
00280         The next line resets the mode to normal. Because the special modes
00281         reset the line length we have a little problem with the special modes
00282         and customized line length. We try to improve by removing the next line
00283     */
00284     /*
00285         AC.OutputMode = 0; */
00286         AC.LineLength = x;
00287         if ( *s != 0 ) {
00288             error = 1;
00289             MesPrint("&Illegal linesize field in format statement");
00290         }
00291     else {
00292         key = FindKeyWord(s,formatoptions,
00293             sizeof(formatoptions)/sizeof(KEYWORD));
00294         if ( key ) {
00295             if ( key->type == FORTRANMODE || key->type == PFORTRANMODE || key->type ==
00296                 DOUBLEFORTRANMODE
00297                 || key->type == QUADRUPLEFORTRANMODE || key->type == VORTRANMODE ) {
00298                 if ( AC.LineLength > 72 ) AC.LineLength = 72;
00299             }
00300             if ( key->flags == 0 ) {
00301                 if ( key->type == FORTRANMODE || key->type == PFORTRANMODE
00302                     || key->type == DOUBLEFORTRANMODE || key->type == ALLINTEGERDOUBLE
00303                     || key->type == QUADRUPLEFORTRANMODE || key->type == VORTRANMODE ) {
00304                     AC.IsFortran90 = ISNOTFORTRAN90;
00305                     if ( AC.Fortran90Kind ) {
00306                         M_free(AC.Fortran90Kind,"Fortran90 Kind");
00307                         AC.Fortran90Kind = 0;
00308                     }
00309                     if ( ( key->type == ALLINTEGERDOUBLE ) && AO.DoubleFlag != 0 ) {
00310                         AO.DoubleFlag |= 4;
00311                     }
00312                     else {
00313                         AO.DoubleFlag = 0;
00314                         AC.OutputMode = key->type & NODOUBLEMASK;
00315                         if ( ( key->type & DOUBLEPRECISIONFLAG ) != 0 ) {
00316                             AO.DoubleFlag = 1;
00317                         }
00318                         else if ( ( key->type & QUADRUPLEPRECISIONFLAG ) != 0 ) {
00319                             AO.DoubleFlag = 2;
00320                         }
00321                     }
00322                 }
00323             }
00324             else if ( key->flags == 1 ) {
00325                 AC.OutputMode = AC.OutNumberType = key->type;
00326             }
00327             else if ( key->flags == 2 ) {
00328                 while ( FG.cTable[*s] == 0 ) s++;
00329                 if ( *s == 0 ) AC.OutNumberType = 10;
00330                 else if ( *s == ',' ) {
00331                     s++;
00332                     x = 0;
00333                     while ( FG.cTable[*s] == 1 ) x = 10*x + *s++ - '0';
00334                     if ( *s != 0 ) {
00335                         error = 1;
00336                         MesPrint("&Illegal float format specifier");
00337                     }
00338                     else {
00339                         if ( x < 3 ) {
00340                             x = 3;
00341                             MesPrint("& ... float format value corrected to 3");
00342                         }
00343                         if ( x > 100 ) {
00344                             x = 100;
00345                             MesPrint("& ... float format value corrected to 100");
00346                         }
00347                         AC.OutNumberType = x;
00348                     }
00349                 }
00350             }
00351             else if ( key->flags == 3 ) {

```



```

00351         AC.OutputSpaces = key->type;
00352     }
00353     else if ( key->flags == 4 ) {
00354         AC.IsFortran90 = ISFORTRAN90;
00355         if ( AC.Fortran90Kind ) {
00356             M_free(AC.Fortran90Kind,"Fortran90 Kind");
00357             AC.Fortran90Kind = 0;
00358         }
00359         while ( FG.cTable[*s] <= 1 ) s++;
00360         if ( *s == ',' ) {
00361             s++; ss = s;
00362             while ( *ss && *ss != ',' ) ss++;
00363             if ( *ss == ',' ) {
00364                 MesPrint("&No white space or comma's allowed in Fortran90 option: %s",s);
00365                 error = 1;
00366             }
00367             else {
00368                 AC.Fortran90Kind = strDup1(s,"Fortran90 Kind");
00369             }
00370             AO.DoubleFlag = 0;
00371             AC.OutputMode = key->type & NODOUBLEMASK;
00372         }
00373 #ifdef WITHFLOAT
00374         else if ( key->flags == 5 ) {
00375             /*
00376              Syntax: Format FloatPrecision [precision];
00377              Format FloatPrecision off;
00378             */
00379             while ( FG.cTable[*s] == 0 ) s++;
00380             while ( *s == ' ' || *s == '\t' || *s == ',' ) s++;
00381             if ( *s == 0 ) {
00382                 AO.FloatPrec = 0;
00383             }
00384             else if ( tolower(*s) == 'o' && tolower(s[1]) == 'f'
00385                 && tolower(s[2]) == 'f' ) {
00386                 ss = s;
00387                 s += 3;
00388                 while ( *s == ' ' || *s == '\t' || *s == ',' ) s++;
00389                 if ( *s ) { s = ss; goto WrongOption; }
00390                 AO.FloatPrec = -1;
00391             }
00392             else if ( FG.cTable[*s] == 1 ) {
00393                 ss = s;
00394                 ParseNumber(AO.FloatPrec,s)
00395             /*
00396              The precision can either be in digits or bits.
00397              AO.FloatPrec is in digits.
00398             */
00399             if ( tolower(*s) == 'd' ) { s++; }
00400             else if ( tolower(*s) == 'b' ) { AO.FloatPrec = AO.FloatPrec*log10(2.0); s++; }
00401             else { s = ss; goto WrongOption; }
00402             while ( *s == ' ' || *s == '\t' || *s == ',' ) s++;
00403             if ( *s ) { s = ss; goto WrongOption; }
00404         }
00405         else {
00406 WrongOption: MesPrint("&Illegal option in Format FloatPrecision: %s",s);
00407                 error = 1;
00408             }
00409         }
00410 #endif
00411     }
00412     else if ( ( *s == 'c' || *s == 'C' ) && ( FG.cTable[s[1]] == 1 ) ) {
00413         UBYTE *ss = s+1;
00414         WORD x = 0;
00415         while ( *ss >= '0' && *ss <= '9' ) x = 10*x + *ss++ - '0';
00416         if ( *ss != 0 ) goto Unknown;
00417         AC.OutputMode = CMODE;
00418         AC.Cnumpows = x;
00419     }
00420     else {
00421 Unknown: MesPrint("&Unknown option: %s",s); error = 1;
00422     }
00423 }
00424 return(error);
00425 }
00426
00427 /*
00428 #] CoFormat :
00429 #[ CoCollect :
00430
00431 Collect,functionname
00432 */
00433
00434 int CoCollect (UBYTE *s)
00435 {
00436 /* -----change 17-feb-2003 Added percentage */

```

```

00437     WORD numfun;
00438     int type,x = 0;
00439     UBYTE *t = SkipAName(s), *t1, *t2;
00440     AC.AltCollectFun = 0;
00441     if ( t == 0 ) goto syntaxerror;
00442     t1 = t; while ( *t1 == ',' || *t1 == ' ' || *t1 == '\t' ) t1++;
00443     *t = 0; t = t1;
00444     if ( *t1 && ( FG.cTable[*t1] == 0 || *t1 == '[' ) ) {
00445         t2 = SkipAName(t1);
00446         if ( t2 == 0 ) goto syntaxerror;
00447         t = t2;
00448         while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
00449         *t2 = 0;
00450     }
00451     else t1 = 0;
00452     if ( *t && FG.cTable[*t] == 1 ) {
00453         while ( *t >= '0' && *t <= '9' ) x = 10*x + *t++ - '0';
00454         if ( x > 100 ) x = 100;
00455         while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
00456         if ( *t ) goto syntaxerror;
00457     }
00458     else {
00459         if ( *t ) goto syntaxerror;
00460         x = 100;
00461     }
00462     if ( ( ( type = GetName(AC.varnames,s,&numfun,WITHAUTO) ) != CFUNCTION )
00463     || ( functions[numfun].spec != 0 ) ) {
00464         MesPrint("%s should be a regular function",s);
00465         if ( type < 0 ) {
00466             if ( GetName(AC.exprnames,s,&numfun,NOAUTO) == NAMENOTFOUND )
00467                 AddFunction(s,0,0,0,0,0,-1,-1);
00468         }
00469         return(1);
00470     }
00471     AC.CollectFun = numfun+FUNCTION;
00472     AC.CollectPercentage = (WORD)x;
00473     if ( t1 ) {
00474         if ( ( ( type = GetName(AC.varnames,t1,&numfun,WITHAUTO) ) != CFUNCTION )
00475         || ( functions[numfun].spec != 0 ) ) {
00476             MesPrint("%s should be a regular function",t1);
00477             if ( type < 0 ) {
00478                 if ( GetName(AC.exprnames,t1,&numfun,NOAUTO) == NAMENOTFOUND )
00479                     AddFunction(t1,0,0,0,0,0,-1,-1);
00480             }
00481             return(1);
00482         }
00483         AC.AltCollectFun = numfun+FUNCTION;
00484     }
00485     return(0);
00486 syntaxerror:
00487     MesPrint("&Collect statement needs one or two functions (and a percentage) for its argument(s)");
00488     return(1);
00489 }
00490
00491 /*
00492  #] CoCollect :
00493  #[ setonoff :
00494 */
00495
00496 int setonoff(UBYTE *s, int *flag, int onvalue, int offvalue)
00497 {
00498     if ( StrICmp(s,(UBYTE *)"on") == 0 ) *flag = onvalue;
00499     else if ( StrICmp(s,(UBYTE *)"off") == 0 ) *flag = offvalue;
00500     else {
00501         MesPrint("&Unknown option: %s, on or off expected",s);
00502         return(1);
00503     }
00504     return(0);
00505 }
00506
00507 /*
00508  #] setonoff :
00509  #[ CoCompress :
00510 */
00511
00512 int CoCompress(UBYTE *s)
00513 {
00514     GETIDENTITY
00515     UBYTE *t, c;
00516     if ( StrICmp(s,(UBYTE *)"on") == 0 ) {
00517         AC.NoCompress = 0;
00518         AR.gzipCompress = 0;
00519     }
00520     else if ( StrICmp(s,(UBYTE *)"off") == 0 ) {
00521         AC.NoCompress = 1;
00522         AR.gzipCompress = 0;
00523     }

```

```

00524     else {
00525         t = s; while ( FG.cTable[*t] <= 1 ) t++;
00526         c = *t; *t = 0;
00527         if ( StrICmp(s, (UBYTE *) "gzip") == 0 ) {
00528             #ifndef WITHZLIB
00529                 Warning("gzip compression not supported on this platform");
00530             #endif
00531             s = t; *s = c;
00532             if ( *s == 0 ) {
00533                 AR.gzipCompress = GZIPDEFAULT; /* Normally should be 6 */
00534                 return(0);
00535             }
00536             while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00537             t = s;
00538             if ( FG.cTable[*s] == 1 ) {
00539                 AR.gzipCompress = *s - '0';
00540                 s++;
00541                 while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00542                 if ( *s == 0 ) return(0);
00543             }
00544             MesPrint("&Unknown gzip option: %s, a digit was expected",t);
00545             return(1);
00546         }
00547     }
00548     else {
00549         MesPrint("&Unknown option: %s, on, off or gzip expected",s);
00550         return(1);
00551     }
00552 }
00553 return(0);
00554 }
00555
00556 /*
00557  #] CoCompress :
00558  #[ CoFlags :
00559  */
00560
00561 int CoFlags(UBYTE *s,int value)
00562 {
00563     int i, error = 0;
00564     if ( *s != ',' ) {
00565         MesPrint("&Proper syntax is: On/Off Flag,number[s];");
00566         error = 1;
00567     }
00568     while ( *s == ',' ) {
00569         do { s++; } while ( *s == ',' );
00570         i = 0;
00571         if ( FG.cTable[*s] != 1 ) {
00572             MesPrint("&Proper syntax is: On/Off Flag,number[s];");
00573             error = 1;
00574             break;
00575         }
00576         while ( FG.cTable[*s] == 1 ) { i = 10*i + *s++ - '0'; }
00577         if ( i <= 0 || i > MAXFLAGS ) {
00578             MesPrint("&The number of a flag in On/Off Flag should be in the range
00579 0-%d", (int)MAXFLAGS);
00580             error = 1;
00581             break;
00582         }
00583         AC.debugFlags[i] = value;
00584     }
00585     if ( *s ) {
00586         MesPrint("&Proper syntax is: On/Off Flag,number[s];");
00587         error = 1;
00588     }
00589     return(error);
00590 }
00591
00592 /*
00593  #] CoFlags :
00594  #[ CoOff :
00595  */
00596
00597 int CoOff(UBYTE *s)
00598 {
00599     GETIDENTITY
00600     UBYTE *t, c;
00601     int i, num = sizeof(onoffoptions)/sizeof(KEYWORD);
00602     for (;;) {
00603         while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00604         if ( *s == 0 ) return(0);
00605         if ( chartype[*s] != 0 ) {
00606             MesPrint("&Illegal character or option encountered in OFF statement");
00607             return(-1);
00608         }
00609         t = s; while ( chartype[*s] == 0 ) s++;
00610         c = *s; *s = 0;

```

```

00610         for ( i = 0; i < num; i++ ) {
00611             if ( StrICont(t, (UBYTE *) (onoffoptions[i].name)) == 0 ) break;
00612         }
00613         if ( i >= num ) {
00614             MesPrint("&Unrecognized option in OFF statement: %s", t);
00615             *s = c; return(-1);
00616         }
00617         else if ( StrICont(t, (UBYTE *) "compress") == 0 ) {
00618             AR.zipCompress = 0;
00619         }
00620         else if ( StrICont(t, (UBYTE *) "checkpoint") == 0 ) {
00621             PrintDeprecation("the checkpoint mechanism", "issues/626");
00622             AC.CheckpointInterval = 0;
00623             if ( AC.CheckpointRunBefore ) { free(AC.CheckpointRunBefore); AC.CheckpointRunBefore =
NULL; }
00624             if ( AC.CheckpointRunAfter ) { free(AC.CheckpointRunAfter); AC.CheckpointRunAfter = NULL;
}
00625             if ( AC.NoShowInput == 0 ) MesPrint("Checkpoints deactivated.");
00626         }
00627         else if ( StrICont(t, (UBYTE *) "threads") == 0 ) {
00628             AS.MultiThreaded = 0;
00629         }
00630         else if ( StrICont(t, (UBYTE *) "flag") == 0 ) {
00631             *s = c;
00632             return(CoFlags(s, 0));
00633         }
00634         else if ( StrICont(t, (UBYTE *) "innertest") == 0 ) {
00635             *s = c;
00636             AC.InnerTest = 0;
00637             if ( AC.TestValue ) {
00638                 M_free(AC.TestValue, "InnerTest");
00639                 AC.TestValue = 0;
00640             }
00641         }
00642         else if ( StrICont(t, (UBYTE *) "sortreallocate") == 0 ) {
00643             if ( AC.SortReallocateFlag == 2 ) {
00644                 /* The flag has been set by #sortreallocate, and it was off before. Leave it as 2,
00645                    so that the reallocation still happens in the current module. It will be turned
00646                    off after the reallocation is done. */
00647                 return(0);
00648             }
00649         }
00650         *s = c;
00651         *onoffoptions[i].var = onoffoptions[i].flags;
00652         AR.SortType = AC.SortType;
00653         AC.mparallelfalg = AC.parallelfalg | AM.hparallelfalg;
00654     }
00655 }
00656
00657 /*
00658 #] CoOff :
00659 #[ CoOn :
00660 */
00661
00662 int CoOn(UBYTE *s)
00663 {
00664     GETIDENTITY
00665     UBYTE *t, c;
00666     int i, num = sizeof(onoffoptions)/sizeof(KEYWORD);
00667     LONG interval;
00668     for (;;) {
00669         while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00670         if ( *s == 0 ) return(0);
00671         if ( chartype[*s] != 0 ) {
00672             MesPrint("&Illegal character or option encountered in ON statement");
00673             return(-1);
00674         }
00675         t = s; while ( chartype[*s] == 0 ) s++;
00676         c = *s; *s = 0;
00677         for ( i = 0; i < num; i++ ) {
00678             if ( StrICont(t, (UBYTE *) (onoffoptions[i].name)) == 0 ) break;
00679         }
00680         if ( i >= num ) {
00681             MesPrint("&Unrecognized option in ON statement: %s", t);
00682             *s = c; return(-1);
00683         }
00684         if ( StrICont(t, (UBYTE *) "backtrace") == 0 ) {
00685             #ifndef ENABLE_BACKTRACE
00686                 Warning("backtrace not supported on this platform");
00687             #endif
00688         }
00689         else if ( StrICont(t, (UBYTE *) "compress") == 0 ) {
00690             AR.zipCompress = 0;
00691             *s = c;
00692             while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00693             if ( *s ) {
00694                 t = s;

```

```

00695         while ( FG.cTable[*s] <= 1 ) s++;
00696         c = *s; *s = 0;
00697         if ( StrICmp(t, (UBYTE *)"gzip") == 0 ) {
00698             #ifndef WITHZLIB
00699                 Warning("gzip compression not supported on this platform");
00700             #endif
00701             #ifdef WITHZSTD
00702                 /* If gzip is specified, turn off zstd compression. zlib still goes via the
00703                    wrapper. */
00704                 ZWRAP_useZSTDcompression(0);
00705             #endif
00706         }
00707         else if ( StrICmp(t, (UBYTE *)"zstd") == 0 ) {
00708             #ifdef WITHZSTD
00709                 ZWRAP_useZSTDcompression(1);
00710             #else
00711                 Warning("zstd compression not supported on this platform");
00712             #endif
00713         }
00714         else {
00715             MesPrint("&Unrecognized option in ON compress statement: %s",t);
00716             return(-1);
00717         }
00718         /* Whether we are using zlib or zstd, accept and use a compression level. */
00719         *s = c;
00720         while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00721         if ( FG.cTable[*s] == 1 ) {
00722             AR.gzipCompress = *s++ - '0';
00723             while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00724             if ( *s ) {
00725                 MesPrint("&Unrecognized option in ON compress gzip/zstd statement: %s",t);
00726                 return(-1);
00727             }
00728             else if ( *s == 0 ) {
00729                 AR.gzipCompress = GZIPDEFAULT;
00730             }
00731             else {
00732                 MesPrint("&Unrecognized option in ON compress gzip/zstd statement: %s, single
00733 digit expected",t);
00734                 return(-1);
00735             }
00736         }
00737         else if ( StrICont(t, (UBYTE *)"checkpoint") == 0 ) {
00738             PrintDeprecation("the checkpoint mechanism", "issues/626");
00739             AC.CheckpointInterval = 0;
00740             if ( AC.CheckpointRunBefore ) { free(AC.CheckpointRunBefore); AC.CheckpointRunBefore =
00741 NULL; }
00742             if ( AC.CheckpointRunAfter ) { free(AC.CheckpointRunAfter); AC.CheckpointRunAfter = NULL;
00743 }
00744             *s = c;
00745             while ( *s ) {
00746                 while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00747                 if ( FG.cTable[*s] == 1 ) {
00748                     interval = 0;
00749                     t = s;
00750                     do { interval = 10*interval + *s++ - '0'; } while ( FG.cTable[*s] == 1 );
00751                     if ( *s == 's' || *s == 'S' ) {
00752                         s++;
00753                     }
00754                     else if ( *s == 'm' || *s == 'M' ) {
00755                         interval *= 60; s++;
00756                     }
00757                     else if ( *s == 'h' || *s == 'H' ) {
00758                         interval *= 3600; s++;
00759                     }
00760                     else if ( *s == 'd' || *s == 'D' ) {
00761                         interval *= 86400; s++;
00762                     }
00763                     if ( *s != ',' && FG.cTable[*s] != 6 && FG.cTable[*s] != 10 ) {
00764                         MesPrint("&Unrecognized time interval in ON Checkpoint statement: %s", t);
00765                         return(-1);
00766                     }
00767                     AC.CheckpointInterval = interval * 100; /* in 1/100 of seconds */
00768                 }
00769             }
00770             else if ( FG.cTable[*s] == 0 ) {
00771                 int type;
00772                 t = s;
00773                 while ( FG.cTable[*s] == 0 ) s++;
00774                 c = *s; *s = 0;
00775                 if ( StrICmp(t, (UBYTE *)"run") == 0 ) {
00776                     type = 3;
00777                 }
00778                 else if ( StrICmp(t, (UBYTE *)"runafter") == 0 ) {
00779                     type = 2;
00780                 }
00781             }

```

```

00778         else if ( StrICmp(t, (UBYTE *)"runbefore") == 0 ) {
00779             type = 1;
00780         }
00781         else {
00782             MesPrint("&Unrecognized option in ON Checkpoint statement: %s", t);
00783             *s = c; return(-1);
00784         }
00785         *s = c;
00786         if ( *s != '=' && FG.cTable[*s] != 9 ) {
00787             MesPrint("&Unrecognized option in ON Checkpoint statement: %s", t);
00788             return(-1);
00789         }
00790         ++s;
00791         t = ++s;
00792         while ( *s ) {
00793             if ( FG.cTable[*s] == 9 ) {
00794                 c = *s; *s = 0;
00795                 if ( type & 1 ) {
00796                     if ( AC.CheckpointRunBefore ) {
00797                         free(AC.CheckpointRunBefore); AC.CheckpointRunBefore = NULL;
00798                     }
00799                     if ( s-t > 0 ) {
00800                         AC.CheckpointRunBefore = Malloc1(s-t+1, "AC.CheckpointRunBefore");
00801                         StrCopy(t, (UBYTE*)AC.CheckpointRunBefore);
00802                     }
00803                 }
00804                 if ( type & 2 ) {
00805                     if ( AC.CheckpointRunAfter ) {
00806                         free(AC.CheckpointRunAfter); AC.CheckpointRunAfter = NULL;
00807                     }
00808                     if ( s-t > 0 ) {
00809                         AC.CheckpointRunAfter = Malloc1(s-t+1, "AC.CheckpointRunAfter");
00810                         StrCopy(t, (UBYTE*)AC.CheckpointRunAfter);
00811                     }
00812                 }
00813                 *s = c;
00814                 break;
00815             }
00816             ++s;
00817         }
00818         if ( FG.cTable[*s] != 9 ) {
00819             MesPrint("&Unrecognized option in ON Checkpoint statement: %s", t);
00820             return(-1);
00821         }
00822         ++s;
00823     }
00824 }
00825 /*
00826 if ( AC.NoShowInput == 0 ) {
00827     MesPrint("Checkpoints activated.");
00828     if ( AC.CheckpointInterval ) {
00829         MesPrint("-> Minimum saving interval: %1 seconds.", AC.CheckpointInterval/100);
00830     }
00831     else {
00832         MesPrint("-> No minimum saving interval given. Saving after EVERY module.");
00833     }
00834     if ( AC.CheckpointRunBefore ) {
00835         MesPrint("-> Calling script \"%s\" before saving.", AC.CheckpointRunBefore);
00836     }
00837     if ( AC.CheckpointRunAfter ) {
00838         MesPrint("-> Calling script \"%s\" after saving.", AC.CheckpointRunAfter);
00839     }
00840 }
00841 */
00842 }
00843 else if ( StrICont(t, (UBYTE *)"indentSpace") == 0 ) {
00844     *s = c;
00845     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00846     if ( *s ) {
00847         i = 0;
00848         while ( FG.cTable[*s] == 1 ) { i = 10*i + *s++ - '0'; }
00849         if ( *s ) {
00850             MesPrint("&Unrecognized option in ON IndentSpace statement: %s", t);
00851             return(-1);
00852         }
00853         if ( i > 40 ) {
00854             Warning("IndentSpace parameter adjusted to 40");
00855             i = 40;
00856         }
00857         AO.IndentSpace = i;
00858     }
00859     else {
00860         AO.IndentSpace = AM.ggIndentSpace;
00861     }
00862     return(0);
00863 }
00864 else if ( ( StrICont(t, (UBYTE *)"fewerstats") == 0 ) ||

```

```

00865         ( StrICont(t, (UBYTE *)"fewerstatistics") == 0 ) ) {
00866     *s = c;
00867     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00868     if ( *s ) {
00869         i = 0;
00870         while ( FG.cTable[*s] == 1 ) { i = 10*i + *s++ - '0'; }
00871         if ( *s ) {
00872             MesPrint("&Unrecognized option in ON FewerStatistics statement: %s",t);
00873             return(-1);
00874         }
00875         if ( i > AM.S0->MaxPatches ) {
00876             if ( AC.WarnFlag )
00877                 MesPrint("&Warning: FewerStatistics parameter greater than MaxPatches(=%d).
Adjusted to %d"
00878                     ,AM.S0->MaxPatches, (AM.S0->MaxPatches+1)/2);
00879             i = (AM.S0->MaxPatches+1)/2;
00880         }
00881         AC.ShortStatsMax = i;
00882     }
00883     else {
00884         AC.ShortStatsMax = 10; /* default value */
00885     }
00886     return(0);
00887 }
00888 else if ( StrICont(t, (UBYTE *)"threads") == 0 ) {
00889     if ( AM.totalnumberofthreads > 1 ) AS.MultiThreaded = 1;
00890 }
00891 else if ( StrICont(t, (UBYTE *)"flag") == 0 ) {
00892     *s = c;
00893     return(CoFlags(s,1));
00894 }
00895 else if ( StrICont(t, (UBYTE *)"innertest") == 0 ) {
00896     UBYTE *t;
00897     *s = c;
00898     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00899     if ( *s ) {
00900         t = s; while ( *t ) t++;
00901         while ( t[-1] == ' ' || t[-1] == '\t' ) t--;
00902         c = *t; *t = 0;
00903         if ( AC.TestValue ) M_free(AC.TestValue, "InnerTest");
00904         AC.TestValue = strDupl(s, "InnerTest");
00905         *t = c;
00906         s = t;
00907         while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
00908     }
00909     else {
00910         if ( AC.TestValue ) {
00911             M_free(AC.TestValue, "InnerTest");
00912             AC.TestValue = 0;
00913         }
00914     }
00915 }
00916 else if ( StrICont(t, (UBYTE *)"flint") == 0 ) {
00917     #ifndef WITHFLINT
00918         MesPrint("&Warning: FORM was not built with FLINT support.");
00919         MesPrint("Statement has no effect.");
00920     #endif
00921 }
00922 else { *s = c; }
00923 *onoffoptions[i].var = onoffoptions[i].type;
00924 AR.SortType = AC.SortType;
00925 AC.mparallelfalg = AC.parallelfalg | AM.hparallelfalg;
00926 }
00927 }
00928
00929 /*
00930  #] CoOn :
00931  #[ CoInsideFirst :
00932  */
00933
00934 int CoInsideFirst(UBYTE *s) { return(setonoff(s,&AC.insidefirst,1,0)); }
00935
00936 /*
00937  #] CoInsideFirst :
00938  #[ CoProperCount :
00939  */
00940
00941 int CoProperCount(UBYTE *s) { return(setonoff(s,&AC.BottomLevel,1,0)); }
00942
00943 /*
00944  #] CoProperCount :
00945  #[ CoDelete :
00946  */
00947
00948 int CoDelete(UBYTE *s)
00949 {
00950     int error = 0;

```

```

00951     if ( StrICmp(s, (UBYTE *)"storage") == 0 ) {
00952         if ( DeleteStore(1) < 0 ) {
00953             MesPrint("&Cannot restart storage file");
00954             error = 1;
00955         }
00956     }
00957     else {
00958         UBYTE *t = s, c;
00959         while ( *t && *t != ',' && *t != '>' ) t++;
00960         c = *t; *t = 0;
00961         if ( ( StrICmp(s, (UBYTE *)"extrasymbols") == 0 )
00962             || ( StrICmp(s, (UBYTE *)"extrasymbol") == 0 ) ) {
00963             WORD x = 0;
00964             /*
00965              * Either deletes all extra symbols or deletes above a given number
00966              */
00967             *t = c; s = t;
00968             if ( *s == '>' ) {
00969                 s++;
00970                 if ( FG.cTable[*s] != 1 ) goto unknown;
00971                 while ( *s <= '9' && *s >= '0' ) x = 10*x + *s++ - '0';
00972                 if ( *s ) goto unknown;
00973             }
00974             else if ( *s ) goto unknown;
00975             if ( x < AM.gnumextrasyms ) x = AM.gnumextrasyms;
00976             PruneExtraSymbols(x);
00977         }
00978         else {
00979             *t = c;
00980             unknown:
00981             MesPrint("&Unknown option: %s", s);
00982             error = 1;
00983         }
00984     }
00985     return(error);
00986 }
00987
00988 /*
00989 #] CoDelete :
00990 #[ CoKeep :
00991 */
00992
00993 int CoKeep(UBYTE *s)
00994 {
00995     if ( StrICmp(s, (UBYTE *)"brackets") == 0 ) AC.ComDefer = 1;
00996     else { MesPrint("&Unknown option: '%s'", s); return(1); }
00997     return(0);
00998 }
00999
01000 /*
01001 #] CoKeep :
01002 #[ CoFixIndex :
01003 */
01004
01005 int CoFixIndex(UBYTE *s)
01006 {
01007     int x, y, error = 0;
01008     while ( *s ) {
01009         if ( FG.cTable[*s] != 1 ) {
01010             proper: MesPrint("&Proper syntax is: FixIndex,number:value[,number,value];");
01011             return(1);
01012         }
01013         ParseNumber(x, s)
01014         if ( *s != ':' ) goto proper;
01015         s++;
01016         if ( *s != '-' && *s != '+' && FG.cTable[*s] != 1 ) goto proper;
01017         ParseSignedNumber(y, s)
01018         if ( *s && *s != ',' ) goto proper;
01019         while ( *s == ',' ) s++;
01020         if ( x >= AM.OffsetIndex ) {
01021             MesPrint("&Fixed index out of allowed range. Change ConstIndex in setup file?");
01022             MesPrint("&Current value of ConstIndex = %d", AM.OffsetIndex-1);
01023             error = 1;
01024         }
01025         if ( y != (int)((WORD)y) ) {
01026             MesPrint("&Value of d_(%d,%d) outside range for this computer", x, x);
01027             error = 1;
01028         }
01029         if ( error == 0 ) AC.FixIndices[x] = y;
01030     }
01031     return(error);
01032 }
01033
01034 /*
01035 #] CoFixIndex :
01036 #[ CoMetric :
01037 */

```



```

01038
01039 int CoMetric(UBYTE *s)
01040 { DUMMYUSE(s); MesPrint("&The metric statement does not do anything yet"); return(1); }
01041
01042 /*
01043 #] CoMetric :
01044 #[ DoPrint :
01045 */
01046
01047 int DoPrint(UBYTE *s, int par)
01048 {
01049     int i, error = 0, numdol = 0, type;
01050     WORD handle = -1;
01051     UBYTE *name, c, *t;
01052     EXPRESSIONS e;
01053     WORD numexpr, tofile = 0, *w, par2 = 0;
01054     CBUF *C = cbuf + AC.cbufnum;
01055     while ( *s == ',' ) s++;
01056     if ( ( *s == '+' || *s == '-' ) && ( s[1] == 'f' || s[1] == 'F' ) ) {
01057         t = s + 2; while ( *t == ' ' || *t == ',' ) t++;
01058         if ( *t == '"' ) {
01059             if ( *s == '+' ) { tofile = 1; handle = AC.LogHandle; }
01060             s = t;
01061         }
01062     }
01063     else if ( *s == '<' ) {
01064         UBYTE *filename;
01065         s++; filename = s;
01066         while ( *s && *s != '>' ) s++;
01067         if ( *s == 0 ) {
01068             MesPrint("&Improper filename in print statement");
01069             return(1);
01070         }
01071         *s++ = 0;
01072         tofile = 1;
01073         if ( ( handle = GetChannel((char *)filename,1) ) < 0 ) return(1);
01074         SKIPBLANKS(s) if ( *s == ',' ) s++; SKIPBLANKS(s)
01075         if ( *s == '+' && ( s[1] == 's' || s[1] == 'S' ) ) {
01076             s += 2;
01077             par2 |= PRINTONETERM;
01078             if ( *s == 's' || *s == 'S' ) {
01079                 s++;
01080                 par2 |= PRINTONEFUNCTION;
01081                 if ( *s == 's' || *s == 'S' ) {
01082                     s++;
01083                     par2 |= PRINTALL;
01084                 }
01085             }
01086             SKIPBLANKS(s) if ( *s == ',' ) s++; SKIPBLANKS(s)
01087         }
01088     }
01089     if ( par == PRINTON && *s == '"' ) {
01090         WORD code[3];
01091         if ( tofile == 1 ) code[0] = TYPEFPRINT;
01092         else code[0] = TYPEPRINT;
01093         code[1] = handle;
01094         code[2] = par2;
01095         s++; name = s;
01096         while ( *s && *s != '"' ) {
01097             if ( *s == '\\' ) s++;
01098             if ( *s == '%' && s[1] == '$' ) numdol++;
01099             s++;
01100         }
01101         if ( *s != '"' ) {
01102             MesPrint("&String in print statement should be enclosed in \"");
01103             return(1);
01104         }
01105         *s = 0;
01106         AddComString(3,code,name,1);
01107         *s++ = '"';
01108         while ( *s == ',' ) {
01109             s++;
01110             if ( *s == '$' ) {
01111                 s++; name = s; while ( FG.cTable[*s] <= 1 ) s++;
01112                 c = *s; *s = 0;
01113                 type = GetName(AC.dollarnames,name,&numexpr,NOAUTO);
01114                 if ( type == NAMENOTFOUND ) {
01115                     MesPrint("&$ variable %s not (yet) defined",name);
01116                     error = 1;
01117                 }
01118                 else {
01119                     C->lhs[C->numlhs][1] += 2;
01120                     *(C->Pointer)++ = DOLLAREXPRESSION;
01121                     *(C->Pointer)++ = numexpr;
01122                     numdol--;
01123                 }
01124             }

```

```

01125         else {
01126             MesPrint("&Illegal object in print statement");
01127             error = 1;
01128             return(error);
01129         }
01130         *s = c;
01131         if ( c == '[' ) {
01132             w = C->Pointer;
01133             s++;
01134             s = GetDoParam(s,&(C->Pointer),-1);
01135             if ( s == 0 ) return(1);
01136             if ( *s != ']' ) {
01137                 MesPrint("&unmatched [] in $ factor");
01138                 return(1);
01139             }
01140             C->lhs[C->numlhs][1] += C->Pointer - w;
01141             s++;
01142         }
01143     }
01144     if ( *s != 0 ) {
01145         MesPrint("&Illegal object in print statement");
01146         error = 1;
01147     }
01148     if ( numdol > 0 ) {
01149         MesPrint("&More $ variables asked for than provided");
01150         error = 1;
01151     }
01152     *(C->Pointer)++ = 0;
01153     return(error);
01154 }
01155 if ( *s == 0 ) { /* All active expressions */
01156 AllExpr:
01157     for ( e = Expressions, i = NumExpressions; i > 0; i--, e++ ) {
01158         if ( e->status == LOCALEXPRESSION || e->status ==
01159             GLOBALEXPRESSION || e->status == UNHIDELEXPRESSION
01160             || e->status == UNHIDEEXPRESSION ) e->printflag = par;
01161     }
01162     return(error);
01163 }
01164 while ( *s ) {
01165     if ( *s == '+' ) {
01166         s++;
01167         if ( tolower(*s) == 'f' ) par |= PRINTLFILE;
01168         else if ( tolower(*s) == 's' ) {
01169             if ( tolower(s[1]) == 's' ) {
01170                 if ( tolower(s[2]) == 's' ) {
01171                     par |= PRINTONEFUNCTION | PRINTONETERM | PRINTALL;
01172                     s++;
01173                 }
01174                 else if ( ( par & 3 ) < 2 ) par |= PRINTONEFUNCTION | PRINTONETERM;
01175                 s++;
01176             }
01177             else {
01178                 if ( ( par & 3 ) < 2 ) par |= PRINTONETERM;
01179             }
01180         }
01181         else {
01182 illeg:         MesPrint("&Illegal option in (n)print statement");
01183                 error = 1;
01184             }
01185             s++;
01186             if ( *s == 0 ) goto AllExpr;
01187         }
01188         else if ( *s == '-' ) {
01189             s++;
01190             if ( tolower(*s) == 'f' ) par &= ~PRINTLFILE;
01191             else if ( tolower(*s) == 's' ) {
01192                 if ( tolower(s[1]) == 's' ) {
01193                     if ( tolower(s[2]) == 's' ) {
01194                         par &= ~PRINTALL;
01195                         s++;
01196                     }
01197                     else if ( ( par & 3 ) < 2 ) {
01198                         par &= ~PRINTONEFUNCTION;
01199                         par &= ~PRINTALL;
01200                     }
01201                     s++;
01202                 }
01203             }
01204             else {
01205                 if ( ( par & 3 ) < 2 ) {
01206                     par &= ~PRINTONETERM;
01207                     par &= ~PRINTONEFUNCTION;
01208                     par &= ~PRINTALL;
01209                 }
01210             }
01211             else goto illeg;

```

```

01212         s++;
01213         if ( *s == 0 ) goto AllExpr;
01214     }
01215     else if ( FG.cTable[*s] == 0 || *s == '[' ) {
01216         name = s;
01217         if ( ( s = SkipAName(s) ) == 0 ) {
01218             MesPrint("&Improper name in (n)print statement");
01219             return(1);
01220         }
01221         c = *s; *s = 0;
01222         if ( ( GetName(AC.exprnames,name,&numexpr,NOAUTO) == CEXPRESSION )
01223             && ( Expressions[numexpr].status == LOCALEXPRESSION
01224                 || Expressions[numexpr].status == GLOBALEXPRESSION ) ) {
01225 FoundExpr;;
01226             if ( c == '[' && s[1] == ']' ) {
01227                 Expressions[numexpr].printflag = par | PRINTCONTENTS;
01228                 *s++ = c; c = *++s;
01229             }
01230             else
01231                 Expressions[numexpr].printflag = par;
01232         }
01233         else if ( GetLastExprName(name,&numexpr)
01234             && ( Expressions[numexpr].status == LOCALEXPRESSION
01235                 || Expressions[numexpr].status == GLOBALEXPRESSION
01236                 || Expressions[numexpr].status == UNHIDELEXPRESSION
01237                 || Expressions[numexpr].status == UNHIDEGEXPRESSION
01238             ) ) {
01239             goto FoundExpr;
01240         }
01241         else {
01242             MesPrint("%s is not the name of an active expression",name);
01243             error = 1;
01244         }
01245         *s++ = c;
01246         if ( c == 0 ) return(0);
01247         if ( c == '-' || c == '+' ) s--;
01248     }
01249     else if ( *s == ',' ) s++;
01250     else {
01251         MesPrint("&Illegal object in (n)print statement");
01252         return(1);
01253     }
01254 }
01255 return(0);
01256 }
01257
01258 /*
01259 #] DoPrint :
01260 #[ CoPrint :
01261 */
01262
01263 int CoPrint(UBYTE *s) { return(DoPrint(s,PRINTON)); }
01264
01265 /*
01266 #] CoPrint :
01267 #[ CoPrintB :
01268 */
01269
01270 int CoPrintB(UBYTE *s) { return(DoPrint(s,PRINTCONTENT)); }
01271
01272 /*
01273 #] CoPrintB :
01274 #[ CoNPrint :
01275 */
01276
01277 int CoNPrint(UBYTE *s) { return(DoPrint(s,PRINTOFF)); }
01278
01279 /*
01280 #] CoNPrint :
01281 #[ CoPushHide :
01282 */
01283
01284 int CoPushHide(UBYTE *s)
01285 {
01286     GETIDENTITY
01287     WORD *ScratchBuf;
01288     int i;
01289     if ( AR.Fscr[2].PObuffer == 0 ) {
01290         ScratchBuf = (WORD *)Malloca1(AM.HideSize*sizeof(WORD),"hidesize");
01291         AR.Fscr[2].POsize = AM.HideSize * sizeof(WORD);
01292         AR.Fscr[2].POfull = AR.Fscr[2].POfill = AR.Fscr[2].PObuffer = ScratchBuf;
01293         AR.Fscr[2].POstop = AR.Fscr[2].PObuffer + AM.HideSize;
01294         PUTZERO(AR.Fscr[2].POposition);
01295     }
01296     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
01297     AC.HideLevel += 2;
01298     if ( *s ) {

```

```

01299         MesPrint("&PushHide statement should have no arguments");
01300         return(-1);
01301     }
01302     for ( i = 0; i < NumExpressions; i++ ) {
01303         switch ( Expressions[i].status ) {
01304             case DROPLEXPRESSION:
01305             case SKIPLEXPRESSION:
01306             case LOCALEXPRESSION:
01307                 Expressions[i].status = HIDELEXPRESSION;
01308                 Expressions[i].hidelevel = AC.HideLevel-1;
01309                 break;
01310             case DROPGEXPRESSION:
01311             case SKIPGEXPRESSION:
01312             case GLOBALEXPRESSION:
01313                 Expressions[i].status = HIDEGEXPRESSION;
01314                 Expressions[i].hidelevel = AC.HideLevel-1;
01315                 break;
01316             default:
01317                 break;
01318         }
01319     }
01320     return(0);
01321 }
01322
01323 /*
01324 #] CoPushHide :
01325 #[ CoPopHide :
01326 */
01327
01328 int CoPopHide(UBYTE *s)
01329 {
01330     int i;
01331     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
01332     if ( AC.HideLevel <= 0 ) {
01333         MesPrint("&PopHide statement without corresponding PushHide statement");
01334         return(-1);
01335     }
01336     AC.HideLevel -= 2;
01337     if ( *s ) {
01338         MesPrint("&PopHide statement should have no arguments");
01339         return(-1);
01340     }
01341     for ( i = 0; i < NumExpressions; i++ ) {
01342         switch ( Expressions[i].status ) {
01343             case HIDDENLEXPRESSION:
01344                 if ( Expressions[i].hidelevel > AC.HideLevel )
01345                     Expressions[i].status = UNHIDELEXPRESSION;
01346                 break;
01347             case HIDDENGEXPRESSION:
01348                 if ( Expressions[i].hidelevel > AC.HideLevel )
01349                     Expressions[i].status = UNHIDEGEXPRESSION;
01350                 break;
01351             default:
01352                 break;
01353         }
01354     }
01355     return(0);
01356 }
01357
01358 /*
01359 #] CoPopHide :
01360 #[ SetExprCases :
01361 */
01362
01363 int SetExprCases(int par, int setunset, int val)
01364 {
01365     switch ( par ) {
01366         case SKIP:
01367             switch ( val ) {
01368                 case SKIPLEXPRESSION:
01369                     if ( !setunset ) val = LOCALEXPRESSION;
01370                     break;
01371                 case SKIPGEXPRESSION:
01372                     if ( !setunset ) val = GLOBALEXPRESSION;
01373                     break;
01374                 case LOCALEXPRESSION:
01375                     if ( setunset ) val = SKIPLEXPRESSION;
01376                     break;
01377                 case GLOBALEXPRESSION:
01378                     if ( setunset ) val = SKIPGEXPRESSION;
01379                     break;
01380                 case INTOHIDEGEXPRESSION:
01381                 case INTOHIDELEXPRESSION:
01382                 default:
01383                     break;
01384             }
01385         break;

```

```

01386     case DROP:
01387         switch ( val ) {
01388             case SKIPLEXPRESSION:
01389             case LOCALEXPRESSION:
01390             case HIDELEXPRESSION:
01391                 if ( setunset ) val = DROPLEXPRESSION;
01392                 break;
01393             case DROPLEXPRESSION:
01394                 if ( !setunset ) val = LOCALEXPRESSION;
01395                 break;
01396             case SKIPGEXPRESSION:
01397             case GLOBALEXPRESSION:
01398             case HIDEGEXPRESSION:
01399                 if ( setunset ) val = DROPGEXPRESSION;
01400                 break;
01401             case DROPGEXPRESSION:
01402                 if ( !setunset ) val = GLOBALEXPRESSION;
01403                 break;
01404             case HIDDENLEXPRESSION:
01405             case UNHIDELEXPRESSION:
01406                 if ( setunset ) val = DROPHLEXPRESSION;
01407                 break;
01408             case HIDDENGEXPRESSION:
01409             case UNHIDEGEXPRESSION:
01410                 if ( setunset ) val = DROPHGEXPRESSION;
01411                 break;
01412             case DROPHLEXPRESSION:
01413                 if ( !setunset ) val = HIDDENLEXPRESSION;
01414                 break;
01415             case DROPHGEXPRESSION:
01416                 if ( !setunset ) val = HIDDENGEXPRESSION;
01417                 break;
01418             case INTOHIDEGEXPRESSION:
01419             case INTOHIDELEXPRESSION:
01420             default:
01421                 break;
01422         }
01423         break;
01424     case HIDE:
01425         switch ( val ) {
01426             case DROPLEXPRESSION:
01427             case SKIPLEXPRESSION:
01428             case LOCALEXPRESSION:
01429                 if ( setunset ) val = HIDELEXPRESSION;
01430                 break;
01431             case HIDELEXPRESSION:
01432                 if ( !setunset ) val = LOCALEXPRESSION;
01433                 break;
01434             case DROPGEXPRESSION:
01435             case SKIPGEXPRESSION:
01436             case GLOBALEXPRESSION:
01437                 if ( setunset ) val = HIDEGEXPRESSION;
01438                 break;
01439             case HIDEGEXPRESSION:
01440                 if ( !setunset ) val = GLOBALEXPRESSION;
01441                 break;
01442             case INTOHIDEGEXPRESSION:
01443             case INTOHIDELEXPRESSION:
01444             default:
01445                 break;
01446         }
01447         break;
01448     case UNHIDE:
01449         switch ( val ) {
01450             case HIDDENLEXPRESSION:
01451             case DROPHLEXPRESSION:
01452                 if ( setunset ) val = UNHIDELEXPRESSION;
01453                 break;
01454             case UNHIDELEXPRESSION:
01455                 if ( !setunset ) val = HIDDENLEXPRESSION;
01456                 break;
01457             case HIDDENGEXPRESSION:
01458             case DROPHGEXPRESSION:
01459                 if ( setunset ) val = UNHIDEGEXPRESSION;
01460                 break;
01461             case UNHIDEGEXPRESSION:
01462                 if ( !setunset ) val = HIDDENGEXPRESSION;
01463                 break;
01464             case INTOHIDEGEXPRESSION:
01465             case INTOHIDELEXPRESSION:
01466             default:
01467                 break;
01468         }
01469         break;
01470     case INTOHIDE:
01471         switch ( val ) {
01472             case HIDDENLEXPRESSION:

```

```

01473         case HIDDENEXPRESSION:
01474             MesPrint("&Expression is already hidden");
01475             return(-1);
01476         case DROPHLEXPRESSION:
01477         case DROPHGEXPRESSION:
01478         case UNHIDELEXPRESSION:
01479         case UNHIDEGEXPRESSION:
01480             if ( setunset ) {
01481                 MesPrint("&Cannot unhide/drop and put intohide expression in the same
module");
01482                 return(-1);
01483             }
01484             break;
01485         case LOCALEXPRESSION:
01486         case DROPLEXPRESSION:
01487         case SKIPLEXPRESSION:
01488         case HIDELEXPRESSION:
01489             if ( setunset ) val = INTOHIDELEXPRESSION;
01490             break;
01491         case GLOBALEXPRESSION:
01492         case DROPGEXPRESSION:
01493         case SKIPGEXPRESSION:
01494         case HIDEGEXPRESSION:
01495             if ( setunset ) val = INTOHIDEGEXPRESSION;
01496             break;
01497         case INTOHIDELEXPRESSION:
01498             if ( !setunset ) val = LOCALEXPRESSION;
01499             break;
01500         case INTOHIDEGEXPRESSION:
01501             if ( !setunset ) val = GLOBALEXPRESSION;
01502             break;
01503         default:
01504             break;
01505     }
01506     break;
01507     default:
01508         break;
01509 }
01510 return(val);
01511 }
01512
01513 /*
01514 #] SetExprCases :
01515 #[ SetExpr :
01516 */
01517
01518 int SetExpr(UBYTE *s, int setunset, int par)
01519 {
01520     WORD *w, numexpr;
01521     int error = 0, i;
01522     UBYTE *name, c;
01523     if ( *s == 0 ) {
01524         for ( i = 0; i < NumExpressions; i++ ) {
01525             w = &(Expressions[i].status);
01526             *w = SetExprCases(par,setunset,*w);
01527             if ( *w < 0 ) error = 1;
01528             if ( ( par == HIDE || par == INTOHIDE ) && setunset == 1 )
01529                 Expressions[i].hidelevel = AC.HideLevel;
01530         }
01531         return(0);
01532     }
01533     while ( *s ) {
01534         if ( *s == ',' ) { s++; continue; }
01535         if ( *s == '0' ) { s++; continue; }
01536         name = s;
01537         if ( ( s = SkipAName(s) ) == 0 ) {
01538             MesPrint("&Improper name for an expression: '%s'",name);
01539             return(1);
01540         }
01541         c = *s; *s = 0;
01542         if ( GetName(AC.exprnames,name,&numexpr,NOAUTO) == CEXPRESSION ) {
01543             w = &(Expressions[numexpr].status);
01544             *w = SetExprCases(par,setunset,*w);
01545             if ( *w < 0 ) error = 1;
01546             if ( ( par == HIDE || par == INTOHIDE ) && setunset == 1 )
01547                 Expressions[numexpr].hidelevel = AC.HideLevel;
01548         }
01549         else if ( GetName(AC.varnames,name,&numexpr,NOAUTO) != NAMENOTFOUND ) {
01550             MesPrint("&%s is not an expression",name);
01551             error = 1;
01552         }
01553         *s = c;
01554     }
01555     return(error);
01556 }
01557
01558 /*

```

```

01559     #] SetExpr :
01560     #[ CoDrop :
01561     */
01562
01563 int CoDrop(UBYTE *s) { return(SetExpr(s,1,DROP)); }
01564
01565 /*
01566     #] CoDrop :
01567     #[ CoNoDrop :
01568     */
01569
01570 int CoNoDrop(UBYTE *s) { return(SetExpr(s,0,DROP)); }
01571
01572 /*
01573     #] CoNoDrop :
01574     #[ CoSkip :
01575     */
01576
01577 int CoSkip(UBYTE *s) { return(SetExpr(s,1,SKIP)); }
01578
01579 /*
01580     #] CoSkip :
01581     #[ CoNoSkip :
01582     */
01583
01584 int CoNoSkip(UBYTE *s) { return(SetExpr(s,0,SKIP)); }
01585
01586 /*
01587     #] CoNoSkip :
01588     #[ CoHide :
01589     */
01590
01591 int CoHide(UBYTE *inp) {
01592     GETIDENTITY
01593     WORD *ScratchBuf;
01594     if ( AR.Fscr[2].PObuffer == 0 ) {
01595         ScratchBuf = (WORD *)Malloc1(AM.HideSize*sizeof(WORD),"hidesize");
01596         AR.Fscr[2].POsize = AM.HideSize * sizeof(WORD);
01597         AR.Fscr[2].POfull = AR.Fscr[2].POfill = AR.Fscr[2].PObuffer = ScratchBuf;
01598         AR.Fscr[2].POstop = AR.Fscr[2].PObuffer + AM.HideSize;
01599         PUTZERO(AR.Fscr[2].POposition);
01600     }
01601     return(SetExpr(inp,1,HIDE));
01602 }
01603
01604 /*
01605     #] CoHide :
01606     #[ CoIntoHide :
01607     */
01608
01609 int CoIntoHide(UBYTE *inp) {
01610     GETIDENTITY
01611     WORD *ScratchBuf;
01612     if ( AR.Fscr[2].PObuffer == 0 ) {
01613         ScratchBuf = (WORD *)Malloc1(AM.HideSize*sizeof(WORD),"hidesize");
01614         AR.Fscr[2].POsize = AM.HideSize * sizeof(WORD);
01615         AR.Fscr[2].POfull = AR.Fscr[2].POfill = AR.Fscr[2].PObuffer = ScratchBuf;
01616         AR.Fscr[2].POstop = AR.Fscr[2].PObuffer + AM.HideSize;
01617         PUTZERO(AR.Fscr[2].POposition);
01618     }
01619     return(SetExpr(inp,1,INTOHIDE));
01620 }
01621
01622 /*
01623     #] CoIntoHide :
01624     #[ CoNoIntoHide :
01625     */
01626
01627 int CoNoIntoHide(UBYTE *inp) { return(SetExpr(inp,0,INTOHIDE)); }
01628
01629 /*
01630     #] CoNoIntoHide :
01631     #[ CoNoHide :
01632     */
01633
01634 int CoNoHide(UBYTE *inp) { return(SetExpr(inp,0,HIDE)); }
01635
01636 /*
01637     #] CoNoHide :
01638     #[ CoUnHide :
01639     */
01640
01641 int CoUnHide(UBYTE *inp) { return(SetExpr(inp,1,UNHIDE)); }
01642
01643 /*
01644     #] CoUnHide :
01645     #[ CoNoUnHide :

```

```

01646 */
01647
01648 int CoNoUnHide(UBYTE *inp) { return(SetExpr(inp,0,UNHIDE)); }
01649
01650 /*
01651     #] CoNoUnHide :
01652     #[ AddToCom :
01653 */
01654
01655 void AddToCom(int n, WORD *array)
01656 {
01657     CBUF *C = cbuf+AC.cbunum;
01658 #ifdef COMPUFDEBUG
01659     MesPrint("  %a",n,array);
01660 #endif
01661     while ( C->Pointer+n >= C->Top ) DoubleCbuffer(AC.cbunum,C->Pointer,18);
01662     while ( --n >= 0 ) *(C->Pointer)++ = *array++;
01663 }
01664
01665 /*
01666     #] AddToCom :
01667     #[ AddComString :
01668 */
01669
01670 int AddComString(int n, WORD *array, UBYTE *thestring, int par)
01671 {
01672     CBUF *C = cbuf+AC.cbunum;
01673     UBYTE *s = thestring, *w;
01674 #ifdef COMPUFDEBUG
01675     WORD *cc;
01676     UBYTE *ww;
01677 #endif
01678     int i, numchars = 0, size, zeroes;
01679     while ( *s ) {
01680         if ( *s == '\\\\' ) s++;
01681         else if ( par == 1 &&
01682             ( ( *s == '%' && s[1] != 't' && s[1] != 'T' && s[1] != '$' &&
01683               s[1] != 'w' && s[1] != 'W' && s[1] != 'r' && s[1] != 0 ) || *s == '#'
01684             || *s == '@' || *s == '&' ) ) {
01685             numchars++;
01686         }
01687         s++; numchars++;
01688     }
01689     AddLHS(AC.cbunum);
01690     size = numchars/sizeof(WORD)+1;
01691     while ( C->Pointer+size+n+2 >= C->Top ) DoubleCbuffer(AC.cbunum,C->Pointer,19);
01692 #ifdef COMPUFDEBUG
01693     cc = C->Pointer;
01694 #endif
01695     *(C->Pointer)++ = array[0];
01696     *(C->Pointer)++ = size+n+2;
01697     for ( i = 1; i < n; i++ ) *(C->Pointer)++ = array[i];
01698     *(C->Pointer)++ = size;
01699 #ifdef COMPUFDEBUG
01700     ww =
01701 #endif
01702     w = (UBYTE *) (C->Pointer);
01703     zeroes = size*sizeof(WORD)-numchars;
01704     s = thestring;
01705     while ( *s ) {
01706         if ( *s == '\\\\' ) s++;
01707         else if ( par == 1 && ( *s == '%' &&
01708             s[1] != 't' && s[1] != 'T' && s[1] != '$' &&
01709             s[1] != 'w' && s[1] != 'W' && s[1] != 'r' && s[1] != 0 ) || *s == '#'
01710             || *s == '@' || *s == '&' ) ) {
01711             *w++ = '%';
01712         }
01713         *w++ = *s++;
01714     }
01715     while ( --zeroes >= 0 ) *w++ = 0;
01716     C->Pointer += size;
01717 #ifdef COMPUFDEBUG
01718     MesPrint("LH: %a",size+1+n,cc);
01719     MesPrint("    %s",thestring);
01720 #endif
01721     return(0);
01722 }
01723
01724 /*
01725     #] AddComString :
01726     #[ Add2ComStrings :
01727 */
01728
01729 int Add2ComStrings(int n, WORD *array, UBYTE *string1, UBYTE *string2)
01730 {
01731     CBUF *C = cbuf+AC.cbunum;
01732     UBYTE *s1 = string1, *s2 = string2, *w;

```



```

01733     int i, num1chars = 0, num2chars = 0, size1, size2, zeroes1, zeroes2;
01734     AddLHS(AC.cbufnum);
01735     while ( *s1 ) { s1++; num1chars++; }
01736     size1 = num1chars/sizeof(WORD)+1;
01737     if ( s2 ) {
01738         while ( *s2 ) { s2++; num2chars++; }
01739         size2 = num2chars/sizeof(WORD)+1;
01740     }
01741     else size2 = 0;
01742     while ( C->Pointer+size1+size2+n+3 >= C->Top ) DoubleCbuffer(AC.cbufnum,C->Pointer,20);
01743     *(C->Pointer)++ = array[0];
01744     *(C->Pointer)++ = size1+size2+n+3;
01745     for ( i = 1; i < n; i++ ) *(C->Pointer)++ = array[i];
01746     *(C->Pointer)++ = size1;
01747     w = (UBYTE *) (C->Pointer);
01748     zeroes1 = size1*sizeof(WORD)-num1chars;
01749     s1 = string1;
01750     while ( *s1 ) { *w++ = *s1++; }
01751     while ( --zeroes1 >= 0 ) *w++ = 0;
01752     C->Pointer += size1;
01753     *(C->Pointer)++ = size2;
01754     if ( size2 ) {
01755         w = (UBYTE *) (C->Pointer);
01756         zeroes2 = size2*sizeof(WORD)-num2chars;
01757         s2 = string2;
01758         while ( *s2 ) { *w++ = *s2++; }
01759         while ( --zeroes2 >= 0 ) *w++ = 0;
01760         C->Pointer += size2;
01761     }
01762     return(0);
01763 }
01764
01765 /*
01766 #] Add2ComStrings :
01767 #[ CoDiscard :
01768 */
01769
01770 int CoDiscard(UBYTE *s)
01771 {
01772     if ( *s == 0 ) {
01773         Add2Com(TYPEDISCARD)
01774         return(0);
01775     }
01776     MesPrint("&Illegal argument in discard statement: '%s'",s);
01777     return(1);
01778 }
01779
01780 /*
01781 #] CoDiscard :
01782 #[ CoContract :
01783
01784 Syntax:
01785     Contract
01786     Contract:#
01787     Contract #
01788     Contract:##, #
01789 */
01790
01791 static WORD carray[5] = { TYPEOPERATION,5,CONTRACT,0,0 };
01792
01793 int CoContract(UBYTE *s)
01794 {
01795     int x;
01796     if ( *s == ':' ) {
01797         s++;
01798         ParseNumber(x,s)
01799         if ( *s != ',' && *s ) {
01800 proper:      MesPrint("&Illegal number in contract statement");
01801             return(1);
01802         }
01803         if ( *s ) s++;
01804         carray[4] = x;
01805     }
01806     else carray[4] = 0;
01807     if ( FG.cTable[*s] == 1 ) {
01808         ParseNumber(x,s)
01809         if ( *s ) goto proper;
01810         carray[3] = x;
01811     }
01812     else if ( *s ) goto proper;
01813     else carray[3] = -1;
01814     return(AddNtoL(5,carray));
01815 }
01816
01817 /*
01818 #] CoContract :
01819 #[ CoGoTo :

```

```

01820 */
01821
01822 int CoGoTo(UBYTE *inp)
01823 {
01824     UBYTE *s = inp;
01825     int x;
01826     while ( FG.cTable[*s] <= 1 ) s++;
01827     if ( *s ) {
01828         MesPrint("&Label should be an alpha-numeric string");
01829         return(1);
01830     }
01831     x = GetLabel(inp);
01832     Add3Com(TYPEGOTO,x);
01833     return(0);
01834 }
01835
01836 /*
01837 #] CoGoTo :
01838 #[ CoLabel :
01839 */
01840
01841 int CoLabel(UBYTE *inp)
01842 {
01843     UBYTE *s = inp;
01844     int x;
01845     while ( FG.cTable[*s] <= 1 ) s++;
01846     if ( *s ) {
01847         MesPrint("&Label should be an alpha-numeric string");
01848         return(1);
01849     }
01850     x = GetLabel(inp);
01851     if ( AC.Labels[x] >= 0 ) {
01852         MesPrint("&Label %s defined more than once",AC.LabelNames[x]);
01853         return(1);
01854     }
01855     AC.Labels[x] = cbuf[AC.cbufnum].numlhs;
01856     return(0);
01857 }
01858
01859 /*
01860 #] CoLabel :
01861 #[ DoArgument :
01862
01863 Layout:
01864     par,full size,numlhs(+1),par,scale
01865     scale is for normalize
01866 */
01867
01868 int DoArgument(UBYTE *s, int par)
01869 {
01870     GETIDENTITY
01871     UBYTE *name, *t, *v, c;
01872     WORD *oldworkpointer = AT.WorkPointer, *w, *ww, number, *scale;
01873     int error = 0, zeroflag, type, x;
01874     AC.lhdollarflag = 0;
01875     while ( *s == ',' ) s++;
01876     w = AT.WorkPointer;
01877     *w++ = par;
01878     w++;
01879     switch ( par ) {
01880     case TYPEARG:
01881         if ( AC.arglevel >= MAXNEST ) {
01882             MesPrint("@Nesting of argument statements more than %d levels"
01883                 , (WORD)MAXNEST);
01884             return(-1);
01885         }
01886         AC.argsumcheck[AC.arglevel] = NestingChecksum();
01887         AC.argstack[AC.arglevel] = cbuf[AC.cbufnum].Pointer
01888             - cbuf[AC.cbufnum].Buffer + 2;
01889         AC.arglevel++;
01890         *w++ = cbuf[AC.cbufnum].numlhs;
01891         break;
01892     case TYPENORM:
01893     case TYPENORM4:
01894     case TYPESPLITARG:
01895     case TYPESPLITFIRSTARG:
01896     case TYPESPLITLASTARG:
01897     case TYPEFACTARG:
01898     case TYPEARGTOEXTRASymbol:
01899         *w++ = cbuf[AC.cbufnum].numlhs+1;
01900         break;
01901     }
01902     *w++ = par;
01903     scale = w;
01904     *w++ = 1;
01905     *w++ = 0;
01906     if ( *s == '^' ) {

```

```

01907         s++; ParseSignedNumber(x,s)
01908         while ( *s == ',' ) s++;
01909         *scale = x;
01910     }
01911     if ( *s == '(' ) {
01912         t = s+1; SKIPBRA3(s)      /* We did check the brackets already */
01913         if ( par == TYPEARG ) {
01914             MesPrint("&Illegal () entry in argument statement");
01915             error = 1; s++; goto skipbracks;
01916         }
01917         else if ( par == TYPESPLITFIRSTARG ) {
01918             MesPrint("&Illegal () entry in splitfirstarg statement");
01919             error = 1; s++; goto skipbracks;
01920         }
01921         else if ( par == TYPESPLITLASTARG ) {
01922             MesPrint("&Illegal () entry in splitlastarg statement");
01923             error = 1; s++; goto skipbracks;
01924         }
01925         v = t;
01926         while ( v < s ) {
01927             if ( *v == '?' ) {
01928                 MesPrint("&Wildcarding not allowed in this type of statement");
01929                 error = 1; break;
01930             }
01931             v++;
01932         }
01933         v = s++;
01934         if ( *t == '(' && v[-1] == ')' ) {
01935             t++; v--;
01936             if ( par == TYPESPLITARG ) oldworkpointer[0] = TYPESPLITARG2;
01937             else if ( par == TYPEFACTARG ) oldworkpointer[0] = TYPEFACTARG2;
01938             else if ( par == TYPENORM4 ) oldworkpointer[0] = TYPENORM4;
01939             else if ( par == TYPENORM ) {
01940                 if ( *t == '-' ) { oldworkpointer[0] = TYPENORM3; t++; }
01941                 else { oldworkpointer[0] = TYPENORM2; *scale = 0; }
01942             }
01943         }
01944         if ( error == 0 ) {
01945             CBUF *C = cbuf+AC.cbunum;
01946             WORD oldnumrhs = C->numrhs, oldnumlhs = C->numlhs;
01947             WORD prototype[SUBEXPSIZE+40]; /* Up to 10 nested sums! */
01948             WORD *m, *mm;
01949             int i, retcode;
01950             LONG oldpointer = C->Pointer - C->Buffer;
01951             *v = 0;
01952             prototype[0] = SUBEXPRESSION;
01953             prototype[1] = SUBEXPSIZE;
01954             prototype[2] = C->numrhs+1;
01955             prototype[3] = 1;
01956             prototype[4] = AC.cbunum;
01957             AT.WorkPointer += TYPEARGHEADSIZE+1;
01958             AddLHS(AC.cbunum);
01959             if ( ( retcode = CompileAlgebra(t,LHSIDE,prototype) ) < 0 )
01960                 error = 1;
01961             else {
01962                 prototype[2] = retcode;
01963                 ww = C->lhs[retcode];
01964                 AC.lhdollarflag = 0;
01965                 if ( *ww == 0 ) {
01966                     *w++ = -2; *w++ = 0;
01967                 }
01968                 else if ( ww[ww[0]] != 0 ) {
01969                     MesPrint("&There should be only one term between ()");
01970                     error = 1;
01971                 }
01972                 else if ( NewSort(BHEAD0) ) { if ( !error ) error = 1; }
01973                 else if ( NewSort(BHEAD0) ) {
01974                     LowerSortLevel();
01975                     if ( !error ) error = 1;
01976                 }
01977                 else {
01978                     AN.RepPoint = AT.RepCount + 1;
01979                     m = AT.WorkPointer;
01980                     mm = ww; i = *mm;
01981                     while ( --i >= 0 ) *m++ = *mm++;
01982                     mm = AT.WorkPointer; AT.WorkPointer = m;
01983                     AR.Cnumlhs = C->numlhs;
01984                     if ( Generator(BHEAD mm,C->numlhs) ) {
01985                         LowerSortLevel(); error = 1;
01986                     }
01987                     else if ( EndSort(BHEAD mm,0) < 0 ) {
01988                         error = 1;
01989                         AT.WorkPointer = mm;
01990                     }
01991                     else if ( *mm == 0 ) {
01992                         *w++ = -2; *w++ = 0;
01993                         AT.WorkPointer = mm;

```

```

01994         }
01995         else if ( mm[mm[0]] != 0 ) {
01996             error = 1;
01997             AT.WorkPointer = mm;
01998         }
01999         else {
02000             AT.WorkPointer = mm;
02001             m = mm+*mm;
02002             if ( par == TYPEFACTARG ) {
02003                 if ( *mm != ABS(m[-1])+1 ) {
02004                     *mm -= ABS(m[-1]); /* Strip coefficient */
02005                 }
02006                 mm[-1] = -*mm-1; w += *mm+1;
02007             }
02008             else {
02009                 *mm -= ABS(m[-1]); /* Strip coefficient */
02010             /*
02011                 if ( *mm == 1 ) { *w++ = -2; *w++ = 0; }
02012                 else
02013             /*
02014                 { mm[-1] = -*mm-1; w += *mm+1; }
02015             }
02016             oldworkpointer[1] = w - oldworkpointer;
02017         }
02018         LowerSortLevel();
02019     }
02020     oldworkpointer[5] = AC.lhdollarflag;
02021 }
02022 *v = ')';
02023 C->numrhs = oldnumrhs;
02024 C->numlhs = oldnumlhs;
02025 C->Pointer = C->Buffer + oldpointer;
02026 }
02027 }
02028 skipbracks:
02029     if ( *s == 0 ) { *w++ = 0; *w++ = 2; *w++ = 1; }
02030     else {
02031         do {
02032             if ( *s == ',' ) { s++; continue; }
02033             ww = w; *w++ = 0; w++;
02034             if ( FG.cTable[*s] > 1 && *s != '[' && *s != '{' ) {
02035                 MesPrint("&Illegal parameters in statement");
02036                 error = 1;
02037                 break;
02038             }
02039             while ( FG.cTable[*s] == 0 || *s == '[' || *s == '{' ) {
02040                 if ( *s == '{' ) {
02041                     name = s+1;
02042                     SKIPBRA2(s)
02043                     c = *s; *s = 0;
02044                     number = DoTempSet(name,s);
02045                     name--; *s++ = c; c = *s; *s = 0;
02046                     goto doset;
02047                 }
02048                 else {
02049                     name = s;
02050                     if ( ( s = SkipAName(s) ) == 0 ) {
02051                         MesPrint("&Illegal name '%s'",name);
02052                         return(1);
02053                     }
02054                     c = *s; *s = 0;
02055                     if ( ( type = GetName(AC.varnames,name,&number,WITHAUTO) ) == CSET ) {
02056                         doset:
02057                         if ( Sets[number].type != CFUNCTION ) goto nofun;
02058                         WORD *r1, *r2;
02059                         r1 = SetElements + Sets[number].first;
02060                         r2 = SetElements + Sets[number].last;
02061                         while ( r1 < r2 ) {
02062                             if ( *r1++ == FLOATFUN ) {
02063                                 MesPrint("&Illegal use of argument environment and float_.");
02064                                 error = 1;
02065                             }
02066                         }
02067                     #endif
02068                     *w++ = CSET; *w++ = number;
02069                 }
02070                 else if ( type == CFUNCTION ) {
02071                     #ifdef WITHFLOAT
02072                     if ( (number + FUNCTION) == FLOATFUN ) {
02073                         MesPrint("&Illegal use of argument environment and float_.");
02074                         error = 1;
02075                     }
02076                     #endif
02077                     *w++ = CFUNCTION; *w++ = number + FUNCTION;
02078                 }
02079                 else {
02080                     nofun:
                     MesPrint("&%s is not a function or a set of functions"

```

```

02081         ,name);
02082         error = 1;
02083     }
02084     }
02085     *s = c;
02086     while ( *s == ',' ) s++;
02087 }
02088 ww[1] = w - ww;
02089 ww = w; w++; zeroflag = 0;
02090 while ( FG.cTable[*s] == 1 ) {
02091     ParseNumber(x,s)
02092     if ( *s && *s != ',' ) {
02093         MesPrint("&Illegal separator after number");
02094         error = 1;
02095         while ( *s && *s != ',' ) s++;
02096     }
02097     while ( *s == ',' ) s++;
02098     if ( x == 0 ) zeroflag = 1;
02099     if ( !zeroflag ) *w++ = (WORD)x;
02100 }
02101 *ww = w - ww;
02102 } while ( *s );
02103 }
02104 oldworkpointer[1] = w - oldworkpointer;
02105 if ( par == TYPEARG ) { /* To make sure. The Pointer might move in the future */
02106     AC.argstack[AC.arglevel-1] = cbuf[AC.cbunum].Pointer
02107         - cbuf[AC.cbunum].Buffer + 2;
02108 }
02109 AddNtoL(oldworkpointer[1],oldworkpointer);
02110 AT.WorkPointer = oldworkpointer;
02111 return(error);
02112 }
02113
02114 /*
02115 #] DoArgument :
02116 #[ CoArgument :
02117 */
02118
02119 int CoArgument(UBYTE *s) { return(DoArgument(s,TYPEARG)); }
02120
02121 /*
02122 #] CoArgument :
02123 #[ CoEndArgument :
02124 */
02125
02126 int CoEndArgument(UBYTE *s)
02127 {
02128     CBUF *C = cbuf+AC.cbunum;
02129     while ( *s == ',' ) s++;
02130     if ( *s ) {
02131         MesPrint("&Illegal syntax for EndArgument statement");
02132         return(1);
02133     }
02134     if ( AC.arglevel <= 0 ) {
02135         MesPrint("&EndArgument without corresponding Argument statement");
02136         return(1);
02137     }
02138     AC.arglevel--;
02139     cbuf[AC.cbunum].Buffer[AC.argstack[AC.arglevel]] = C->numlhs;
02140     if ( AC.argsumcheck[AC.arglevel] != NestingChecksum() ) {
02141         MesNesting();
02142         return(1);
02143     }
02144     return(0);
02145 }
02146
02147 /*
02148 #] CoEndArgument :
02149 #[ CoInside :
02150 */
02151
02152 int CoInside(UBYTE *s) { return(ExecInside(s)); }
02153
02154 /*
02155 #] CoInside :
02156 #[ CoEndInside :
02157 */
02158
02159 int CoEndInside(UBYTE *s)
02160 {
02161     CBUF *C = cbuf+AC.cbunum;
02162     while ( *s == ',' ) s++;
02163     if ( *s ) {
02164         MesPrint("&Illegal syntax for EndInside statement");
02165         return(1);
02166     }
02167     if ( AC.insidelevel <= 0 ) {

```

```

02168         MesPrint("&EndInside without corresponding Inside statement");
02169         return(1);
02170     }
02171     AC.insidelevel--;
02172     cbuf[AC.cbufnum].Buffer[AC.insidestack[AC.insidelevel]] = C->numlhs;
02173     if ( AC.insidesumcheck[AC.insidelevel] != NestingChecksum() ) {
02174         MesNesting();
02175         return(1);
02176     }
02177     return(0);
02178 }
02179
02180 /*
02181  #] CoEndInside :
02182  #[ CoNormalize :
02183  */
02184
02185 int CoNormalize(UBYTE *s) { return(DoArgument(s,TYPENORM)); }
02186
02187 /*
02188  #] CoNormalize :
02189  #[ CoMakeInteger :
02190  */
02191
02192 int CoMakeInteger(UBYTE *s) { return(DoArgument(s,TYPENORM4)); }
02193
02194 /*
02195  #] CoMakeInteger :
02196  #[ CoSplitArg :
02197  */
02198
02199 int CoSplitArg(UBYTE *s) { return(DoArgument(s,TYPESPLITARG)); }
02200
02201 /*
02202  #] CoSplitArg :
02203  #[ CoSplitFirstArg :
02204  */
02205
02206 int CoSplitFirstArg(UBYTE *s) { return(DoArgument(s,TYPESPLITFIRSTARG)); }
02207
02208 /*
02209  #] CoSplitFirstArg :
02210  #[ CoSplitLastArg :
02211  */
02212
02213 int CoSplitLastArg(UBYTE *s) { return(DoArgument(s,TYPESPLITLASTARG)); }
02214
02215 /*
02216  #] CoSplitLastArg :
02217  #[ CoFactArg :
02218  */
02219
02220 int CoFactArg(UBYTE *s) {
02221     if ( ( AC.topolynomialflag & TOPOLYNOMIALFLAG ) != 0 ) {
02222         MesPrint("&ToPolynomial statement and FactArg statement are not allowed in the same module");
02223         return(1);
02224     }
02225     AC.topolynomialflag |= FACTARGFLAG;
02226     return(DoArgument(s,TYPEFACTARG));
02227 }
02228
02229 /*
02230  #] CoFactArg :
02231  #[ DoSymmetrize :
02232
02233     Syntax:
02234     Symmetrize Fun[:[number]] [Fields]      -> par = 0;
02235     AntiSymmetrize Fun[:[number]] [Fields]   -> par = 1;
02236     CycleSymmetrize Fun[:[number]] [Fields] -> par = 2;
02237     RCycleSymmetrize Fun[:[number]] [Fields]-> par = 3;
02238  */
02239
02240 int DoSymmetrize(UBYTE *s, int par)
02241 {
02242     GETIDENTITY
02243     int extra = 0, error = 0, err, fix, x, groupsize, num, i;
02244     UBYTE *name, c;
02245     WORD funnum, *w, *ww, type;
02246     for(;;) {
02247         name = s;
02248         if ( ( s = SkipAName(s) ) == 0 ) {
02249             MesPrint("&Improper function name");
02250             return(1);
02251         }
02252         c = *s; *s = 0;
02253         if ( c != ',' || ( FG.cTable[s[1]] != 0 && s[1] != '[' ) ) break;
02254         if ( par <= 0 && StrICmp(name,(UBYTE *)"cyclic") == 0 ) extra = 2;

```

```

02255         else if ( par <= 0 && StrICmp(name, (UBYTE *) "rcyclic") == 0 ) extra = 6;
02256     else {
02257         MesPrint("&Illegal option: '%s'", name);
02258         error = 1;
02259     }
02260     *s++ = c;
02261 }
02262 if ( ( err = GetVar(name, &type, &funnum, CFUNCTION, WITHAUTO) ) == NAMENOTFOUND ) {
02263     MesPrint("&Undefined function: %s", name);
02264     AddFunction(name, 0, 0, 0, 0, 0, -1, -1);
02265     *s++ = c;
02266     return(1);
02267 }
02268 funnum += FUNCTION;
02269 if ( err == -1 ) error = 1;
02270 *s = c;
02271 if ( *s == ':' ) {
02272     s++;
02273     if ( *s == ',' || *s == '(' || *s == 0 ) fix = -1;
02274     else if ( FG.cTable[*s] == 1 ) {
02275         ParseNumber(fix, s)
02276         if ( fix == 0 )
02277             Warning("Restriction to zero arguments removed");
02278     }
02279     else {
02280         MesPrint("&Illegal character after :");
02281         return(1);
02282     }
02283 }
02284 else fix = 0;
02285 w = AT.WorkPointer;
02286 *w++ = TYPEOPERATION;
02287 w++;
02288 *w++ = SYMMETRIZE;
02289 *w++ = par | extra;
02290 *w++ = funnum;
02291 *w++ = fix;
02292 /*
02293  And now the argument lists. We have either ,#,#,... or (#,#,...,#), (#,...
02294 */
02295 w += 2; ww = w; groupsize = -1;
02296 while ( *s == ',' ) s++;
02297 while ( *s ) {
02298     if ( *s == '(' ) {
02299         s++; num = 0;
02300         while ( *s && *s != ')' ) {
02301             if ( *s == ',' ) { s++; continue; }
02302             if ( FG.cTable[*s] != 1 ) goto illarg;
02303             ParseNumber(x, s)
02304             if ( x <= 0 || ( fix > 0 && x > fix ) ) goto illnum;
02305             num++;
02306             *w++ = x-1;
02307         }
02308         if ( *s == 0 ) {
02309             MesPrint("&Improper termination of statement");
02310             return(1);
02311         }
02312         if ( groupsize < 0 ) groupsize = num;
02313         else if ( groupsize != num ) goto group;
02314         s++;
02315     }
02316     else if ( FG.cTable[*s] == 1 ) {
02317         if ( groupsize < 0 ) groupsize = 1;
02318         else if ( groupsize != 1 ) {
02319             MesPrint("&All groups should have the same number of arguments");
02320             return(1);
02321         }
02322         ParseNumber(x, s)
02323         if ( x <= 0 || ( fix > 0 && x > fix ) ) {
02324             MesPrint("&Illegal argument number: %d", x);
02325             return(1);
02326         }
02327         *w++ = x-1;
02328     }
02329     else {
02330         MesPrint("&Illegal argument");
02331         return(1);
02332     }
02333     while ( *s == ',' ) s++;
02334 }
02335 /*
02336  Now the completion
02337 */
02338 if ( w == ww ) {
02339     ww[-1] = 1;
02340     ww[-2] = 0;
02341     if ( fix > 0 ) {

```

```

02342         for ( i = 0; i < fix; i++ ) *w++ = i;
02343         ww[-2] = fix; /* Bugfix 31-oct-2001. Reported by York Schroeder */
02344     }
02345 }
02346 else {
02347     ww[-1] = groupsize;
02348     ww[-2] = (w-ww)/groupsize;
02349 }
02350 AT.WorkPointer[1] = w - AT.WorkPointer;
02351 AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
02352 return(error);
02353 }
02354
02355 /*
02356 #] DoSymmetrize :
02357 #[ CoSymmetrize :
02358 */
02359
02360 int CoSymmetrize(UBYTE *s) { return(DoSymmetrize(s,SYMMETRIC)); }
02361
02362 /*
02363 #] CoSymmetrize :
02364 #[ CoAntiSymmetrize :
02365 */
02366
02367 int CoAntiSymmetrize(UBYTE *s) { return(DoSymmetrize(s,ANTISYMMETRIC)); }
02368
02369 /*
02370 #] CoAntiSymmetrize :
02371 #[ CoCycleSymmetrize :
02372 */
02373
02374 int CoCycleSymmetrize(UBYTE *s) { return(DoSymmetrize(s,CYCLESYMMETRIC)); }
02375
02376 /*
02377 #] CoCycleSymmetrize :
02378 #[ CoRCycleSymmetrize :
02379 */
02380
02381 int CoRCycleSymmetrize(UBYTE *s) { return(DoSymmetrize(s,RCYCLESYMMETRIC)); }
02382
02383 /*
02384 #] CoRCycleSymmetrize :
02385 #[ CoWrite :
02386 */
02387
02388 int CoWrite(UBYTE *s)
02389 {
02390     GETIDENTITY
02391     UBYTE *option;
02392     KEYWORDV *key;
02393     option = s;
02394     if ( ( ( s = SkipAName(s) ) == 0 ) || *s != 0 ) {
02395         MesPrint("&Proper use of write statement is: write option");
02396         return(1);
02397     }
02398     key = (KEYWORDV *)FindInKeyWord(option,(KEYWORD
02399 *)writeoptions,sizeof(writeoptions)/sizeof(KEYWORD));
02400     if ( key == 0 ) {
02401         MesPrint("&Unrecognized option in write statement");
02402         return(1);
02403     }
02404     *key->var = key->type;
02405     AR.SortType = AC.SortType;
02406     return(0);
02407 }
02408
02409 /*
02410 #] CoWrite :
02411 #[ CoNWrite :
02412 */
02413
02414 int CoNWrite(UBYTE *s)
02415 {
02416     GETIDENTITY
02417     UBYTE *option;
02418     KEYWORDV *key;
02419     option = s;
02420     if ( ( ( s = SkipAName(s) ) == 0 ) || *s != 0 ) {
02421         MesPrint("&Proper use of nwrite statement is: nwrite option");
02422         return(1);
02423     }
02424     key = (KEYWORDV *)FindInKeyWord(option,(KEYWORD
02425 *)writeoptions,sizeof(writeoptions)/sizeof(KEYWORD));
02426     if ( key == 0 ) {
02427         MesPrint("&Unrecognized option in nwrite statement");
02428         return(1);

```



```

02427     }
02428     *key->var = key->flags;
02429     AR.SortType = AC.SortType;
02430     return(0);
02431 }
02432
02433 /*
02434  #] CoNWrite :
02435  #[ CoRatio :
02436 */
02437
02438 static WORD ratstring[6] = { TYPEOPERATION, 6, RATIO, 0, 0, 0 };
02439
02440 int CoRatio(UBYTE *s)
02441 {
02442     UBYTE c, *t;
02443     int i, type, error = 0;
02444     WORD numsym, *rs;
02445     rs = ratstring+3;
02446     for ( i = 0; i < 3; i++ ) {
02447         if ( *s ) {
02448             t = s;
02449             s = SkipAName(s);
02450             c = *s; *s = 0;
02451             if ( ( ( type = GetName(AC.varnames,t,&numsym,WITHAUTO) ) != CSYMBOL )
02452                 && type != CDUBIOUS ) {
02453                 MesPrint("%s is not a symbol",t);
02454                 error = 4;
02455                 if ( type < 0 ) numsym = AddSymbol(t,-MAXPOWER,MAXPOWER,0,0);
02456             }
02457             *s = c;
02458             if ( *s == ',' ) s++;
02459         }
02460         else {
02461             if ( error == 0 )
02462                 MesPrint("&The ratio statement needs three symbols for its arguments");
02463             error++;
02464             numsym = 0;
02465         }
02466         *rs++ = numsym;
02467     }
02468     AddNtoL(6, ratstring);
02469     return(error);
02470 }
02471
02472 /*
02473  #] CoRatio :
02474  #[ CoRedefine :
02475
02476  We have a preprocessor variable and a (new) value for it.
02477  This value is inside a string that must be stored.
02478 */
02479
02480 int CoRedefine(UBYTE *s)
02481 {
02482     UBYTE *name, c, *args = 0;
02483     int numprevar;
02484     WORD code[2];
02485     name = s;
02486     if ( FG.cTable[*s] || ( s = SkipAName(s) ) == 0 || s[-1] == '_' ) {
02487         MesPrint("&Illegal name for preprocessor variable in redefine statement");
02488         return(1);
02489     }
02490     c = *s; *s = 0;
02491     for ( numprevar = NumPre-1; numprevar >= 0; numprevar-- ) {
02492         if ( StrCmp(name, PreVar[numprevar].name) == 0 ) break;
02493     }
02494     if ( numprevar < 0 ) {
02495         MesPrint("&There is no preprocessor variable with the name '%s'", name);
02496         *s = c;
02497         return(1);
02498     }
02499     *s = c;
02500
02501 /*
02502  The next code worries about arguments.
02503  It is a direct copy of the code in TheDefine in the preprocessor.
02504 */
02505     if ( *s == '(' ) { /* arguments. scan for correctness */
02506         s++; args = s;
02507         for (;;) {
02508             if ( chartype[*s] != 0 ) goto illarg;
02509             s++;
02510             while ( chartype[*s] <= 1 ) s++;
02511             while ( *s == ' ' || *s == '\t' ) s++;
02512             if ( *s == ')' ) break;
02513             if ( *s != ',' ) goto illargs;

```

```

02514         while ( *s == ' ' || *s == '\\t' ) s++;
02515     }
02516     *s++ = 0;
02517     while ( *s == ' ' || *s == '\\t' ) s++;
02518 }
02519 while ( *s == ',' ) s++;
02520 if ( *s != '"' ) {
02521     encl: MesPrint("&Value for %s should be enclosed in double quotes"
02522         ,PreVar[numprevar].name);
02523     return(1);
02524 }
02525 s++; name = s; /* actually name points to the new string */
02526 while ( *s && *s != '"' ) { if ( *s == '\\\\' ) s++; s++; }
02527 if ( *s != '"' ) goto encl;
02528 *s = 0;
02529 code[0] = TYPEREDEFPRE; code[1] = numprevar;
02530 /*
02531     AddComString(2,code,name,0);
02532 */
02533     Add2ComStrings(2,code,name,args);
02534     *s = '"';
02535 #ifdef PARALLELCODE
02536 /*
02537     Now we prepare the input numbering system for pthreads.
02538     We need a list of preprocessor variables that are redefined in this
02539     module.
02540 */
02541     {
02542         int j;
02543         WORD *newpf;
02544         LONG *newin;
02545         for ( j = 0; j < AC.numpfirstnum; j++ ) {
02546             if ( numprevar == AC.pfirstnum[j] ) break;
02547         }
02548         if ( j >= AC.numpfirstnum ) { /* add to list */
02549             if ( j >= AC.sizepfirstnum ) {
02550                 if ( AC.sizepfirstnum <= 0 ) { AC.sizepfirstnum = 10; }
02551                 else { AC.sizepfirstnum = 2 * AC.sizepfirstnum; }
02552                 newin = (LONG *)Malloc1(AC.sizepfirstnum*(sizeof(WORD)+sizeof(LONG)),"AC.pfirstnum");
02553                 newpf = (WORD *) (newin+AC.sizepfirstnum);
02554                 for ( j = 0; j < AC.numpfirstnum; j++ ) {
02555                     newpf[j] = AC.pfirstnum[j];
02556                     newin[j] = AC.inputnumbers[j];
02557                 }
02558                 if ( AC.inputnumbers ) M_free(AC.inputnumbers,"AC.pfirstnum");
02559                 AC.inputnumbers = newin;
02560                 AC.pfirstnum = newpf;
02561             }
02562             AC.pfirstnum[AC.numpfirstnum] = numprevar;
02563             AC.inputnumbers[AC.numpfirstnum] = -1;
02564             AC.numpfirstnum++;
02565         }
02566     }
02567 #endif
02568     return(0);
02569 illarg:;
02570     MesPrint("&Illegally formed name in argument of redefine statement");
02571     return(1);
02572 illargs:;
02573     MesPrint("&Illegally formed arguments in redefine statement");
02574     return(1);
02575 }
02576
02577 /*
02578     #] CoRedefine :
02579     #[ CoRenumber :
02580
02581     renumber    or renumber,0    Only exchanges (n^2 until no improvement)
02582     renumber,1    All permutations (could be slow)
02583 */
02584
02585 int CoRenumber(UBYTE *s)
02586 {
02587     int x;
02588     UBYTE *inp;
02589     while ( *s == ',' ) s++;
02590     inp = s;
02591     if ( *s == 0 ) { x = 0; }
02592     else ParseNumber(x,s)
02593     if ( *s == 0 && x >= 0 && x <= 1 ) {
02594         Add3Com(TYPERENUMBER,x);
02595         return(0);
02596     }
02597     MesPrint("&Illegal argument in Renumber statement: '%s'",inp);
02598     return(1);
02599 }
02600

```

```

02601 /*
02602      #] CoReNumber :
02603      #[ CoSum :
02604 */
02605
02606 int CoSum(UBYTE *s)
02607 {
02608     CBUF *C = cbuf+AC.cbufnum;
02609     UBYTE *ss = 0, c, *t;
02610     int error = 0, i = 0, type, x;
02611     WORD numindex,number;
02612     while ( *s ) {
02613         t = s;
02614         if ( *s == '$' ) {
02615             t++; s++; while ( FG.cTable[*s] < 2 ) s++;
02616             c = *s; *s = 0;
02617             if ( ( number = GetDollar(t) ) < 0 ) {
02618                 MesPrint("&Undefined variable %s",t);
02619                 if ( !error ) error = 1;
02620                 number = AddDollar(t,0,0,0);
02621             }
02622             numindex = -number;
02623         }
02624         else {
02625             if ( ( s = SkipAName(s) ) == 0 ) return(1);
02626             c = *s; *s = 0;
02627             if ( ( ( type = GetOName(AC.exprnames,t,&numindex,NOAUTO) ) != NAMENOTFOUND )
02628                 || ( ( type = GetOName(AC.varnames,t,&numindex,WITHAUTO) ) != CINDEX ) ) {
02629                 if ( type != NAMENOTFOUND ) error = NameConflict(type,t);
02630                 else {
02631                     MesPrint("&%s should have been declared as an index",t);
02632                     error = 1;
02633                     numindex = AddIndex(s,AC.lDefDim,AC.lDefDim4) + AM.OffsetIndex;
02634                 }
02635             }
02636             Add3Com(TYPESUM,numindex);
02637             i = 3; *s = c;
02638             if ( *s == 0 ) break;
02639             if ( *s != ',' ) {
02640                 MesPrint("&Illegal separator between objects in sum statement.");
02641                 return(1);
02642             }
02643             s++;
02644             if ( FG.cTable[*s] == 0 || *s == '[' || *s == '$' ) {
02645                 while ( FG.cTable[*s] == 0 || *s == '[' || *s == '$' ) {
02646                     if ( *s == '$' ) {
02647                         s++;
02648                         ss = t = s;
02649                         while ( FG.cTable[*s] < 2 ) s++;
02650                         c = *s; *s = 0;
02651                         if ( ( number = GetDollar(t) ) < 0 ) {
02652                             MesPrint("&Undefined variable %s",t);
02653                             if ( !error ) error = 1;
02654                             number = AddDollar(t,0,0,0);
02655                         }
02656                         numindex = -number;
02657                     }
02658                     else {
02659                         ss = t = s;
02660                         if ( ( s = SkipAName(s) ) == 0 ) return(1);
02661                         c = *s; *s = 0;
02662                         if ( ( ( type = GetOName(AC.exprnames,t,&numindex,NOAUTO) ) != NAMENOTFOUND )
02663                             || ( ( type = GetOName(AC.varnames,t,&numindex,WITHAUTO) ) != CINDEX ) ) {
02664                             if ( type != NAMENOTFOUND ) error = NameConflict(type,t);
02665                             else {
02666                                 MesPrint("&%s should have been declared as an index",t);
02667                                 error = 1;
02668                                 numindex = AddIndex(s,AC.lDefDim,AC.lDefDim4) + AM.OffsetIndex;
02669                             }
02670                         }
02671                     }
02672                 }
02673                 AddToCB(C,numindex)
02674                 i++;
02675                 C->Pointer[-i+1] = i;
02676                 *s = c;
02677                 if ( *s == 0 ) return(error);
02678                 if ( *s != ',' ) {
02679                     MesPrint("&Illegal separator between objects in sum statement.");
02680                     return(1);
02681                 }
02682                 s++;
02683             }
02684             if ( FG.cTable[*s] == 1 ) {
02685                 C->Pointer[-i+1]--; C->Pointer--; s = ss;
02686             }
02687         }
02688     }

```

```

02688     else if ( FG.cTable[*s] == 1 ) {
02689         while ( FG.cTable[*s] == 1 ) {
02690             t = s;
02691             x = *s++ - '0';
02692             while( FG.cTable[*s] == 1 ) x = 10*x + *s++ - '0';
02693             if ( *s && *s != ',' ) {
02694                 MesPrint("&%s is not a legal fixed index",t);
02695                 return(1);
02696             }
02697             else if ( x >= AM.OffsetIndex ) {
02698                 MesPrint("&%d is too large to be a fixed index",x);
02699                 error = 1;
02700             }
02701             else {
02702                 AddToCB(C,x)
02703                 i++;
02704                 C->Pointer[-i] = TYPESUMFIX;
02705                 C->Pointer[-i+1] = i;
02706             }
02707             if ( *s == 0 ) break;
02708             s++;
02709         }
02710     }
02711     else {
02712         MesPrint("&Illegal object in sum statement");
02713         error = 1;
02714     }
02715 }
02716 return(error);
02717 }
02718
02719 /*
02720 #] CoSum :
02721 #[ CoToTensor :
02722 */
02723
02724 static WORD cttarray[7] = { TYPEOPERATION,7,TENVEC,0,0,1,0 };
02725
02726 int CoToTensor(UBYTE *s)
02727 {
02728     UBYTE c, *t;
02729     int type, j, nargs, error = 0;
02730     WORD number, dol[2] = { 0, 0 };
02731     cttarray[1] = 6; /* length */
02732     cttarray[3] = 0; /* tensor */
02733     cttarray[4] = 0; /* vector */
02734     cttarray[5] = 1; /* option flags */
02735     /* cttarray[6] = 0; set veto */
02736     /*
02737     Count the number of the arguments. The validity of them is not checked here.
02738     */
02739     nargs = 0;
02740     t = s;
02741     for (;;) {
02742         while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
02743         if ( *s == 0 ) break;
02744         if ( *s == '!' ) {
02745             s++;
02746             if ( *s == '(' ) {
02747                 SKIPBRA2(s)
02748                 s++;
02749             } else {
02750                 if ( ( s = SkipAName(s) ) == 0 ) goto syntax_error;
02751             }
02752         } else {
02753             if ( ( s = SkipAName(s) ) == 0 ) goto syntax_error;
02754         }
02755         nargs++;
02756     }
02757     if ( nargs < 2 ) goto not_enough_arguments;
02758     s = t;
02759     /*
02760     Parse options, which are given as the arguments except the last two.
02761     */
02762     for ( j = 2; j < nargs; j++ ) {
02763         while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
02764         if ( *s == '!' ) {
02765             /*
02766             Handle !set or !{vector,...}. Note: If two or more sets are
02767             specified, then only the last one is used.
02768             */
02769             s++;
02770             cttarray[1] = 7;
02771             cttarray[5] |= 8;
02772             if ( FG.cTable[*s] == 0 || *s == '[' || *s == '_' ) {
02773                 t = s;
02774                 if ( ( s = SkipAName(s) ) == 0 ) goto syntax_error;

```

```

02775         c = *s; *s = 0;
02776         type = GetName(AC.varnames,t,&number,WITHAUTO);
02777         if ( type == CVECTOR ) {
02778             /*
02779                 As written in the manual, "!p" (without "{}") should work.
02780             */
02781             cttarray[6] = DoTempSet(t,s);
02782             *s = c;
02783             goto check_tempset;
02784         }
02785         else if ( type != CSET ) {
02786             MesPrint("&%s is not the name of a set or a vector",t);
02787             error = 1;
02788         }
02789         *s = c;
02790         cttarray[6] = number;
02791     }
02792     else if ( *s == '{' ) {
02793         t = ++s; SKIPBRA2(s) *s = 0;
02794         cttarray[6] = DoTempSet(t,s);
02795         *s++ = '}';
02796     check_tempset:
02797         if ( cttarray[6] < 0 ) {
02798             error = 1;
02799         }
02800         if ( AC.wildflag ) {
02801             MesPrint("&Improper use of wildcard(s) in set specification");
02802             error = 1;
02803         }
02804     }
02805     } else {
02806         /*
02807             Other options.
02808         */
02809         t = s;
02810         if ( ( s = SkipAName(s) ) == 0 ) goto syntax_error;
02811         c = *s; *s = 0;
02812         if ( StrICmp(t,(UBYTE *)"nosquare") == 0 ) cttarray[5] |= 2;
02813         else if ( StrICmp(t,(UBYTE *)"functions") == 0 ) cttarray[5] |= 4;
02814         else {
02815             MesPrint("&Unrecognized option in ToTensor statement: '%s'",t);
02816             *s = c;
02817             return(1);
02818         }
02819         *s = c;
02820     }
02821 }
02822 /*
02823     Now parse a vector and a tensor. The ordering doesn't matter.
02824 */
02825 for ( j = 0; j < 2; j++ ) {
02826     while ( *s == ' ' || *s == ',' || *s == '\\t' ) s++;
02827     t = s;
02828     if ( ( s = SkipAName(s) ) == 0 ) goto syntax_error;
02829     c = *s; *s = 0;
02830     if ( t[0] == '$' ) {
02831         dol[j] = GetDollar(t+1);
02832         if ( dol[j] < 0 ) dol[j] = AddDollar(t+1,DOLUNDEFINED,0,0);
02833     } else {
02834         type = GetName(AC.varnames,t,&number,WITHAUTO);
02835         if ( type == CVECTOR ) {
02836             cttarray[4] = number + AM.OffsetVector;
02837         }
02838         else if ( type == CFUNCTION && ( functions[number].spec > 0 ) ) {
02839             cttarray[3] = number + FUNCTION;
02840         }
02841         else {
02842             MesPrint("&%s is not a vector or a tensor",t);
02843             error = 1;
02844         }
02845     }
02846     *s = c;
02847 }
02848 if ( cttarray[3] == 0 || cttarray[4] == 0 ) {
02849     if ( dol[0] == 0 && dol[1] == 0 ) {
02850         goto not_enough_arguments;
02851     }
02852     else if ( cttarray[3] ) {
02853         if ( dol[1] ) cttarray[4] = dol[1];
02854         else if ( dol[0] ) cttarray[4] = dol[0];
02855         else {
02856             goto not_enough_arguments;
02857         }
02858     }
02859     else if ( cttarray[4] ) {
02860         if ( dol[1] ) cttarray[3] = -dol[1];
02861         else if ( dol[0] ) cttarray[3] = -dol[0];

```

```

02862         else {
02863             goto not_enough_arguments;
02864         }
02865     }
02866     else {
02867         if ( dol[0] == 0 || dol[1] == 0 ) {
02868             goto not_enough_arguments;
02869         }
02870         else {
02871             cttarray[3] = -dol[0]; cttarray[4] = dol[1];
02872         }
02873     }
02874 }
02875 AddNtoL(cttarray[1],cttarray);
02876 return(error);
02877
02878 syntax_error:
02879     MesPrint("&Syntax error in ToTensor statement");
02880     return(1);
02881
02882 not_enough_arguments:
02883     MesPrint("&ToTensor statement needs a vector and a tensor");
02884     return(1);
02885 }
02886
02887 /*
02888 #] CoToTensor :
02889 #[ CoToVector :
02890 */
02891
02892 static WORD ctvarray[6] = { TYPEOPERATION,6,TENVEC,0,0,0 };
02893
02894 int CoToVector(UBYTE *s)
02895 {
02896     UBYTE *t, c;
02897     int j, type, error = 0;
02898     WORD number, dol[2];
02899     dol[0] = dol[1] = 0;
02900     ctvarray[3] = ctvarray[4] = ctvarray[5] = 0;
02901     for ( j = 0; j < 2; j++ ) {
02902         t = s;
02903         if ( ( s = SkipAName(s) ) == 0 ) {
02904 proper:         MesPrint("&Arguments of ToVector statement should be a vector and a tensor");
02905                 return(1);
02906             }
02907             c = *s; *s = 0;
02908             if ( *t == '$' ) {
02909                 dol[j] = GetDollar(t+1);
02910                 if ( dol[j] < 0 ) dol[j] = AddDollar(t+1,DOLUNDEFINED,0,0);
02911             }
02912             else if ( ( type = GetName(AC.varnames,t,&number,WITHAUTO) ) == CVECTOR )
02913                 ctvarray[4] = number + AM.OffsetVector;
02914             else if ( type == CFUNCTION && ( functions[number].spec > 0 ) )
02915                 ctvarray[3] = number+FUNCTION;
02916             else {
02917                 MesPrint("&%s is not a vector or a tensor",t);
02918                 error = 1;
02919             }
02920             *s = c; if ( *s && *s != ',' ) goto proper;
02921             if ( *s ) s++;
02922         }
02923         if ( *s != 0 ) goto proper;
02924         if ( ctvarray[3] == 0 || ctvarray[4] == 0 ) {
02925             if ( dol[0] == 0 && dol[1] == 0 ) {
02926                 MesPrint("&ToVector statement needs a vector and a tensor");
02927                 error = 1;
02928             }
02929             else if ( ctvarray[3] ) {
02930                 if ( dol[1] ) ctvarray[4] = dol[1];
02931                 else if ( dol[0] ) ctvarray[4] = dol[0];
02932                 else {
02933                     MesPrint("&ToVector statement needs a vector and a tensor");
02934                     error = 1;
02935                 }
02936             }
02937             else if ( ctvarray[4] ) {
02938                 if ( dol[1] ) ctvarray[3] = -dol[1];
02939                 else if ( dol[0] ) ctvarray[3] = -dol[0];
02940                 else {
02941                     MesPrint("&ToVector statement needs a vector and a tensor");
02942                     error = 1;
02943                 }
02944             }
02945             else {
02946                 if ( dol[0] == 0 || dol[1] == 0 ) {
02947                     MesPrint("&ToVector statement needs a vector and a tensor");
02948                     error = 1;

```

```

02949         }
02950         else {
02951             ctvarray[3] = -dol[0]; ctvarray[4] = dol[1];
02952         }
02953     }
02954 }
02955 AddNtoL(6,ctvarray);
02956 return(error);
02957 }
02958
02959 /*
02960 #] CoToVector :
02961 #[ CoTrace4 :
02962 */
02963
02964 int CoTrace4(UBYTE *s)
02965 {
02966     int error = 0, type, option = CHISHOLM;
02967     UBYTE *t, c;
02968     WORD numindex, one = 1;
02969     KEYWORD *key;
02970     for (;;) {
02971         t = s;
02972         if ( FG.cTable[*s] == 1 ) break;
02973         if ( ( s = SkipAName(s) ) == 0 ) {
02974 proper:      MesPrint("&Proper syntax for Trace4 is 'Trace4[,options],index;'");
02975             return(1);
02976         }
02977         if ( *s == 0 ) break;
02978         c = *s; *s = 0;
02979         if ( ( key = FindKeyWord(t,trace4options,
02980             sizeof(trace4options)/sizeof(KEYWORD)) ) == 0 ) break;
02981         else {
02982             option |= key->type;
02983             option &= ~key->flags;
02984         }
02985         if ( ( *s++ = c ) != ',' ) {
02986             MesPrint("&Illegal separator in Trace4 statement");
02987             return(1);
02988         }
02989         if ( *s == 0 ) goto proper;
02990     }
02991     s = t;
02992     if ( FG.cTable[*s] == 1 ) {
02993 retry:      ParseNumber(numindex,s)
02994         if ( *s != 0 ) {
02995             MesPrint("&Last argument of Trace4 should be an index");
02996             return(1);
02997         }
02998         if ( numindex >= AM.OffsetIndex ) {
02999             MesPrint("&fixed index >= %d. Change value of OffsetIndex in setup file"
03000                 ,AM.OffsetIndex);
03001             return(1);
03002         }
03003     }
03004 }
03005 else if ( *s == '$' ) {
03006     if ( ( type = GetName(AC.dollarnames,s+1,&numindex,NOAUTO) ) == CDOLLAR )
03007         numindex = -numindex;
03008     else {
03009         MesPrint("&%s is undefined",s);
03010         numindex = AddDollar(s+1,DOLINDEX,&one,1);
03011         return(1);
03012     }
03013 tests:    s = SkipAName(s);
03014     if ( *s != 0 ) {
03015         MesPrint("&Trace4 should have a single index or $variable for its argument");
03016         return(1);
03017     }
03018 }
03019 else if ( ( type = GetName(AC.varnames,s,&numindex,WITHAUTO) ) == CINDEX ) {
03020     numindex += AM.OffsetIndex;
03021     goto tests;
03022 }
03023 else if ( type != -1 ) {
03024     if ( type != CDUBIOUS ) {
03025         if ( ( FG.cTable[*s] != 0 ) && ( *s != '[' ) ) {
03026             if ( *s == '+' && FG.cTable[s[1]] == 1 ) { s++; goto retry; }
03027             goto proper;
03028         }
03029         NameConflict(type,s);
03030         type = MakeDubious(AC.varnames,s,&numindex);
03031     }
03032     return(1);
03033 }
03034 else {
03035     MesPrint("&%s is not an index",s);

```

```

03036         numindex = AddIndex(s,AC.lDefDim,AC.lDefDim4) + AM.OffsetIndex;
03037         return(1);
03038     }
03039     if ( error ) return(error);
03040     if ( ( option & CHISHOLM ) != 0 )
03041         Add4Com(TYPECHISHOLM,numindex,(option & ALSOREVERSE));
03042     Add5Com(TYPEOPERATION,TAKETRACE,4 + (option & NOTRICK),numindex);
03043     return(0);
03044 }
03045
03046 /*
03047 #] CoTrace4 :
03048 #[ CoTraceN :
03049 */
03050
03051 int CoTraceN(UBYTE *s)
03052 {
03053     WORD numindex, one = 1;
03054     int type;
03055     if ( FG.cTable[*s] == 1 ) {
03056     retry:
03057         ParseNumber(numindex,s)
03058         if ( *s != 0 ) {
03059     proper:
03060             MesPrint("&TraceN should have a single index for its argument");
03061             return(1);
03062         }
03063         if ( numindex >= AM.OffsetIndex ) {
03064             MesPrint("&fixed index >= %d. Change value of OffsetIndex in setup file"
03065                 ,AM.OffsetIndex);
03066             return(1);
03067         }
03068         else if ( *s == '$' ) {
03069             if ( ( type = GetName(AC.dollarnames,s+1,&numindex,NOAUTO) ) == CDOLLAR )
03070                 numindex = -numindex;
03071             else {
03072                 MesPrint("&%s is undefined",s);
03073                 numindex = AddDollar(s+1,DOLINDEX,&one,1);
03074                 return(1);
03075             }
03076     tests:
03077             s = SkipAName(s);
03078             if ( *s != 0 ) {
03079                 MesPrint("&TraceN should have a single index or $variable for its argument");
03080                 return(1);
03081             }
03082             else if ( ( type = GetName(AC.varnames,s,&numindex,WITHAUTO) ) == CINDEX ) {
03083                 numindex += AM.OffsetIndex;
03084                 goto tests;
03085             }
03086             else if ( type != -1 ) {
03087                 if ( type != CDUBIOUS ) {
03088                     if ( ( FG.cTable[*s] != 0 ) && ( *s != '[' ) ) {
03089                         if ( *s == '+' && FG.cTable[s[1]] == 1 ) { s++; goto retry; }
03090                         goto proper;
03091                     }
03092                     NameConflict(type,s);
03093                     type = MakeDubious(AC.varnames,s,&numindex);
03094                 }
03095                 return(1);
03096             }
03097             else {
03098                 MesPrint("&%s is not an index",s);
03099                 numindex = AddIndex(s,AC.lDefDim,AC.lDefDim4) + AM.OffsetIndex;
03100                 return(1);
03101             }
03102             Add5Com(TYPEOPERATION,TAKETRACE,0,numindex);
03103             return(0);
03104         }
03105
03106 /*
03107 #] CoTraceN :
03108 #[ CoChisholm :
03109 */
03110
03111 int CoChisholm(UBYTE *s)
03112 {
03113     int error = 0, type, option = CHISHOLM;
03114     UBYTE *t, c;
03115     WORD numindex, one = 1;
03116     KEYWORD *key;
03117     for (;;) {
03118         t = s;
03119         if ( FG.cTable[*s] == 1 ) break;
03120         if ( ( s = SkipAName(s) ) == 0 ) {
03121     proper:
03122             MesPrint("&Proper syntax for Chisholm is 'Chisholm[,options],index;'");
03123             return(1);

```



```

03123     }
03124     if ( *s == 0 ) break;
03125     c = *s; *s = 0;
03126     if ( ( key = FindKeyWord(t,chisoptions,
03127         sizeof(chisoptions)/sizeof(KEYWORD)) ) == 0 ) break;
03128     else {
03129         option |= key->type;
03130         option &= ~key->flags;
03131     }
03132     if ( ( *s++ = c ) != ',' ) {
03133         MesPrint("&Illegal separator in Chisholm statement");
03134         return(1);
03135     }
03136     if ( *s == 0 ) goto proper;
03137 }
03138 s = t;
03139 if ( FG.cTable[*s] == 1 ) {
03140     ParseNumber(numindex,s)
03141     if ( *s != 0 ) {
03142         MesPrint("&Last argument of Chisholm should be an index");
03143         return(1);
03144     }
03145     if ( numindex >= AM.OffsetIndex ) {
03146         MesPrint("&fixed index >= %d. Change value of OffsetIndex in setup file"
03147             ,AM.OffsetIndex);
03148         return(1);
03149     }
03150 }
03151 else if ( *s == '$' ) {
03152     if ( ( type = GetName(AC.dollarnames,s+1,&numindex,NOAUTO) ) == CDOLLAR )
03153         numindex = -numindex;
03154     else {
03155         MesPrint("&%s is undefined",s);
03156         numindex = AddDollar(s+1,DOLINDEX,&one,1);
03157         return(1);
03158     }
03159 tests: s = SkipAName(s);
03160     if ( *s != 0 ) {
03161         MesPrint("&Chisholm should have a single index or $variable for its argument");
03162         return(1);
03163     }
03164 }
03165 else if ( ( type = GetName(AC.varnames,s,&numindex,WITHAUTO) ) == CINDEX ) {
03166     numindex += AM.OffsetIndex;
03167     goto tests;
03168 }
03169 else if ( type != -1 ) {
03170     if ( type != CDUBIOUS ) {
03171         NameConflict(type,s);
03172         type = MakeDubious(AC.varnames,s,&numindex);
03173     }
03174     return(1);
03175 }
03176 else {
03177     MesPrint("&%s is not an index",s);
03178     numindex = AddIndex(s,AC.lDefDim,AC.lDefDim4) + AM.OffsetIndex;
03179     return(1);
03180 }
03181 if ( error ) return(error);
03182 Add4Com(TYPECHISHOLM,numindex,(option & ALSOREVERSE));
03183 return(0);
03184 }
03185
03186 /*
03187 #] CoChisholm :
03188 #[ DoChain :
03189
03190 Syntax: Chainxx functionname;
03191 */
03192
03193 int DoChain(UBYTE *s, int option)
03194 {
03195     WORD numfunc,type;
03196     if ( *s == '$' ) {
03197         if ( ( type = GetName(AC.dollarnames,s+1,&numfunc,NOAUTO) ) == CDOLLAR )
03198             numfunc = -numfunc;
03199         else {
03200             MesPrint("&%s is undefined",s);
03201             numfunc = AddDollar(s+1,DOLINDEX,&one,1);
03202             return(1);
03203         }
03204     tests: s = SkipAName(s);
03205         if ( *s != 0 ) {
03206             MesPrint("&ChainIn/ChainOut should have a single function or $variable for its argument");
03207             return(1);
03208         }
03209     }

```

```

03210     else if ( ( type = GetName(AC.varnames,s,&numfunc,WITHAUTO) ) == CFUNCTION ) {
03211         numfunc += FUNCTION;
03212         goto tests;
03213     }
03214     else if ( type != -1 ) {
03215         if ( type != CDUBIOUS ) {
03216             NameConflict(type,s);
03217             type = MakeDubious(AC.varnames,s,&numfunc);
03218         }
03219         return(1);
03220     }
03221     else {
03222         MesPrint("%s is not a function",s);
03223         numfunc = AddFunction(s,0,0,0,0,0,-1,-1) + FUNCTION;
03224         return(1);
03225     }
03226     Add3Com(option,numfunc);
03227     return(0);
03228 }
03229
03230 /*
03231 #] DoChain :
03232 #[ CoChainin :
03233
03234 Syntax: Chainin functionname;
03235 */
03236
03237 int CoChainin(UBYTE *s)
03238 {
03239     return(DoChain(s,TYPECHAININ));
03240 }
03241
03242 /*
03243 #] CoChainin :
03244 #[ CoChainout :
03245
03246 Syntax: Chainout functionname;
03247 */
03248
03249 int CoChainout(UBYTE *s)
03250 {
03251     return(DoChain(s,TYPECHAINOUT));
03252 }
03253
03254 /*
03255 #] CoChainout :
03256 #[ CoExit :
03257 */
03258
03259 int CoExit(UBYTE *s)
03260 {
03261     UBYTE *name;
03262     WORD code = TYPEEXIT;
03263     while ( *s == ',' ) s++;
03264     if ( *s == 0 ) {
03265         Add3Com(TYPEEXIT,0);
03266         return(0);
03267     }
03268     name = s+1;
03269     s++;
03270     while ( *s ) { if ( *s == '\\\\' ) s++; s++; }
03271     if ( name[-1] != '"' || s[-1] != '"' ) {
03272         MesPrint("&Illegal syntax for exit statement");
03273         return(1);
03274     }
03275     s[-1] = 0;
03276     AddComString(1,&code,name,0);
03277     s[-1] = '"';
03278     return(0);
03279 }
03280
03281 /*
03282 #] CoExit :
03283 #[ CoInParallel :
03284 */
03285
03286 int CoInParallel(UBYTE *s)
03287 {
03288     return(DoInParallel(s,1));
03289 }
03290
03291 /*
03292 #] CoInParallel :
03293 #[ CoNotInParallel :
03294 */
03295
03296 int CoNotInParallel(UBYTE *s)

```

```

03297 {
03298     return(DoInParallel(s,0));
03299 }
03300
03301 /*
03302  #] CoNotInParallel :
03303  #[ DoInParallel :
03304
03305     InParallel;
03306     InParallel,names;
03307     NotInParallel;
03308     NotInParallel,names;
03309 */
03310
03311 int DoInParallel(UBYTE *s, int par)
03312 {
03313     #ifdef PARALLELCODE
03314         EXPRESSIONS e;
03315         WORD i;
03316     #endif
03317     WORD number;
03318     UBYTE *t, c;
03319     int error = 0;
03320     #ifndef WITHPTHREADS
03321         DUMMYUSE(par);
03322     #endif
03323     if ( *s == 0 ) {
03324         AC.inparallelflag = par;
03325     #ifdef PARALLELCODE
03326         for ( i = NumExpressions-1; i >= 0; i-- ) {
03327             e = Expressions+i;
03328             if ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION
03329                 || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION
03330             ) {
03331                 e->partodo = par;
03332             }
03333         }
03334     #endif
03335     }
03336     else {
03337         for(;;) { /* Look for a (comma separated) list of variables */
03338             while ( *s == ',' ) s++;
03339             if ( *s == 0 ) break;
03340             if ( *s == '[' || FG.cTable[*s] == 0 ) {
03341                 t = s;
03342                 if ( ( s = SkipAName(s) ) == 0 ) {
03343                     MesPrint("&Improper name for an expression: '%s'",t);
03344                     return(1);
03345                 }
03346                 c = *s; *s = 0;
03347                 if ( GetName(AC.exprnames,t,&number,NOAUTO) == CEXPRESSION ) {
03348                     #ifdef PARALLELCODE
03349                         e = Expressions+number;
03350                         if ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION
03351                             || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION
03352                         ) {
03353                             e->partodo = par;
03354                         }
03355                     #endif
03356                 }
03357                 else if ( GetName(AC.varnames,t,&number,NOAUTO) != NAMENOTFOUND ) {
03358                     MesPrint("&%s is not an expression",t);
03359                     error = 1;
03360                 }
03361                 *s = c;
03362             }
03363             else {
03364                 MesPrint("&Illegal object in InParallel statement");
03365                 error = 1;
03366                 while ( *s && *s != ',' ) s++;
03367                 if ( *s == 0 ) break;
03368             }
03369         }
03370     }
03371 }
03372 return(error);
03373 }
03374
03375 /*
03376  #] DoInParallel :
03377  #[ CoInExpression :
03378 */
03379
03380 int CoInExpression(UBYTE *s)
03381 {
03382     GETIDENTITY
03383     UBYTE *t, c;

```

```

03384     WORD *w, number;
03385     int error = 0;
03386     w = AT.WorkPointer;
03387     if ( AC.inexprlevel >= MAXNEST ) {
03388         MesPrint("@Nesting of inexpression statements more than %d levels", (WORD)MAXNEST);
03389         return(-1);
03390     }
03391     AC.inexprsumcheck[AC.inexprlevel] = NestingChecksum();
03392     AC.inexprstack[AC.inexprlevel] = cbuf[AC.cbunum].Pointer
03393         - cbuf[AC.cbunum].Buffer + 2;
03394     AC.inexprlevel++;
03395     *w++ = TYPEINEXPRESSION;
03396     w++; w++;
03397     for(;;) { /* Look for a (comma separated) list of variables */
03398         while ( *s == ',' ) s++;
03399         if ( *s == 0 ) break;
03400         if ( *s == '[' || FG.cTable[*s] == 0 ) {
03401             t = s;
03402             if ( ( s = SkipAName(s) ) == 0 ) {
03403                 MesPrint("&Improper name for an expression: '%s'",t);
03404                 return(1);
03405             }
03406             c = *s; *s = 0;
03407             if ( GetName(AC.exprnames,t,&number,NOAUTO) == CEXPRESSION ) {
03408                 *w++ = number;
03409             }
03410             else if ( GetName(AC.varnames,t,&number,NOAUTO) != NAMENOTFOUND ) {
03411                 MesPrint("%s is not an expression",t);
03412                 error = 1;
03413             }
03414             *s = c;
03415         }
03416         else {
03417             MesPrint("&Illegal object in InExpression statement");
03418             error = 1;
03419             while ( *s && *s != ',' ) s++;
03420             if ( *s == 0 ) break;
03421         }
03422     }
03423     AT.WorkPointer[1] = w - AT.WorkPointer;
03424     AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
03425     return(error);
03426 }
03427 /*
03428 #] CoInExpression :
03429 #[ CoEndInExpression :
03430 */
03431 int CoEndInExpression(UBYTE *s)
03432 {
03433     CBUF *C = cbuf+AC.cbunum;
03434     while ( *s == ',' ) s++;
03435     if ( *s ) {
03436         MesPrint("&Illegal syntax for EndInExpression statement");
03437         return(1);
03438     }
03439     if ( AC.inexprlevel <= 0 ) {
03440         MesPrint("&EndInExpression without corresponding InExpression statement");
03441         return(1);
03442     }
03443     AC.inexprlevel--;
03444     cbuf[AC.cbunum].Buffer[AC.inexprstack[AC.inexprlevel]] = C->numlhs;
03445     if ( AC.inexprsumcheck[AC.inexprlevel] != NestingChecksum() ) {
03446         MesNesting();
03447         return(1);
03448     }
03449     return(0);
03450 }
03451 /*
03452 #] CoEndInExpression :
03453 #[ CoSetExitFlag :
03454 */
03455 int CoSetExitFlag(UBYTE *s)
03456 {
03457     if ( *s ) {
03458         MesPrint("&Illegal syntax for the SetExitFlag statement");
03459         return(1);
03460     }
03461     Add2Com(TYPESETEXIT);
03462     return(0);
03463 }
03464 /*
03465 #] CoSetExitFlag :

```

```

03471     #[ CoTryReplace :
03472     /*
03473     int CoTryReplace(UBYTE *p)
03474     {
03475         GETIDENTITY
03476         UBYTE *name, c;
03477         WORD *w, error = 0, i, which = -1, c1, minvec = 0;
03478         w = AT.WorkPointer;
03479         *w++ = TYPETRY;
03480         *w++ = 3;
03481         *w++ = 0;
03482         *w++ = REPLACEMENT;
03483         *w++ = FUNHEAD;
03484         FILLFUN(w)
03485     /*
03486     Now we have to read a function argument for the replace_ function.
03487     Current arguments that we allow involve only single arguments
03488     that do not expand further. No brackets!
03489     /*
03490     while ( *p ) {
03491     /*
03492         No numbers yet
03493     /*
03494         if ( *p == '-' && minvec == 0 && which == (CVECTOR+1) ) {
03495             minvec = 1; p++;
03496         }
03497         if ( *p == '[' || FG.cTable[*p] == 0 ) {
03498             name = p;
03499             if ( ( p = SkipAName(p) ) == 0 ) return(1);
03500             c = *p; *p = 0;
03501             i = GetName(AC.varnames,name,&c1,WITHAUTO);
03502             if ( which >= 0 && i >= 0 && i != CDUBIOUS && which != (i+1) ) {
03503                 MesPrint("&Illegal combination of objects in TryReplace");
03504                 error = 1;
03505             }
03506             else if ( minvec && i != CVECTOR && i != CDUBIOUS ) {
03507                 MesPrint("&Currently a - sign can be used only with a vector in TryReplace");
03508                 error = 1;
03509             }
03510             else switch ( i ) {
03511                 case CSYMBOL: *w++ = -SYMBOL; *w++ = c1; break;
03512                 case CVECTOR:
03513                     if ( minvec ) *w++ = -MINVECTOR;
03514                     else *w++ = -VECTOR;
03515                     *w++ = c1 + AM.OffsetVector;
03516                     minvec = 0;
03517                     break;
03518                 case CINDEX: *w++ = -INDEX; *w++ = c1 + AM.OffsetIndex;
03519                     if ( c1 >= AM.WilInd && c == '?' ) { *p++ = c; c = *p; }
03520                     break;
03521                 case CFUNCTION: *w++ = -c1-FUNCTION; break;
03522                 case CDUBIOUS: minvec = 0; error = 1; break;
03523                 default:
03524                     MesPrint("&Illegal object type in TryReplace: %s",name);
03525                     error = 1;
03526                     i = 0;
03527                     break;
03528             }
03529             if ( which < 0 ) which = i+1;
03530             else which = -1;
03531             *p = c;
03532             if ( *p == ',' ) p++;
03533             continue;
03534         }
03535         else {
03536             MesPrint("&Illegal object in TryReplace");
03537             error = 1;
03538             while ( *p && *p != ',' ) {
03539                 if ( *p == '(' ) SKIPBRA3(p)
03540                 else if ( *p == '{' ) SKIPBRA2(p)
03541                 else if ( *p == '[' ) SKIPBRAL(p)
03542                 else p++;
03543             }
03544         }
03545         if ( *p == ',' ) p++;
03546         if ( which < 0 ) which = 0;
03547         else which = -1;
03548     }
03549     if ( which >= 0 ) {
03550         MesPrint("&Odd number of arguments in TryReplace");
03551         error = 1;
03552     }
03553     i = w - AT.WorkPointer;
03554     AT.WorkPointer[1] = i;
03555     AT.WorkPointer[2] = i - 3;
03556     AT.WorkPointer[4] = i - 3;
03557     AddNtoL((int)i,AT.WorkPointer);

```

```

03558     return(error);
03559 }
03560
03561 /*
03562  #] CoTryReplace :
03563  #[ CoModulus :
03564
03565     Old syntax: Modulus [-] number [:number]
03566     New syntax: Modulus [option(s)] number
03567     Options are: NoFunctions/CoefficientsOnly/AlsoFunctions
03568                 PlusMin/Positive
03569                 InverseTable
03570                 PrintPowersOf(number)
03571                 AlsoPowers/NoPowers
03572                 AlsoDollars/NoDollars
03573     Notice: We change the defaults. This may cause problems to some.
03574 */
03575
03576 int CoModulus(UBYTE *inp)
03577 {
03578     GETIDENTITY
03579     int Retval = 0, sign = 1;
03580     UBYTE *p, c;
03581     while ( *inp == ' ' || *inp == '\t' ) inp++;
03582     if ( *inp == 0 ) {
03583         SwitchOff:
03584         if ( AC.modpowers ) M_free(AC.modpowers,"AC.modpowers");
03585         AC.modpowers = 0;
03586         AN.ncmod = AC.ncmod = 0;
03587         if ( AC.halfmod ) M_free(AC.halfmod,"halfmod");
03588         AC.halfmod = 0; AC.nhalfmod = 0;
03589         if ( AC.modinverses ) M_free(AC.modinverses,"modinverses");
03590         AC.modinverses = 0;
03591         AC.modmode = 0;
03592         return(0);
03593     }
03594     #ifdef WITHFLOAT
03595     if ( AT.aux_ != 0 ) {
03596         MesPrint("&Simultaneous use of floating point and modulus arithmetic makes no sense.");
03597         Retval = 1;
03598     }
03599     #endif
03600     AC.modmode = 0;
03601     if ( *inp == '-' ) {
03602         sign = -1;
03603         inp++;
03604     }
03605     else {
03606         while ( FG.cTable[*inp] == 0 ) {
03607             p = inp;
03608             while ( FG.cTable[*inp] == 0 ) inp++;
03609             c = *inp; *inp = 0;
03610             if ( StrICmp(p,(UBYTE *)"nofunctions") == 0 ) {
03611                 AC.modmode &= ~ALSOFUNARGS;
03612             }
03613             else if ( StrICmp(p,(UBYTE *)"alsofunctions") == 0 ) {
03614                 AC.modmode |= ALSOFUNARGS;
03615             }
03616             else if ( StrICmp(p,(UBYTE *)"coefficientsonly") == 0 ) {
03617                 AC.modmode &= ~ALSOFUNARGS;
03618                 AC.modmode &= ~ALSOPOWERS;
03619                 sign = -1;
03620             }
03621             else if ( StrICmp(p,(UBYTE *)"plusmin") == 0 ) {
03622                 AC.modmode |= POSNEG;
03623             }
03624             else if ( StrICmp(p,(UBYTE *)"positive") == 0 ) {
03625                 AC.modmode &= ~POSNEG;
03626             }
03627             else if ( StrICmp(p,(UBYTE *)"inversetable") == 0 ) {
03628                 AC.modmode |= INVERSETABLE;
03629             }
03630             else if ( StrICmp(p,(UBYTE *)"noinversetable") == 0 ) {
03631                 AC.modmode &= ~INVERSETABLE;
03632             }
03633             else if ( StrICmp(p,(UBYTE *)"nodollars") == 0 ) {
03634                 AC.modmode &= ~ALSODOLLARS;
03635             }
03636             else if ( StrICmp(p,(UBYTE *)"alsodollars") == 0 ) {
03637                 AC.modmode |= ALSODOLLARS;
03638             }
03639             else if ( StrICmp(p,(UBYTE *)"printpowersof") == 0 ) {
03640                 *inp = c;
03641                 if ( *inp != '(' ) {
03642                     badsyntax:
03643                     MesPrint("&Bad syntax in argument of PrintPowersOf(number) in Modulus statement");
03644                     return(1);

```

```

03645         }
03646         while ( *inp == ',' || *inp == ' ' || *inp == '\t' ) inp++;
03647         inp++; p = inp;
03648         if ( FG.cTable[*inp] != 1 ) goto badsyntax;
03649         do { inp++; } while ( FG.cTable[*inp] == 1 );
03650         c = *inp; *inp = 0;
03651         if ( GetLong(p, (UWORD *)AC.powmod,&AC.npowmod) ) Retval = -1;
03652         if ( TakeModulus((UWORD *)AC.powmod,&AC.npowmod,AC.cmod,AC.ncmod,NOUNPACK) ) Retval = -1;
03653         if ( AC.npowmod == 0 ) {
03654             MesPrint("&Improper value for generator");
03655             Retval = -1;
03656         }
03657         if ( MakeModTable() ) Retval = -1;
03658         AC.DirtPow = 1;
03659         *inp = c;
03660         while ( *inp == ',' || *inp == ' ' || *inp == '\t' ) inp++;
03661         if ( *inp != ')' ) goto badsyntax;
03662         inp++;
03663         c = *inp;
03664     }
03665     else if ( StrICmp(p,(UBYTE *)"alsopowers") == 0 ) {
03666         AC.modmode |= ALSOPOWERS;
03667         sign = 1;
03668     }
03669     else if ( StrICmp(p,(UBYTE *)"nopowers") == 0 ) {
03670         AC.modmode &= ~ALSOPOWERS;
03671         sign = -1;
03672     }
03673     else {
03674         MesPrint("&Unrecognized option %s in Modulus statement",inp);
03675         return(1);
03676     }
03677     *inp = c;
03678     while ( *inp == ',' || *inp == ' ' || *inp == '\t' ) inp++;
03679     if ( *inp == 0 ) {
03680         MesPrint("&Modulus statement with no value!!!");
03681         return(1);
03682     }
03683 }
03684 }
03685 p = inp;
03686 if ( FG.cTable[*inp] != 1 ) {
03687     MesPrint("&Invalid value for modulus:%s",inp);
03688     if ( AC.modpowers ) M_free(AC.modpowers,"AC.modpowers");
03689     AC.modpowers = 0;
03690     AN.ncmod = AC.ncmod = 0;
03691     if ( AC.halfmod ) M_free(AC.halfmod,"halfmod");
03692     AC.halfmod = 0; AC.nhalfmod = 0;
03693     if ( AC.modinverses ) M_free(AC.modinverses,"modinverses");
03694     AC.modinverses = 0;
03695     return(1);
03696 }
03697 do { inp++; } while ( FG.cTable[*inp] == 1 );
03698 c = *inp; *inp = 0;
03699 Retval = GetLong(p, (UWORD *)AC.cmod,&AC.ncmod);
03700 if ( Retval == 0 && AC.ncmod == 0 ) goto SwitchOff;
03701 if ( sign < 0 ) AC.ncmod = -AC.ncmod;
03702 AN.ncmod = AC.ncmod;
03703 if ( ( AC.modmode & INVERSETABLE ) != 0 ) MakeInverses();
03704 if ( AC.halfmod ) M_free(AC.halfmod,"halfmod");
03705 AC.halfmod = 0; AC.nhalfmod = 0;
03706 return(Retval);
03707 }
03708
03709 /*
03710 #] CoModulus :
03711 #[ CoRepeat :
03712 */
03713
03714 int CoRepeat(UBYTE *inp)
03715 {
03716     int error = 0;
03717     AC.RepSumCheck[AC.RepLevel] = NestingChecksum();
03718     AC.RepLevel++;
03719     if ( AC.RepLevel > AM.RepMax ) {
03720         MesPrint("&Too many repeat levels. Maximum is %d",AM.RepMax);
03721         return(1);
03722     }
03723     Add3Com(TYPEREPEAT,-1) /* Means indefinite */
03724     while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
03725     if ( *inp ) {
03726         error = CompileStatement(inp);
03727         if ( CoEndRepeat(inp) ) error = 1;
03728     }
03729     return(error);
03730 }
03731

```

```

03732 /*
03733      #] CoRepeat :
03734      #[ CoEndRepeat :
03735 */
03736
03737 int CoEndRepeat(UBYTE *inp)
03738 {
03739     CBUF *C = cbuf+AC.cbunum;
03740     int level, error = 0, repeatlevel = 0;
03741     DUMMYUSE(inp);
03742     AC.RepLevel--;
03743     if ( AC.RepLevel < 0 ) {
03744         MesPrint("&EndRepeat without Repeat");
03745         AC.RepLevel = 0;
03746         return(1);
03747     }
03748     else if ( AC.RepSumCheck[AC.RepLevel] != NestingChecksum() ) {
03749         MesNesting();
03750         error = 1;
03751     }
03752     level = C->numlhs+1;
03753     while ( level > 0 ) {
03754         if ( C->lhs[--level][0] == TYPEPERPEAT ) {
03755             if ( repeatlevel == 0 ) {
03756                 Add3Com(TYPEENDREPEAT,level)
03757                 return(error);
03758             }
03759             repeatlevel--;
03760         }
03761         else if ( C->lhs[level][0] == TYPEENDREPEAT ) repeatlevel++;
03762     }
03763     return(1);
03764 }
03765
03766 /*
03767      #] CoEndRepeat :
03768      #[ DoBrackets :
03769
03770      Reads in the bracket information.
03771      Storage is in the form of a regular term.
03772      No subterms and arguments are allowed.
03773 */
03774
03775 int DoBrackets(UBYTE *inp, int par)
03776 {
03777     GETIDENTITY
03778     UBYTE *p, *pp, c;
03779     WORD *to, i, type, *w, error = 0;
03780     WORD c1,c2, *WorkSave;
03781     int biflag;
03782     p = inp;
03783     WorkSave = to = AT.WorkPointer;
03784     to++;
03785     if ( AT.BrackBuf == 0 ) {
03786         AR.MaxBracket = 100;
03787         AT.BrackBuf = (WORD *)Malloca(sizeof(WORD)*(AR.MaxBracket+1),"bracket buffer");
03788     }
03789     *AT.BrackBuf = 0;
03790     AR.BracketOn = 0;
03791     AC.bracketindexflag = 0;
03792     AT.bracketindexflag = 0;
03793     if ( *p == '+' || *p == '-' ) p++;
03794     if ( p[-1] == ',' && *p ) p--;
03795     if ( p[-1] == '+' && *p ) { biflag = 1; if ( *p != ',' ) { *--p = ','; } }
03796     else if ( p[-1] == '-' && *p ) { biflag = -1; if ( *p != ',' ) { *--p = ','; } }
03797     else biflag = 0;
03798     while ( *p == ',' ) {
03799         redo: AR.BracketOn++;
03800         while ( *p == ',' ) p++;
03801         if ( *p == 0 ) break;
03802         if ( *p == '0' ) {
03803             p++; while ( *p == '0' ) p++;
03804             continue;
03805         }
03806         inp = pp = p;
03807         p = SkipAName(p);
03808         if ( p == 0 ) return(1);
03809         c = *p;
03810         *p = 0;
03811         type = GetName(AC.varnames,inp,&c1,WITHAUTO);
03812         if ( c == '.' ) {
03813             if ( type == CVECTOR || type == CDUBIOUS ) {
03814                 *p++ = c;
03815                 inp = p;
03816                 p = SkipAName(p);
03817                 if ( p == 0 ) return(1);
03818                 c = *p;

```



```

03819         *p = 0;
03820         type = GetName(AC.varnames,inp,&c2,WITHAUTO);
03821         if ( type != CVECTOR && type != CDUBIOUS ) {
03822             MesPrint("&Not a vector in dotproduct in bracket statement: %s",inp);
03823             error = 1;
03824         }
03825         else type = CDOTPRODUCT;
03826     }
03827     else {
03828         MesPrint("&Illegal use of . after %s in bracket statement",inp);
03829         error = 1;
03830         *p++ = c;
03831         goto redo;
03832     }
03833 }
03834 switch ( type ) {
03835     case CSYMBOL :
03836         *to++ = SYMBOL; *to++ = 4; *to++ = c1; *to++ = 1; break;
03837     case CVECTOR :
03838         *to++ = INDEX; *to++ = 3; *to++ = AM.OffsetVector + c1; break;
03839     case CFUNCTION :
03840         *to++ = c1+FUNCTION; *to++ = FUNHEAD; *to++ = 0;
03841         FILLFUN3(to)
03842         break;
03843     case CDOTPRODUCT :
03844         *to++ = DOTPRODUCT; *to++ = 5; *to++ = c1 + AM.OffsetVector;
03845         *to++ = c2 + AM.OffsetVector; *to++ = 1; break;
03846     case CDELTA :
03847         *to++ = DELTA; *to++ = 4; *to++ = EMPTYINDEX; *to++ = EMPTYINDEX; break;
03848     case CSET :
03849         *to++ = SETSET; *to++ = 4; *to++ = c1; *to++ = Sets[c1].type; break;
03850     default :
03851         MesPrint("&Illegal bracket request for %s",pp);
03852         error = 1; break;
03853 }
03854 *p = c;
03855 }
03856 if ( *p ) {
03857     MesCerr("separator",p);
03858     AC.BracketNormalize = 0;
03859     AT.WorkPointer = WorkSave;
03860     error = 1;
03861     return(error);
03862 }
03863 *to++ = 1; *to++ = 1; *to++ = 3;
03864 *AT.WorkPointer = to - AT.WorkPointer;
03865 AT.WorkPointer = to;
03866 AC.BracketNormalize = 1;
03867 if ( BracketNormalize(BHEAD WorkSave) ) { error = 1; AR.BracketOn = 0; }
03868 else {
03869     w = WorkSave;
03870     if ( *w == 4 || !*w ) { AR.BracketOn = 0; }
03871     else {
03872         i = *(w+*w-1);
03873         if ( i < 0 ) i = -i;
03874         *w -= i;
03875         i = *w;
03876         if ( i > AR.MaxBracket ) {
03877             WORD *newbuf;
03878             newbuf = (WORD *)Malloc1(sizeof(WORD)*(i+1),"bracket buffer");
03879             AR.MaxBracket = i;
03880             if ( AT.BrackBuf != 0 ) M_free(AT.BrackBuf,"bracket buffer");
03881             AT.BrackBuf = newbuf;
03882         }
03883         to = AT.BrackBuf;
03884         NCOPY(to,w,i);
03885     }
03886 }
03887 AC.BracketNormalize = 0;
03888 if ( par == 1 ) AR.BracketOn = -AR.BracketOn;
03889 if ( error == 0 ) {
03890     AC.bracketindexflag = biflag;
03891     AT.bracketindexflag = biflag;
03892 }
03893 AT.WorkPointer = WorkSave;
03894 return(error);
03895 }
03896 /*
03897 #] DoBrackets :
03898 #[ CoBracket :
03899 */
03900 int CoBracket(UBYTE *inp)
03901 { return(DoBrackets(inp,0)); }
03902
03903 /*

```

```

03906     #] CoBracket :
03907     #[ CoAntiBracket :
03908 */
03909
03910 int CoAntiBracket(UBYTE *inp)
03911 { return(DoBrackets(inp,1)); }
03912
03913 /*
03914     #] CoAntiBracket :
03915     #[ CoMultiBracket :
03916
03917     Syntax:
03918         MultiBracket:{A|B} bracketinfo:...:{A|B} bracketinfo;
03919 */
03920
03921 int CoMultiBracket(UBYTE *inp)
03922 {
03923     GETIDENTITY
03924     int i, error = 0, error1, type, num;
03925     UBYTE *s, c;
03926     WORD *to, *from;
03927
03928     if ( *inp != ':' ) {
03929         MesPrint("&Illegal Multiple Bracket separator: %s",inp);
03930         return(1);
03931     }
03932     inp++;
03933     if ( AC.MultiBracketBuf == 0 ) {
03934         AC.MultiBracketBuf = (WORD **)Malloca(sizeof(WORD *)*MAXMULTIBRACKETLEVELS,"multi bracket
buffer");
03935         for ( i = 0; i < MAXMULTIBRACKETLEVELS; i++ ) {
03936             AC.MultiBracketBuf[i] = 0;
03937         }
03938     }
03939     else {
03940         for ( i = 0; i < MAXMULTIBRACKETLEVELS; i++ ) {
03941             if ( AC.MultiBracketBuf[i] ) {
03942                 M_free(AC.MultiBracketBuf[i],"bracket buffer i");
03943                 AC.MultiBracketBuf[i] = 0;
03944             }
03945         }
03946         AC.MultiBracketLevels = 0;
03947     }
03948     AC.MultiBracketLevels = 0;
03949 /*
03950     Start with disabling the regular brackets.
03951 */
03952     if ( AT.BrackBuf == 0 ) {
03953         AR.MaxBracket = 100;
03954         AT.BrackBuf = (WORD *)Malloca(sizeof(WORD)*(AR.MaxBracket+1),"bracket buffer");
03955     }
03956     *AT.BrackBuf = 0;
03957     AR.BraceOn = 0;
03958     AC.bracketindexflag = 0;
03959     AT.bracketindexflag = 0;
03960 /*
03961     Now loop through the various levels, separated by the colons.
03962 */
03963     for ( i = 0; i < MAXMULTIBRACKETLEVELS; i++ ) {
03964         if ( *inp == 0 ) goto RegEnd;
03965 /*
03966         1: skip to ':', determine bracket or antibracket
03967 */
03968         s = inp;
03969         while ( *s && *s != ':' ) {
03970             if ( *s == '[' ) { SKIPBRA1(s) s++; }
03971             else if ( *s == '{' ) { SKIPBRA2(s) s++; }
03972             else s++;
03973         }
03974         c = *s; *s = 0;
03975         if ( StrICont(inp,(UBYTE *)"antibrackets") == 0 ) { type = 1; }
03976         else if ( StrICont(inp,(UBYTE *)"brackets") == 0 ) { type = 0; }
03977         else {
03978             MesPrint("&Illegal (anti)bracket specification in MultiBracket statement");
03979             if ( error == 0 ) error = 1;
03980             goto NextLevel;
03981         }
03982         while ( FG.cTable[*inp] == 0 ) inp++;
03983         if ( *inp != ',' ) {
03984             MesPrint("&Illegal separator after (anti)bracket specification in MultiBracket
statement");
03985             if ( error == 0 ) error = 1;
03986             goto NextLevel;
03987         }
03988         inp++;
03989 /*
03990         2: call DoBrackets.

```

```

03991 */
03992     error1 = DoBrackets(inp, type);
03993     if ( error < 0 ) return(error1);
03994     if ( error1 > error ) error = error1;
03995 /*
03996     3: copy bracket information to the multi bracket arrays
03997 */
03998     if ( AR.BracketOn ) {
03999         num = AT.BrackBuf[0];
04000         to = AC.MultiBracketBuf[i] = (WORD *)Malloc1((num+2)*sizeof(WORD),"bracket buffer i");
04001         from = AT.BrackBuf;
04002         *to++ = AR.BracketOn;
04003         NCOPY(to,from,num);
04004         *to = 0;
04005     }
04006 /*
04007     4: set ready for the next level
04008 */
04009 NextLevel:
04010     *s = c; if ( c == ':' ) s++;
04011     inp = s;
04012     *AT.BrackBuf = 0;
04013     AR.BracketOn = 0;
04014 }
04015 if ( *inp != 0 ) {
04016     MesPrint("&More than %d levels in MultiBracket statement",(WORD)MAXMULTIBRACKETLEVELS);
04017     if ( error == 0 ) error = 1;
04018 }
04019 RegEnd:
04020     AC.MultiBracketLevels = i;
04021     *AT.BrackBuf = 0;
04022     AR.BracketOn = 0;
04023     AC.bracketindexflag = 0;
04024     AT.bracketindexflag = 0;
04025     return(error);
04026 }
04027
04028 /*
04029 #] CoMultiBracket :
04030 #[ CountComp :
04031
04032     This routine reads the count statement. The syntax is:
04033     count minimum,object,size[,object,size]
04034     Objects can be:
04035         symbol
04036         dotproduct
04037         vector
04038         function
04039     Vectors can have the auxiliary flags:
04040         +v +f +d +?setname
04041
04042     Output for the compiler:
04043     TYPECOUNT,size,minimum,objects
04044     with the objects:
04045     SYMBOL,4,number,size
04046     DOTPRODUCT,5,v1,v2,size
04047     FUNCTION,4,number,size
04048     VECTOR,5,number,bits,size or VECTOR,6,number,bits,setnumber,size
04049
04050     Currently only used in the if statement
04051 */
04052
04053 WORD *CountComp(UBYTE *inp, WORD *to)
04054 {
04055     GETIDENTITY
04056     UBYTE *p, c;
04057     WORD *w, mini = 0, type, c1, c2;
04058     int error = 0;
04059     p = inp;
04060     w = to;
04061     AR.Eside = 2;
04062     *w++ = TYPECOUNT;
04063     *w++ = 0;
04064     *w++ = 0;
04065     while ( *p == ',' ) {
04066         p++; inp = p;
04067         if ( *p == '[' || FG.cTable[*p] == 0 ) {
04068             if ( ( p = SkipAName(inp) ) == 0 ) return(0);
04069             c = *p; *p = 0;
04070             type = GetName(AC.varnames,inp,&c1,WITHAUTO);
04071             if ( c == '.' ) {
04072                 if ( type == CVECTOR || type == CDUBIOUS ) {
04073                     *p++ = c;
04074                     inp = p;
04075                     p = SkipAName(p);
04076                     if ( p == 0 ) return(0);
04077                     c = *p;

```

```

04078         *p = 0;
04079         type = GetName(AC.varnames,inp,&c2,WITHAUTO);
04080         if ( type != CVECTOR && type != CDUBIOUS ) {
04081             MesPrint("&Not a vector in dotproduct in if statement: %s",inp);
04082             error = 1;
04083         }
04084         else type = CDOTPRODUCT;
04085     }
04086     else {
04087         MesPrint("&Illegal use of . after %s in if statement",inp);
04088         if ( type == NAMENOTFOUND )
04089             MesPrint("&%s is not a properly declared variable",inp);
04090         error = 1;
04091         *p++ = c;
04092         while ( *p && *p != ')' && *p != ',' ) p++;
04093         if ( *p == ',' && FG.cTable[p[1]] == 1 ) {
04094             p++;
04095             while ( *p && *p != ')' && *p != ',' ) p++;
04096         }
04097         continue;
04098     }
04099 }
04100 *p = c;
04101 switch ( type ) {
04102     case CSYMBOL:
04103         *w++ = SYMBOL; *w++ = 4; *w++ = c1;
04104         if ( *p != ',' ) {
04105             MesCerr("sequence",p);
04106             while ( *p && *p != ')' && *p != ',' ) p++;
04107             error = 1;
04108         }
04109         p++; inp = p;
04110         ParseSignedNumber(mini,p)
04111         if ( FG.cTable[p[-1]] != 1 || ( *p && *p != ')' && *p != ',' ) ) {
04112             while ( *p && *p != ')' && *p != ',' ) p++;
04113             error = 1;
04114             c = *p; *p = 0;
04115             MesPrint("&Improper value in count: %s",inp);
04116             *p = c;
04117             while ( *p && *p != ')' && *p != ',' ) p++;
04118         }
04119         *w++ = mini;
04120         break;
04121     case CFUNCTION:
04122         *w++ = FUNCTION; *w++ = 4; *w++ = c1+FUNCTION; goto Sgetnum;
04123     case CDOTPRODUCT:
04124         *w++ = DOTPRODUCT; *w++ = 5;
04125         *w++ = c2 + AM.OffsetVector;
04126         *w++ = c1 + AM.OffsetVector;
04127         goto Sgetnum;
04128     case CVECTOR:
04129         *w++ = VECTOR; *w++ = 5;
04130         *w++ = c1 + AM.OffsetVector;
04131         if ( *p == ',' ) {
04132             *w++ = VECTBIT | DOTPBIT | FUNBIT;
04133             goto Sgetnum;
04134         }
04135     else if ( *p == '+' ) {
04136         p++;
04137         *w = 0;
04138         while ( *p && *p != ',' ) {
04139             if ( *p == 'v' || *p == 'V' ) {
04140                 *w |= VECTBIT; p++;
04141             }
04142             else if ( *p == 'd' || *p == 'D' ) {
04143                 *w |= DOTPBIT; p++;
04144             }
04145             else if ( *p == 'f' || *p == 'F'
04146                 || *p == 't' || *p == 'T' ) {
04147                 *w |= FUNBIT; p++;
04148             }
04149             else if ( *p == '?' ) {
04150                 p++; inp = p;
04151                 if ( *p == '{' ) { /* } */
04152                     SKIPBRA2(p)
04153                     if ( p == 0 ) return(0);
04154                     if ( ( c1 = DoTempSet(inp+1,p) ) < 0 ) return(0);
04155                     if ( Sets[c1].type != CFUNCTION ) {
04156                         MesPrint("&set type conflict: Function expected");
04157                         return(0);
04158                     }
04159                     type = CSET;
04160                     c = **p;
04161                 }
04162             else {
04163                 p = SkipAName(p);
04164                 if ( p == 0 ) return(0);

```

```

04165             c = *p; *p = 0;
04166             type = GetName(AC.varnames,inp,&c1,WITHAUTO);
04167         }
04168         if ( type != CSET && type != CDUBIOUS ) {
04169             MesPrint("&%%s is not a set",inp);
04170             error = 1;
04171         }
04172         w[-2] = 6;
04173         *w++ |= SETBIT;
04174         *w++ = c1;
04175         *p = c;
04176         goto Sgetnum;
04177     }
04178     else {
04179         MesCerr("specifier for vector",p);
04180         error = 1;
04181     }
04182 }
04183 w++;
04184 goto Sgetnum;
04185 }
04186 else {
04187     MesCerr("specifier for vector",p);
04188     while ( *p && *p != ')' && *p != ',' ) p++;
04189     error = 1;
04190     *w++ = VECTBIT | DOTPBIT | FUNBIT;
04191     goto Sgetnum;
04192 }
04193 case CDUBIOUS:
04194     goto skipfield;
04195 default:
04196     *p = 0;
04197     MesPrint("&%%s is not a symbol, function, vector or dotproduct",inp);
04198     error = 1;
04199 skipfield:
04200     while ( *p && *p != ')' && *p != ',' ) p++;
04201     if ( *p && FG.cTable[p[1]] == 1 ) {
04202         p++;
04203         while ( *p && *p != ')' && *p != ',' ) p++;
04204     }
04205     break;
04206 }
04207 else {
04208     MesCerr("name",p);
04209     while ( *p && *p != ',' ) p++;
04210     error = 1;
04211 }
04212 }
04213 to[1] = w-to;
04214 if ( *p == ')' ) p++;
04215 if ( *p ) { MesCerr("end of statement",p); return(0); }
04216 if ( error ) return(0);
04217 return(w);
04218 }
04219
04220 /*
04221 #] CountComp :
04222 #[ CoIf :
04223
04224     Reads the if statement: There must be a pair of parentheses.
04225     Much work is delegated to the routines in compi2 and CountComp.
04226     The goto is kept hanging as it is forward.
04227     The address in which the label must be written is pushed on
04228     the AC.IfStack.
04229
04230     Here we allow statements of the type
04231     if ( condition ) single statement;
04232     compile the if statement.
04233     test character at end
04234     if not ; or )
04235     copy the statement after the proper parenthesis to the
04236     beginning of the AC.iBuffer.
04237     Have it compiled.
04238     generate an endif statement.
04239 */
04240
04241 static UWORD *CIsratC = 0;
04242
04243 int CoIf(UBYTE *inp)
04244 {
04245     GETIDENTITY
04246     int error = 0, level;
04247     WORD *w, *ww, *u, *s, *OldWork, *OldSpace = AT.WorkSpace;
04248     WORD gotexp = 0; /* Indicates whether there can be a condition */
04249     WORD lenpp, lenlev, ncoef, i, number;
04250     UBYTE *p, *pp, *ppp, c;
04251     CBUF *C = cbuf+AC.cbufnum;

```

```

04252     LONG x;
04253 #ifdef WITHFLOAT
04254     int spec;
04255 #endif
04256     if ( *inp == '(' && inp[1] == ',' ) inp += 2;
04257     else if ( *inp == '(' ) inp++; /* Usually we enter at the bracket */
04258
04259     if ( CIsratC == 0 )
04260         CIsratC = (UWORD *)Malloc1((AM.MaxTal+2)*sizeof(UWORD), "CoIf");
04261     lenpp = 0;
04262     lenlev = 1;
04263     if ( AC.IfLevel >= AC.MaxIf ) DoubleIfBuffers();
04264     AC.IfCount[lenpp++] = 0;
04265 /*
04266     IfStack is used for organizing the 'go to' for the various if levels
04267 */
04268     *AC.IfStack++ = C->Pointer-C->Buffer+2;
04269 /*
04270     IfSumCheck is used to test for illegal nesting of if, argument or repeat.
04271 */
04272     AC.IfSumCheck[AC.IfLevel] = NestingChecksum();
04273     AC.IfLevel++;
04274     w = OldWork = AT.WorkPointer;
04275     *w++ = TYPEIF;
04276     w += 2;
04277     p = inp;
04278     for(;;) {
04279         inp = p;
04280         level = 0;
04281 ReDo:
04282         if ( FG.cTable[*p] == 1 ) { /* Number */
04283             if ( gotexp == 1 ) { MesCerr("position for ",p); error = 1; }
04284 #ifdef WITHFLOAT
04285             pp = CheckFloat(p,&spec);
04286             if ( pp > p ) { /* Got one */
04287 HaveFloat:
04288                 if ( spec == -1 ) {
04289                     MesPrint("&The floating point system has not been started: %s",p);
04290                     if ( !error ) error = 1;
04291                 }
04292                 else {
04293                     WORD *ow = AT.WorkPointer;
04294                     AT.WorkPointer = w;
04295                     c = *pp; *pp = 0;
04296                     ReadFloat((SBYTE *)p); /* Is now at AT.WorkPointer */
04297                     *pp = c;
04298                     p = pp;
04299                     AT.WorkPointer[0] = IFFLOATNUMBER;
04300                     w = AT.WorkPointer + AT.WorkPointer[1];
04301                     AT.WorkPointer = ow;
04302                     if ( level ) w[FUNHEAD+3] = -w[FUNHEAD+3];
04303                 }
04304                 goto DoneWithNumber;
04305             }
04306 /*
04307             Notation: Same as FLOATFUN but FLOATFUN replaced by IFFLOATNUMBER.
04308 */
04309 #endif
04310             u = w;
04311             *w++ = LONGNUMBER;
04312 /*
04313             Notation:
04314             LONGNUMBER,size,reducedsize*sign,numerator,denominator
04315             with the length of denominator and numerator equal to reducedsize
04316 */
04317             w += 2;
04318             if ( GetLong(p, (UWORD *)w,&ncoef) ) { ncoef = 1; error = 1; }
04319             w[-1] = ncoef;
04320             while ( FG.cTable[***p] == 1 );
04321             if ( *p == '/' ) {
04322                 p++;
04323                 if ( FG.cTable[*p] != 1 ) {
04324                     MesCerr("sequence",p); error = 1; goto OnlyNum;
04325                 }
04326                 if ( GetLong(p,CIsratC,&ncoef) ) {
04327                     ncoef = 1; error = 1;
04328                 }
04329                 while ( FG.cTable[***p] == 1 );
04330                 if ( ncoef == 0 ) {
04331                     MesPrint("&Division by zero!");
04332                     error = 1;
04333                 }
04334             }
04335             else {
04336                 if ( w[-1] != 0 ) {
04337                     if ( Simplify(BHEAD (UWORD *)w,(WORD *) (w-1),
04338                         CIsratC,&ncoef) ) error = 1;

```

```

04339         else {
04340             i = w[-1];
04341             if ( i >= ncoef ) {
04342                 i = w[-1];
04343                 w += i;
04344                 i -= ncoef;
04345                 s = (WORD *)CIscratC;
04346                 NCOPY(w,s,ncoef);
04347                 while ( --i >= 0 ) *w++ = 0;
04348             }
04349             else {
04350                 w += i;
04351                 i = ncoef - i;
04352                 while ( --i >= 0 ) *w++ = 0;
04353                 s = (WORD *)CIscratC;
04354                 NCOPY(w,s,ncoef);
04355             }
04356         }
04357     }
04358 }
04359 }
04360 else {
04361 OnlyNum:
04362     w += ncoef;
04363     if ( ncoef > 0 ) {
04364         ncoef--; *w++ = 1;
04365         while ( --ncoef >= 0 ) *w++ = 0;
04366     }
04367 }
04368 u[1] = WORDDIF(w,u);
04369 u[2] = (u[1] - 3)/2;
04370 if ( level ) u[2] = -u[2];
04371 #ifdef WITHFLOAT
04372 DoneWithNumber:
04373 #endif
04374     gotexp = 1;
04375 }
04376 else if ( *p == '+' ) { p++; goto ReDo; }
04377 else if ( *p == '-' ) { level ^= 1; p++; goto ReDo; }
04378 else if ( *p == 'c' || *p == 'C' ) { /* Count or Coefficient */
04379     if ( gotexp == 1 ) { MesCerr("position for ",p); error = 1; }
04380     while ( FG.cTable[***p] == 0 );
04381     c = *p; *p = 0;
04382     if ( !StrICmp(inp, (UBYTE *) "count" ) ) {
04383         *p = c;
04384         if ( c != '(' ) {
04385             MesPrint("&no ( after count");
04386             error = 1;
04387             goto endofif;
04388         }
04389         inp = p;
04390         SKIPBRA4(p);
04391         c = ***p; *p = 0; *inp = ',';
04392         w = CountComp(inp,w);
04393         *p = c; *inp = '(';
04394         if ( w == 0 ) { error = 1; goto endofif; }
04395         gotexp = 1;
04396     }
04397     else if ( ConWord(inp, (UBYTE *) "coefficient" ) && ( p - inp ) > 3 ) {
04398         *w++ = COEFFI;
04399         *w++ = 2;
04400         *p = c;
04401         gotexp = 1;
04402     }
04403     else goto NoGood;
04404     inp = p;
04405 }
04406 else if ( *p == 'm' || *p == 'M' ) { /* match */
04407     if ( gotexp == 1 ) { MesCerr("position for ",p); error = 1; }
04408     while ( !FG.cTable[***p] );
04409     c = *p; *p = 0;
04410     if ( !StrICmp(inp, (UBYTE *) "match" ) ) {
04411         *p = c;
04412         if ( c != '(' ) {
04413             MesPrint("&no ( after match");
04414             error = 1;
04415             goto endofif;
04416         }
04417         p++; inp = p;
04418         SKIPBRA4(p);
04419         *p = '=';
04420     }
04421     /*
04422     Now we can call the reading of the lhs of an id statement.
04423     This has to be modified in the future.
04424     */
04425     AT.WorkSpace = AT.WorkPointer = w;
    ppp = inp;

```

```

04426         while ( FG.cTable[*ppp] == 0 && ppp < p ) ppp++;
04427         if ( *ppp == ',' ) AC.idoption = 0;
04428         else AC.idoption = SUBMULTI;
04429         level = CoIdExpression(inp,TYPEIF);
04430         AT.WorkSpace = OldSpace;
04431         AT.WorkPointer = OldWork;
04432         if ( level != 0 ) {
04433             if ( level < 0 ) { error = -1; goto endofif; }
04434             error = 1;
04435         }
04436         /*
04437         If we pop numlhs we are in good shape
04438         */
04439         s = u = C->lhs[C->numlhs];
04440         while ( u < C->Pointer ) *w++ = *u++;
04441         C->numlhs--; C->Pointer = s;
04442         *p++ = ',';
04443         inp = p;
04444         gotexp = 1;
04445     }
04446     else if ( !StrICmp(inp, (UBYTE *) "multipleof") ) {
04447         if ( gotexp == 1 ) { MesCerr("position for ",p); error = 1; }
04448         *p = c;
04449         if ( c != '(' ) {
04450             MesPrint("&no ( after multipleof");
04451             error = 1; goto endofif;
04452         }
04453         p++;
04454         if ( FG.cTable[*p] != 1 ) {
04455             MesPrint("&multipleof needs a short positive integer argument");
04456             error = 1; goto endofif;
04457         }
04458         ParseNumber(x,p)
04459         if ( *p != ',' || x <= 0 || x > MAXPOSITIVE ) goto Nomulof;
04460         p++;
04461         *w++ = MULTIPLEOF; *w++ = 3; *w++ = (WORD)x;
04462         inp = p;
04463         gotexp = 1;
04464     }
04465     else {
04466         NoGood: MesPrint("&Unrecognized word: %s",inp);
04467         *p = c;
04468         error = 1;
04469         level = 0;
04470         if ( c == '(' ) SKIPBRA4(p)
04471         inp = ++p;
04472         gotexp = 1;
04473     }
04474 }
04475 else if ( *p == 'f' || *p == 'F' ) { /* FindLoop */
04476     if ( gotexp == 1 ) { MesCerr("position for ",p); error = 1; }
04477     while ( FG.cTable[**p] == 0 );
04478     c = *p; *p = 0;
04479     if ( !StrICmp(inp, (UBYTE *) "findloop") ) {
04480         *p = c;
04481         if ( c != '(' ) {
04482             MesPrint("&no ( after findloop");
04483             error = 1;
04484             goto endofif;
04485         }
04486         inp = p;
04487         SKIPBRA4(p);
04488         c = **p; *p = 0; *inp = ',';
04489         if ( CoFindLoop(inp) ) { error = 1; goto endofif; }
04490         s = u = C->lhs[C->numlhs];
04491         while ( u < C->Pointer ) *w++ = *u++;
04492         C->numlhs--; C->Pointer = s;
04493         *p = c; *inp = '(';
04494         if ( w == 0 ) { error = 1; goto endofif; }
04495         gotexp = 1;
04496     }
04497     else if ( !StrICmp(inp, (UBYTE *) "flag") ) {
04498         UBYTE cc = c, *pppp;
04499         *p = cc;
04500         if ( cc != '(' ) {
04501             MesPrint("&no ( after flag");
04502             error = 1;
04503             goto endofif;
04504         }
04505         inp = p;
04506         SKIPBRA4(p);
04507         cc = **p; *p = 0; *inp = ','; pppp = p;
04508         ww = w;
04509         *w++ = IFUSERFLAG; *w++ = 0;
04510         while ( *inp ) {
04511             int x = 0;
04512             while ( *inp == ',' ) inp++;

```



```

04513         if ( *inp == 0 || *inp == ')' ) break;
04514         while ( *inp >= '0' && *inp <= '9' ) x = 10*x+( *inp++-'0' );
04515         if ( x < 1 || x > BITSINWORD ) {
04516             MesPrint("&Flag number %d outside the permitted range 1-%d.",BITSINWORD);
04517             error = 1;
04518         }
04519         *w++ = x-1;
04520     }
04521     ww[1] = w-ww;
04522     p = pppp; *p = cc; *inp = '(';
04523     gotexp = 1;
04524     if ( ww[1] <= 2 ) {
04525         MesPrint("&The userflag condition in the if statement needs arguments.");
04526         error = 1;
04527     }
04528     inp = p;
04529     gotexp = 1;
04530 }
04531 else goto NoGood;
04532 }
04533 else if ( *p == 'e' || *p == 'E' ) { /* Expression */
04534     if ( gotexp == 1 ) { MesCerr("position for ",p); error = 1; }
04535     while ( FG.cTable[ *p ] == 0 );
04536     c = *p; *p = 0;
04537     if ( !StrICmp(inp, (UBYTE *) "expression" ) ) {
04538         *p = c;
04539         if ( c != '(' ) {
04540             MesPrint("&no ( after expression");
04541             error = 1;
04542             goto endofif;
04543         }
04544         p++; ww = w; *w++ = IFEXPRESSION; w++;
04545         while ( *p != ')' ) {
04546             if ( *p == ',' ) { p++; continue; }
04547             if ( *p == '[' || FG.cTable[*p] == 0 ) {
04548                 pp = p;
04549                 if ( ( p = SkipAName(p) ) == 0 ) {
04550                     MesPrint("&Improper name for an expression: '%s'",pp);
04551                     error = 1;
04552                     goto endofif;
04553                 }
04554                 c = *p; *p = 0;
04555                 if ( GetName(AC.exprnames,pp,&number,NOAUTO) == CEXPRESSION ) {
04556                     *w++ = number;
04557                 }
04558                 else if ( GetName(AC.varnames,pp,&number,NOAUTO) != NAMENOTFOUND ) {
04559                     MesPrint("&%s is not an expression",pp);
04560                     error = 1;
04561                     *w++ = number;
04562                 }
04563                 *p = c;
04564             }
04565             else {
04566                 MesPrint("&Illegal object in Expression in if-statement");
04567                 error = 1;
04568                 while ( *p && *p != ',' && *p != ')' ) p++;
04569                 if ( *p == 0 || *p == ')' ) break;
04570             }
04571         }
04572         ww[1] = w - ww;
04573         p++;
04574         gotexp = 1;
04575     }
04576     else goto NoGood;
04577     inp = p;
04578 }
04579 else if ( *p == 'i' || *p == 'I' ) { /* IsFactorized */
04580     if ( gotexp == 1 ) { MesCerr("position for ",p); error = 1; }
04581     while ( FG.cTable[ *p ] == 0 );
04582     c = *p; *p = 0;
04583     if ( !StrICmp(inp, (UBYTE *) "isfactorized" ) ) {
04584         *p = c;
04585         if ( c != '(' ) { /* No expression means current expression */
04586             ww = w; *w++ = IFISFACTORIZED; w++;
04587         }
04588     }
04589     else {
04590         p++; ww = w; *w++ = IFISFACTORIZED; w++;
04591         while ( *p != ')' ) {
04592             if ( *p == ',' ) { p++; continue; }
04593             if ( *p == '[' || FG.cTable[*p] == 0 ) {
04594                 pp = p;
04595                 if ( ( p = SkipAName(p) ) == 0 ) {
04596                     MesPrint("&Improper name for an expression: '%s'",pp);
04597                     error = 1;
04598                     goto endofif;
04599                 }
04600             }
04601             else {
04602                 MesPrint("&Illegal object in Expression in if-statement");
04603                 error = 1;
04604                 while ( *p && *p != ',' && *p != ')' ) p++;
04605                 if ( *p == 0 || *p == ')' ) break;
04606             }
04607         }
04608         ww[1] = w - ww;
04609         p++;
04610         gotexp = 1;
04611     }
04612     else goto NoGood;
04613     inp = p;
04614 }
04615 }
04616 }
04617 }
04618 }
04619 }
04620 }
04621 }
04622 }
04623 }
04624 }
04625 }
04626 }
04627 }
04628 }
04629 }
04630 }
04631 }
04632 }
04633 }
04634 }
04635 }
04636 }
04637 }
04638 }
04639 }
04640 }
04641 }
04642 }
04643 }
04644 }
04645 }
04646 }
04647 }
04648 }
04649 }
04650 }
04651 }
04652 }
04653 }
04654 }
04655 }
04656 }
04657 }
04658 }
04659 }
04660 }
04661 }
04662 }
04663 }
04664 }
04665 }
04666 }
04667 }
04668 }
04669 }
04670 }
04671 }
04672 }
04673 }
04674 }
04675 }
04676 }
04677 }
04678 }
04679 }
04680 }
04681 }
04682 }
04683 }
04684 }
04685 }
04686 }
04687 }
04688 }
04689 }
04690 }
04691 }
04692 }
04693 }
04694 }
04695 }
04696 }
04697 }
04698 }
04699 }
04700 }
04701 }
04702 }
04703 }
04704 }
04705 }
04706 }
04707 }
04708 }
04709 }
04710 }
04711 }
04712 }
04713 }
04714 }
04715 }
04716 }
04717 }
04718 }
04719 }
04720 }
04721 }
04722 }
04723 }
04724 }
04725 }
04726 }
04727 }
04728 }
04729 }
04730 }
04731 }
04732 }
04733 }
04734 }
04735 }
04736 }
04737 }
04738 }
04739 }
04740 }
04741 }
04742 }
04743 }
04744 }
04745 }
04746 }
04747 }
04748 }
04749 }
04750 }
04751 }
04752 }
04753 }
04754 }
04755 }
04756 }
04757 }
04758 }
04759 }
04760 }
04761 }
04762 }
04763 }
04764 }
04765 }
04766 }
04767 }
04768 }
04769 }
04770 }
04771 }
04772 }
04773 }
04774 }
04775 }
04776 }
04777 }
04778 }
04779 }
04780 }
04781 }
04782 }
04783 }
04784 }
04785 }
04786 }
04787 }
04788 }
04789 }
04790 }
04791 }
04792 }
04793 }
04794 }
04795 }
04796 }
04797 }
04798 }
04799 }
04800 }
04801 }
04802 }
04803 }
04804 }
04805 }
04806 }
04807 }
04808 }
04809 }
04810 }
04811 }
04812 }
04813 }
04814 }
04815 }
04816 }
04817 }
04818 }
04819 }
04820 }
04821 }
04822 }
04823 }
04824 }
04825 }
04826 }
04827 }
04828 }
04829 }
04830 }
04831 }
04832 }
04833 }
04834 }
04835 }
04836 }
04837 }
04838 }
04839 }
04840 }
04841 }
04842 }
04843 }
04844 }
04845 }
04846 }
04847 }
04848 }
04849 }
04850 }
04851 }
04852 }
04853 }
04854 }
04855 }
04856 }
04857 }
04858 }
04859 }
04860 }
04861 }
04862 }
04863 }
04864 }
04865 }
04866 }
04867 }
04868 }
04869 }
04870 }
04871 }
04872 }
04873 }
04874 }
04875 }
04876 }
04877 }
04878 }
04879 }
04880 }
04881 }
04882 }
04883 }
04884 }
04885 }
04886 }
04887 }
04888 }
04889 }
04890 }
04891 }
04892 }
04893 }
04894 }
04895 }
04896 }
04897 }
04898 }
04899 }
04900 }
04901 }
04902 }
04903 }
04904 }
04905 }
04906 }
04907 }
04908 }
04909 }
04910 }
04911 }
04912 }
04913 }
04914 }
04915 }
04916 }
04917 }
04918 }
04919 }
04920 }
04921 }
04922 }
04923 }
04924 }
04925 }
04926 }
04927 }
04928 }
04929 }
04930 }
04931 }
04932 }
04933 }
04934 }
04935 }
04936 }
04937 }
04938 }
04939 }
04940 }
04941 }
04942 }
04943 }
04944 }
04945 }
04946 }
04947 }
04948 }
04949 }
04950 }
04951 }
04952 }
04953 }
04954 }
04955 }
04956 }
04957 }
04958 }
04959 }
04960 }
04961 }
04962 }
04963 }
04964 }
04965 }
04966 }
04967 }
04968 }
04969 }
04970 }
04971 }
04972 }
04973 }
04974 }
04975 }
04976 }
04977 }
04978 }
04979 }
04980 }
04981 }
04982 }
04983 }
04984 }
04985 }
04986 }
04987 }
04988 }
04989 }
04990 }
04991 }
04992 }
04993 }
04994 }
04995 }
04996 }
04997 }
04998 }
04999 }
05000 }

```

```

04600         if ( GetName(AC.exprnames,pp,&number,NOAUTO) == CEXPRESSION ) {
04601             *w++ = number;
04602         }
04603         else if ( GetName(AC.varnames,pp,&number,NOAUTO) != NAMENOTFOUND ) {
04604             MesPrint("&#s is not an expression",pp);
04605             error = 1;
04606             *w++ = number;
04607         }
04608         *p = c;
04609     }
04610     else {
04611         MesPrint("&#Illegal object in IsFactorized in if-statement");
04612         error = 1;
04613         while ( *p && *p != ',' && *p != ')' ) p++;
04614         if ( *p == 0 || *p == ')' ) break;
04615     }
04616 }
04617 p++;
04618 }
04619 ww[1] = w - ww;
04620 gotexp = 1;
04621 }
04622 else goto NoGood;
04623 inp = p;
04624 }
04625 else if ( *p == 'o' || *p == 'O' ) { /* Occurs */
04626 /*
04627     Tests whether variables occur inside a term.
04628     At the moment this is done one by one.
04629     If we want to do them in groups we should do the reading
04630     a bit different: each as a variable in a term, and then
04631     use Normalize to get the variables grouped and in order.
04632     That way FindVar (in if.c) can work more efficiently.
04633     Still to be done!!!
04634     TASK: Nice little task for someone to learn.
04635 */
04636     UBYTE cc;
04637     if ( gotexp == 1 ) { MesCerr("position for ",p); error = 1; }
04638     while ( FG.cTable[++p] == 0 );
04639     c = cc = *p; *p = 0;
04640     if ( !StrICmp(inp,(UBYTE *)"occurs") ) {
04641         WORD c1, c2, type;
04642         *p = cc;
04643         if ( cc != '(' ) {
04644             MesPrint("&#no ( after occurs");
04645             error = 1;
04646             goto endofif;
04647         }
04648         inp = p;
04649         SKIPBRA4(p);
04650         cc = ++p; *p = 0; *inp = ','; pp = p;
04651         ww = w;
04652         *w++ = IFOCCURS; *w++ = 0;
04653         while ( *inp ) {
04654             while ( *inp == ',' ) inp++;
04655             if ( *inp == 0 || *inp == ')' ) break;
04656 /*
04657             Now read a list of names
04658             We can have symbols, vectors, dotproducts, indices, functions.
04659             There could also be dummy indices and/or extra symbols.
04660 */
04661             if ( *inp == '[' || FG.cTable[*inp] == 0 ) {
04662                 if ( ( p = SkipAName(inp) ) == 0 ) return(0);
04663                 c = *p; *p = 0;
04664                 type = GetName(AC.varnames,inp,&c1,WITHAUTO);
04665                 if ( c == '.' ) {
04666                     if ( type == CVECTOR || type == CDUBIOUS ) {
04667                         *p++ = c;
04668                         inp = p;
04669                         p = SkipAName(p);
04670                         if ( p == 0 ) return(0);
04671                         c = *p;
04672                         *p = 0;
04673                         type = GetName(AC.varnames,inp,&c2,WITHAUTO);
04674                         if ( type != CVECTOR && type != CDUBIOUS ) {
04675                             MesPrint("&#Not a vector in dotproduct in if statement: %s",inp);
04676                             error = 1;
04677                         }
04678                         else type = CDOTPRODUCT;
04679                     }
04680                     else {
04681                         MesPrint("&#Illegal use of . after %s in if statement",inp);
04682                         if ( type == NAMENOTFOUND )
04683                             MesPrint("&#%s is not a properly declared variable",inp);
04684                         error = 1;
04685                         *p++ = c;
04686                         while ( *p && *p != ')' && *p != ',' ) p++;

```

```

04687             if ( *p == ',' && FG.cTable[p[1]] == 1 ) {
04688                 p++;
04689                 while ( *p && *p != ')' && *p != ',' ) p++;
04690             }
04691             continue;
04692         }
04693     }
04694     *p = c;
04695     switch ( type ) {
04696     case CSYMBOL: /* To worry about extra symbols */
04697         *w++ = SYMBOL;
04698         *w++ = c1;
04699         break;
04700     case CINDEX:
04701         *w++ = INDEX;
04702         *w++ = c1 + AM.OffsetIndex;
04703         break;
04704     case CVECTOR:
04705         *w++ = VECTOR;
04706         *w++ = c1 + AM.OffsetVector;
04707         break;
04708     case CDOTPRODUCT:
04709         *w++ = DOTPRODUCT;
04710         *w++ = c1 + AM.OffsetVector;
04711         *w++ = c2 + AM.OffsetVector;
04712         break;
04713     case CFUNCTION:
04714         *w++ = FUNCTION;
04715         *w++ = c1 + FUNCTION;
04716         break;
04717     default:
04718         MesPrint("&Illegal variable %s in occurs condition in if
statement",inp);
04719         error = 1;
04720         break;
04721     }
04722     inp = p;
04723 }
04724 else {
04725     MesPrint("&Illegal object %s in occurs condition in if statement",inp);
04726     error = 1;
04727     break;
04728 }
04729 }
04730 ww[1] = w-ww;
04731 p = pp; *p = cc; *inp = '(';
04732 gotexp = 1;
04733 if ( ww[1] <= 2 ) {
04734     MesPrint("&The occurs condition in the if statement needs arguments.");
04735     error = 1;
04736 }
04737 }
04738 else goto NoGood;
04739 inp = p;
04740 }
04741 else if ( *p == '$' ) {
04742     if ( gotexp == 1 ) { MesCerr("position for ",p); error = 1; }
04743     p++; inp = p;
04744     while ( FG.cTable[*p] == 0 || FG.cTable[*p] == 1 ) p++;
04745     c = *p; *p = 0;
04746     if ( ( i = GetDollar(inp) ) < 0 ) {
04747         MesPrint("&undefined dollar expression %s",inp);
04748         error = 1;
04749         i = AddDollar(inp,DOLUNDEFINED,0,0);
04750     }
04751     *p = c;
04752     *w++ = IFDOLLAR; *w++ = 3; *w++ = i;
04753 /*
04754     And then the IFDOLLAREXTRA pieces for [1] [$y] etc
04755 */
04756     if ( *p == '[' ) {
04757         p++;
04758         if ( ( w = GetIfDollarFactor(&p,w) ) == 0 ) {
04759             error = 1;
04760             goto endofif;
04761         }
04762         else if ( *p != ']' ) {
04763             error = 1;
04764             goto endofif;
04765         }
04766         p++;
04767     }
04768     inp = p;
04769     gotexp = 1;
04770 }
04771 #ifdef WITHFLOAT
04772     else if ( *p == '.' ) {

```

```

04773         pp = CheckFloat(p,&spec);
04774         if ( pp > p ) goto HaveFloat;
04775     }
04776 #endif
04777     else if ( *p == '(' ) {
04778         if ( gotexp ) {
04779             MesCerr("parenthesis",p);
04780             error = 1;
04781             goto endofif;
04782         }
04783         gotexp = 0;
04784         if ( ++lenlev >= AC.MaxIf ) DoubleIfBuffers();
04785         AC.IfCount[lenpp++] = w-OldWork;
04786         *w++ = SUBEXPR;
04787         w += 2;
04788         p++;
04789     }
04790     else if ( *p == ')' ) {
04791         if ( gotexp == 0 ) { MesCerr("position for )",p); error = 1; }
04792         gotexp = 1;
04793         u = AC.IfCount[--lenpp]+OldWork;
04794         lenlev--;
04795         u[1] = w - u;
04796         if ( lenlev <= 0 ) { /* End if condition */
04797             AT.WorkSpace = OldSpace;
04798             AT.WorkPointer = OldWork;
04799             AddNtol(OldWork[1],OldWork);
04800             p++;
04801             if ( *p == ')' ) {
04802                 MesPrint("&unmatched parenthesis in if/while ()");
04803                 error = 1;
04804                 while ( *++p == ')' );
04805             }
04806             if ( *p ) {
04807                 level = CompileStatement(p);
04808                 if ( level ) error = level;
04809                 while ( *p ) p++;
04810                 if ( CoEndIf(p) && error == 0 ) error = 1;
04811             }
04812             return(error);
04813         }
04814         p++;
04815     }
04816     else if ( *p == '>' ) {
04817         if ( gotexp == 0 ) goto NoExp;
04818         if ( p[1] == '=' ) { *w++ = GREATEREQUAL; *w++ = 2; p += 2; }
04819         else { *w++ = GREATER; *w++ = 2; p++; }
04820         gotexp = 0;
04821     }
04822     else if ( *p == '<' ) {
04823         if ( gotexp == 0 ) goto NoExp;
04824         if ( p[1] == '=' ) { *w++ = LESSEQUAL; *w++ = 2; p += 2; }
04825         else { *w++ = LESS; *w++ = 2; p++; }
04826         gotexp = 0;
04827     }
04828     else if ( *p == '=' ) {
04829         if ( gotexp == 0 ) goto NoExp;
04830         if ( p[1] == '=' ) p++;
04831         *w++ = EQUAL; *w++ = 2; p++;
04832         gotexp = 0;
04833     }
04834     else if ( *p == '!' && p[1] == '=' ) {
04835         if ( gotexp == 0 ) { p++; goto NoExp; }
04836         *w++ = NOTEQUAL; *w++ = 2; p += 2;
04837         gotexp = 0;
04838     }
04839     else if ( *p == '|' && p[1] == '|' ) {
04840         if ( gotexp == 0 ) { p++; goto NoExp; }
04841         *w++ = ORCOND; *w++ = 2; p += 2;
04842         gotexp = 0;
04843     }
04844     else if ( *p == '&' && p[1] == '&' ) {
04845         if ( gotexp == 0 ) {
04846             p++;
04847             p++;
04848             MesCerr("sequence",p);
04849             error = 1;
04850         }
04851         else {
04852             *w++ = ANDCOND; *w++ = 2; p += 2;
04853             gotexp = 0;
04854         }
04855     }
04856     else if ( *p == 0 ) {
04857         MesPrint("&Unmatched parentheses");
04858         error = 1;
04859         goto endofif;

```

```

04860     }
04861     else {
04862         if ( FG.cTable[*p] == 0 ) {
04863             WORD ij;
04864             inp = p;
04865             while ( ( ij = FG.cTable[**p] ) == 0 || ij == 1 );
04866             c = *p; *p = 0;
04867             goto NoGood;
04868         }
04869         MesCerr("sequence",p);
04870         error = 1;
04871         p++;
04872     }
04873 }
04874 endifif;;
04875 return(error);
04876 }
04877
04878 /*
04879 #] CoIf :
04880 #[ CoElse :
04881 */
04882
04883 int CoElse(UBYTE *p)
04884 {
04885     int error = 0;
04886     CBUF *C = cbuf+AC.cbunum;
04887     if ( *p != 0 ) {
04888         while ( *p == ',' ) p++;
04889         if ( tolower(*p) == 'i' && tolower(p[1]) == 'f' && p[2] == '(' )
04890             return(CoElseIf(p+2));
04891         MesPrint("&No extra text allowed as part of an else statement");
04892         error = 1;
04893     }
04894     if ( AC.IfLevel <= 0 ) { MesPrint("&else statement without if"); return(1); }
04895     if ( AC.IfSumCheck[AC.IfLevel-1] != NestingChecksum() - 1 ) {
04896         MesNesting();
04897         error = 1;
04898     }
04899     Add3Com(TYPEELSE,AC.IfLevel)
04900     C->Buffer[AC.IfStack[-1]] = C->numlhs;
04901     AC.IfStack[-1] = C->Pointer - C->Buffer - 1;
04902     return(error);
04903 }
04904
04905 /*
04906 #] CoElse :
04907 #[ CoElseIf :
04908 */
04909
04910 int CoElseIf(UBYTE *inp)
04911 {
04912     CBUF *C = cbuf+AC.cbunum;
04913     if ( AC.IfLevel <= 0 ) { MesPrint("&elseif statement without if"); return(1); }
04914     Add3Com(TYPEELSE,-AC.IfLevel)
04915     AC.IfLevel--;
04916     C->Buffer[*--AC.IfStack] = C->numlhs;
04917     return(CoIf(inp));
04918 }
04919
04920 /*
04921 #] CoElseIf :
04922 #[ CoEndIf :
04923
04924     It puts a RHS-level at the position indicated in the AC.IfStack.
04925     This corresponds to the label belonging to a forward goto.
04926     It is the goto that belongs either to the failing condition
04927     of the if (no else statement), or the completion of the
04928     success path (with else statement)
04929     The code is a jump to the next statement. It is there to prevent
04930     problems with
04931     if ( .. )
04932         if ( .. )
04933         endif;
04934     elseif ( .. )
04935 */
04936
04937 int CoEndIf(UBYTE *inp)
04938 {
04939     CBUF *C = cbuf+AC.cbunum;
04940     WORD i = C->numlhs, to, k = -AC.IfLevel;
04941     int error = 0;
04942     while ( *inp == ',' ) inp++;
04943     if ( *inp != 0 ) {
04944         error = 1;
04945         MesPrint("&No extra text allowed as part of an endif/elseif statement");
04946     }

```

```

04947     if ( AC.IfLevel <= 0 ) {
04948         MesPrint("&Endif statement without corresponding if"); return(1);
04949     }
04950     AC.IfLevel--;
04951     C->Buffer[*--AC.IfStack] = i+1;
04952     if ( AC.IfSumCheck[AC.IfLevel] != NestingChecksum() ) {
04953         MesNesting();
04954         error = 1;
04955     }
04956     Add3Com(TYPEENDIF,i+1)
04957 /*
04958     Now the search for the TYPEELSE in front of the elseif statements
04959 */
04960     to = C->numlhs;
04961     while ( i > 0 ) {
04962         if ( C->lhs[i][0] == TYPEELSE && C->lhs[i][2] == to ) to = i;
04963         if ( C->lhs[i][0] == TYPEIF ) {
04964             if ( C->lhs[i][2] == to ) {
04965                 i--;
04966                 if ( i <= 0 || C->lhs[i][0] != TYPEELSE
04967                     || C->lhs[i][2] != k ) break;
04968                 C->lhs[i][2] = C->numlhs;
04969                 to = i;
04970             }
04971         }
04972         i--;
04973     }
04974     return(error);
04975 }
04976
04977 /*
04978 #] CoEndIf :
04979 #[ CoWhile :
04980 */
04981 int CoWhile(UBYTE *inp)
04982 {
04983     CBUF *C = cbuf+AC.cbunum;
04984     WORD startnum = C->numlhs + 1;
04985     int error;
04986     AC.WhileLevel++;
04987     error = CoIf(inp);
04988     if ( C->numlhs > startnum && C->lhs[startnum][2] == C->numlhs
04989         && C->lhs[C->numlhs][0] == TYPEENDIF ) {
04990         C->lhs[C->numlhs][2] = startnum-1;
04991         AC.WhileLevel--;
04992     }
04993     else C->lhs[startnum][2] = startnum;
04994     return(error);
04995 }
04996
04997 /*
04998 #] CoWhile :
04999 #[ CoEndWhile :
05000 */
05001
05002 int CoEndWhile(UBYTE *inp)
05003 {
05004     int error = 0;
05005     WORD i;
05006     CBUF *C = cbuf+AC.cbunum;
05007     if ( AC.WhileLevel <= 0 ) {
05008         MesPrint("&EndWhile statement without corresponding While"); return(1);
05009     }
05010     AC.WhileLevel--;
05011     i = C->Buffer[AC.IfStack[-1]];
05012     error = CoEndIf(inp);
05013     C->lhs[C->numlhs][2] = i - 1;
05014     return(error);
05015 }
05016
05017 /*
05018 #] CoEndWhile :
05019 #[ DoFindLoop :
05020
05021     Function,arguments=number,loopsize=number,outfun=function,include=index;
05022 */
05023
05024 static char *messfind[] = {
05025     "Findloop(function,arguments=#,loopsize(=#|<#)[,include=index])"
05026     ,"Replaceloop,function,arguments=#,loopsize(=#|<#),outfun=function[,include=index]"
05027 };
05028
05029 static WORD comfindloop[7] = { TYPEFINDLOOP,7,0,0,0,0,0 };
05030
05031 int DoFindLoop(UBYTE *inp, int mode)
05032 {
05033     UBYTE *s, c;

```

```

05034     WORD funnum, nargs = 0, nloop = 0, indexnum = 0, outfun = 0;
05035     int type, aflag, lflag, indflag, outflag, error = 0, sym;
05036     while ( *inp == ',' ) inp++;
05037     if ( ( s = SkipAName(inp) ) == 0 ) {
05038 syntax:
05039         MesPrint("&Proper syntax is:");
05040         MesPrint("%s",messfind[mode]);
05041         return(1);
05042     }
05043     c = *s; *s = 0;
05044     if ( ( ( type = GetName(AC.varnames,inp,&funnum,WITHAUTO) ) == NAMENOTFOUND )
05045         || type != CFUNCTION || ( ( sym = (functions[funnum].symmetric) & ~REVERSEORDER )
05046         != SYMMETRIC && sym != ANTISYMMETRIC ) ) {
05047         MesPrint("&%s should be a (anti)symmetric function or tensor",inp);
05048         error = 1;
05049     }
05050     funnum += FUNCTION;
05051     *s = c; inp = s;
05052     aflag = lflag = indflag = outflag = 0;
05053     while ( *inp == ',' ) {
05054         while ( *inp == ',' ) inp++;
05055         s = inp;
05056         if ( ( s = SkipAName(inp) ) == 0 ) goto syntax;
05057         c = *s; *s = 0;
05058         if ( StrICont(inp,(UBYTE *)"arguments") == 0 ) {
05059             if ( c != '=' ) goto syntax;
05060             *s++ = c;
05061             NeedNumber(nargs,s,syntax)
05062             aflag++;
05063             inp = s;
05064         }
05065         else if ( StrICont(inp,(UBYTE *)"loopsize") == 0 ) {
05066             if ( c != '=' && c != '<' ) goto syntax;
05067             *s++ = c;
05068             if ( FG.cTable[*s] == 1 ) {
05069                 NeedNumber(nloop,s,syntax)
05070                 if ( nloop < 2 ) {
05071                     MesPrint("&loopsize should be at least 2");
05072                     error = 1;
05073                 }
05074                 if ( c == '<' ) nloop = -nloop;
05075             }
05076             else if ( tolower(*s) == 'a' && tolower(s[1]) == 'l'
05077                 && tolower(s[2]) == 'l' && FG.cTable[s[3]] > 1 ) {
05078                 nloop = -1; s += 3;
05079                 if ( c != '=' ) goto syntax;
05080             }
05081             inp = s;
05082             lflag++;
05083         }
05084         else if ( StrICont(inp,(UBYTE *)"include") == 0 ) {
05085             if ( c != '=' ) goto syntax;
05086             *s++ = c;
05087             if ( ( inp = SkipAName(s) ) == 0 ) goto syntax;
05088             c = *inp; *inp = 0;
05089             if ( ( type = GetName(AC.varnames,s,&indexnum,WITHAUTO) ) != CINDEX ) {
05090                 MesPrint("&%s is not a proper index",s);
05091                 error = 1;
05092             }
05093             else if ( indexnum < WILDOFFSET
05094                 && indices[indexnum].dimension == 0 ) {
05095                 MesPrint("&%s should be a summable index",s);
05096                 error = 1;
05097             }
05098             indexnum += AM.OffsetIndex;
05099             *inp = c;
05100             indflag++;
05101         }
05102         else if ( StrICont(inp,(UBYTE *)"outfun") == 0 ) {
05103             if ( c != '=' ) goto syntax;
05104             *s++ = c;
05105             if ( ( inp = SkipAName(s) ) == 0 ) goto syntax;
05106             c = *inp; *inp = 0;
05107             if ( ( type = GetName(AC.varnames,s,&outfun,WITHAUTO) ) != CFUNCTION ) {
05108                 MesPrint("&%s is not a proper function or tensor",s);
05109                 error = 1;
05110             }
05111             outfun += FUNCTION;
05112             outflag++;
05113             *inp = c;
05114         }
05115         else {
05116             MesPrint("&Unrecognized option in FindLoop or ReplaceLoop: %s",inp);
05117             error = 1;
05118             *s = c; inp = s;
05119             while ( *inp && *inp != ',' ) inp++;
05120         }

```

```

05121     }
05122     if ( *inp != 0 && mode == REPLACELOOP ) goto syntax;
05123     if ( mode == FINDLOOP && outflag > 0 ) {
05124         MesPrint("&outflag option is illegal in FindLoop");
05125         error = 1;
05126     }
05127     if ( mode == REPLACELOOP && outflag == 0 ) goto syntax;
05128     if ( aflag == 0 || lflag == 0 ) goto syntax;
05129     comfindloop[3] = funnum;
05130     comfindloop[4] = nloop;
05131     comfindloop[5] = nargs;
05132     comfindloop[6] = outfun;
05133     comfindloop[1] = 7;
05134     if ( indflag ) {
05135         if ( mode == 0 ) comfindloop[2] = indexnum + 5;
05136         else comfindloop[2] = -indexnum - 5;
05137     }
05138     else comfindloop[2] = mode;
05139     AddNtoL(comfindloop[1],comfindloop);
05140     return(error);
05141 }
05142
05143 /*
05144 #] DoFindLoop :
05145 #[ CoFindLoop :
05146 */
05147
05148 int CoFindLoop(UBYTE *inp)
05149 { return(DoFindLoop(inp,FINDLOOP)); }
05150
05151 /*
05152 #] CoFindLoop :
05153 #[ CoReplaceLoop :
05154 */
05155
05156 int CoReplaceLoop(UBYTE *inp)
05157 {
05158     int error = DoFindLoop(inp,REPLACELOOP);
05159     if ( error ) {
05160         Terminate(-1);
05161     }
05162     return(error);
05163 }
05164
05165 /*
05166 #] CoReplaceLoop :
05167 #[ CoFunPowers :
05168 */
05169
05170 static UBYTE *FunPowOptions[] = {
05171     (UBYTE *) "nofunpowers"
05172     , (UBYTE *) "commutingonly"
05173     , (UBYTE *) "allfunpowers"
05174 };
05175
05176 int CoFunPowers(UBYTE *inp)
05177 {
05178     UBYTE *option, c;
05179     int i, maxoptions = sizeof(FunPowOptions)/sizeof(UBYTE *);
05180     while ( *inp == ',' ) inp++;
05181     option = inp;
05182     inp = SkipAName(inp); c = *inp; *inp = 0;
05183     for ( i = 0; i < maxoptions; i++ ) {
05184         if ( StrICont(option,FunPowOptions[i]) == 0 ) {
05185             if ( c ) {
05186                 *inp = c;
05187                 MesPrint("&Illegal FunPowers statement");
05188                 return(1);
05189             }
05190             AC.funpowers = i;
05191             return(0);
05192         }
05193     }
05194     MesPrint("&Illegal option in FunPowers statement: %s",option);
05195     return(1);
05196 }
05197
05198 /*
05199 #] CoFunPowers :
05200 #[ CoUnitTrace :
05201 */
05202
05203 int CoUnitTrace(UBYTE *s)
05204 {
05205     WORD num;
05206     if ( FG.cTable[*s] == 1 ) {
05207         ParseNumber(num,s)

```



```

05208         if ( *s != 0 ) {
05209 nogood:      MesPrint("&Value of UnitTrace should be a (positive) number or a symbol");
05210               return(1);
05211         }
05212         AC.lUnitTrace[0] = SNUMBER;
05213         AC.lUnitTrace[2] = num;
05214     }
05215     else {
05216         if ( GetName(AC.varnames,s,&num,WITHAUTO) == CSYMBOL ) {
05217             AC.lUnitTrace[0] = SYMBOL;
05218             AC.lUnitTrace[2] = num;
05219             num = -num;
05220         }
05221         else goto nogood;
05222         s = SkipAName(s);
05223         if ( *s ) goto nogood;
05224     }
05225     AC.lUnitTrace = num;
05226     return(0);
05227 }
05228
05229 /*
05230 #] CoUnitTrace :
05231 #[ CoTerm :
05232
05233 Note: termstack holds the offset of the term statement in the compiler
05234 buffer. termsortstack holds the offset of the last sort statement
05235      (or the corresponding term statement)
05236 */
05237
05238 int CoTerm(UBYTE *s)
05239 {
05240     GETIDENTITY
05241     WORD *w = AT.WorkPointer;
05242     int error = 0;
05243     while ( *s == ',' ) s++;
05244     if ( *s ) {
05245         MesPrint("&Illegal syntax for Term statement");
05246         return(1);
05247     }
05248     if ( AC.termlevel+1 >= AC.maxtermlevel ) {
05249         if ( AC.maxtermlevel <= 0 ) {
05250             AC.maxtermlevel = 20;
05251             AC.termstack = (LONG *)Malloc1(AC.maxtermlevel*sizeof(LONG),"termstack");
05252             AC.termstack = (LONG *)Malloc1(AC.maxtermlevel*sizeof(LONG),"termsortstack");
05253             AC.termsumcheck = (WORD *)Malloc1(AC.maxtermlevel*sizeof(WORD),"termsumcheck");
05254         }
05255         else {
05256             DoubleBuffer((void **)AC.termstack,(void **)AC.termstack+AC.maxtermlevel,
05257                 sizeof(LONG),"doubling termstack");
05258             DoubleBuffer((void **)AC.termstack,(void **)AC.termstack+AC.maxtermlevel,
05259                 sizeof(LONG),"doubling termsortstack");
05260             DoubleBuffer((void **)AC.termstack,(void **)AC.termstack+AC.maxtermlevel,
05261                 sizeof(WORD),"doubling termsumcheck");
05262             AC.maxtermlevel *= 2;
05263         }
05264     }
05265     AC.termsumcheck[AC.termlevel] = NestingChecksum();
05266     AC.termstack[AC.termlevel] = cbuf[AC.cbunum].Pointer
05267         - cbuf[AC.cbunum].Buffer + 2;
05268     AC.termstack[AC.termlevel] = AC.termstack[AC.termlevel] + 1;
05269     AC.termlevel++;
05270     *w++ = TYPETERM;
05271     w++;
05272     *w++ = cbuf[AC.cbunum].numlhs;
05273     *w++ = cbuf[AC.cbunum].numlhs;
05274     AT.WorkPointer[1] = w - AT.WorkPointer;
05275     AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
05276     return(error);
05277 }
05278
05279 /*
05280 #] CoTerm :
05281 #[ CoEndTerm :
05282
05283 */
05284
05285 int CoEndTerm(UBYTE *s)
05286 {
05287     CBUF *C = cbuf+AC.cbunum;
05288     while ( *s == ',' ) s++;
05289     if ( *s ) {
05290         MesPrint("&Illegal syntax for EndTerm statement");
05291         return(1);
05292     }
05293     if ( AC.termlevel <= 0 ) {

```

```

05295         MesPrint("&EndTerm without corresponding Argument statement");
05296         return(1);
05297     }
05298     AC.termlevel--;
05299     cbuf[AC.cbunum].Buffer[AC.termstack[AC.termlevel]] = C->numlhs;
05300     cbuf[AC.cbunum].Buffer[AC.termstack[AC.termlevel]] = C->numlhs;
05301     if ( AC.termsumcheck[AC.termlevel] != NestingChecksum() ) {
05302         MesNesting();
05303         return(1);
05304     }
05305     return(0);
05306 }
05307
05308 /*
05309 #] CoEndTerm :
05310 #[ CoSort :
05311 */
05312
05313 int CoSort(UBYTE *s)
05314 {
05315     GETIDENTITY
05316     WORD *w = AT.WorkPointer;
05317     int error = 0;
05318     while ( *s == ',' ) s++;
05319     if ( *s ) {
05320         MesPrint("&Illegal syntax for Sort statement");
05321         error = 1;
05322     }
05323     if ( AC.termlevel <= 0 ) {
05324         MesPrint("&The Sort statement can only be used inside a term environment");
05325         error = 1;
05326     }
05327     if ( error ) return(error);
05328     *w++ = TYPESORT;
05329     w++;
05330     w++;
05331     cbuf[AC.cbunum].Buffer[AC.termstack[AC.termlevel-1]] =
05332         *w = cbuf[AC.cbunum].numlhs+1;
05333     w++;
05334     AC.termstack[AC.termlevel-1] = cbuf[AC.cbunum].Pointer
05335         - cbuf[AC.cbunum].Buffer + 3;
05336     if ( AC.termsumcheck[AC.termlevel-1] != NestingChecksum() - 1 ) {
05337         MesNesting();
05338         return(1);
05339     }
05340     AT.WorkPointer[1] = w - AT.WorkPointer;
05341     AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
05342     return(error);
05343 }
05344
05345 /*
05346 #] CoSort :
05347 #[ CoPolyFun :
05348
05349 Collect,functionname
05350 */
05351
05352 int CoPolyFun(UBYTE *s)
05353 {
05354     GETIDENTITY
05355     WORD numfun;
05356     int type, error = 0;
05357     UBYTE *t;
05358     AR.PolyFun = AC.lPolyFun = 0;
05359     AR.PolyFunInv = AC.lPolyFunInv = 0;
05360     AR.PolyFunType = AC.lPolyFunType = 0;
05361     AR.PolyFunExp = AC.lPolyFunExp = 0;
05362     AR.PolyFunVar = AC.lPolyFunVar = 0;
05363     AR.PolyFunPow = AC.lPolyFunPow = 0;
05364     if ( *s == 0 ) { return(0); }
05365     t = SkipAName(s);
05366     if ( t == 0 || *t != 0 ) {
05367         MesPrint("&PolyFun statement needs a single commuting function for its argument");
05368         return(1);
05369     }
05370     if ( ( ( type = GetName(AC.varnames,s,&numfun,WITHAUTO) ) != CFUNCTION )
05371         || ( functions[numfun].spec != 0 ) || ( functions[numfun].commute != 0 ) ) {
05372         MesPrint("&%s should be a regular commuting function",s);
05373         if ( type < 0 ) {
05374             if ( GetName(AC.exprnames,s,&numfun,NOAUTO) == NAMENOTFOUND )
05375                 AddFunction(s,0,0,0,0,0,-1,-1);
05376         }
05377         error = 1;
05378     }
05379     else {
05380         AR.PolyFun = AC.lPolyFun = numfun+FUNCTION;
05381         AR.PolyFunType = AC.lPolyFunType = 1;

```

```

05382     }
05383 #ifdef WITHFLOAT
05384     if ( mpfaux_ != 0 ) {
05385         MesPrint("&Simultaneous use of PolyFun and float_ is not allowed.");
05386         error = 1;
05387     }
05388 #endif
05389     return(error);
05390 }
05391
05392 /*
05393 #] CoPolyFun :
05394 #[ CoPolyRatFun :
05395
05396     PolyRatFun [,functionname[,functionname] (option)]
05397 */
05398
05399 int CoPolyRatFun(UBYTE *s)
05400 {
05401     GETIDENTITY
05402     WORD numfun;
05403     int type, error = 0;
05404     UBYTE *t, c;
05405     AR.PolyFun = AC.lPolyFun = 0;
05406     AR.PolyFunInv = AC.lPolyFunInv = 0;
05407     AR.PolyFunType = AC.lPolyFunType = 0;
05408     AR.PolyFunExp = AC.lPolyFunExp = 0;
05409     AR.PolyFunVar = AC.lPolyFunVar = 0;
05410     AR.PolyFunPow = AC.lPolyFunPow = 0;
05411     if ( *s == 0 ) return(error);
05412     t = SkipAName(s);
05413     if ( t == 0 ) goto NumErr;
05414     c = *t; *t = 0;
05415 #ifdef WITHFLOAT
05416     if ( mpfaux_ != 0 ) {
05417         MesPrint("&Simultaneous use of PolyFun and float_ is not allowed.");
05418         error = 1;
05419     }
05420 #endif
05421     if ( ( ( type = GetName(AC.varnames,s,&numfun,WITHAUTO) ) != CFUNCTION )
05422     || ( functions[numfun].spec != 0 ) || ( functions[numfun].commute != 0 ) ) {
05423         MesPrint("&%s should be a regular commuting function",s);
05424         if ( type < 0 ) {
05425             if ( GetName(AC.exprnames,s,&numfun,NOAUTO) == NAMENOTFOUND )
05426                 AddFunction(s,0,0,0,0,0,-1,-1);
05427         }
05428         return(1);
05429     }
05430     AR.PolyFun = AC.lPolyFun = numfun+FUNCTION;
05431     AR.PolyFunInv = AC.lPolyFunInv = 0;
05432     AR.PolyFunType = AC.lPolyFunType = 2;
05433     AC.PolyRatFunChanged = 1;
05434     if ( c == 0 ) return(error);
05435     *t = c;
05436     if ( *t == '-' ) { AC.PolyRatFunChanged = 0; t++; }
05437     while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05438     if ( *t == 0 ) return(error);
05439     if ( *t != '(' ) {
05440         s = t;
05441         t = SkipAName(s);
05442         if ( t == 0 ) goto NumErr;
05443         c = *t; *t = 0;
05444         if ( ( ( type = GetName(AC.varnames,s,&numfun,WITHAUTO) ) != CFUNCTION )
05445         || ( functions[numfun].spec != 0 ) || ( functions[numfun].commute != 0 ) ) {
05446             MesPrint("&%s should be a regular commuting function",s);
05447             if ( type < 0 ) {
05448                 if ( GetName(AC.exprnames,s,&numfun,NOAUTO) == NAMENOTFOUND )
05449                     AddFunction(s,0,0,0,0,0,-1,-1);
05450             }
05451             return(1);
05452         }
05453         AR.PolyFunInv = AC.lPolyFunInv = numfun+FUNCTION;
05454         if ( c == 0 ) return(error);
05455         *t = c;
05456         if ( *t == '-' ) { AC.PolyRatFunChanged = 0; t++; }
05457         while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05458         if ( *t == 0 ) return(error);
05459     }
05460     if ( *t == '(' ) {
05461         t++;
05462         while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05463     }
05464     /*
05465     Next we need a keyword like
05466     (divergence,ep)
05467     (expand,ep,maxpow)
05468     */
05469     s = t;

```

```

05469     t = SkipAName(s);
05470     if ( t == 0 ) goto NumErr;
05471     c = *t; *t = 0;
05472     if ( ( StrICmp(s, (UBYTE *)"divergence") == 0 )
05473 || ( StrICmp(s, (UBYTE *)"finddivergence") == 0 ) ) {
05474         if ( c != ',' ) {
05475             MesPrint("&Illegal option field in PolyRatFun statement.");
05476             return(1);
05477         }
05478         *t = c;
05479         while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05480         s = t;
05481         t = SkipAName(s);
05482         if ( t == 0 ) goto NumErr;
05483         c = *t; *t = 0;
05484         if ( ( type = GetName(AC.varnames,s,&AC.lPolyFunVar,WITHAUTO) ) != CSYMBOL ) {
05485             MesPrint("&Illegal symbol %s in option field in PolyRatFun statement.",s);
05486             return(1);
05487         }
05488         *t = c;
05489         while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05490         if ( *t != ')' ) {
05491             MesPrint("&Illegal termination of option in PolyRatFun statement.");
05492             return(1);
05493         }
05494         AR.PolyFunExp = AC.lPolyFunExp = 1;
05495         AR.PolyFunVar = AC.lPolyFunVar;
05496         symbols[AC.lPolyFunVar].minpower = -MAXPOWER;
05497         symbols[AC.lPolyFunVar].maxpower = MAXPOWER;
05498     }
05499     else if ( StrICmp(s, (UBYTE *)"expand") == 0 ) {
05500         WORD x = 0, etype = 2;
05501         if ( c != ',' ) {
05502             MesPrint("&Illegal option field in PolyRatFun statement.");
05503             return(1);
05504         }
05505         *t = c;
05506         while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05507         s = t;
05508         t = SkipAName(s);
05509         if ( t == 0 ) goto NumErr;
05510         c = *t; *t = 0;
05511         if ( ( type = GetName(AC.varnames,s,&AC.lPolyFunVar,WITHAUTO) ) != CSYMBOL ) {
05512             MesPrint("&Illegal symbol %s in option field in PolyRatFun statement.",s);
05513             return(1);
05514         }
05515         *t = c;
05516         while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05517         if ( *t > '9' || *t < '0' ) {
05518             MesPrint("&Illegal option field in PolyRatFun statement.");
05519             return(1);
05520         }
05521         while ( *t <= '9' && *t >= '0' ) x = 10*x + *t++ - '0';
05522         while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05523         if ( *t != ')' ) {
05524             s = t;
05525             t = SkipAName(s);
05526             if ( t == 0 ) goto ParErr;
05527             c = *t; *t = 0;
05528             if ( StrICmp(s, (UBYTE *)"fixed") == 0 ) {
05529                 etype = 3;
05530             }
05531             else if ( StrICmp(s, (UBYTE *)"relative") == 0 ) {
05532                 etype = 2;
05533             }
05534             else {
05535                 MesPrint("&Illegal termination of option in PolyRatFun statement.");
05536                 return(1);
05537             }
05538             *t = c;
05539             while ( *t == ',' || *t == ' ' || *t == '\t' ) t++;
05540             if ( *t != ')' ) {
05541                 MesPrint("&Illegal termination of option in PolyRatFun statement.");
05542                 return(1);
05543             }
05544         }
05545         AR.PolyFunExp = AC.lPolyFunExp = etype;
05546         AR.PolyFunVar = AC.lPolyFunVar;
05547         AR.PolyFunPow = AC.lPolyFunPow = x;
05548         symbols[AC.lPolyFunVar].minpower = -MAXPOWER;
05549         symbols[AC.lPolyFunVar].maxpower = MAXPOWER;
05550     }
05551     else {
05552 ParErr: MesPrint("&Illegal option %s in PolyRatFun statement.",s);
05553         return(1);
05554     }
05555     t++;

```

```

05556         while ( *t == ',' || *t == ' ' || *t == '\\t' ) t++;
05557         if ( *t == 0 ) return(error);
05558     }
05559     NumErr++;
05560     MesPrint("&PolyRatFun statement needs one or two commuting function(s) for its argument(s)");
05561     return(1);
05562 }
05563
05564 /*
05565  #] CoPolyRatFun :
05566  #[ CoMerge :
05567 */
05568
05569 int CoMerge(UBYTE *inp)
05570 {
05571     UBYTE *s = inp;
05572     int type;
05573     WORD numfunc, option = 0;
05574     if ( tolower(s[0]) == 'o' && tolower(s[1]) == 'n' && tolower(s[2]) == 'c' &&
05575         tolower(s[3]) == 'e' && tolower(s[4]) == ',' ) {
05576         option = 1; s += 5;
05577     }
05578     else if ( tolower(s[0]) == 'a' && tolower(s[1]) == 'l' && tolower(s[2]) == 'l' &&
05579         tolower(s[3]) == ',' ) {
05580         option = 0; s += 4;
05581     }
05582     if ( *s == '$' ) {
05583         if ( ( type = GetName(AC.dollarnames,s+1,&numfunc,NOAUTO) ) == CDOLLAR )
05584             numfunc = -numfunc;
05585         else {
05586             MesPrint("%s is undefined",s);
05587             numfunc = AddDollar(s+1,DOLINDEX,&one,1);
05588             return(1);
05589         }
05590     tests: s = SkipAName(s);
05591     if ( *s != 0 ) {
05592         MesPrint("&Merge/shuffle should have a single function or $variable for its argument");
05593         return(1);
05594     }
05595     }
05596     else if ( ( type = GetName(AC.varnames,s,&numfunc,WITHAUTO) ) == CFUNCTION ) {
05597         numfunc += FUNCTION;
05598         goto tests;
05599     }
05600     else if ( type != -1 ) {
05601         if ( type != CDUBIOUS ) {
05602             NameConflict(type,s);
05603             type = MakeDubious(AC.varnames,s,&numfunc);
05604         }
05605         return(1);
05606     }
05607     else {
05608         MesPrint("%s is not a function",s);
05609         numfunc = AddFunction(s,0,0,0,0,0,-1,-1) + FUNCTION;
05610         return(1);
05611     }
05612     Add4Com(TYPEMERGE,numfunc,option);
05613     return(0);
05614 }
05615
05616 /*
05617  #] CoMerge :
05618  #[ CoStuffle :
05619
05620     Important for future options: The bit, given by 256 (bit 8) is reserved
05621     internally for keeping track of the sign in the number of Stuffle
05622     additions.
05623 */
05624
05625 int CoStuffle(UBYTE *inp)
05626 {
05627     UBYTE *s = inp, *ss, c;
05628     int type;
05629     WORD numfunc, option = 0;
05630     if ( tolower(s[0]) == 'o' && tolower(s[1]) == 'n' && tolower(s[2]) == 'c' &&
05631         tolower(s[3]) == 'e' && tolower(s[4]) == ',' ) {
05632         option = 1; s += 5;
05633     }
05634     else if ( tolower(s[0]) == 'a' && tolower(s[1]) == 'l' && tolower(s[2]) == 'l' &&
05635         tolower(s[3]) == ',' ) {
05636         option = 0; s += 4;
05637     }
05638     ss = SkipAName(s);
05639     c = *ss; *ss = 0;
05640     if ( *s == '$' ) {
05641         if ( ( type = GetName(AC.dollarnames,s+1,&numfunc,NOAUTO) ) == CDOLLAR )
05642             numfunc = -numfunc;

```

```

05643         else {
05644             MesPrint("%s is undefined",s);
05645             numfunc = AddDollar(s+1,DOLINDEX,&one,1);
05646             return(1);
05647         }
05648 tests: *ss = c;
05649         if ( *ss != '+' && *ss != '-' && ss[1] != 0 ) {
05650             MesPrint("&Stuffle should have a single function or $variable for its argument, followed
by either + or -");
05651             return(1);
05652         }
05653         if ( *ss == '-' ) option += 2;
05654     }
05655     else if ( ( type = GetName(AC.varnames,s,&numfunc,WITHAUTO) ) == CFUNCTION ) {
05656         numfunc += FUNCTION;
05657         goto tests;
05658     }
05659     else if ( type != -1 ) {
05660         if ( type != CDUBIOUS ) {
05661             NameConflict(type,s);
05662             type = MakeDubious(AC.varnames,s,&numfunc);
05663         }
05664         return(1);
05665     }
05666     else {
05667         MesPrint("%s is not a function",s);
05668         numfunc = AddFunction(s,0,0,0,0,-1,-1) + FUNCTION;
05669         return(1);
05670     }
05671     Add4Com(TYPESTUFFLE,numfunc,option);
05672     return(0);
05673 }
05674
05675 /*
05676 #] CoStuffle :
05677 #[ CoProcessBucket :
05678 */
05679
05680 int CoProcessBucket(UBYTE *s)
05681 {
05682     LONG x;
05683     while ( *s == ',' || *s == '=' ) s++;
05684     ParseNumber(x,s)
05685     if ( *s && *s != ' ' && *s != '\t' ) {
05686         MesPrint("&Numerical value expected for ProcessBucketSize");
05687         return(1);
05688     }
05689     AC.ProcessBucketSize = x;
05690     return(0);
05691 }
05692
05693 /*
05694 #] CoProcessBucket :
05695 #[ CoThreadBucket :
05696 */
05697
05698 int CoThreadBucket(UBYTE *s)
05699 {
05700     LONG x;
05701     while ( *s == ',' || *s == '=' ) s++;
05702     ParseNumber(x,s)
05703     if ( *s && *s != ' ' && *s != '\t' ) {
05704         MesPrint("&Numerical value expected for ThreadBucketSize");
05705         return(1);
05706     }
05707     if ( x <= 0 ) {
05708         Warning("Negative of zero value not allowed for ThreadBucketSize. Adjusted to 1.");
05709         x = 1;
05710     }
05711     AC.ThreadBucketSize = x;
05712 #ifdef WITHPTHREADS
05713     if ( AS.MultiThreaded ) MakeThreadBuckets(-1,1);
05714 #endif
05715     return(0);
05716 }
05717
05718 /*
05719 #] CoThreadBucket :
05720 #[ DoArgPlode :
05721
05722 Syntax: a list of functions.
05723 If the functions have an argument it must be a function.
05724 In the case f(g) we treat f(g(...)) with g any argument.
05725 (not yet implemented)
05726 */
05727
05728 int DoArgPlode(UBYTE *s, int par)

```

```

05729 {
05730     GETIDENTITY
05731     WORD numfunc, type, error = 0, *w, n;
05732     UBYTE *t,c;
05733     int i;
05734     w = AT.WorkPointer;
05735     *w++ = par;
05736     w++;
05737     while ( *s == ',' ) s++;
05738     while ( *s ) {
05739         if ( *s == '$' ) {
05740             MesPrint("&We don't do dollar variables yet in ArgImplode/ArgExplode");
05741             return(1);
05742         }
05743         t = s;
05744         if ( ( s = SkipAName(s) ) == 0 ) return(1);
05745         c = *s; *s = 0;
05746         if ( ( type = GetName(AC.varnames,t,&numfunc,WITHAUTO) ) == CFUNCTION ) {
05747             numfunc += FUNCTION;
05748         }
05749         else if ( type != -1 ) {
05750             if ( type != CDUBIOUS ) {
05751                 NameConflict(type,t);
05752                 type = MakeDubious(AC.varnames,t,&numfunc);
05753             }
05754             error = 1;
05755         }
05756         else {
05757             MesPrint("&%s is not a function",t);
05758             numfunc = AddFunction(s,0,0,0,0,0,-1,-1) + FUNCTION;
05759             return(1);
05760         }
05761         *s = c;
05762         *w++ = numfunc;
05763         *w++ = FUNHEAD;
05764         #if FUNHEAD > 2
05765         for ( i = 2; i < FUNHEAD; i++ ) *w++ = 0;
05766     #endif
05767         if ( *s && *s != ',' ) {
05768             MesPrint("&Illegal character in ArgImplode/ArgExplode statement: %s",s);
05769             return(1);
05770         }
05771         while ( *s == ',' ) s++;
05772     }
05773     n = w - AT.WorkPointer;
05774     AT.WorkPointer[1] = n;
05775     AddNtoL(n,AT.WorkPointer);
05776     return(error);
05777 }
05778
05779 /*
05780 #] DoArgPlode :
05781 #[ CoArgExplode :
05782 */
05783
05784 int CoArgExplode(UBYTE *s) { return(DoArgPlode(s,TYPEARGEXPLODE)); }
05785
05786 /*
05787 #] CoArgExplode :
05788 #[ CoArgImplode :
05789 */
05790
05791 int CoArgImplode(UBYTE *s) { return(DoArgPlode(s,TYPEARGIMPLODE)); }
05792
05793 /*
05794 #] CoArgImplode :
05795 #[ CoClearTable :
05796 */
05797
05798 int CoClearTable(UBYTE *s)
05799 {
05800     UBYTE c, *t;
05801     int j, type, error = 0;
05802     WORD numfun;
05803     TABLES T, TT;
05804     if ( *s == 0 ) {
05805         MesPrint("&The ClearTable statement needs at least one (table) argument.");
05806         return(1);
05807     }
05808     while ( *s ) {
05809         t = s;
05810         s = SkipAName(s);
05811         c = *s; *s = 0;
05812         if ( ( ( type = GetName(AC.varnames,t,&numfun,WITHAUTO) ) != CFUNCTION )
05813             && type != CDUBIOUS ) {
05814             MesPrint("&%s is not a table",t);
05815             error = 4;

```

```

05816         if ( type < 0 ) numfun = AddFunction(t,0,0,0,0,0,-1,-1);
05817         *s = c;
05818         if ( *s == ',' ) s++;
05819         continue;
05820     }
05821 /*
05822     else if ( ( ( T = functions[numfun].tabl ) == 0 )
05823         || ( T->sparse == 0 ) ) goto nofunc;
05824 */
05825     else if ( ( T = functions[numfun].tabl ) == 0 ) goto nofunc;
05826     numfun += FUNCTION;
05827     *s = c;
05828     if ( *s == ',' ) s++;
05829 /*
05830     Now we clear the table.
05831 */
05832     if ( T->sparse ) {
05833         if ( T->boomlijst ) M_free(T->boomlijst,"TableTree");
05834         for ( j = 0; j < T->buffersfill; j++ ) { /* was <= */
05835             finishcbuf(T->buffers[j]);
05836         }
05837         if ( T->buffers ) M_free(T->buffers,"Table buffers");
05838         finishcbuf(T->bufnum);
05839
05840         T->boomlijst = 0;
05841         T->numtree = 0; T->rootnum = 0; T->MaxTreeSize = 0;
05842         T->boomlijst = 0;
05843         T->bufnum = inicbufs();
05844         T->bufferssize = 8;
05845         T->buffers = (WORD *)Malloca(sizeof(WORD)*T->bufferssize,"Table buffers");
05846         T->buffersfill = 0;
05847         T->buffers[T->buffersfill++] = T->bufnum;
05848
05849         T->totind = 0;          /* At the moment there are this many */
05850         T->reserved = 0;
05851
05852         ClearTableTree(T);
05853
05854         if ( T->spare ) {
05855             if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
05856             T->tablepointers = 0;
05857             TT = T->spare;
05858             if ( TT->tablepointers ) M_free(TT->tablepointers,"tablepointers");
05859             for ( j = 0; j < TT->buffersfill; j++ ) {
05860                 finishcbuf(TT->buffers[j]);
05861             }
05862             if ( TT->boomlijst ) M_free(TT->boomlijst,"TableTree");
05863             if ( TT->buffers ) M_free(TT->buffers,"Table buffers");
05864             if ( TT->mm ) M_free(TT->mm,"tableminmax");
05865             if ( TT->flags ) M_free(TT->flags,"tableflags");
05866             M_free(TT,"table");
05867             SpareTable(T);
05868         }
05869     }
05870     else EmptyTable(T);
05871 }
05872 return(error);
05873 }
05874
05875 /*
05876 #] CoClearTable :
05877 #[ CoDenominators :
05878 */
05879
05880 int CoDenominators(UBYTE *s)
05881 {
05882     WORD numfun;
05883     int type;
05884     UBYTE *t = SkipAName(s), *t1;
05885     if ( t == 0 ) goto syntaxerror;
05886     t1 = t; while ( *t1 == ',' || *t1 == ' ' || *t1 == '\t' ) t1++;
05887     if ( *t1 ) goto syntaxerror;
05888     *t = 0;
05889     if ( ( ( type = GetName(AC.varnames,s,&numfun,WITHAUTO) ) != CFUNCTION )
05890         || ( functions[numfun].spec != 0 ) ) {
05891         if ( type < 0 ) {
05892             if ( GetName(AC.exprnames,s,&numfun,NOAUTO) == NAMENOTFOUND )
05893                 AddFunction(s,0,0,0,0,0,-1,-1);
05894         }
05895         goto syntaxerror;
05896     }
05897     Add3Com(TYPEDENOMINATORS,numfun+FUNCTION);
05898     return(0);
05899 syntaxerror:
05900     MesPrint("&Denominators statement needs one regular function for its argument");
05901     return(1);
05902 }

```



```

05903
05904 /*
05905     #] CoDenominators :
05906     #[ CoDropCoefficient :
05907 */
05908
05909 int CoDropCoefficient(UBYTE *s)
05910 {
05911     if ( *s == 0 ) {
05912         Add2Com(TYPEDROPCOEFFICIENT)
05913         return(0);
05914     }
05915     MesPrint("&Illegal argument in DropCoefficient statement: '%s'",s);
05916     return(1);
05917 }
05918 /*
05919     #] CoDropCoefficient :
05920     #[ CoDropSymbols :
05921 */
05922
05923 int CoDropSymbols(UBYTE *s)
05924 {
05925     if ( *s == 0 ) {
05926         Add2Com(TYPEDROPSYMBOLS)
05927         return(0);
05928     }
05929     MesPrint("&Illegal argument in DropSymbols statement: '%s'",s);
05930     return(1);
05931 }
05932 /*
05933     #] CoDropSymbols :
05934     #[ CoToPolynomial :
05935
05936     Converts the current term as much as possible to symbols.
05937     Keeps a list of all objects converted to symbols in AM.sbufnum.
05938     Note that this cannot be executed in parallel because we have only
05939     a single compiler buffer for this. Hence we switch on the noparallel
05940     module option.
05941
05942     Option(s):
05943         OnlyFunctions [,name1][,name2][,...,namem];
05944 */
05945
05946 int CoToPolynomial(UBYTE *inp)
05947 {
05948     int error = 0;
05949     while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
05950     if ( ( AC.topolynomialflag & ~TOPOLYNOMIALFLAG ) != 0 ) {
05951         MesPrint("&ToPolynomial statement and FactArg statement are not allowed in the same module");
05952         return(1);
05953     }
05954     if ( AO.OptimizeResult.code != NULL ) {
05955         MesPrint("&Using ToPolynomial statement when there are still optimization results active.");
05956         MesPrint("&Please use #ClearOptimize instruction first.");
05957         MesPrint("&This will loose the optimized expression.");
05958         return(1);
05959     }
05960     if ( *inp == 0 ) {
05961         Add3Com(TYPETOPOLYNOMIAL,DOALL)
05962     }
05963     else {
05964         int numargs = 0;
05965         WORD *funnums = 0, type, num;
05966         UBYTE *s, c;
05967         s = SkipAName(inp);
05968         if ( s == 0 ) return(1);
05969         c = *s; *s = 0;
05970         if ( StrICmp(inp, (UBYTE *)"onlyfunctions") ) {
05971             MesPrint("&Illegal option %s in ToPolynomial statement",inp);
05972             *s = c;
05973             return(1);
05974         }
05975         *s = c;
05976         inp = s;
05977         while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
05978         s = inp;
05979         while ( *s ) s++;
05980     /*
05981     Get definitely enough space for the numbers of the functions
05982     */
05983     funnums = (WORD *)Malloc1(((LONG)(s-inp)+3)*sizeof(WORD), "ToPlynomial");
05984     while ( *inp ) {
05985         s = SkipAName(inp);
05986         if ( s == 0 ) return(1);
05987         c = *s; *s = 0;
05988         type = GetName(AC.varnames,inp,&num,WITHAUTO);
05989         if ( type != CFUNCTION ) {

```

```

05990         MesPrint("&%s is not a function in ToPolynomial statement",inp);
05991         error = 1;
05992     }
05993     funnums[3+numargs++] = num+FUNCTION;
05994     *s = c;
05995     inp = s;
05996     while ( *inp == ' ' || *inp == ',' || *inp == '\\t' ) inp++;
05997 }
05998 funnums[0] = TYPETOPOLYNOMIAL;
05999 funnums[1] = numargs+3;
06000 funnums[2] = ONLYFUNCTIONS;
06001
06002 AddNtoL(numargs+3,funnums);
06003 if ( funnums ) M_free(funnums,"ToPolynomial");
06004 }
06005 AC.topolynomialflag |= TOPOLYNOMIALFLAG;
06006 #ifdef WITHMPI
06007 /* In ParFORM, ToPolynomial has to be executed on the master. */
06008 AC.mparallelflag |= NOPARALLEL_CONVPOLY;
06009 #endif
06010 return(error);
06011 }
06012
06013 /*
06014 #] CoToPolynomial :
06015 #[ CoFromPolynomial :
06016
06017 Converts the current term as much as possible back from extra symbols
06018 to their original values. Does not look inside functions.
06019 */
06020
06021 int CoFromPolynomial(UBYTE *inp)
06022 {
06023     while ( *inp == ' ' || *inp == ',' || *inp == '\\t' ) inp++;
06024     if ( *inp == 0 ) {
06025         if ( AO.OptimizeResult.code != NULL ) {
06026             MesPrint("&Using FromPolynomial statement when there are still optimization results
active.");
06027             MesPrint("&Please use #ClearOptimize instruction first.");
06028             MesPrint("&This will loose the optimized expression.");
06029             return(1);
06030         }
06031         Add2Com(TYPEFROMPOLYNOMIAL)
06032         return(0);
06033     }
06034     MesPrint("&Illegal argument in FromPolynomial statement: '%s'",inp);
06035     return(1);
06036 }
06037
06038 /*
06039 #] CoFromPolynomial :
06040 #[ CoArgToExtraSymbol :
06041
06042 Converts the specified function arguments into extra symbols.
06043
06044 Syntax: ArgToExtraSymbol [ToNumber] [<argument specifications>]
06045 */
06046
06047 int CoArgToExtraSymbol(UBYTE *s)
06048 {
06049     CBUF *c = cbuf + AC.cbufnum;
06050     WORD *lhs;
06051
06052     /* TODO: resolve interference with rational arithmetic. (#138) */
06053     if ( ( AC.topolynomialflag & ~TOPOLYNOMIALFLAG ) != 0 ) {
06054         MesPrint("&ArgToExtraSymbol statement and FactArg statement are not allowed in the same
module");
06055         return(1);
06056     }
06057     if ( AO.OptimizeResult.code != NULL ) {
06058         MesPrint("&Using ArgToExtraSymbol statement when there are still optimization results
active.");
06059         MesPrint("&Please use #ClearOptimize instruction first.");
06060         MesPrint("&This will loose the optimized expression.");
06061         return(1);
06062     }
06063
06064     SkipSpaces(&s);
06065     int tonumber = ConsumeOption(&s, "tonumber");
06066
06067     int ret = DoArgument(s,TYPEARGTOEXTRASymbol);
06068     if ( ret ) return(ret);
06069
06070     /*
06071     * The "scale" parameter is unused. Instead, we put the "tonumber"
06072     * parameter.
06073     */

```

```

06074     lhs = C->lhs[C->numlhs];
06075     if ( lhs[4] != 1 ) {
06076         Warning("scale parameter (^n) is ignored in ArgToExtraSymbol");
06077     }
06078     lhs[4] = tonumber;
06079
06080     AC.topolynomialflag |= TOPOLYNOMIALFLAG; /* This flag is also used in ParFORM. */
06081 #ifdef WITHMPI
06082     /*
06083      * In ParFORM, the conversion to extra symbols has to be performed on
06084      * the master.
06085      */
06086     AC.mparallelfailflag |= NOPARALLEL_CONVPOLY;
06087 #endif
06088
06089     return(0);
06090 }
06091
06092 /*
06093  #] CoArgToExtraSymbol :
06094  #[ CoExtraSymbols :
06095  */
06096
06097 int CoExtraSymbols(UBYTE *inp)
06098 {
06099     UBYTE *arg1, *arg2, c, *s;
06100     WORD i, j, type, number;
06101     while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
06102     if ( FG.cTable[*inp] != 0 ) {
06103         MesPrint("&Illegal argument in ExtraSymbols statement: '%s'",inp);
06104         return(1);
06105     }
06106     arg1 = inp;
06107     while ( FG.cTable[*inp] == 0 ) inp++;
06108     c = *inp; *inp = 0;
06109     if ( ( StrICmp(arg1,(UBYTE *)"array") == 0 )
06110         || ( StrICmp(arg1,(UBYTE *)"vector") == 0 ) ) {
06111         AC.extrasymbols = 1;
06112     }
06113     else if ( StrICmp(arg1,(UBYTE *)"underscore") == 0 ) {
06114         AC.extrasymbols = 0;
06115     }
06116     /*
06117     else if ( StrICmp(arg1,(UBYTE *)"nothing") == 0 ) {
06118         AC.extrasymbols = 2;
06119     }
06120     */
06121     else {
06122         MesPrint("&Illegal keyword in ExtraSymbols statement: '%s'",arg1);
06123         return(1);
06124     }
06125     *inp = c;
06126     while ( *inp == ' ' || *inp == ',' || *inp == '\t' ) inp++;
06127     if ( FG.cTable[*inp] != 0 ) {
06128         MesPrint("&Illegal argument in ExtraSymbols statement: '%s'",inp);
06129         return(1);
06130     }
06131     arg2 = inp;
06132     while ( FG.cTable[*inp] <= 1 ) inp++;
06133     if ( *inp != 0 ) {
06134         MesPrint("&Illegal end of ExtraSymbols statement: '%s'",inp);
06135         return(1);
06136     }
06137     /*
06138     Now check whether this object has been declared already.
06139     That would not be allowed.
06140     */
06141     if ( AC.extrasymbols == 1 ) {
06142         type = GetName(AC.varnames,arg2,&number,NOAUTO);
06143         if ( type != NAMENOTFOUND ) {
06144             MesPrint("&ExtraSymbols statement: '%s' has already been declared before",arg2);
06145             return(1);
06146         }
06147     }
06148     else if ( AC.extrasymbols == 0 ) {
06149         if ( *arg2 == 'N' ) {
06150             s = arg2+1;
06151             while ( FG.cTable[*s] == 1 ) s++;
06152             if ( *s == 0 ) {
06153                 MesPrint("&ExtraSymbols statement: '%s' creates conflicts with summed indices",arg2);
06154                 return(1);
06155             }
06156         }
06157     }
06158     if ( AC.extrasym ) { M_free(AC.extrasym,"extrasym"); AC.extrasym = 0; }
06159     i = inp - arg2 + 1;
06160     AC.extrasym = (UBYTE *)Malloc1(i*sizeof(UBYTE),"extrasym");

```

```

06161     for ( j = 0; j < i; j++ ) AC.extrasym[j] = arg2[j];
06162     return(0);
06163 }
06164
06165 /*
06166     #] CoExtraSymbols :
06167     #[ GetIfDollarFactor :
06168 */
06169
06170 WORD *GetIfDollarFactor(UBYTE **inp, WORD *w)
06171 {
06172     LONG x;
06173     WORD number;
06174     UBYTE *name, c, *s;
06175     s = *inp;
06176     if ( FG.cTable[*s] == 1 ) {
06177         x = 0;
06178         while ( FG.cTable[*s] == 1 ) {
06179             x = 10*x + *s++ - '0';
06180             if ( x >= MAXPOSITIVE ) {
06181                 MesPrint("&Value in dollar factor too large");
06182                 while ( FG.cTable[*s] == 1 ) s++;
06183                 *inp = s;
06184                 return(0);
06185             }
06186         }
06187         *w++ = IFDOLLAREXTRA;
06188         *w++ = 3;
06189         *w++ = -x-1;
06190         *inp = s;
06191         return(w);
06192     }
06193     if ( *s != '$' ) {
06194         MesPrint("&Factor indicator for $-variable should be a number or a $-variable.");
06195         return(0);
06196     }
06197     s++; name = s;
06198     while ( FG.cTable[*s] < 2 ) s++;
06199     c = *s; *s = 0;
06200     if ( GetName(AC.dollarnames,name,&number,NOAUTO) == NAMENOTFOUND ) {
06201         MesPrint("&dollar in if statement should have been defined previously");
06202         return(0);
06203     }
06204     *s = c;
06205     *w++ = IFDOLLAREXTRA;
06206     *w++ = 3;
06207     *w++ = number;
06208     if ( c == '[' ) {
06209         s++;
06210         *inp = s;
06211         if ( ( w = GetIfDollarFactor(inp,w) ) == 0 ) return(0);
06212         s = *inp;
06213         if ( *s != ']' ) {
06214             MesPrint("&unmatched [] in $ in if statement");
06215             return(0);
06216         }
06217         s++;
06218         *inp = s;
06219     }
06220     return(w);
06221 }
06222
06223 /*
06224     #] GetIfDollarFactor :
06225     #[ GetDoParam :
06226 */
06227
06228 UBYTE *GetDoParam(UBYTE *inp, WORD **wp, int par)
06229 {
06230     LONG x;
06231     WORD number;
06232     UBYTE *name, c;
06233     if ( FG.cTable[*inp] == 1 ) {
06234         x = 0;
06235         while ( *inp >= '0' && *inp <= '9' ) {
06236             x = 10*x + *inp++ - '0';
06237             if ( x > MAXPOSITIVE ) {
06238                 if ( par == -1 ) {
06239                     MesPrint("&Value in dollar factor too large");
06240                 }
06241                 else {
06242                     MesPrint("&Value in do loop boundaries too large");
06243                 }
06244                 while ( FG.cTable[*inp] == 1 ) inp++;
06245                 return(0);
06246             }
06247         }

```

```

06248         if ( par > 0 ) {
06249             *(*wp)++ = SNUMBER;
06250             *(*wp)++ = (WORD)x;
06251         }
06252         else {
06253             *(*wp)++ = DOLLAREXP2;
06254             *(*wp)++ = -( (WORD)x)-1;
06255         }
06256         return(inp);
06257     }
06258     if ( *inp != '$' ) {
06259         return(0);
06260     }
06261     inp++; name = inp;
06262     while ( FG.cTable[*inp] < 2 ) inp++;
06263     c = *inp; *inp = 0;
06264     if ( GetName(AC.dollarnames,name,&number,NOAUTO) == NAMENOTFOUND ) {
06265         if ( par == -1 ) {
06266             MesPrint("&dollar in print statement should have been defined previously");
06267         }
06268         else {
06269             MesPrint("&dollar in do loop boundaries should have been defined previously");
06270         }
06271         return(0);
06272     }
06273     *inp = c;
06274     if ( par > 0 ) {
06275         *(*wp)++ = DOLLAREXP2;
06276         *(*wp)++ = number;
06277     }
06278     else {
06279         *(*wp)++ = DOLLAREXP2;
06280         *(*wp)++ = number;
06281     }
06282     if ( c == '[' ) {
06283         inp++;
06284         inp = GetDoParam(inp,wp,0);
06285         if ( inp == 0 ) return(0);
06286         if ( *inp != ']' ) {
06287             if ( par == -1 ) {
06288                 MesPrint("&unmatched [] in $ in print statement");
06289             }
06290             else {
06291                 MesPrint("&unmatched [] in do loop boundaries");
06292             }
06293             return(0);
06294         }
06295         inp++;
06296     }
06297     return(inp);
06298 }
06299
06300 /*
06301 #] GetDoParam :
06302 #[ CoDo :
06303 */
06304
06305 int CoDo(UBYTE *inp)
06306 {
06307     GETIDENTITY
06308     CBUF *C = cbuf+AC.cbunum;
06309     WORD *w, numparam;
06310     int error = 0, i;
06311     UBYTE *name, c;
06312     if ( AC.doloopstack == 0 ) {
06313         AC.doloopstacksize = 20;
06314         AC.doloopstack = (WORD *)Malloc1(AC.doloopstacksize*2*sizeof(WORD),"doloop stack");
06315         AC.doloopnest = AC.doloopstack + AC.doloopstacksize;
06316     }
06317     if ( AC.dolooplevel >= AC.doloopstacksize ) {
06318         WORD *newstack, *newnest, newsize;
06319         newsize = AC.doloopstacksize * 2;
06320         newstack = (WORD *)Malloc1(newsize*2*sizeof(WORD),"doloop stack");
06321         newnest = newstack + newsize;
06322         for ( i = 0; i < newsize; i++ ) {
06323             newstack[i] = AC.doloopstack[i];
06324             newnest[i] = AC.doloopnest[i];
06325         }
06326         M_free(AC.doloopstack,"doloop stack");
06327         AC.doloopstack = newstack;
06328         AC.doloopnest = newnest;
06329         AC.doloopstacksize = newsize;
06330     }
06331     AC.doloopnest[AC.dolooplevel] = NestingChecksum();
06332
06333     w = AT.WorkPointer;
06334     *w++ = TYPEDOLOOP;

```

```

06335     w++; /* Space for the length of the statement */
06336 /*
06337 Now the $loopvariable
06338 */
06339 while ( *inp == ',' ) inp++;
06340 if ( *inp != '$' ) {
06341     error = 1;
06342     MesPrint("&do loop parameter should be a dollar variable");
06343 }
06344 else {
06345     inp++;
06346     name = inp;
06347     if ( FG.cTable[*inp] != 0 ) {
06348         error = 1;
06349         MesPrint("&illegal name for do loop parameter");
06350     }
06351     while ( FG.cTable[*inp] < 2 ) inp++;
06352     c = *inp; *inp = 0;
06353     if ( GetName(AC.dollarnames,name,&numparam,NOAUTO) == NAMENOTFOUND ) {
06354         numparam = AddDollar(name,DOLUNDEFINED,0,0);
06355     }
06356     *w++ = numparam;
06357     *inp = c;
06358     AddPotModdollar(numparam);
06359 }
06360 w++; /* space for the level of the enddo statement */
06361 while ( *inp == ',' ) inp++;
06362 if ( *inp != '=' ) goto IllSyntax;
06363 inp++;
06364 while ( *inp == ',' ) inp++;
06365 /*
06366 The start value
06367 */
06368 inp = GetDoParam(inp,&w,1);
06369 if ( inp == 0 || *inp != ',' ) goto IllSyntax;
06370 while ( *inp == ',' ) inp++;
06371 /*
06372 The end value
06373 */
06374 inp = GetDoParam(inp,&w,1);
06375 if ( inp == 0 || ( *inp != 0 && *inp != ',' ) ) goto IllSyntax;
06376 /*
06377 The increment value
06378 */
06379 if ( *inp != ',' ) {
06380     if ( *inp == 0 ) { *w++ = SNUMBER; *w++ = 1; }
06381     else goto IllSyntax;
06382 }
06383 else {
06384     while ( *inp == ',' ) inp++;
06385     inp = GetDoParam(inp,&w,1);
06386 }
06387 if ( inp == 0 || *inp != 0 ) goto IllSyntax;
06388 *w = 0;
06389 AT.WorkPointer[1] = w - AT.WorkPointer;
06390 /*
06391 Put away and set information for placing enddo information.
06392 */
06393 AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
06394 AC.doloopstack[AC.dolooplevel++] = C->numlhs;
06395
06396 return(error);
06397
06398 IllSyntax:
06399 MesPrint("&Illegal syntax for do statement");
06400 return(1);
06401 }
06402
06403 /*
06404 #] CoDo :
06405 #[ CoEndDo :
06406 */
06407
06408 int CoEndDo(UBYTE *inp)
06409 {
06410     CBUF *C = cbuf+AC.cbufnum;
06411     WORD scratch[3];
06412     while ( *inp == ',' ) inp++;
06413     if ( *inp ) {
06414         MesPrint("&Illegal syntax for EndDo statement");
06415         return(1);
06416     }
06417     if ( AC.dolooplevel <= 0 ) {
06418         MesPrint("&EndDo without corresponding Do statement");
06419         return(1);
06420     }
06421     AC.dolooplevel--;

```

```

06422     scratch[0] = TYPEENDDOLOOP;
06423     scratch[1] = 3;
06424     scratch[2] = AC.doloopstack[AC.dolooplevel];
06425     AddNtoL(3,scratch);
06426     cbuf[AC.cbunum].lhs[AC.doloopstack[AC.dolooplevel]][3] = C->numlhs;
06427     if ( AC.doloopnest[AC.dolooplevel] != NestingChecksum() ) {
06428         MesNesting();
06429         return(1);
06430     }
06431     return(0);
06432 }
06433
06434 /*
06435  #] CoEndDo :
06436  #[ CoFactDollar :
06437  */
06438
06439 int CoFactDollar(UBYTE *inp)
06440 {
06441     WORD numdollar;
06442     if ( *inp == '$' ) {
06443         if ( GetName(AC.dollarnames,inp+1,&numdollar,NOAUTO) != CDOLLAR ) {
06444             MesPrint("&%s is undefined",inp);
06445             numdollar = AddDollar(inp+1,DOLINDEX,&one,1);
06446             return(1);
06447         }
06448         inp = SkipAName(inp+1);
06449         if ( *inp != 0 ) {
06450             MesPrint("&FactDollar should have a single $variable for its argument");
06451             return(1);
06452         }
06453         AddPotModdollar(numdollar);
06454     }
06455     else {
06456         MesPrint("&%s is not a $-variable",inp);
06457         return(1);
06458     }
06459     Add3Com(TYPEFACTOR,numdollar);
06460     return(0);
06461 }
06462
06463 /*
06464  #] CoFactDollar :
06465  #[ CoFactorize :
06466  */
06467
06468 int CoFactorize(UBYTE *s) { return(DoFactorize(s,1)); }
06469
06470 /*
06471  #] CoFactorize :
06472  #[ CoNFactorize :
06473  */
06474
06475 int CoNFactorize(UBYTE *s) { return(DoFactorize(s,0)); }
06476
06477 /*
06478  #] CoNFactorize :
06479  #[ CoUnFactorize :
06480  */
06481
06482 int CoUnFactorize(UBYTE *s) { return(DoFactorize(s,3)); }
06483
06484 /*
06485  #] CoUnFactorize :
06486  #[ CoNUnFactorize :
06487  */
06488
06489 int CoNUnFactorize(UBYTE *s) { return(DoFactorize(s,2)); }
06490
06491 /*
06492  #] CoNUnFactorize :
06493  #[ DoFactorize :
06494  */
06495
06496 int DoFactorize(UBYTE *s,int par)
06497 {
06498     EXPRESSIONS e;
06499     WORD i;
06500     WORD number;
06501     UBYTE *t, c;
06502     int error = 0, keepzeroflag = 0;
06503     if ( *s == '(' ) {
06504         s++;
06505         while ( *s != ')' && *s ) {
06506             if ( FG.cTable[*s] == 0 ) {
06507                 t = s; while ( FG.cTable[*s] == 0 ) s++;
06508                 c = *s; *s = 0;

```

```

06509         if ( StrICmp((UBYTE *)"keepzero",t) == 0 ) {
06510             keepzeroflag = 1;
06511         }
06512         else {
06513             MesPrint("&Illegal option in [N][Un]Factorize statement: %s",t);
06514             error = 1;
06515         }
06516         *s = c;
06517     }
06518     while ( *s == ',' ) s++;
06519     if ( *s && *s != ')' && FG.cTable[*s] != 0 ) {
06520         MesPrint("&Illegal character in option field of [N][Un]Factorize statement");
06521         error = 1;
06522         return(error);
06523     }
06524 }
06525 if ( *s ) s++;
06526 while ( *s == ',' || *s == ' ' ) s++;
06527 }
06528 if ( *s == 0 ) {
06529     for ( i = NumExpressions-1; i >= 0; i-- ) {
06530         e = Expressions+i;
06531         if ( e->replace >= 0 ) {
06532             e = Expressions + e->replace;
06533         }
06534         if ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION
06535             || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION
06536             || e->status == INTOHIDELEXPRESSION || e->status == INTOHIDEGEXPRESSION
06537         ) {
06538             switch ( par ) {
06539                 case 0:
06540                     e->vflags &= ~TOBEFACTORED;
06541                     break;
06542                 case 1:
06543                     e->vflags |= TOBEFACTORED;
06544                     e->vflags &= ~TOBEUNFACTORED;
06545                     break;
06546                 case 2:
06547                     e->vflags &= ~TOBEUNFACTORED;
06548                     break;
06549                 case 3:
06550                     e->vflags |= TOBEUNFACTORED;
06551                     e->vflags &= ~TOBEFACTORED;
06552                     break;
06553             }
06554         }
06555         if ( ( e->vflags & TOBEFACTORED ) != 0 ) {
06556             if ( keepzeroflag ) e->vflags |= KEEPZERO;
06557             else e->vflags &= ~KEEPZERO;
06558         }
06559         else e->vflags &= ~KEEPZERO;
06560     }
06561 }
06562 else {
06563     for(;;) { /* Look for a (comma separated) list of variables */
06564         while ( *s == ',' ) s++;
06565         if ( *s == 0 ) break;
06566         if ( *s == '[' || FG.cTable[*s] == 0 ) {
06567             t = s;
06568             if ( ( s = SkipAName(s) ) == 0 ) {
06569                 MesPrint("&Improper name for an expression: '%s'",t);
06570                 return(1);
06571             }
06572             c = *s; *s = 0;
06573             if ( GetName(AC.exprnames,t,&number,NOAUTO) == CEXPRESSION ) {
06574                 e = Expressions+number;
06575                 if ( e->replace >= 0 ) {
06576                     e = Expressions + e->replace;
06577                 }
06578                 if ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION
06579                     || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION
06580                     || e->status == INTOHIDELEXPRESSION || e->status == INTOHIDEGEXPRESSION
06581                 ) {
06582                     switch ( par ) {
06583                         case 0:
06584                             e->vflags &= ~TOBEFACTORED;
06585                             break;
06586                         case 1:
06587                             e->vflags |= TOBEFACTORED;
06588                             e->vflags &= ~TOBEUNFACTORED;
06589                             break;
06590                         case 2:
06591                             e->vflags &= ~TOBEUNFACTORED;
06592                             break;
06593                         case 3:
06594                             e->vflags |= TOBEUNFACTORED;
06595                             e->vflags &= ~TOBEFACTORED;

```



```

06596                                     break;
06597                                     }
06598                                     }
06599                                     if ( ( e->vflags & TOBEFACTORED ) != 0 ) {
06600                                         if ( keepzeroflag ) e->vflags |= KEEPZERO;
06601                                         else e->vflags &= ~KEEPZERO;
06602                                     }
06603                                     else e->vflags &= ~KEEPZERO;
06604                                     }
06605                                     else if ( GetName(AC.varnames,t,&number,NOAUTO) != NAMENOTFOUND ) {
06606                                         MesPrint("%s is not an expression",t);
06607                                         error = 1;
06608                                     }
06609                                     *s = c;
06610                                 }
06611                                 else {
06612                                     MesPrint("&Illegal object in (N)Factorize statement");
06613                                     error = 1;
06614                                     while ( *s && *s != ',' ) s++;
06615                                     if ( *s == 0 ) break;
06616                                 }
06617                             }
06618                         }
06619                     }
06620                     return(error);
06621 }
06622
06623 /*
06624 #] DoFactorize :
06625 #[ CoOptimizeOption :
06626 */
06627
06628 int CoOptimizeOption(UBYTE *s)
06629 {
06630     UBYTE *name, *t1, *t2, c1, c2, *value, *u;
06631     int error = 0, x;
06632     double d;
06633     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
06634     while ( *s ) {
06635         name = s; while ( FG.cTable[*s] == 0 ) s++;
06636         t1 = s; c1 = *t1;
06637         while ( *s == ' ' || *s == '\t' ) s++;
06638         if ( *s != '=' ) {
06639             correctuse:
06640                 MesPrint("&Correct use in Format,Optimize statement is Optionname=value");
06641                 error = 1;
06642                 while ( *s == ' ' || *s == ',' || *s == '\t' || *s == '=' ) s++;
06643                 *t1 = c1;
06644                 continue;
06645             }
06646             *t1 = 0;
06647             s++;
06648             while ( *s == ' ' || *s == '\t' ) s++;
06649             if ( *s == 0 ) goto correctuse;
06650             value = s;
06651             while ( FG.cTable[*s] <= 1 || *s=='.' || *s=='*' || *s == '(' || *s == ')' ) {
06652                 if ( *s == '(' ) { SKIPBRA4(s) }
06653                 s++;
06654             }
06655             t2 = s; c2 = *t2;
06656             while ( *s == ' ' || *s == '\t' ) s++;
06657             if ( *s && *s != ',' ) goto correctuse;
06658             if ( *s ) {
06659                 s++;
06660                 while ( *s == ' ' || *s == '\t' ) s++;
06661             }
06662             *t2 = 0;
06663         }
06664     /*
06665     Now we have name=value with name and value zero terminated strings.
06666     */
06667     if ( StrICmp(name,(UBYTE *)"horner") == 0 ) {
06668         if ( StrICmp(value,(UBYTE *)"occurrence") == 0 ) {
06669             AO.Optimize.horner = O_OCCURRENCE;
06670         }
06671         else if ( StrICmp(value,(UBYTE *)"mcts") == 0 ) {
06672             AO.Optimize.horner = O_MCTS;
06673         }
06674         else if ( StrICmp(value,(UBYTE *)"sa") == 0 ) {
06675             AO.Optimize.horner = O_SIMULATED_ANNEALING;
06676         }
06677         else {
06678             AO.Optimize.horner = -1;
06679             MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
06680             error = 1;
06681         }
06682     }

```

```

06683     else if ( StrICmp(name, (UBYTE *)"hornerdirection") == 0 ) {
06684         if ( StrICmp(value, (UBYTE *)"forward") == 0 ) {
06685             AO.Optimize.hornerdirection = O_FORWARD;
06686         }
06687         else if ( StrICmp(value, (UBYTE *)"backward") == 0 ) {
06688             AO.Optimize.hornerdirection = O_BACKWARD;
06689         }
06690         else if ( StrICmp(value, (UBYTE *)"forwardorbackward") == 0 ) {
06691             AO.Optimize.hornerdirection = O_FORWARDORBACKWARD;
06692         }
06693         else if ( StrICmp(value, (UBYTE *)"forwardandbackward") == 0 ) {
06694             AO.Optimize.hornerdirection = O_FORWARDANDBACKWARD;
06695         }
06696         else {
06697             AO.Optimize.method = -1;
06698             MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s", name, value);
06699             error = 1;
06700         }
06701     }
06702     else if ( StrICmp(name, (UBYTE *)"method") == 0 ) {
06703         if ( StrICmp(value, (UBYTE *)"none") == 0 ) {
06704             AO.Optimize.method = O_NONE;
06705         }
06706         else if ( StrICmp(value, (UBYTE *)"cse") == 0 ) {
06707             AO.Optimize.method = O_CSE;
06708         }
06709         else if ( StrICmp(value, (UBYTE *)"csegreedy") == 0 ) {
06710             AO.Optimize.method = O_CSEGREEDY;
06711         }
06712         else if ( StrICmp(value, (UBYTE *)"greedy") == 0 ) {
06713             AO.Optimize.method = O_GREEDY;
06714         }
06715         else {
06716             AO.Optimize.method = -1;
06717             MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s", name, value);
06718             error = 1;
06719         }
06720     }
06721     else if ( StrICmp(name, (UBYTE *)"timelimit") == 0 ) {
06722         x = 0;
06723         u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06724         if ( *u != 0 ) {
06725             MesPrint("&Option TimeLimit in Format,Optimize statement should be a positive number:
06726             %s", value);
06727             AO.Optimize.mctstimelimit = 0;
06728             AO.Optimize.greedytimelimit = 0;
06729             error = 1;
06730         }
06731         else {
06732             AO.Optimize.mctstimelimit = x/2;
06733             AO.Optimize.greedytimelimit = x/2;
06734         }
06735     }
06736     else if ( StrICmp(name, (UBYTE *)"mctstimelimit") == 0 ) {
06737         x = 0;
06738         u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06739         if ( *u != 0 ) {
06740             MesPrint("&Option MCTSTimeLimit in Format,Optimize statement should be a positive
06741             number: %s", value);
06742             AO.Optimize.mctstimelimit = 0;
06743             error = 1;
06744         }
06745         else {
06746             AO.Optimize.mctstimelimit = x;
06747         }
06748     }
06749     else if ( StrICmp(name, (UBYTE *)"mctsnunexpand") == 0 ) {
06750         int y;
06751         x = 0;
06752         u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06753         if ( *u == '*' || *u == 'x' || *u == 'X' ) {
06754             u++; y = x;
06755             x = 0;
06756             while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06757         }
06758         else { y = 1; }
06759         if ( *u != 0 ) {
06760             MesPrint("&Option MCTSNunExpand in Format,Optimize statement should be a positive
06761             number: %s", value);
06762             AO.Optimize.mctsnunexpand= 0;
06763             AO.Optimize.mctsnunrepeat= 1;
06764             error = 1;
06765         }
06766         else {
06767             AO.Optimize.mctsnunexpand= x;
06768             AO.Optimize.mctsnunrepeat= y;
06769         }
06770     }

```

```

06767     }
06768     else if ( StrICmp(name, (UBYTE *) "mctsnnumrepeat") == 0 ) {
06769         x = 0;
06770         u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06771         if ( *u != 0 ) {
06772             MesPrint("&Option MCTSNnumExpand in Format,Optimize statement should be a positive
number: %s",value);
06773             AO.Optimize.mctsnnumrepeat= 1;
06774             error = 1;
06775         }
06776         else {
06777             AO.Optimize.mctsnnumrepeat= x;
06778         }
06779     }
06780     else if ( StrICmp(name, (UBYTE *) "mctsnnumkeep") == 0 ) {
06781         x = 0;
06782         u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06783         if ( *u != 0 ) {
06784             MesPrint("&Option MCTSNnumKeep in Format,Optimize statement should be a positive
number: %s",value);
06785             AO.Optimize.mctsnnumkeep= 0;
06786             error = 1;
06787         }
06788         else {
06789             AO.Optimize.mctsnnumkeep= x;
06790         }
06791     }
06792     else if ( StrICmp(name, (UBYTE *) "mctskonstant") == 0 ) {
06793         d = 0;
06794         if ( sscanf ((char*)value, "%lf", &d) != 1 ) {
06795             MesPrint("&Option MCTSKonstant in Format,Optimize statement should be a positive
number: %s",value);
06796             AO.Optimize.mctskonstant.fval = 0;
06797             error = 1;
06798         }
06799         else {
06800             AO.Optimize.mctskonstant.fval = d;
06801         }
06802     }
06803     else if ( StrICmp(name, (UBYTE *) "greedytimelimit") == 0 ) {
06804         x = 0;
06805         u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06806         if ( *u != 0 ) {
06807             MesPrint("&Option GreedyTimeLimit in Format,Optimize statement should be a positive
number: %s",value);
06808             AO.Optimize.greedytimelimit = 0;
06809             error = 1;
06810         }
06811         else {
06812             AO.Optimize.greedytimelimit = x;
06813         }
06814     }
06815     else if ( StrICmp(name, (UBYTE *) "greedyminnum") == 0 ) {
06816         x = 0;
06817         u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06818         if ( *u != 0 ) {
06819             MesPrint("&Option GreedyMinNum in Format,Optimize statement should be a positive
number: %s",value);
06820             AO.Optimize.greedyminnum= 0;
06821             error = 1;
06822         }
06823         else {
06824             AO.Optimize.greedyminnum= x;
06825         }
06826     }
06827     else if ( StrICmp(name, (UBYTE *) "greedymaxperc") == 0 ) {
06828         x = 0;
06829         u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06830         if ( *u != 0 ) {
06831             MesPrint("&Option GreedyMaxPerc in Format,Optimize statement should be a positive
number: %s",value);
06832             AO.Optimize.greedymaxperc= 0;
06833             error = 1;
06834         }
06835         else {
06836             AO.Optimize.greedymaxperc= x;
06837         }
06838     }
06839     else if ( StrICmp(name, (UBYTE *) "stats") == 0 ) {
06840         if ( StrICmp(value, (UBYTE *) "on") == 0 ) {
06841             AO.Optimize.printstats = 1;
06842         }
06843         else if ( StrICmp(value, (UBYTE *) "off") == 0 ) {
06844             AO.Optimize.printstats = 0;
06845         }
06846         else {
06847             AO.Optimize.printstats = 0;

```

```

06848         MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
06849         error = 1;
06850     }
06851 }
06852 else if ( StrICmp(name,(UBYTE *)"printscheme") == 0 ) {
06853     if ( StrICmp(value,(UBYTE *)"on") == 0 ) {
06854         AO.Optimize.schemeflags |= 1;
06855     }
06856     else if ( StrICmp(value,(UBYTE *)"off") == 0 ) {
06857         AO.Optimize.schemeflags &= ~1;
06858     }
06859     else {
06860         AO.Optimize.schemeflags &= ~1;
06861         MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
06862         error = 1;
06863     }
06864 }
06865 else if ( StrICmp(name,(UBYTE *)"debugflag") == 0 ) {
06866     /*
06867         This option is for debugging purposes only. Not in the manual!
06868         0x1: Print statements in reverse order.
06869         0x2: Print the scheme of the variables.
06870     */
06871     x = 0;
06872     u = value;
06873     if ( FG.cTable[*u] == 1 ) {
06874         while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06875         if ( *u != 0 ) {
06876             MesPrint("&Numerical value for DebugFlag in Format,Optimize statement should be a
nonnegative number: %s",value);
06877             AO.Optimize.debugflags = 0;
06878             error = 1;
06879         }
06880         else {
06881             AO.Optimize.debugflags = x;
06882         }
06883     }
06884     else if ( StrICmp(value,(UBYTE *)"on") == 0 ) {
06885         AO.Optimize.debugflags = 1;
06886     }
06887     else if ( StrICmp(value,(UBYTE *)"off") == 0 ) {
06888         AO.Optimize.debugflags = 0;
06889     }
06890     else {
06891         AO.Optimize.debugflags = 0;
06892         MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
06893         error = 1;
06894     }
06895 }
06896 else if ( StrICmp(name,(UBYTE *)"scheme") == 0 ) {
06897     UBYTE *ss, *s1, c;
06898     WORD type, numsym;
06899     AO.schemenum = 0;
06900     u = value;
06901     if ( *u != '(' ) {
06902         noscheme:
06903         MesPrint("&Option Scheme in Format,Optimize statement should be an array of names or
integers between (): %s",value);
06904         error = 1;
06905         break;
06906     }
06907     u++; ss = u;
06908     while ( *ss == ' ' || *ss == '\t' || *ss == ',' ) ss++;
06909     if ( FG.cTable[*ss] == 0 || *ss == '$' || *ss == '[' ) { /* Name */
06910         s1 = u; SKIPBRA3(s1)
06911         if ( *s1 != ')' ) goto noscheme;
06912         while ( ss < s1 ) { if ( *ss++ == ',' ) AO.schemenum++; }
06913         *ss++ = 0; while ( *ss == ' ' ) ss++;
06914         if ( *ss != 0 ) goto noscheme;
06915         ss = u;
06916         if ( AO.schemenum < 1 ) {
06917             MesPrint("&Option Scheme in Format,Optimize statement should have at least one
name or number between ()");
06918             error = 1;
06919             break;
06920         }
06921         if ( AO.inscheme ) M_free(AO.inscheme,"Horner input scheme");
06922         AO.inscheme = (WORD *)Malloc1((AO.schemenum+1)*sizeof(WORD),"Horner input scheme");
06923         while ( *ss == ' ' || *ss == '\t' || *ss == ',' ) ss++;
06924         AO.schemenum = 0;
06925         for(;;) {
06926             if ( *ss == 0 ) break;
06927             s1 = ss; ss = SkipAName(s1); c = *ss; *ss = 0;
06928
06929             if ( ss[-1] == '_' ) {
06930                 /*
06931                     Now AC.extrasym followed by a number and _

```

```

06932 */
06933     UBYTE *u1, *u2;
06934     u1 = s1; u2 = AC.extrasym;
06935     while ( *u1 == *u2 ) { u1++; u2++; }
06936     if ( *u2 == 0 ) { /* Good start */
06937         numsym = 0;
06938         while ( *u1 >= '0' && *u1 <= '9' ) numsym = 10*numsym + *u1++ - '0';
06939         if ( u1 != ss-1 || numsym == 0 || AC.extrasymbols != 0 ) {
06940             MesPrint("&Improper use of extra symbol in scheme format option");
06941             goto noscheme;
06942         }
06943         numsym = MAXVARIABLES-numsym;
06944         ss++;
06945         goto GotTheNumber;
06946     }
06947 }
06948 else if ( *s1 == '$' ) {
06949     GETIDENTITY
06950     int numdollar;
06951     if ( ( numdollar = GetDollar(s1+1) ) < 0 ) {
06952         MesPrint("&Undefined variable %s",s1);
06953         error = 5;
06954     }
06955     else if ( ( numsym = DolToSymbol(BHEAD numdollar) ) < 0 ) {
06956         MesPrint("&$$s does not evaluate to a symbol",s1);
06957         error = 5;
06958     }
06959     *ss = c;
06960     goto GotTheNumber;
06961 }
06962 else if ( c == '(' ) {
06963     if ( StrCmp(s1,AC.extrasym) == 0 ) {
06964         if ( (AC.extrasymbols&1) != 1 ) {
06965             MesPrint("&Improper use of extra symbol in scheme format option");
06966             goto noscheme;
06967         }
06968         *ss++ = c;
06969         numsym = 0;
06970         while ( *ss >= '0' && *ss <= '9' ) numsym = 10*numsym + *ss++ - '0';
06971         if ( *ss != ')' ) {
06972             MesPrint("&Extra symbol should have a number for its argument.");
06973             goto noscheme;
06974         }
06975         numsym = MAXVARIABLES-numsym;
06976         ss++;
06977         goto GotTheNumber;
06978     }
06979 }
06980 type = GetName(AC.varnames,s1,&numsym,WITHAUTO);
06981 if ( ( type != CSYMBOL ) && type != CDUBIOUS ) {
06982     MesPrint("&$$s is not a symbol",s1);
06983     error = 4;
06984     if ( type < 0 ) numsym = AddSymbol(s1,-MAXPOWER,MAXPOWER,0,0);
06985 }
06986 *ss = c;
06987 GotTheNumber:
06988     AO.inscheme[AO.schemenum++] = numsym;
06989     while ( *ss == ' ' || *ss == '\t' || *ss == ',' ) ss++;
06990 }
06991 }
06992 }
06993 else if ( StrICmp(name,(UBYTE *)"mctsdecaymode") == 0 ) {
06994     x = 0;
06995     u = value;
06996     if ( FG.cTable[*u] == 1 ) {
06997         while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
06998         if ( *u != 0 ) {
06999             MesPrint("&Option MCTSDecayMode in Format,Optimize statement should be a
nonnegative integer: %s",value);
07000             AO.Optimize.mctsdecaymode = 0;
07001             error = 1;
07002         }
07003         else {
07004             AO.Optimize.mctsdecaymode = x;
07005         }
07006     }
07007     else {
07008         AO.Optimize.mctsdecaymode = 0;
07009         MesPrint("&Unrecognized option value in Format,Optimize statement: %s=%s",name,value);
07010         error = 1;
07011     }
07012 }
07013 else if ( StrICmp(name,(UBYTE *)"saiter") == 0 ) {
07014     x = 0;
07015     u = value; while ( *u >= '0' && *u <= '9' ) x = 10*x + *u++ - '0';
07016     if ( *u != 0 ) {
07017         MesPrint("&Option SAIter in Format,Optimize statement should be a positive integer:

```

```

    %s",value);
07018         AO.Optimize.saIter = 0;
07019         error = 1;
07020     }
07021     else {
07022         AO.Optimize.saIter= x;
07023     }
07024 }
07025 else if ( StrICmp(name,(UBYTE *)"samaxt") == 0 ) {
07026     d = 0;
07027     if ( sscanf ((char*)value, "%lf", &d) != 1 ) {
07028         MesPrint("&Option SAMaxT in Format,Optimize statement should be a positive number:
    %s",value);
07029         AO.Optimize.saMaxT.fval = 0;
07030         error = 1;
07031     }
07032     else {
07033         AO.Optimize.saMaxT.fval = d;
07034     }
07035 }
07036 else if ( StrICmp(name,(UBYTE *)"samint") == 0 ) {
07037     d = 0;
07038     if ( sscanf ((char*)value, "%lf", &d) != 1 ) {
07039         MesPrint("&Option SAMinT in Format,Optimize statement should be a positive number:
    %s",value);
07040         AO.Optimize.saMinT.fval = 0;
07041         error = 1;
07042     }
07043     else {
07044         AO.Optimize.saMinT.fval = d;
07045     }
07046 }
07047 else {
07048     MesPrint("&Unrecognized option name in Format,Optimize statement: %s",name);
07049     error = 1;
07050 }
07051 *t1 = c1; *t2 = c2;
07052 }
07053 return(error);
07054 }
07055
07056 /*
07057 #] CoOptimizeOption :
07058 #[ DoPutInside :
07059
07060 Syntax:
07061     PutIn[side],functionname[,brackets] -> par = 1
07062     AntiPutIn[side],functionname,antibrackets -> par = -1
07063 */
07064
07065 int CoPutInside(UBYTE *inp) { return(DoPutInside(inp,1)); }
07066 int CoAntiPutInside(UBYTE *inp) { return(DoPutInside(inp,-1)); }
07067
07068 int DoPutInside(UBYTE *inp, int par)
07069 {
07070     GETIDENTITY
07071     UBYTE *p, c;
07072     WORD *to, type, c1,c2,funnum, *WorkSave;
07073     int error = 0;
07074     while ( *inp == ' ' || *inp == '\t' || *inp == ',' ) inp++;
07075 /*
07076     First we need the name of a function. (Not a tensor or table!)
07077 */
07078     p = SkipAName(inp);
07079     if ( p == 0 ) return(1);
07080     c = *p; *p = 0;
07081     type = GetName(AC.varnames,inp,&funnum,WITHAUTO);
07082     if ( type != CFUNCTION || functions[funnum].tabl != 0 || functions[funnum].spec ) {
07083         MesPrint("&PutInside/AntiPutInside expects a regular function for its first argument");
07084         MesPrint("&Argument is %s",inp);
07085         error = 1;
07086     }
07087     funnum += FUNCTION;
07088     *p = c;
07089     inp = p;
07090     while ( *inp == ' ' || *inp == '\t' || *inp == ',' ) inp++;
07091     if ( *inp == 0 ) {
07092         if ( par == 1 ) {
07093             WORD tocompiler[4];
07094             tocompiler[0] = TYPEPUTINSIDE;
07095             tocompiler[1] = 4;
07096             tocompiler[2] = 0;
07097             tocompiler[3] = funnum;
07098             AddNtoL(4,tocompiler);
07099         }
07100         else {
07101             MesPrint("&AntiPutInside needs inside information.");

```

```

07102         error = 1;
07103     }
07104     return(error);
07105 }
07106 WorkSave = to = AT.WorkPointer;
07107 *to++ = TYPEPUTINSIDE;
07108 *to++ = 4;
07109 *to++ = par;
07110 *to++ = funnum;
07111 to++;
07112 while ( *inp ) {
07113     while ( *inp == ' ' || *inp == '\t' || *inp == ',' ) inp++;
07114     if ( *inp == 0 ) break;
07115     p = SkipAName(inp);
07116     if ( p == 0 ) { error = 1; break; }
07117     c = *p; *p = 0;
07118     type = GetName(AC.varnames,inp,&c1,WITHAUTO);
07119     if ( c == '.' ) {
07120         if ( type == CVECTOR || type == CDUBIOUS ) {
07121             *p++ = c;
07122             inp = p;
07123             p = SkipAName(inp);
07124             if ( p == 0 ) return(1);
07125             c = *p; *p = 0;
07126             type = GetName(AC.varnames,inp,&c2,WITHAUTO);
07127             if ( type != CVECTOR && type != CDUBIOUS ) {
07128                 MesPrint("&Not a vector in dotproduct in PutInside/AntiPutInside statement:
%s",inp);
07129                 error = 1;
07130             }
07131             else type = CDOTPRODUCT;
07132         }
07133         else {
07134             MesPrint("&Illegal use of . after %s in PutInside/AntiPutInside statement",inp);
07135             error = 1;
07136             *p = c; inp = p;
07137             continue;
07138         }
07139     }
07140     switch ( type ) {
07141         case CSYMBOL :
07142             *to++ = SYMBOL; *to++ = 4; *to++ = c1; *to++ = 1; break;
07143         case CVECTOR :
07144             *to++ = INDEX; *to++ = 3; *to++ = AM.OffsetVector + c1; break;
07145         case CFUNCTION :
07146             *to++ = c1+FUNCTION; *to++ = FUNHEAD; *to++ = 0;
07147             FILLFUN3(to)
07148             break;
07149         case CDOTPRODUCT :
07150             *to++ = DOTPRODUCT; *to++ = 5; *to++ = c1 + AM.OffsetVector;
07151             *to++ = c2 + AM.OffsetVector; *to++ = 1; break;
07152         case CDELTA :
07153             *to++ = DELTA; *to++ = 4; *to++ = EMPTYINDEX; *to++ = EMPTYINDEX; break;
07154         default :
07155             MesPrint("&Illegal variable request for %s in PutInside/AntiPutInside statement",inp);
07156             error = 1; break;
07157     }
07158     *p = c;
07159     inp = p;
07160 }
07161 *to++ = 1; *to++ = 1; *to++ = 3;
07162 AT.WorkPointer[1] = to - AT.WorkPointer;
07163 AT.WorkPointer[4] = AT.WorkPointer[1]-4;
07164 AT.WorkPointer = to;
07165 AC.BraceNormalize = 1;
07166 if ( Normalize(BHEAD WorkSave+4) ) { error = 1; }
07167 else {
07168     WorkSave[1] = WorkSave[4]+4;
07169     to = WorkSave + WorkSave[1] - 1;
07170     c1 = ABS(*to);
07171     WorkSave[1] -= c1;
07172     WorkSave[4] -= c1;
07173     AddNtoL(WorkSave[1],WorkSave);
07174 }
07175 AC.BraceNormalize = 0;
07176 AT.WorkPointer = WorkSave;
07177 return(error);
07178 }
07179
07180 /*
07181 #] DoPutInside :
07182 #[ CoSwitch :
07183
07184 Syntax: Switch $var;
07185 Be careful with illegal nestings with repeat, if, while.
07186 */
07187

```

```

07188 int CoSwitch(UBYTE *s)
07189 {
07190     WORD numdollar;
07191     SWITCH *sw;
07192     if ( *s == '$' ) {
07193         if ( GetName(AC.dollarnames,s+1,&numdollar,NOAUTO) != CDOLLAR ) {
07194             MesPrint("%s is undefined in switch statement",s);
07195             numdollar = AddDollar(s+1,DOLINDEX,&one,1);
07196             return(1);
07197         }
07198         s = SkipAName(s+1);
07199         if ( *s != 0 ) {
07200             MesPrint("&Switch should have a single $variable for its argument");
07201             return(1);
07202         }
07203         /*      AddPotModdollar(numdollar);      */
07204     }
07205     else {
07206         MesPrint("%s is not a $-variable in switch statement",s);
07207         return(1);
07208     }
07209     /*
07210     Now create the switch table. We will add to it each time we run
07211     into a new case. It will all be sorted out the moment we run into
07212     the endswitch statement.
07213     */
07214     AC.SwitchLevel++;
07215     if ( AC.SwitchInArray >= AC.MaxSwitch ) DoubleSwitchBuffers();
07216     AC.SwitchHeap[AC.SwitchLevel] = AC.SwitchInArray;
07217     sw = AC.SwitchArray + AC.SwitchInArray;
07218
07219     sw->iflevel = AC.IfLevel;
07220     sw->whilelevel = AC.WhileLevel;
07221     sw->nestingsum = NestingChecksum();
07222
07223     Add4Com(TYPESWITCH,numdollar,AC.SwitchInArray);
07224
07225     AC.SwitchInArray++;
07226     return(0);
07227 }
07228
07229 /*
07230 #] CoSwitch :
07231 #[ CoCase :
07232 */
07233
07234 int CoCase(UBYTE *s)
07235 {
07236     SWITCH *sw = AC.SwitchArray + AC.SwitchHeap[AC.SwitchLevel];
07237     WORD x = 0, sign = 1;
07238     while ( *s == ',' ) s++;
07239     SKIPBLANKS(s);
07240     while ( *s == '-' || *s == '+' ) {
07241         if ( *s == '-' ) sign = -sign;
07242         s++;
07243     }
07244     while ( FG.cTable[*s] == 1 ) { x = 10*x + *s++ - '0'; }
07245     x = sign*x;
07246
07247     if ( sw->iflevel != AC.IfLevel || sw->whilelevel != AC.WhileLevel
07248         || sw->nestingsum != NestingChecksum() ) {
07249         MesPrint("%Illegal nesting of switch/case/default with
07250 if/while/repeat/loop/argument/term/...");
07251         return(-1);
07252     }
07253     /*
07254     Now add a case to the table with the current 'address'.
07255     */
07256     if ( sw->numcases >= sw->tablesize ) {
07257         int i;
07258         SWITCHTABLE *newtable;
07259         WORD newsize;
07260         if ( sw->tablesize == 0 ) newsize = 10;
07261         else newsize = 2*sw->tablesize;
07262         newtable = (SWITCHTABLE *)Malloc1(newsize*sizeof(SWITCHTABLE),"Switch table");
07263         if ( sw->table ) {
07264             for ( i = 0; i < sw->tablesize; i++ ) newtable[i] = sw->table[i];
07265             M_free(sw->table,"Switch table");
07266         }
07267         sw->table = newtable;
07268         sw->tablesize = newsize;
07269     }
07270     if ( sw->numcases == 0 ) { sw->mincase = sw->maxcase = x; }
07271     else if ( x > sw->maxcase ) sw->maxcase = x;
07272     else if ( x < sw->mincase ) sw->mincase = x;
07273     sw->table[sw->numcases].ncase = x;
07274     sw->table[sw->numcases].value = cbuf[AC.cbufnum].numlhs;

```



```

07274     sw->table[sw->numcases].compbuffer = AC.cbufnum;
07275     sw->numcases++;
07276     return(0);
07277 }
07278
07279 /*
07280  #] CoCase :
07281  #[ CoBreak :
07282 */
07283
07284 int CoBreak(UBYTE *s)
07285 {
07286     /*
07287      This involves a 'postponed' jump to the end. This can be done
07288      in a special routine during execution.
07289      That routine should also pop the switch level.
07290     */
07291     SWITCH *sw = AC.SwitchArray + AC.SwitchHeap[AC.SwitchLevel];
07292     if ( sw->iflevel != AC.IfLevel || sw->whilelevel != AC.WhileLevel
07293         || sw->nestingsum != NestingChecksum() ) {
07294         MesPrint("&Illegal nesting of switch/case/default with
if/while/repeat/loop/argument/term/...");
07295         return(-1);
07296     }
07297     if ( *s ) {
07298         MesPrint("&No parameters allowed in Break statement");
07299         return(-1);
07300     }
07301     Add3Com(TYPEENDSWITCH,AC.SwitchHeap[AC.SwitchLevel]);
07302     return(0);
07303 }
07304
07305 /*
07306  #] CoBreak :
07307  #[ CoDefault :
07308 */
07309
07310 int CoDefault(UBYTE *s)
07311 {
07312     /*
07313      A bit like case, except that the address gets stored directly in the
07314      SWITCH struct.
07315     */
07316     SWITCH *sw = AC.SwitchArray + AC.SwitchHeap[AC.SwitchLevel];
07317     if ( sw->iflevel != AC.IfLevel || sw->whilelevel != AC.WhileLevel
07318         || sw->nestingsum != NestingChecksum() ) {
07319         MesPrint("&Illegal nesting of switch/case/default with
if/while/repeat/loop/argument/term/...");
07320         return(-1);
07321     }
07322     if ( *s ) {
07323         MesPrint("&No parameters allowed in Default statement");
07324         return(-1);
07325     }
07326     sw->defaultcase.ncase = 0;
07327     sw->defaultcase.value = cbuf[AC.cbufnum].numlhs;
07328     sw->defaultcase.compbuffer = AC.cbufnum;
07329     return(0);
07330 }
07331
07332 /*
07333  #] CoDefault :
07334  #[ CoEndSwitch :
07335 */
07336
07337 int CoEndSwitch(UBYTE *s)
07338 {
07339     /*
07340      We store this address in the SWITCH struct.
07341      Next we sort the table by ncase.
07342      Then we decide whether the table is DENSE or SPARSE.
07343      If it is dense we change the allocation and spread the cases is necessary.
07344      Finally we pop levels.
07345     */
07346     SWITCH *sw = AC.SwitchArray + AC.SwitchHeap[AC.SwitchLevel];
07347     WORD i;
07348     WORD totcases = sw->maxcase-sw->mincase+1;
07349     while ( *s == ',' ) s++;
07350     SKIPBLANKS(s)
07351     if ( *s ) {
07352         MesPrint("&No parameters allowed in EndSwitch statement");
07353         return(-1);
07354     }
07355     if ( sw->iflevel != AC.IfLevel || sw->whilelevel != AC.WhileLevel
07356         || sw->nestingsum != NestingChecksum() ) {
07357         MesPrint("&Illegal nesting of switch/case/default with
if/while/repeat/loop/argument/term/...");

```

```

07358     return(-1);
07359 }
07360 if ( sw->defaultcase.value == 0 ) CoDefault(s);
07361 if ( totcases > sw->numcases*AM.jumpratio ) { /* The factor is experimental */
07362     sw->caseoffset = 0;
07363     sw->typetable = SPARSETABLE;
07364 /*
07365     Now we need to sort sw->table
07366 */
07367     SwitchSplitMerge(sw->table,sw->numcases);
07368 }
07369 else { /* DENSE */
07370     SWITCHTABLE *ntable;
07371     sw->caseoffset = sw->mincase;
07372     sw->typetable = DENSETABLE;
07373     ntable = (SWITCHTABLE *)Malloc1(totcases*sizeof(SWITCHTABLE),"Switch table");
07374     for ( i = 0; i < totcases; i++ ) {
07375         ntable[i].ncase = i+sw->caseoffset;
07376         ntable[i].value = sw->defaultcase.value;
07377         ntable[i].compbuffer = sw->defaultcase.compbuffer;
07378     }
07379     for ( i = 0; i < sw->numcases; i++ ) {
07380         ntable[sw->table[i].ncase-sw->caseoffset] = sw->table[i];
07381     }
07382     M_free(sw->table,"Switch table");
07383     sw->table = ntable;
07384     sw->numcases = totcases;
07385 }
07386 sw->endswitch.ncase = 0;
07387 sw->endswitch.value = cbuf[AC.cbunum].numlhs;
07388 sw->endswitch.compbuffer = AC.cbunum;
07389 if ( sw->defaultcase.value == 0 ) {
07390     sw->defaultcase = sw->endswitch;
07391 }
07392 Add3Com(TYPEENDSWITCH,AC.SwitchHeap[AC.SwitchLevel]);
07393 /*
07394     Now we need to pop.
07395 */
07396 AC.SwitchLevel--;
07397 return(0);
07398 }
07399
07400 /*
07401     #] CoEndSwitch :
07402     #[ CoSetUserFlag :
07403 */
07404
07405 int CoSetUserFlag(UBYTE *s)
07406 {
07407     int error = 0;
07408     while ( *s && ( *s == ',' || *s == ' ' || *s == '\t' ) ) s++;
07409     while ( *s && ( FG.cTable[*s] == 1 ) ) {
07410         int x = 0;
07411         while ( *s && ( FG.cTable[*s] == 1 ) ) x = 10*x+(s++ - '0');
07412         if ( x < 1 || x > BITSINWORD ) {
07413             MesPrint("&Flag number %d outside the permitted range 1-%d.",BITSINWORD);
07414             error = 1;
07415         }
07416         else {
07417             Add3Com(TYPESETUSERFLAG,x-1);
07418         }
07419         while ( *s && ( *s == ',' || *s == ' ' || *s == '\t' ) ) s++;
07420     }
07421     if ( *s ) {
07422         MesPrint("&Illegal character in SetUserFlag statement: %s",s);
07423         error = 1;
07424     }
07425     return(error);
07426 }
07427
07428 /*
07429     #] CoSetUserFlag :
07430     #[ CoClearUserFlag :
07431 */
07432
07433 int CoClearUserFlag(UBYTE *s)
07434 {
07435     int error = 0;
07436     while ( *s && ( *s == ',' || *s == ' ' || *s == '\t' ) ) s++;
07437     while ( *s && ( FG.cTable[*s] == 1 ) ) {
07438         int x = 0;
07439         while ( *s && ( FG.cTable[*s] == 1 ) ) x = 10*x+(s++ - '0');
07440         if ( x < 1 || x > BITSINWORD ) {
07441             MesPrint("&Flag number %d outside the permitted range 1-%d.",BITSINWORD);
07442             error = 1;
07443         }
07444         else {

```

```

07445         Add3Com(TYPECLEARUSERFLAG,x);
07446     }
07447     while ( *s && ( *s == ',' || *s == ' ' || *s == '\t' ) ) s++;
07448 }
07449 if ( *s ) {
07450     MesPrint("&Illegal character in SetUserFlag statement: %s",s);
07451     error = 1;
07452 }
07453 return(error);
07454 }
07455
07456 /*
07457 #] CoClearUserFlag :
07458 #[ CoCreateAllLoops :
07459
07460 Syntax:
07461     CoCreateAllLoops,in-function,out-function,type-argument,ifnolooop;
07462 Types allowed:
07463     vector, index, symbol, snumber
07464 ifnolooop:
07465     ifnolooop=0 or ifnolooop=1
07466 in-function can be a tensor. In that case type can be only vector or index.
07467 out-function can be a tensor. In that case type can be only vector or index.
07468 */
07469
07470 int CoCreateAllLoops(UBYTE *s)
07471 {
07472     GETIDENTITY
07473     UBYTE *iname, *outname, *stype, c;
07474     WORD infun, outfun, x, type, tensorflag, typenum;
07475     WORD *WorkSave, *to;
07476     while ( *s == ',' || *s == ' ' ) s++;
07477     iname = s; s = SkipAName(s);
07478     c = *s; *s = 0;
07479     if ( ( ( type = GetName(AC.varnames,inname,&infun,WITHAUTO) ) != CFUNCTION )
07480 || ( ( functions[infun].spec != 0 ) && ( functions[infun].spec != TENSORFUNCTION ) ) ) ) {
07481         MesPrint("&%s should be a regular function or a tensor.",iname);
07482         if ( type < 0 ) {
07483             if ( GetName(AC.exprnames,s,&infun,NOAUTO) == NAMENOTFOUND )
07484                 AddFunction(s,0,0,0,0,0,-1,-1);
07485         }
07486         return(1);
07487     }
07488     infun += FUNCTION;
07489     *s++ = c;
07490     while ( *s == ',' || *s == ' ' ) s++;
07491     outname = s; s = SkipAName(s);
07492     c = *s; *s = 0;
07493     if ( ( ( type = GetName(AC.varnames,outname,&outfun,WITHAUTO) ) != CFUNCTION )
07494 || ( ( functions[outfun].spec != 0 ) && ( functions[outfun].spec != TENSORFUNCTION ) ) ) ) {
07495         MesPrint("&%s should be a regular function or a tensor.",outname);
07496         if ( type < 0 ) {
07497             if ( GetName(AC.exprnames,s,&outfun,NOAUTO) == NAMENOTFOUND )
07498                 AddFunction(s,0,0,0,0,0,-1,-1);
07499         }
07500         return(1);
07501     }
07502     outfun += FUNCTION;
07503     *s++ = c;
07504     if ( functions[infun].spec == TENSORFUNCTION ||
07505         functions[outfun].spec == TENSORFUNCTION ) tensorflag = 1;
07506     else tensorflag = 0;
07507 /*
07508 Now the type: type=...
07509 */
07510     while ( *s == ',' || *s == ' ' ) s++;
07511     stype = s;
07512     while ( FG.cTable[*s] == 0 ) s++;
07513     c = *s; *s = 0;
07514     if ( StrICmp(stype,(UBYTE *)"type") != 0 || c != '=' ) {
07515         MesPrint("&In CreateAllLoops statement: expected type=vartype.");
07516         return(1);
07517     }
07518     *s++ = c;
07519     stype = s;
07520     while ( FG.cTable[*s] == 0 ) s++;
07521     c = *s; *s = 0;
07522     if ( StrICmp(stype,(UBYTE *)"vector") == 0 ) {
07523         typenum = -VECTOR;
07524     }
07525     else if ( StrICmp(stype,(UBYTE *)"index") == 0 ) {
07526         typenum = -INDEX;
07527     }
07528     else if ( StrICmp(stype,(UBYTE *)"symbol") == 0 ) {
07529         if ( tensorflag ) goto notintensor;
07530         typenum = -SYMBOL;
07531     }

```

```

07532     else if ( StrICmp(stype,(UBYTE *)"snumber") == 0 ) {
07533         if ( tensorflag ) goto notintensor;
07534         typenum = -SNUMBER;
07535     }
07536     else {
07537         MesPrint("&Unknown/not allowed variable type in CreateAllLoops: %s",stype);
07538         return(1);
07539     }
07540     *s = c;
07541     while ( *s == ',' || *s == ' ' ) s++;
07542
07543     stype = s;
07544     while ( FG.cTable[*s] == 0 ) s++;
07545     c = *s; *s = 0;
07546     if ( StrICmp(stype,(UBYTE *)"ifnolooop") != 0 || c != '=' ) {
07547         MesPrint("&Unrecognised option in CreateAllLoops statement: %s",stype);
07548         return(1);
07549     }
07550     *s++ = c;
07551     x = -1;
07552     if ( FG.cTable[*s] == 1 ) {
07553         x = 0;
07554         do { x = 10*x + (*s++-'0'); } while (FG.cTable[*s] == 1);
07555     }
07556     if ( x != 0 && x != 1 ) {
07557         MesPrint("&Only options allowed for ifnolooop are 0 or 1.");
07558         return(1);
07559     }
07560     WorkSave = to = AT.WorkPointer;
07561     *to++ = TYPEALLLOOPS;
07562     *to++ = 6;
07563     *to++ = infun;
07564     *to++ = outfun;
07565     *to++ = typenum;
07566     *to++ = x;
07567
07568     AddNtoL(WorkSave[1],WorkSave);
07569
07570     return(0);
07571 notintensor:
07572     MesPrint("&Variable type not allowed in tensors: %s",stype);
07573     return(1);
07574 }
07575
07576 /*
07577 #] CoCreateAllLoops :
07578 #[ CoCreateAllPaths :
07579
07580 Syntax:
07581     CreateAllPaths,end-function,intermediate-function,out-function,type-argument,ifnopath;
07582 Types allowed:
07583     vector, index, symbol, snumber
07584 ifnolooop:
07585     ifnolooop=0 or ifnolooop=1
07586 in-function can be a tensor. In that case type can be only vector or index.
07587 out-function can be a tensor. In that case type can be only vector or index.
07588 */
07589
07590 int CoCreateAllPaths(UBYTE *s)
07591 {
07592     GETIDENTITY
07593     UBYTE *endname,*inname,*outname,*stype,c;
07594     WORD endfun, infun, outfun, x, type, tensorflag, typenum;
07595     WORD *WorkSave,*to;
07596     while ( *s == ',' || *s == ' ' ) s++;
07597     endname = s; s = SkipAName(s);
07598     c = *s; *s = 0;
07599     if ( ( ( type = GetName(AC.varnames,endname,&endfun,WITHAUTO) ) != CFUNCTION )
07600 || ( ( functions[endfun].spec != 0 ) && ( functions[endfun].spec != TENSORFUNCTION ) ) ) {
07601         MesPrint("&%s should be a regular function or a tensor.",endname);
07602         if ( type < 0 ) {
07603             if ( GetName(AC.exprnames,s,&endfun,NOAUTO) == NAMENOTFOUND )
07604                 AddFunction(s,0,0,0,0,0,-1,-1);
07605         }
07606         return(1);
07607     }
07608     endfun += FUNCTION;
07609     *s++ = c;
07610     while ( *s == ',' || *s == ' ' ) s++;
07611     inname = s; s = SkipAName(s);
07612     c = *s; *s = 0;
07613     if ( ( ( type = GetName(AC.varnames,inname,&infun,WITHAUTO) ) != CFUNCTION )
07614 || ( ( functions[infun].spec != 0 ) && ( functions[infun].spec != TENSORFUNCTION ) ) ) {
07615         MesPrint("&%s should be a regular function or a tensor.",inname);
07616         if ( type < 0 ) {
07617             if ( GetName(AC.exprnames,s,&infun,NOAUTO) == NAMENOTFOUND )
07618                 AddFunction(s,0,0,0,0,0,-1,-1);

```

```

07619     }
07620     return(1);
07621 }
07622 infun += FUNCTION;
07623 *s++ = c;
07624 while ( *s == ',' || *s == ' ' ) s++;
07625 outname = s; s = SkipAName(s);
07626 c = *s; *s = 0;
07627 if ( ( type = GetName(AC.varnames,outname,&outfun,WITHAUTO) ) != CFUNCTION )
07628 || ( ( functions[outfun].spec != 0 ) && ( functions[outfun].spec != TENSORFUNCTION ) ) ) {
07629     MesPrint("%s should be a regular function or a tensor.",outname);
07630     if ( type < 0 ) {
07631         if ( GetName(AC.exprnames,s,&outfun,NOAUTO) == NAMENOTFOUND )
07632             AddFunction(s,0,0,0,0,0,-1,-1);
07633     }
07634     return(1);
07635 }
07636 outfun += FUNCTION;
07637 *s++ = c;
07638 if ( functions[infun].spec == TENSORFUNCTION ||
07639     functions[outfun].spec == TENSORFUNCTION ) tensorflag = 1;
07640 else tensorflag = 0;
07641 /*
07642 Now the type: type=....
07643 */
07644 while ( *s == ',' || *s == ' ' ) s++;
07645 stype = s;
07646 while ( FG.cTable[*s] == 0 ) s++;
07647 c = *s; *s = 0;
07648 if ( StrICmp(stype,(UBYTE *)"type") != 0 || c != '=' ) {
07649     MesPrint("&In CreateAllPaths statement: expected type=vartype.");
07650     return(1);
07651 }
07652 *s++ = c;
07653 stype = s;
07654 while ( FG.cTable[*s] == 0 ) s++;
07655 c = *s; *s = 0;
07656 if ( StrICmp(stype,(UBYTE *)"vector") == 0 ) {
07657     typenum = -VECTOR;
07658 }
07659 else if ( StrICmp(stype,(UBYTE *)"index") == 0 ) {
07660     typenum = -INDEX;
07661 }
07662 else if ( StrICmp(stype,(UBYTE *)"symbol") == 0 ) {
07663     if ( tensorflag ) goto notintensor;
07664     typenum = -SYMBOL;
07665 }
07666 else if ( StrICmp(stype,(UBYTE *)"snumber") == 0 ) {
07667     if ( tensorflag ) goto notintensor;
07668     typenum = -SNUMBER;
07669 }
07670 else {
07671     MesPrint("%&Unknown/not allowed variable type in CreateAllPaths: %s",stype);
07672     return(1);
07673 }
07674 *s = c;
07675 while ( *s == ',' || *s == ' ' ) s++;
07676
07677 stype = s;
07678 while ( FG.cTable[*s] == 0 ) s++;
07679 c = *s; *s = 0;
07680 if ( StrICmp(stype,(UBYTE *)"ifnopath") != 0 || c != '=' ) {
07681     MesPrint("%&Unrecognised option in CreateAllPaths statement: %s",stype);
07682     return(1);
07683 }
07684 *s++ = c;
07685 x = -1;
07686 if ( FG.cTable[*s] == 1 ) {
07687     x = 0;
07688     do { x = 10*x + (*s++-'0'); } while (FG.cTable[*s] == 1);
07689 }
07690 if ( x != 0 && x != 1 ) {
07691     MesPrint("%&Only options allowed for ifnopath are 0 or 1.");
07692     return(1);
07693 }
07694 WorkSave = to = AT.WorkPointer;
07695 *to++ = TYPEALLPATHS;
07696 *to++ = 7;
07697 *to++ = endfun;
07698 *to++ = infun;
07699 *to++ = outfun;
07700 *to++ = typenum;
07701 *to++ = x;
07702
07703 AddNtoL(WorkSave[1],WorkSave);
07704
07705 return(0);

```

```

07706 notintensor:
07707     MesPrint("&CreateAllPaths: Variable type not allowed in tensors: %s",stype);
07708     return(1);
07709 }
07710
07711 /*
07712 #] CoCreateAllPaths :
07713 #[ CoCreateAll :
07714
07715 Syntax: subkey, two or three functions, type, ifnone
07716 */
07717
07718 int CoCreateAll(UBYTE *s)
07719 {
07720     UBYTE *subkey;
07721     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
07722     subkey = s;
07723     while ( FG.cTable[*s] == 0 ) s++;
07724     if ( *s != ' ' && *s != ',' && *s != '\t' ) {
07725         MesPrint("&Illegal subkey in CoCreate statement.");
07726         return(1);
07727     }
07728     /* c = *s; */ *s++ = 0;
07729     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
07730     if ( StrICmp(subkey,(UBYTE *)"loops") == 0 ) {
07731         return(CoCreateAllLoops(s));
07732     }
07733     else if ( StrICmp(subkey,(UBYTE *)"paths") == 0 ) {
07734         return(CoCreateAllPaths(s));
07735     }
07736     /*
07737     else if ( StrICmp(subkey,(UBYTE *)"motics") == 0 ) {
07738     }
07739     else if ( StrICmp(subkey,(UBYTE *)"onepi") == 0 ) {
07740     }
07741     else if ( StrICmp(subkey,(UBYTE *)"cuts") == 0 ) {
07742     }
07743     */
07744     else {
07745         MesPrint("&Illegal subkey in CoCreate statement: %s.",subkey);
07746         return(1);
07747     }
07748 }
07749
07750 /*
07751 #] CoCreateAll :
07752 */

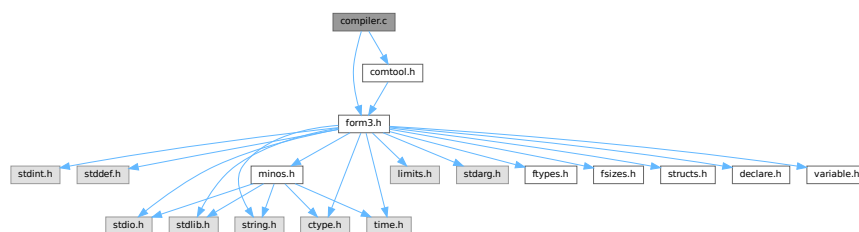
```

4.11 compiler.c File Reference

```

#include "form3.h"
#include "comtool.h"
Include dependency graph for compiler.c:

```



Data Structures

- struct [SuBbUf](#)

Macros

- `#define` [OPTION0](#) 1
- `#define` [OPTION1](#) 2
- `#define` [OPTION2](#) 3
- `#define` [REDUCESUBEXPBUFFERS](#)

Typedefs

- `typedef struct` [SuBbUf](#) **SUBBUF**

Functions

- `void` [inictable](#) (void)
- `KEYWORD *` [findcommand](#) (UBYTE *in)
- `int` [ParenthesesTest](#) (UBYTE *sin)
- `UBYTE *` [SkipAName](#) (UBYTE *s)
- `UBYTE *` [IsRHS](#) (UBYTE *s, UBYTE c)
- `int` [IsIdStatement](#) (UBYTE *s)
- `int` [CompileAlgebra](#) (UBYTE *s, int leftright, WORD *prototype)
- `int` [CompileStatement](#) (UBYTE *in)
- `int` [TestTables](#) (void)
- `int` [CompileSubExpressions](#) (SBYTE *tokens)
- `int` [CodeGenerator](#) (SBYTE *tokens)
- `int` [CompleteTerm](#) (WORD *term, UWORD *numer, UWORD *denom, WORD nnum, WORD nden, int sign)
- `int` [CodeFactors](#) (SBYTE *tokens)
- `WORD` [GenerateFactors](#) (WORD n, WORD inc)

Variables

- `int` [alfatable1](#) [27]
- `SUBBUF *` [subexpbuffers](#) = 0
- `SUBBUF *` [topsubexpbuffers](#) = 0
- `LONG` [insubexpbuffers](#) = 0

4.11.1 Detailed Description

The heart of the compiler. It contains the tables of statements. It finds the statements in the tables and calls the proper routines. For algebraic expressions it runs the compilation by first calling the tokenizer, splitting things into subexpressions and generating the code. There is a system for recognizing already existing subexpressions. This economizes on the length of the output.

Note: the compiler of FORM doesn't attempt to normalize the input. Hence $x+1$ and $1+x$ are different objects during compilation. Similarly $(a+b-b)$ will not be simplified to (a) .

Definition in file [compiler.c](#).

4.11.2 Macro Definition Documentation

4.11.2.1 OPTION0

```
#define OPTION0 1
```

Definition at line 260 of file [compiler.c](#).

4.11.2.2 OPTION1

```
#define OPTION1 2
```

Definition at line 261 of file [compiler.c](#).

4.11.2.3 OPTION2

```
#define OPTION2 3
```

Definition at line 262 of file [compiler.c](#).

4.11.2.4 REDUCESUBEXPBUFFERS

```
#define REDUCESUBEXPBUFFERS
```

Value:

```
{ if ( (topsubexpbuffers-subexpbuffers) > 256 ) {\
  M_free(subexpbuffers,"subexpbuffers");\
  subexpbuffers = (SUBBUF *)Malloc1(256*sizeof(SUBBUF),"subexpbuffers");\
  topsubexpbuffers = subexpbuffers+256; } insubexpbuffers = 0; }
```

Definition at line 273 of file [compiler.c](#).

4.11.3 Function Documentation

4.11.3.1 inictable()

```
void inictable (
    void )
```

Definition at line 315 of file [compiler.c](#).

4.11.3.2 findcommand()

```
KEYWORD * findcommand (
    UBYTE * in )
```

Definition at line 341 of file [compiler.c](#).

4.11.3.3 ParenthesesTest()

```
int ParenthesesTest (
    UBYTE * sin )
```

Definition at line 385 of file [compiler.c](#).

4.11.3.4 SkipAName()

```
UBYTE * SkipAName (
    UBYTE * s )
```

Skips a name and gives a pointer to the character after the name. If there is not a proper name, it emits a compiler message and then returns a null pointer. This function supports formal names (e.g., [*x+a*]) and \$-variables in addition to ordinary identifiers.

Note

The brackets are assumed to be matched already.

Parameters

<i>in</i>	<i>s</i>	Pointer to a null-terminated input string that starts with a name.

Returns

Pointer to the character after the name, or a null pointer if the name is invalid.

Definition at line 443 of file [compiler.c](#).

4.11.3.5 IsRHS()

```
UBYTE * IsRHS (
    UBYTE * s,
    UBYTE c )
```

Definition at line 476 of file [compiler.c](#).

4.11.3.6 IsIdStatement()

```
int IsIdStatement (
    UBYTE * s )
```

Definition at line 522 of file [compiler.c](#).

4.11.3.7 CompileAlgebra()

```
int CompileAlgebra (
    UBYTE * s,
    int leftright,
    WORD * prototype )
```

Definition at line 536 of file [compiler.c](#).

4.11.3.8 CompileStatement()

```
int CompileStatement (
    UBYTE * in )
```

Definition at line 572 of file [compiler.c](#).

4.11.3.9 TestTables()

```
int TestTables (
    void )
```

Definition at line 706 of file [compiler.c](#).

4.11.3.10 CompileSubExpressions()

```
int CompileSubExpressions (
    SBYTE * tokens )
```

Definition at line 750 of file [compiler.c](#).

4.11.3.11 CodeGenerator()

```
int CodeGenerator (
    SBYTE * tokens )
```

Definition at line 885 of file [compiler.c](#).

4.11.3.12 CompleteTerm()

```
int CompleteTerm (
    WORD * term,
    UWORD * numer,
    UWORD * denom,
    WORD nnum,
    WORD nden,
    int sign )
```

Definition at line 1948 of file [compiler.c](#).

4.11.3.13 CodeFactors()

```
int CodeFactors (
    SBYTE * tokens )
```

Definition at line [1985](#) of file [compiler.c](#).

4.11.3.14 GenerateFactors()

```
WORD GenerateFactors (
    WORD n,
    WORD inc )
```

Definition at line [2282](#) of file [compiler.c](#).

4.11.4 Variable Documentation

4.11.4.1 alfatable1

```
int alfatable1[27]
```

Definition at line [258](#) of file [compiler.c](#).

4.11.4.2 subexpbuffers

```
SUBBUF* subexpbuffers = 0
```

Definition at line [269](#) of file [compiler.c](#).

4.11.4.3 topsubexpbuffers

```
SUBBUF* topsubexpbuffers = 0
```

Definition at line [270](#) of file [compiler.c](#).

4.11.4.4 insubexpbuffers

```
LONG insubexpbuffers = 0
```

Definition at line [271](#) of file [compiler.c](#).

4.12 compiler.c

[Go to the documentation of this file.](#)

```

00001
00015 /* #[] License : */
00016 /*
00017 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00018 *   When using this file you are requested to refer to the publication
00019 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00020 *   This is considered a matter of courtesy as the development was paid
00021 *   for by FOM the Dutch physics granting agency and we would like to
00022 *   be able to track its scientific use to convince FOM of its value
00023 *   for the community.
00024 *
00025 *   This file is part of FORM.
00026 *
00027 *   FORM is free software: you can redistribute it and/or modify it under the
00028 *   terms of the GNU General Public License as published by the Free Software
00029 *   Foundation, either version 3 of the License, or (at your option) any later
00030 *   version.
00031 *
00032 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00033 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00034 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00035 *   details.
00036 *
00037 *   You should have received a copy of the GNU General Public License along
00038 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00039 */
00040 /* #[] License : */
00041 /*
00042 *   #[] includes :
00043 */
00044
00045 #include "form3.h"
00046 #include "comtool.h"
00047
00048 /*
00049 *   com1commands are the commands of which only part of the word has to
00050 *   be present. The order is rather important here.
00051 *   com2commands are the commands that must have their whole word match.
00052 *   here we can do a binary search.
00053 *   {{{
00054 */
00055
00056 static KEYWORD com1commands[] = {
00057     {"also", (TFUN) CoIdOld, STATEMENT, PARTEST}
00058     , {"abracquets", (TFUN) CoAntiBracket, TOOUTPUT, PARTEST}
00059     , {"antisymmetrize", (TFUN) CoAntiSymmetrize, STATEMENT, PARTEST}
00060     , {"antibrackets", (TFUN) CoAntiBracket, TOOUTPUT, PARTEST}
00061     , {"brackets", (TFUN) CoBracket, TOOUTPUT, PARTEST}
00062     , {"cfunctions", (TFUN) CoCFunction, DECLARATION, PARTEST|WITHAUTO}
00063     , {"commuting", (TFUN) CoCFunction, DECLARATION, PARTEST|WITHAUTO}
00064     , {"compress", (TFUN) CoCompress, DECLARATION, PARTEST}
00065     , {"ctensors", (TFUN) CoCTensor, DECLARATION, PARTEST|WITHAUTO}
00066     , {"cyclesymmetrize", (TFUN) CoCycleSymmetrize, STATEMENT, PARTEST}
00067     , {"dimension", (TFUN) CoDimension, DECLARATION, PARTEST}
00068     , {"discard", (TFUN) CoDiscard, STATEMENT, PARTEST}
00069     , {"functions", (TFUN) CoNFunction, DECLARATION, PARTEST|WITHAUTO}
00070     , {"format", (TFUN) CoFormat, TOOUTPUT, PARTEST}
00071     , {"fixindex", (TFUN) CoFixIndex, DECLARATION, PARTEST}
00072     , {"global", (TFUN) CoGlobal, DEFINITION, PARTEST}
00073     , {"gfactorized", (TFUN) CoGlobalFactorized, DEFINITION, PARTEST}
00074     , {"globalfactorized", (TFUN) CoGlobalFactorized, DEFINITION, PARTEST}
00075     , {"goto", (TFUN) CoGoTo, STATEMENT, PARTEST}
00076     , {"indexes", (TFUN) CoIndex, DECLARATION, PARTEST|WITHAUTO}
00077     , {"indices", (TFUN) CoIndex, DECLARATION, PARTEST|WITHAUTO}
00078     , {"identify", (TFUN) CoId, STATEMENT, PARTEST}
00079     , {"idnew", (TFUN) CoIdNew, STATEMENT, PARTEST}
00080     , {"idold", (TFUN) CoIdOld, STATEMENT, PARTEST}
00081     , {"local", (TFUN) CoLocal, DEFINITION, PARTEST}
00082     , {"lfactorized", (TFUN) CoLocalFactorized, DEFINITION, PARTEST}
00083     , {"localfactorized", (TFUN) CoLocalFactorized, DEFINITION, PARTEST}
00084     , {"load", (TFUN) CoLoad, DECLARATION, PARTEST}
00085     , {"label", (TFUN) CoLabel, STATEMENT, PARTEST}
00086     , {"modulus", (TFUN) CoModulus, DECLARATION, PARTEST}
00087     , {"multiply", (TFUN) CoMultiply, STATEMENT, PARTEST}
00088     , {"nfunctions", (TFUN) CoNFunction, DECLARATION, PARTEST|WITHAUTO}
00089     , {"nprint", (TFUN) CoNPrint, TOOUTPUT, PARTEST}
00090     , {"ntensors", (TFUN) CoNTensor, DECLARATION, PARTEST|WITHAUTO}
00091     , {"nwrite", (TFUN) CoNWrite, DECLARATION, PARTEST}
00092     , {"print", (TFUN) CoPrint, MIXED, 0}
00093     , {"redefine", (TFUN) CoRedefine, STATEMENT, 0}
00094     , {"rcyclesymmetrize", (TFUN) CoRCycleSymmetrize, STATEMENT, PARTEST}
00095     , {"symbols", (TFUN) CoSymbol, DECLARATION, PARTEST|WITHAUTO}

```

```

00096     ,{"save",                (TFUN) CoSave,                DECLARATION, PARTEST}
00097     ,{"symmetrize",          (TFUN) CoSymmetrize,          STATEMENT,   PARTEST}
00098     ,{"tensors",             (TFUN) CoTensor,             DECLARATION, PARTEST|WITHAUTO}
00099     ,{"unittrace",           (TFUN) CoUnitTrace,          DECLARATION, PARTEST}
00100     ,{"vectors",             (TFUN) CoVector,            DECLARATION, PARTEST|WITHAUTO}
00101     ,{"write",               (TFUN) CoWrite,             DECLARATION, PARTEST}
00102 };
00103
00104 static KEYWORD com2commands[] = {
00105     {"antiputinside",        (TFUN) CoAntiPutInside,     STATEMENT,   PARTEST}
00106     ,{"apply",               (TFUN) CoApply,            STATEMENT,   PARTEST}
00107     ,{"aputinside",          (TFUN) CoAntiPutInside,     STATEMENT,   PARTEST}
00108     ,{"argexplode",          (TFUN) CoArgExplode,        STATEMENT,   PARTEST}
00109     ,{"argimplode",          (TFUN) CoArgImplode,        STATEMENT,   PARTEST}
00110     ,{"argtoextrasymbol",    (TFUN) CoArgToExtraSymbol,  STATEMENT,   PARTEST}
00111     ,{"argument",            (TFUN) CoArgument,          STATEMENT,   PARTEST}
00112     ,{"assign",              (TFUN) CoAssign,            STATEMENT,   PARTEST}
00113     ,{"auto",                (TFUN) CoAuto,              DECLARATION, PARTEST}
00114     ,{"autodeclare",         (TFUN) CoAuto,              DECLARATION, PARTEST}
00115     ,{"break",               (TFUN) CoBreak,             STATEMENT,   PARTEST}
00116     ,{"canonicalize",        (TFUN) CoCanonicalize,      STATEMENT,   PARTEST}
00117     ,{"case",                (TFUN) CoCase,              STATEMENT,   PARTEST}
00118     ,{"chainin",             (TFUN) CoChainin,           STATEMENT,   PARTEST}
00119     ,{"chainout",            (TFUN) CoChainout,          STATEMENT,   PARTEST}
00120     ,{"chisholm",            (TFUN) CoChisholm,          STATEMENT,   PARTEST}
00121     #ifdef WITHFLOAT
00122     ,{"chop",                (TFUN) CoChop,              STATEMENT,   PARTEST}
00123     #endif
00124     ,{"clearflag",           (TFUN) CoClearUserFlag,      STATEMENT,   PARTEST}
00125     ,{"cleartable",          (TFUN) CoClearTable,         DECLARATION, PARTEST}
00126     ,{"collect",             (TFUN) CoCollect,            SPECIFICATION, PARTEST}
00127     ,{"commuteinset",         (TFUN) CoCommuteInSet,       DECLARATION, PARTEST}
00128     ,{"contract",            (TFUN) CoContract,           STATEMENT,   PARTEST}
00129     ,{"copyspectator",       (TFUN) CoCopySpectator,      DEFINITION,   PARTEST}
00130     ,{"createall",           (TFUN) CoCreateAll,          STATEMENT,   PARTEST}
00131     ,{"createspectator",     (TFUN) CoCreateSpectator,    DECLARATION, PARTEST}
00132     ,{"ctable",              (TFUN) CoCTable,            DECLARATION, PARTEST}
00133     ,{"deallocatetable",     (TFUN) CoDeallocateTable,    DECLARATION, PARTEST}
00134     ,{"default",             (TFUN) CoDefault,           STATEMENT,   PARTEST}
00135     ,{"delete",              (TFUN) CoDelete,            SPECIFICATION, PARTEST}
00136     ,{"denominators",        (TFUN) CoDenominators,       STATEMENT,   PARTEST}
00137     ,{"disorder",            (TFUN) CoDisorder,           STATEMENT,   PARTEST}
00138     ,{"do",                  (TFUN) CoDo,                STATEMENT,   PARTEST}
00139     ,{"drop",                (TFUN) CoDrop,              SPECIFICATION, PARTEST}
00140     ,{"dropcoefficient",     (TFUN) CoDropCoefficient,    STATEMENT,   PARTEST}
00141     ,{"dropsymbols",         (TFUN) CoDropSymbols,        STATEMENT,   PARTEST}
00142     ,{"else",                (TFUN) CoElse,              STATEMENT,   PARTEST}
00143     ,{"elseif",              (TFUN) CoElseIf,            STATEMENT,   PARTEST}
00144     ,{"emptyspectator",      (TFUN) CoEmptySpectator,     SPECIFICATION, PARTEST}
00145     ,{"endargument",         (TFUN) CoEndArgument,        STATEMENT,   PARTEST}
00146     ,{"enddo",               (TFUN) CoEndDo,             STATEMENT,   PARTEST}
00147     ,{"endif",               (TFUN) CoEndIf,             STATEMENT,   PARTEST}
00148     ,{"endinexpression",     (TFUN) CoEndInExpression,    STATEMENT,   PARTEST}
00149     ,{"endinside",           (TFUN) CoEndInside,          STATEMENT,   PARTEST}
00150     ,{"endmodel",            (TFUN) CoEndModel,           DECLARATION, PARTEST}
00151     ,{"endrepeat",           (TFUN) CoEndRepeat,          STATEMENT,   PARTEST}
00152     ,{"endswitch",           (TFUN) CoEndSwitch,          STATEMENT,   PARTEST}
00153     ,{"endterm",             (TFUN) CoEndTerm,            STATEMENT,   PARTEST}
00154     ,{"endwhile",            (TFUN) CoEndWhile,           STATEMENT,   PARTEST}
00155     #ifdef WITHFLOAT
00156     ,{"evaluate",            (TFUN) CoEvaluate,           STATEMENT,   PARTEST}
00157     #endif
00158     ,{"exit",                (TFUN) CoExit,              STATEMENT,   PARTEST}
00159     ,{"extrasymbols",        (TFUN) CoExtraSymbols,       DECLARATION, PARTEST}
00160     ,{"factarg",             (TFUN) CoFactArg,            STATEMENT,   PARTEST}
00161     ,{"factdollar",          (TFUN) CoFactDollar,         STATEMENT,   PARTEST}
00162     ,{"factorize",           (TFUN) CoFactorize,          TOOUTPUT,    PARTEST}
00163     ,{"fill",                (TFUN) CoFill,              DECLARATION, PARTEST}
00164     ,{"fillexpression",       (TFUN) CoFillExpression,     DECLARATION, PARTEST}
00165     ,{"frompolynomial",       (TFUN) CoFromPolynomial,     STATEMENT,   PARTEST}
00166     ,{"funpowers",           (TFUN) CoFunPowers,          DECLARATION, PARTEST}
00167     ,{"hide",                (TFUN) CoHide,              SPECIFICATION, PARTEST}
00168     ,{"if",                  (TFUN) CoIf,                STATEMENT,   PARTEST}
00169     ,{"ifmatch",             (TFUN) CoIfMatch,            STATEMENT,   PARTEST}
00170     ,{"ifnomatch",           (TFUN) CoIfNoMatch,          STATEMENT,   PARTEST}
00171     ,{"ifnotmatch",          (TFUN) CoIfNoMatch,          STATEMENT,   PARTEST}
00172     ,{"inexpression",        (TFUN) CoInExpression,       STATEMENT,   PARTEST}
00173     ,{"inparallel",          (TFUN) CoInParallel,         SPECIFICATION, PARTEST}
00174     ,{"inside",              (TFUN) CoInside,            STATEMENT,   PARTEST}
00175     ,{"insidefirst",         (TFUN) CoInsideFirst,        DECLARATION, PARTEST}
00176     ,{"intohide",            (TFUN) CoIntoHide,           SPECIFICATION, PARTEST}
00177     ,{"keep",                (TFUN) CoKeep,              SPECIFICATION, PARTEST}
00178     ,{"makeinteger",          (TFUN) CoMakeInteger,        STATEMENT,   PARTEST}
00179     ,{"many",                (TFUN) CoMany,              STATEMENT,   PARTEST}
00180     ,{"merge",               (TFUN) CoMerge,             STATEMENT,   PARTEST}
00181     ,{"metric",              (TFUN) CoMetric,            DECLARATION, PARTEST}
00182     ,{"model",               (TFUN) CoModel,             DECLARATION, PARTEST}

```

```

00183     ,{"moduleoption",      (TFUN) CoModuleOption,    ATENDOFMODULE, PARTEST}
00184     ,{"multi",             (TFUN) CoMulti,          STATEMENT,    PARTEST}
00185     ,{"multibracket",      (TFUN) CoMultiBracket,  STATEMENT,    PARTEST}
00186     ,{"ndrop",             (TFUN) CoNoDrop,          SPECIFICATION, PARTEST}
00187     ,{"nfactorize",        (TFUN) CoNFactorize,    TOOUTPUT,     PARTEST}
00188     ,{"nhide",             (TFUN) CoNoHide,          SPECIFICATION, PARTEST}
00189     ,{"nintohide",         (TFUN) CoNoIntoHide,    SPECIFICATION, PARTEST}
00190     ,{"normalize",          (TFUN) CoNormalize,    STATEMENT,    PARTEST}
00191     ,{"notinparallel",     (TFUN) CoNotInParallel, SPECIFICATION, PARTEST}
00192     ,{"nskip",             (TFUN) CoNoSkip,        SPECIFICATION, PARTEST}
00193     ,{"ntable",            (TFUN) CoNTable,        DECLARATION,  PARTEST}
00194     ,{"nunfactorize",      (TFUN) CoUnFactorize,   TOOUTPUT,     PARTEST}
00195     ,{"nunhide",           (TFUN) CoNoUnHide,     SPECIFICATION, PARTEST}
00196     ,{"off",               (TFUN) CoOff,           DECLARATION,  PARTEST}
00197     ,{"on",                (TFUN) CoOn,            DECLARATION,  PARTEST}
00198     ,{"once",              (TFUN) CoOnce,          STATEMENT,    PARTEST}
00199     ,{"only",              (TFUN) CoOnly,          STATEMENT,    PARTEST}
00200     ,{"particle",          (TFUN) CoParticle,      DECLARATION,  PARTEST}
00201     ,{"polyfun",           (TFUN) CoPolyFun,       DECLARATION,  PARTEST}
00202     ,{"polyratfun",        (TFUN) CoPolyRatFun,    DECLARATION,  PARTEST}
00203     ,{"pophide",           (TFUN) CoPopHide,       SPECIFICATION, PARTEST}
00204     ,{"print[]",           (TFUN) CoPrintB,        TOOUTPUT,     PARTEST}
00205     ,{"printtable",        (TFUN) CoPrintTable,    MIXED,        PARTEST}
00206     ,{"processbucketsize", (TFUN) CoProcessBucket, DECLARATION,  PARTEST}
00207     ,{"propercount",       (TFUN) CoProperCount,   DECLARATION,  PARTEST}
00208     ,{"pushhide",          (TFUN) CoPushHide,     SPECIFICATION, PARTEST}
00209     ,{"putinside",         (TFUN) CoPutInside,    STATEMENT,    PARTEST}
00210     ,{"ratio",             (TFUN) CoRatio,         STATEMENT,    PARTEST}
00211     ,{"removespectator",   (TFUN) CoRemoveSpectator, SPECIFICATION, PARTEST}
00212     ,{"renumber",          (TFUN) CoReNUMBER,      STATEMENT,    PARTEST}
00213     ,{"repeat",            (TFUN) CoRepeat,        STATEMENT,    PARTEST}
00214     ,{"replaceloop",       (TFUN) CoReplaceLoop,   STATEMENT,    PARTEST}
00215     ,{"select",            (TFUN) CoSelect,        STATEMENT,    PARTEST}
00216     ,{"set",               (TFUN) CoSet,           DECLARATION,  PARTEST}
00217     ,{"setexitflag",       (TFUN) CoSetExitFlag,   STATEMENT,    PARTEST}
00218     ,{"setflag",           (TFUN) CoSetUserFlag,   STATEMENT,    PARTEST}
00219     ,{"shuffle",           (TFUN) CoMerge,         STATEMENT,    PARTEST}
00220     ,{"skip",              (TFUN) CoSkip,          SPECIFICATION, PARTEST}
00221     ,{"sort",              (TFUN) CoSort,          STATEMENT,    PARTEST}
00222     ,{"splitarg",          (TFUN) CoSplitArg,      STATEMENT,    PARTEST}
00223     ,{"splitfirstarg",     (TFUN) CoSplitFirstArg, STATEMENT,    PARTEST}
00224     ,{"splitlastarg",      (TFUN) CoSplitLastArg,  STATEMENT,    PARTEST}
00225 #ifdef WITHFLOAT
00226     ,{"strictrounding",    (TFUN) CoStrictRounding, STATEMENT,    PARTEST}
00227 #endif
00228     ,{"stuffle",           (TFUN) CoStuffle,       STATEMENT,    PARTEST}
00229     ,{"sum",               (TFUN) CoSum,           STATEMENT,    PARTEST}
00230     ,{"switch",            (TFUN) CoSwitch,        STATEMENT,    PARTEST}
00231     ,{"table",             (TFUN) CoTable,         DECLARATION,  PARTEST}
00232     ,{"tablebase",         (TFUN) CoTableBase,     DECLARATION,  PARTEST}
00233     ,{"tb",                (TFUN) CoTableBase,     DECLARATION,  PARTEST}
00234     ,{"term",              (TFUN) CoTerm,          STATEMENT,    PARTEST}
00235     ,{"testuse",           (TFUN) CoTestUse,       STATEMENT,    PARTEST}
00236     ,{"threadbucketsize",  (TFUN) CoThreadBucket,  DECLARATION,  PARTEST}
00237 #ifdef WITHFLOAT
00238     ,{"tofloat",           (TFUN) CoToFloat,        STATEMENT,    PARTEST}
00239 #endif
00240     ,{"topolynomial",      (TFUN) CoToPolynomial,   STATEMENT,    PARTEST}
00241 #ifdef WITHFLOAT
00242     ,{"torat",             (TFUN) CoToRat,          STATEMENT,    PARTEST}
00243     ,{"torational",        (TFUN) CoToRat,          STATEMENT,    PARTEST}
00244 #endif
00245     ,{"tospectator",       (TFUN) CoToSpectator,   STATEMENT,    PARTEST}
00246     ,{"totensor",          (TFUN) CoToTensor,      STATEMENT,    PARTEST}
00247     ,{"tovector",          (TFUN) CoToVector,      STATEMENT,    PARTEST}
00248     ,{"trace4",            (TFUN) CoTrace4,        STATEMENT,    PARTEST}
00249     ,{"tracen",            (TFUN) CoTraceN,        STATEMENT,    PARTEST}
00250     ,{"transform",         (TFUN) CoTransform,     STATEMENT,    PARTEST}
00251     ,{"tryreplace",        (TFUN) CoTryReplace,    STATEMENT,    PARTEST}
00252     ,{"unfactorize",       (TFUN) CoUnFactorize,   TOOUTPUT,     PARTEST}
00253     ,{"unhide",            (TFUN) CoUnHide,        SPECIFICATION, PARTEST}
00254     ,{"vertex",            (TFUN) CoVertex,        DECLARATION,  PARTEST}
00255     ,{"while",             (TFUN) CoWhile,         STATEMENT,    PARTEST}
00256 };
00257
00258 int alfatable1[27];
00259
00260 #define OPTION0 1
00261 #define OPTION1 2
00262 #define OPTION2 3
00263
00264 typedef struct SuBbUf {
00265     WORD    subexpnum;
00266     WORD    buffernum;
00267 } SUBBUF;
00268
00269 SUBBUF *subexpbuffers = 0;

```

```

00270 SUBBUF *topsubexpbuffers = 0;
00271 LONG insubexpbuffers = 0;
00272
00273 #define REDUCESUBEXPBUFFERS { if ( (topsubexpbuffers-subexpbuffers) > 256 ) {\
00274     M_free(subexpbuffers,"subexpbuffers");\
00275     subexpbuffers = (SUBBUF *)Malloc1(256*sizeof(SUBBUF),"subexpbuffers");\
00276     topsubexpbuffers = subexpbuffers+256; } insubexpbuffers = 0; }
00277
00278 #if BITSINWORD == 32
00279     #define PUTNUMBER128(t,n) { if ( n >= 2097152 ) { \
00280         *t++ = ((n/128)/128)/128; *t++ = ((n/128)/128)%128; *t++ = (n/128)%128; *t++ = n%128;
00281     } \
00282         else if ( n >= 16384 ) { \
00283             *t++ = n/(128*128); *t++ = (n/128)%128; *t++ = n%128; } \
00284             else if ( n >= 128 ) { *t++ = n/128; *t++ = n%128; } \
00285             else *t++ = n; }
00286     #define PUTNUMBER100(t,n) { if ( n >= 1000000 ) { \
00287         *t++ = ((n/100)/100)/100; *t++ = ((n/100)/100)%100; *t++ = (n/100)%100; *t++ = n%100;
00288     } \
00289         else if ( n >= 10000 ) { \
00290             *t++ = n/10000; *t++ = (n/100)%100; *t++ = n%100; } \
00291             else if ( n >= 100 ) { *t++ = n/100; *t++ = n%100; } \
00292             else *t++ = n; }
00293 #elif BITSINWORD == 16
00294     #define PUTNUMBER128(t,n) { if ( n >= 16384 ) { \
00295         *t++ = n/(128*128); *t++ = (n/128)%128; *t++ = n%128; } \
00296         else if ( n >= 128 ) { *t++ = n/128; *t++ = n%128; } \
00297         else *t++ = n; }
00298     #define PUTNUMBER100(t,n) { if ( n >= 10000 ) { \
00299         *t++ = n/10000; *t++ = (n/100)%100; *t++ = n%100; } \
00300         else if ( n >= 100 ) { *t++ = n/100; *t++ = n%100; } \
00301         else *t++ = n; }
00302 #else
00303     #error Only 64-bit and 32-bit platforms are supported.
00304 #endif
00305 /*
00306     #] includes :
00307     #[ Compiler :
00308     #[ inictable :
00309
00310     Routine sets the table for 1-st characters that allow a faster
00311     start in the search in table 1 which should be sequential.
00312     Search in table 2 can be binary.
00313 */
00314 void inictable(void)
00315 {
00316     KEYWORD *k = com1commands;
00317     int i, j, ksize;
00318     ksize = sizeof(com1commands)/sizeof(KEYWORD);
00319     j = 0;
00320     alfatable1[0] = 0;
00321     for ( i = 0; i < 26; i++ ) {
00322         while ( j < ksize && k[j].name[0] == 'a'+i ) j++;
00323         alfatable1[i+1] = j;
00324     }
00325 }
00326 }
00327
00328 /*
00329     #] inictable :
00330     #[ findcommand :
00331
00332     Checks whether a command is in the command table.
00333     If so a pointer to the table element is returned.
00334     If not we return 0.
00335     Note that when a command is not in the table, we have
00336     to test whether it is an id command without id. It should
00337     then have the structure pattern = rhs. This should be done
00338     in the calling routine.
00339 */
00340 KEYWORD *findcommand(UBYTE *in)
00341 {
00342     int hi, med, lo, i;
00343     UBYTE *s, c;
00344     s = in;
00345     while ( FG.cTable[*s] <= 1 ) s++;
00346     if ( s > in && *s == '[' && s[1] == ']' ) s += 2;
00347     if ( *s ) { c = *s; *s = 0; }
00348     else c = 0;
00349
00350     /*
00351     First do a binary search in the second table
00352     */
00353     lo = 0;
00354     hi = sizeof(com2commands)/sizeof(KEYWORD)-1;

```

```

00355     do {
00356         med = ( hi + lo ) / 2;
00357         i = StrICmp(in, (UBYTE *)com2commands[med].name);
00358         if ( i == 0 ) { if ( c ) *s = c; return(com2commands+med); }
00359         if ( i < 0 ) hi = med-1;
00360         else lo = med+1;
00361     } while ( hi >= lo );
00362 /*
00363     Now do a 'hashed' search in the first table. It is sequential.
00364 */
00365     i = tolower(*in) - 'a';
00366     med = alfatable1[i];
00367     hi = alfatable1[i+1];
00368     while ( med < hi ) {
00369         if ( StrICont(in, (UBYTE *)com1commands[med].name) == 0 )
00370             { if ( c ) *s = c; return(com1commands+med); }
00371         med++;
00372     }
00373     if ( c ) *s = c;
00374 /*
00375     Unrecognized. Too bad!
00376 */
00377     return(0);
00378 }
00379
00380 /*
00381     #] findcommand :
00382     #[ ParenthesesTest :
00383 */
00384
00385 int ParenthesesTest(UBYTE *sin)
00386 {
00387     WORD L1 = 0, L2 = 0, L3 = 0;
00388     UBYTE *s = sin;
00389     while ( *s ) {
00390         if ( *s == '[' ) L1++;
00391         else if ( *s == ']' ) {
00392             L1--;
00393             if ( L1 < 0 ) { MesPrint("&Unmatched []"); return(1); }
00394         }
00395         s++;
00396     }
00397     if ( L1 > 0 ) { MesPrint("&Unmatched []"); return(1); }
00398     s = sin;
00399     while ( *s ) {
00400         if ( *s == '[' ) SKIPBRA1(s)
00401         else if ( *s == '(' ) { L2++; s++; }
00402         else if ( *s == ')' ) {
00403             L2--; s++;
00404             if ( L2 < 0 ) { MesPrint("&Unmatched ()"); return(1); }
00405         }
00406         else s++;
00407     }
00408     if ( L2 > 0 ) { MesPrint("&Unmatched ()"); return(1); }
00409     s = sin;
00410     while ( *s ) {
00411         if ( *s == '[' ) SKIPBRA1(s)
00412         else if ( *s == '[' ) SKIPBRA4(s)
00413         else if ( *s == '{' ) { L3++; s++; }
00414         else if ( *s == '}' ) {
00415             L3--; s++;
00416             if ( L3 < 0 ) { MesPrint("&Unmatched {}"); return(1); }
00417         }
00418         else s++;
00419     }
00420     if ( L3 > 0 ) { MesPrint("&Unmatched {}"); return(1); }
00421     return(0);
00422 }
00423 /*
00424     #] ParenthesesTest :
00425     #[ SkipAName :
00426 */
00427 /*
00428
00443 UBYTE *SkipAName(UBYTE *s)
00444 {
00445     UBYTE *t = s;
00446     if ( *s == '[' ) {
00447         SKIPBRA1(s)
00448     }
00449     /*
00450         In principle the brackets match already, so the `if ( *s == 0 )'
00451         code is not really needed, but you never know how the program
00452         is extended later.
00453     */
00454     if ( *s == 0 ) {
00455         MesPrint("&Illegal name: '%s'",t);
00456         return(0);

```



```

00456     }
00457     s++;
00458 }
00459 else if ( FG.cTable[*s] == 0 || *s == '_' || *s == '$' ) {
00460     if ( *s == '$' ) s++;
00461     while ( FG.cTable[*s] <= 1 ) s++;
00462     if ( *s == '_' ) s++;
00463 }
00464 else {
00465     MesPrint("&Illegal name: '%s'",t);
00466     return(0);
00467 }
00468 return(s);
00469 }
00470
00471 /*
00472     #] SkipAName :
00473     #[ IsRHS :
00474 */
00475
00476 UBYTE *IsRHS(UBYTE *s, UBYTE c)
00477 {
00478     while ( *s && *s != c ) {
00479         if ( *s == '[' ) {
00480             SKIPBRA1(s);
00481             if ( *s != ']' ) {
00482                 MesPrint("&Unmatched []");
00483                 return(0);
00484             }
00485         }
00486         else if ( *s == '{' ) {
00487             SKIPBRA2(s);
00488             if ( *s != '}' ) {
00489                 MesPrint("&Unmatched {}");
00490                 return(0);
00491             }
00492         }
00493         else if ( *s == '(' ) {
00494             SKIPBRA3(s);
00495             if ( *s != ')' ) {
00496                 MesPrint("&Unmatched ()");
00497                 return(0);
00498             }
00499         }
00500         else if ( *s == ')' ) {
00501             MesPrint("&Unmatched ()");
00502             return(0);
00503         }
00504         else if ( *s == '}' ) {
00505             MesPrint("&Unmatched {}");
00506             return(0);
00507         }
00508         else if ( *s == ']' ) {
00509             MesPrint("&Unmatched []");
00510             return(0);
00511         }
00512         s++;
00513     }
00514     return(s);
00515 }
00516
00517 /*
00518     #] IsRHS :
00519     #[ IsIdStatement :
00520 */
00521
00522 int IsIdStatement(UBYTE *s)
00523 {
00524     DUMMYUSE(s);
00525     return(0);
00526 }
00527
00528 /*
00529     #] IsIdStatement :
00530     #[ CompileAlgebra :
00531
00532     Returns either the number of the main level RHS (>= 0)
00533     or an error code (< 0)
00534 */
00535
00536 int CompileAlgebra(UBYTE *s, int leftright, WORD *prototype)
00537 {
00538     GETIDENTITY
00539     int error;
00540     WORD *oldproto = AC.ProtoType;
00541     AC.ProtoType = prototype;
00542     if ( AC.TokensWriteFlag ) {

```

```

00543     MesPrint("To tokenize: %s",s);
00544     error = tokenize(s,leftright);
00545     MesPrint(" The contents of the token buffer are:");
00546     WriteTokens(AC.tokens);
00547 }
00548 else error = tokenize(s,leftright);
00549 if ( error == 0 ) {
00550     AR.Eside = leftright;
00551     AC.CompileLevel = 0;
00552     if ( leftright == LHSIDE ) { AC.DumNum = AR.CurDum = 0; }
00553     error = CompileSubExpressions(AC.tokens);
00554     REDUCESUBEXPBUFFERS
00555 }
00556 else {
00557     AC.ProtoType = oldproto;
00558     return(-1);
00559 }
00560 AC.ProtoType = oldproto;
00561 if ( error < 0 ) return(-1);
00562 else if ( leftright == LHSIDE ) return(cbuf[AC.cbufnum].numlhs);
00563 else return(cbuf[AC.cbufnum].numrhs);
00564 }
00565
00566 /*
00567     #] CompileAlgebra :
00568     #[ CompileStatement :
00569
00570 */
00571
00572 int CompileStatement(UBYTE *in)
00573 {
00574     KEYWORD *k;
00575     UBYTE *s;
00576     int error1 = 0, error2;
00577     /* A.iStatement = */ s = in;
00578     if ( *s == 0 ) return(0);
00579     if ( *s == '$' ) {
00580         k = findcommand((UBYTE *) "assign");
00581     }
00582     else {
00583         if ( ( k = findcommand(s) ) == 0 && IsIdStatement(s) == 0 ) {
00584             MesPrint("&Unrecognized statement %s",s);
00585             return(1);
00586         }
00587         if ( k == 0 ) { /* Id statement without id. Note: id must be in table */
00588             k = comlcommands + alfatable1['i'-'a'];
00589             while ( k->name[1] != 'd' || k->name[2] ) k++;
00590         }
00591         else {
00592             while ( FG.cTable[*s] <= 1 ) s++;
00593             if ( s > in && *s == '[' && s[1] == ']' ) s += 2;
00594         }
00595         /*
00596             The next statement is rather mysterious
00597             It is undone in DoPrint and CoMultiply, but it also causes effects
00598             in other (wrong) statements like dimension -4; or Trace4 -1;
00599             The code in pre.c (LoadStatement) has been changed 8-sep-2009
00600             to force a comma after the keyword. This means that the
00601             'mysterious' line is automatically inactive. Hence it is taken out.
00602
00603             if ( *s == '+' || *s == '-' ) s++;
00604
00605             if ( *s == ',' ) s++;
00606         */
00607         /*
00608             First the test on the order of the statements.
00609             This is relatively new (2.2c) and may cause some problems with old
00610             programs. Hence the first error message should explain!
00611         */
00612         if ( AP.PreAssignFlag == 0 && AM.OldOrderFlag == 0 ) {
00613             if ( AP.PreInsideLevel ) {
00614                 if ( k->type != STATEMENT && k->type != MIXED ) {
00615                     MesPrint("&Only executable and print statements are allowed in an %inside/%#endinside
00616                     construction");
00617                     return(-1);
00618                 }
00619             }
00620             else {
00621                 if ( ( AC.compiletype == DECLARATION || AC.compiletype == SPECIFICATION )
00622                     && ( k->type == STATEMENT || k->type == DEFINITION || k->type == TOOUTPUT ) ) {
00623                     if ( AC.tablecheck == 0 ) {
00624                         AC.tablecheck = 1;
00625                         if ( TestTables() ) error1 = 1;
00626                     }
00627                 }
00628                 if ( k->type == MIXED ) {
00629                     if ( AC.compiletype <= DEFINITION ) {

```

```

00629         AC.compiletype = STATEMENT;
00630     }
00631 }
00632 else if ( k->type > AC.compiletype ) {
00633     /*
00634      * We intentionally do NOT update "compiletype" for:
00635      * - Format statements (type = TOOUTPUT)
00636      * - ModuleOption statements (type = ATENDOFMODULE)
00637      *   with sum/maximum/minimum/local (i.e., $-variable-related options)
00638      *
00639      * This relaxes the ordering constraint, allowing statements with
00640      * type >= the current "compiletype" to follow.
00641      */
00642     if ( StrCmp((UBYTE *) (k->name), (UBYTE *) "format") == 0 )
00643         goto NoUpdateCompileType;
00644     if ( StrCmp((UBYTE *) (k->name), (UBYTE *) "moduleoption") == 0 ) {
00645         UBYTE *ss = s;
00646         SkipSpaces(&ss);
00647         if ( ConsumeOption(&ss, "sum")
00648             || ConsumeOption(&ss, "maximum")
00649             || ConsumeOption(&ss, "minimum")
00650             || ConsumeOption(&ss, "local") ) goto NoUpdateCompileType;
00651     }
00652     AC.compiletype = k->type;
00653 NoUpdateCompileType:
00654     ;
00655 }
00656 else if ( k->type < AC.compiletype ) {
00657     switch ( k->type ) {
00658     case DECLARATION:
00659         MesPrint("&Declaration out of order");
00660         MesPrint("& %s", in);
00661         break;
00662     case DEFINITION:
00663         MesPrint("&Definition out of order");
00664         MesPrint("& %s", in);
00665         break;
00666     case SPECIFICATION:
00667         MesPrint("&Specification out of order");
00668         MesPrint("& %s", in);
00669         break;
00670     case STATEMENT:
00671         MesPrint("&Statement out of order");
00672         break;
00673     case TOOUTPUT:
00674         MesPrint("&Output control statement out of order");
00675         MesPrint("& %s", in);
00676         break;
00677     }
00678     AC.compiletype = k->type;
00679     if ( AC.firstctypemessage == 0 ) {
00680         MesPrint("&Proper order inside a module is:");
00681         MesPrint("Declarations, specifications, definitions, statements, output control
statements");
00682         AC.firstctypemessage = 1;
00683     }
00684     error1 = 1;
00685 }
00686 }
00687 }
00688 /*
00689  Now we execute the tests that are prescribed by the flags.
00690 */
00691 if ( AC.AutoDeclareFlag && ( ( k->flags & WITHAUTO ) == 0 ) ) {
00692     MesPrint("&Illegal type of auto-declaration");
00693     return(1);
00694 }
00695 if ( ( ( k->flags & PARTEST ) != 0 ) && ParenthesesTest(s) ) return(1);
00696 error2 = (*k->func)(s);
00697 if ( error2 == 0 ) return(error1);
00698 return(error2);
00699 }
00700
00701 /*
00702     #] CompileStatement :
00703     #[ TestTables :
00704 */
00705
00706 int TestTables(void)
00707 {
00708     FUNCTIONS f = functions;
00709     TABLES t;
00710     WORD j;
00711     int error = 0, i;
00712     LONG x;
00713     i = NumFunctions + FUNCTION - MAXBUILTINFUNCTION - 1;
00714     f = f + MAXBUILTINFUNCTION - FUNCTION + 1;

```

```

00715     if ( AC.MustTestTable > 0 ) {
00716         while ( i > 0 ) {
00717             if ( ( t = f->tabl ) != 0 && t->strict > 0 && !t->sparse ) {
00718                 for ( x = 0, j = 0; x < t->totind; x++ ) {
00719                     if ( t->tablepointers[TABLEEXTENSION*x] < 0 ) j++;
00720                 }
00721                 if ( j > 0 ) {
00722                     if ( j > 1 ) {
00723                         MesPrint("&In table %s there are %d unfilled elements",
00724                             AC.varnames->namebuffer+f->name, j);
00725                     }
00726                     else {
00727                         MesPrint("&In table %s there is one unfilled element",
00728                             AC.varnames->namebuffer+f->name);
00729                     }
00730                     error = 1;
00731                 }
00732             }
00733             i--; f++;
00734         }
00735         AC.MustTestTable--;
00736     }
00737     return(error);
00738 }
00739
00740 /*
00741     #] TestTables :
00742     #[ CompileSubExpressions :
00743
00744     Now we attack the subexpressions from inside out.
00745     We try to see whether we had any of them already.
00746     We have to worry about adding the wildcard sum parameter
00747     to the prototype.
00748 */
00749
00750 int CompileSubExpressions(SBYTE *tokens)
00751 {
00752     GETIDENTITY
00753     SBYTE *fill = tokens, *s = tokens, *t;
00754     WORD number[MAXNUMSIZE], *oldwork, *w1, *w2;
00755     int level, num, i, sumlevel = 0, sumtype = SYMTOSYM;
00756     int retval, error = 0;
00757 /*
00758     Eliminate all subexpressions. They are marked by LPARENTHESIS, RPARENTHESIS
00759 */
00760     AC.CompileLevel++;
00761     while ( *s != TENDOFIT ) {
00762         if ( *s == TFUNOPEN ) {
00763             if ( fill < s ) *fill = TENDOFIT;
00764             t = fill - 1;
00765             while ( t >= tokens && t[0] >= 0 ) t--;
00766             if ( t >= tokens && *t == TFUNCTION ) {
00767                 t++; i = 0; while ( *t >= 0 ) i = 128*i + *t++;
00768                 if ( i == AM.sumnum || i == AM.sumpnum ) {
00769                     t = s + 1;
00770                     if ( *t == TSymbol || *t == TINDEX ) {
00771                         t++; i = 0; while ( *t >= 0 ) i = 128*i + *t++;
00772                         if ( s[1] == TINDEX ) {
00773                             i += AM.OffsetIndex;
00774                             sumtype = INDTOIND;
00775                         }
00776                         else sumtype = SYMTOSYM;
00777                         sumlevel = i;
00778                     }
00779                 }
00780             }
00781             *fill++ = *s++;
00782         }
00783         else if ( *s == TFUNCLOSE ) { sumlevel = 0; *fill++ = *s++; }
00784         else if ( *s == LPARENTHESIS ) {
00785 /*
00786             We must make an exception here.
00787             If the subexpression is just an integer, whatever its length,
00788             we should try to keep it.
00789             This is important when we have a function with an integer
00790             argument. In particular this is relevant for the MZV program.
00791 */
00792             t = s; level = 0;
00793             while ( level >= 0 ) {
00794                 s++;
00795                 if ( *s == LPARENTHESIS ) level++;
00796                 else if ( *s == RPARENTHESIS ) level--;
00797                 else if ( *s == TENDOFIT ) {
00798                     MesPrint("&Unbalanced subexpression parentheses");
00799                     return(-1);
00800                 }
00801             }

```

```

00802         t++; *s = TENDOFIT;
00803     if ( sumlevel > 0 ) { /* Inside sum. Add wildcard to prototype */
00804         oldwork = w1 = AT.WorkPointer;
00805         w2 = AC.ProtoType;
00806         i = w2[1];
00807         while ( --i >= 0 ) *w1++ = *w2++;
00808         oldwork[1] += 4;
00809         *w1++ = sumtype; *w1++ = 4; *w1++ = sumlevel; *w1++ = sumlevel;
00810         w2 = AC.ProtoType; AT.WorkPointer = w1;
00811         AC.ProtoType = oldwork;
00812         num = CompileSubExpressions(t);
00813         AC.ProtoType = w2; AT.WorkPointer = oldwork;
00814     }
00815     else num = CompileSubExpressions(t);
00816     if ( num < 0 ) return(-1);
00817 /*
00818     Note that the subexpression code should always fit.
00819     We had two parentheses and at least two bytes contents.
00820     There cannot be more than 2^21 subexpressions or we get outside
00821     this minimum. Ignoring this might lead to really rare and
00822     hard to find errors, years from now.
00823 */
00824     if ( insubexpbuffers >= MAXSUBEXPRESSIONS ) {
00825         MesPrint("&More than %d subexpressions inside one
expression", (WORD)MAXSUBEXPRESSIONS);
00826         Terminate(-1);
00827     }
00828     if ( subexpbuffers+insubexpbuffers >= topsubexpbuffers ) {
00829         DoubleBuffer((void **) ((void *) (&subexpbuffers))
00830             , (void **) ((void *) (&topsubexpbuffers)), sizeof(SUBBUF), "subexpbuffers");
00831     }
00832     subexpbuffers[insubexpbuffers].subexpnum = num;
00833     subexpbuffers[insubexpbuffers].buffernum = AC.cbufnum;
00834     num = insubexpbuffers++;
00835     *fill++ = TSUBEXP;
00836     i = 0;
00837     do { number[i++] = num & 0x7F; num >>= 7; } while ( num );
00838     while ( --i >= 0 ) *fill++ = (SBYTE)(number[i]);
00839     s++;
00840 }
00841 else if ( *s == TEMPTY ) s++;
00842 else *fill++ = *s++;
00843 }
00844 *fill = TENDOFIT;
00845 /*
00846     At this stage there are no more subexpressions.
00847     Hence we can do the basic compilation.
00848 */
00849     if ( AC.CompileLevel == 1 && AC.ToBeInFactors ) {
00850         error = CodeFactors(tokens);
00851     }
00852     AC.CompileLevel--;
00853     retval = CodeGenerator(tokens);
00854     if ( error < 0 ) return(error);
00855     return(retval);
00856 }
00857
00858 /*
00859     #] CompileSubExpressions :
00860     #[ CodeGenerator :
00861
00862     This routine does the real code generation.
00863     It returns the number of the rhs subexpression.
00864     At this point we do not have to worry about subexpressions,
00865     sets, setelements, simple vs complicated function arguments
00866     simple vs complicated powers etc.
00867
00868     The variable 'first' indicates whether we are starting a new term
00869
00870     The major complication are the set elements of type set[n].
00871     We have marked them as TSETNUM,n,Ttype,setnum
00872     They go into
00873     SETSET,size,subterm,relocation list
00874     in which the subterm should be ready to become a regular
00875     subterm in which the sets have been replaced by their element
00876     The relocation list consists of pairs of numbers:
00877     1: offset in the subterm, 2: the symbol n.
00878     Note that such a subterm can be a whole function with its arguments.
00879     We use the variable inset to indicate that we have something going.
00880     The relocation list is collected in the top of the Workspace.
00881 */
00882
00883     static UWORD *CGscrat7 = 0;
00884
00885     int CodeGenerator(SBYTE *tokens)
00886     {
00887         GETIDENTITY

```

```

00888     SBYTE *s = tokens, c;
00889     int i, sign = 1, first = 1, deno = 1, error = 0, minus, n, needarg, numexp, cc;
00890     int base, sumlevel = 0, sumtype = SYMTOSYM, firstsumarg, inset = 0;
00891     int funflag = 0, settype, x1, x2, mulflag = 0;
00892     WORD *t, *v, *r, *term, nnumerator, ndenominator, *oldwork, x3, y, nin;
00893     WORD *w1, *w2, *tsize = 0, *relo = 0;
00894     UWORD *numerator, *denominator, *innum;
00895     CBUF *C;
00896     POSITION position;
00897     WORD TMproto[SUBEXPSIZE];
00898     /*
00899     #ifdef WITHPTHREADS
00900         RENUMBER renumber;
00901     #endif
00902     */
00903     RENUMBER renumber;
00904     if ( AC.TokensWriteFlag ) WriteTokens(tokens);
00905     if ( CGscrat7 == 0 )
00906         CGscrat7 = (UWORD *)Malloc1((AM.MaxTal+2)*sizeof(WORD),"CodeGenerator");
00907     AddrRHS(AC.cbunum,0);
00908     C = cbuf + AC.cbunum;
00909     numexp = C->numrhs;
00910     C->NumTerms[numexp] = 0;
00911     C->numdum[numexp] = 0;
00912     oldwork = AT.WorkPointer;
00913     numerator = (UWORD *) (AT.WorkPointer);
00914     denominator = numerator + 2*AM.MaxTal;
00915     innum = denominator + 2*AM.MaxTal;
00916     term = (WORD *) (innum + 2*AM.MaxTal);
00917     AT.WorkPointer = term + AM.MaxTer/sizeof(WORD);
00918     if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
00919     cc = 0;
00920     t = term+1;
00921     numerator[0] = denominator[0] = 1;
00922     nnumerator = ndenominator = 1;
00923     while ( *s != TENDOFIT ) {
00924         if ( *s == TPLUS || *s == TMINUS ) {
00925             if ( first || mulflag ) { if ( *s == TMINUS ) sign = -sign; }
00926             else {
00927                 *term = t-term;
00928                 C->NumTerms[numexp]++;
00929                 if ( cc && sign ) C->CanCommu[numexp]++;
00930                 CompleteTerm(term,numerator,denominator,nnumerator,ndenominator,sign);
00931                 first = 1; cc = 0; t = term + 1; deno = 1;
00932                 numerator[0] = denominator[0] = 1;
00933                 nnumerator = ndenominator = 1;
00934                 if ( *s == TMINUS ) sign = -1;
00935                 else sign = 1;
00936             }
00937             s++;
00938         }
00939         else {
00940             mulflag = first = 0; c = *s++;
00941             switch ( c ) {
00942                 case TSYMBOL:
00943                     x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
00944                     if ( *s == TWILDCARD ) { s++; x1 += 2*MAXPOWER; }
00945                     *t++ = SYMBOL; *t++ = 4; *t++ = x1;
00946                     if ( inset ) *relo = 2;
00947             TryPower: if ( *s == TPOWER ) {
00948                 s++;
00949                 if ( *s == TMINUS ) { s++; deno = -deno; }
00950                 c = *s++;
00951                 base = ( c == TNUMBER ) ? 100: 128;
00952                 x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
00953                 if ( c == TSYMBOL ) {
00954                     if ( *s == TWILDCARD ) s++;
00955                     x2 += 2*MAXPOWER;
00956                 }
00957                 *t++ = deno*x2;
00958             }
00959             else *t++ = deno;
00960             fin: deno = 1;
00961             if ( inset ) {
00962                 while ( relo < AT.WorkTop ) *t++ = *relo++;
00963                 inset = 0; tsize[1] = t - tsize;
00964             }
00965             break;
00966             case TINDEX:
00967                 x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
00968                 *t++ = INDEX; *t++ = 3;
00969                 if ( *s == TWILDCARD ) { s++; x1 += WILDOFFSET; }
00970                 if ( inset ) { *t++ = x1; *relo = 2; }
00971                 else *t++ = x1 + AM.OffsetIndex;
00972                 if ( t[-1] > AM.IndDum ) {
00973                     x1 = t[-1] - AM.IndDum;
00974                     if ( x1 > C->numdum[numexp] ) C->numdum[numexp] = x1;

```

```

00975         }
00976         goto fin;
00977     case TGENINDEX:
00978         *t++ = INDEX; *t++ = 3; *t++ = AC.DumNum+WILDOFFSET;
00979         deno = 1;
00980         break;
00981     case TVECTOR:
00982         x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
00983     dovector:
00984         if ( inset == 0 ) x1 += AM.OffsetVector;
00985         if ( *s == TWILDCARD ) { s++; x1 += WILDOFFSET; }
00986         if ( inset ) *relo = 2;
00987         if ( *s == TDOT ) { /* DotProduct ? */
00988             s++;
00989             if ( *s == TSETNUM || *s == TSETDOL ) {
00990                 settype = ( *s == TSETDOL );
00991                 s++; x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
00992                 if ( settype ) x2 = -x2;
00993                 if ( inset == 0 ) {
00994                     tsize = t; *t++ = SETSET; *t++ = 0;
00995                     relo = AT.WorkTop;
00996                 }
00997                 inset += 2;
00998                 *--relo = x2; *--relo = 3;
00999             }
01000             if ( *s != TVECTOR && *s != TDUBIOUS ) {
01001                 MesPrint("&Illegally formed dotproduct");
01002                 error = 1;
01003             }
01004             s++; x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
01005             if ( inset < 2 ) x2 += AM.OffsetVector;
01006             if ( *s == TWILDCARD ) { s++; x2 += WILDOFFSET; }
01007             *t++ = DOTPRODUCT; *t++ = 5; *t++ = x1; *t++ = x2;
01008             goto TryPower;
01009         }
01010     else if ( *s == TFUNOPEN ) {
01011         s++;
01012         if ( *s == TSETNUM || *s == TSETDOL ) {
01013             settype = ( *s == TSETDOL );
01014             s++; x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
01015             if ( settype ) x2 = -x2;
01016             if ( inset == 0 ) {
01017                 tsize = t; *t++ = SETSET; *t++ = 0;
01018                 relo = AT.WorkTop;
01019             }
01020             inset += 2;
01021             *--relo = x2; *--relo = 3;
01022         }
01023         if ( *s == TINDEX || *s == TDUBIOUS ) {
01024             s++;
01025             x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
01026             if ( inset < 2 ) x2 += AM.OffsetIndex;
01027             if ( *s == TWILDCARD ) { s++; x2 += WILDOFFSET; }
01028             *t++ = VECTOR; *t++ = 4; *t++ = x1; *t++ = x2;
01029             if ( t[-1] > AM.IndDum ) {
01030                 x2 = t[-1] - AM.IndDum;
01031                 if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
01032             }
01033         }
01034     else if ( *s == TGENINDEX ) {
01035         *t++ = VECTOR; *t++ = 4; *t++ = x1;
01036         *t++ = AC.DumNum + WILDOFFSET;
01037     }
01038     else if ( *s == TNUMBER || *s == TNUMBER1 ) {
01039         base = ( *s == TNUMBER ) ? 100: 128;
01040         s++;
01041         x2 = 0; while ( *s >= 0 ) { x2 = x2*base + *s++; }
01042         if ( x2 >= AM.OffsetIndex && inset < 2 ) {
01043             MesPrint("&Fixed index in vector greater than %d",
01044                 AM.OffsetIndex);
01045             return(-1);
01046         }
01047         *t++ = VECTOR; *t++ = 4; *t++ = x1; *t++ = x2;
01048     }
01049     else if ( *s == TVECTOR || ( *s == TMINUS && s[1] == TVECTOR ) ) {
01050         if ( *s == TMINUS ) { s++; sign = -sign; }
01051         s++;
01052         x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
01053         if ( inset < 2 ) x2 += AM.OffsetVector;
01054         if ( *s == TWILDCARD ) { s++; x2 += WILDOFFSET; }
01055         *t++ = DOTPRODUCT; *t++ = 5; *t++ = x1; *t++ = x2; *t++ = deno;
01056     }
01057     else {
01058         MesPrint("&Illegal argument for vector");
01059         return(-1);
01060     }
01061     if ( *s != TFUNCLOSE ) {
01062         MesPrint("&Illegal argument for vector");

```

```

01062         return(-1);
01063     }
01064     s++;
01065 }
01066 else {
01067     if ( AC.DumNum ) {
01068         *t++ = VECTOR; *t++ = 4; *t++ = x1;
01069         *t++ = AC.DumNum + WILDOFFSET;
01070     }
01071     else {
01072         *t++ = INDEX; *t++ = 3; *t++ = x1;
01073     }
01074 }
01075 goto fin;
01076 case TDELTA:
01077     if ( *s != TFUNOPEN ) {
01078         MesPrint("&d_ needs two arguments");
01079         error = -1;
01080     }
01081     v = t; *t++ = DELTA; *t++ = 4;
01082     needarg = 2; x3 = x1 = -1;
01083     goto dotensor;
01084 case TFUNCTION:
01085     x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
01086     if ( x1 == AM.sumnum || x1 == AM.sumpnum ) sumlevel = x1;
01087     x1 += FUNCTION;
01088     if ( x1 == FIRSTBRACKET ) {
01089         if ( s[0] == TFUNOPEN && s[1] == TEXPRESSION ) {
01090 doexpr:
01091             s += 2;
01092             *t++ = x1; *t++ = FUNHEAD+2; *t++ = 0;
01093             if ( x1 == AR.PolyFun && AR.PolyFunType == 2 && AR.Eside != LHSIDE )
01094                 t[-1] |= MUSTCLEANPRF;
01095             FILLFUN3(t)
01096             x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
01097             *t++ = -EXPRESSION; *t++ = x2;
01098             /*
01099             The next code is added to facilitate parallel processing
01100             We need to call GetTable here to make sure all processors
01101             have the same numbering of all variables.
01102             */
01103             if ( Expressions[x2].status == STOREDEXPRESSION ) {
01104                 TMproto[0] = EXPRESSION;
01105                 TMproto[1] = SUBEXPSIZE;
01106                 TMproto[2] = x2;
01107                 TMproto[3] = 1;
01108                 { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
01109                 AT.TMaddr = TMproto;
01110                 PUTZERO(position);
01111             }
01112             if ( (
01113 #ifdef WITHPTHREADS
01114                 renumber =
01115                 GetTable(x2,&position,0) ) == 0 ) {
01116                 error = 1;
01117                 MesPrint("&Problems getting information about stored expression %s(1)"
01118                     ,EXPRNAME(x2));
01119             }
01120 #ifdef WITHPTHREADS
01121             M_free(renumber->symb.lo,"VarSpace");
01122             M_free(renumber,"Renumber");
01123 #endif
01124             /*
01125             if ( ( renumber = GetTable(x2,&position,0) ) == 0 ) {
01126                 error = 1;
01127                 MesPrint("&Problems getting information about stored expression %s(1)"
01128                     ,EXPRNAME(x2));
01129             }
01130             if ( renumber->symb.lo != AN.dummyrenumlist )
01131                 M_free(renumber->symb.lo,"VarSpace");
01132             M_free(renumber,"Renumber");
01133             AR.StoreData.dirtyflag = 1;
01134         }
01135         if ( *s != TFUNCLOSE ) {
01136             if ( x1 == FIRSTBRACKET )
01137                 MesPrint("&Problems with argument of FirstBracket_");
01138             else if ( x1 == FIRSTTERM )
01139                 MesPrint("&Problems with argument of FirstTerm_");
01140             else if ( x1 == CONTENTTERM )
01141                 MesPrint("&Problems with argument of FirstTerm_");
01142             else if ( x1 == TERMSINEXPR )
01143                 MesPrint("&Problems with argument of TermsIn_");
01144             else if ( x1 == SIZEOFFUNCTION )
01145                 MesPrint("&Problems with argument of SizeOf_");
01146             else if ( x1 == NUMFACTORS )
01147                 MesPrint("&Problems with argument of NumFactors_");
01148             else

```



```

01149         MesPrint("&Problems with argument of FactorIn_");
01150         error = 1;
01151         while ( *s != TENDOFIT && *s != TFUNCLOSE ) s++;
01152     }
01153     if ( *s == TFUNCLOSE ) s++;
01154     goto fin;
01155 }
01156 }
01157 else if ( x1 == TERMSINEXPR || x1 == SIZEOFFUNCTION || x1 == FACTORIN
01158 || x1 == NUMFACTORS || x1 == FIRSTTERM || x1 == CONTENTTERM ) {
01159     if ( s[0] == TFUNOPEN && s[1] == TEXPRESSSION ) goto doexpr;
01160     if ( s[0] == TFUNOPEN && s[1] == TDOLLAR ) {
01161         s += 2;
01162         *t++ = x1; *t++ = FUNHEAD+2; *t++ = 0;
01163         FILLFUN3(t)
01164         x2 = 0; while ( *s >= 0 ) { x2 = x2*128 + *s++; }
01165         *t++ = -DOLLAREXPRESSSION; *t++ = x2;
01166         if ( *s != TFUNCLOSE ) {
01167             if ( x1 == TERMSINEXPR )
01168                 MesPrint("&Problems with argument of TermsIn_");
01169             else if ( x1 == SIZEOFFUNCTION )
01170                 MesPrint("&Problems with argument of SizeOf_");
01171             else if ( x1 == NUMFACTORS )
01172                 MesPrint("&Problems with argument of NumFactors_");
01173             else
01174                 MesPrint("&Problems with argument of FactorIn_");
01175             error = 1;
01176             while ( *s != TENDOFIT && *s != TFUNCLOSE ) s++;
01177         }
01178         if ( *s == TFUNCLOSE ) s++;
01179         goto fin;
01180     }
01181 }
01182 x3 = x1;
01183 if ( inset && ( t-tsize == 2 ) ) x1 -= FUNCTION;
01184 if ( *s == TWILDCARD ) { x1 += WILDOFFSET; s++; }
01185 if ( functions[x3-FUNCTION].commute ) cc = 1;
01186 if ( *s != TFUNOPEN ) {
01187     *t++ = x1; *t++ = FUNHEAD; *t++ = 0;
01188     if ( x1 == AR.PolyFun && AR.PolyFunType == 2 && AR.Eside != LHSIDE )
01189         t[-1] |= MUSTCLEANPRF;
01190     FILLFUN3(t) sumlevel = 0; goto fin;
01191 }
01192 v = t; *t++ = x1; *t++ = FUNHEAD; *t++ = DIRTYFLAG;
01193 if ( x1 == AR.PolyFun && AR.PolyFunType == 2 && AR.Eside != LHSIDE )
01194     t[-1] |= MUSTCLEANPRF;
01195 FILLFUN3(t)
01196 needarg = -1;
01197 if ( !inset && functions[x3-FUNCTION].spec >= TENSORFUNCTION ) {
01198     do {
01199         if ( needarg == 0 ) {
01200             if ( x1 >= 0 ) {
01201                 x3 = x1;
01202                 if ( x3 >= FUNCTION+WILDOFFSET ) x3 -= WILDOFFSET;
01203                 MesPrint("&Too many arguments in function %s",
01204                     VARNAME(functions, (x3-FUNCTION)) );
01205             }
01206             else
01207                 MesPrint("&d_ needs exactly two arguments");
01208             error = -1;
01209             needarg--;
01210         }
01211         else if ( needarg > 0 ) needarg--;
01212         s++;
01213         c = *s++;
01214         if ( c == TMINUS && *s == TVECTOR ) { sign = -sign; c = *s++; }
01215         base = ( c == TNUMBER ) ? 100: 128;
01216         x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
01217         if ( *s == TWILDCARD && c != TNUMBER ) { x2 += WILDOFFSET; s++; }
01218         if ( c == TSETNUM || c == TSETDOL ) {
01219             if ( c == TSETDOL ) x2 = -x2;
01220             if ( inset == 0 ) {
01221                 w1 = t; t += 2; w2 = t;
01222                 while ( w1 > v ) *--w2 = *--w1;
01223                 tsize = v; relo = AT.WorkTop;
01224                 *v++ = SETSET; *v++ = 0;
01225             }
01226             inset = 2; *--relo = x2; *--relo = t - v;
01227             c = *s++;
01228             x2 = 0; while ( *s >= 0 ) x2 = 128*x2 + *s++;
01229             switch ( c ) {
01230                 case TINDEX:
01231                     *t++ = x2;
01232                     if ( t[-1]+AM.OffsetIndex > AM.IndDum ) {
01233                         x2 = t[-1]+AM.OffsetIndex - AM.IndDum;
01234                         if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
01235

```

```

01236         }
01237         break;
01238     case TVECTOR:
01239         *t++ = x2; break;
01240     case TNUMBER1:
01241         if ( x2 >= 0 && x2 < AM.OffsetIndex ) {
01242             *t++ = x2; break;
01243         }
01244         /* fall through */
01245     default:
01246         MesPrint("&Illegal type of set inside tensor");
01247         error = 1;
01248         *t++ = x2;
01249         break;
01250     }
01251 }
01252 else { switch ( c ) {
01253     case TINDEX:
01254         if ( inset < 2 ) *t++ = x2 + AM.OffsetIndex;
01255         else *t++ = x2;
01256         if ( x2+AM.OffsetIndex > AM.IndDum ) {
01257             x2 = x2+AM.OffsetIndex - AM.IndDum;
01258             if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
01259         }
01260         break;
01261     case TGENINDEX:
01262         *t++ = AC.DumNum + WILDOFFSET;
01263         break;
01264     case TVECTOR:
01265         if ( inset < 2 ) *t++ = x2 + AM.OffsetVector;
01266         else *t++ = x2;
01267         break;
01268     case TWILDARG:
01269         *t++ = FUNNYWILD; *t++ = x2;
01270         v[2] = 0; /*
01271         break;
01272     case TDOLLAR:
01273         *t++ = FUNNYDOLLAR; *t++ = x2;
01274         break;
01275     case TDUBIOUS:
01276         if ( inset < 2 ) *t++ = x2 + AM.OffsetVector;
01277         else *t++ = x2;
01278         break;
01279     case TSGAMMA: /* Special gamma's */
01280         if ( x3 != GAMMA ) {
01281             MesPrint("&5_,6_,7_ can only be used inside g_");
01282             error = -1;
01283         }
01284         *t++ = -x2;
01285         break;
01286     case TNUMBER:
01287     case TNUMBER1:
01288         if ( x2 >= AM.OffsetIndex && inset < 2 ) {
01289             MesPrint("&Value of constant index in tensor too large");
01290             error = -1;
01291         }
01292         *t++ = x2;
01293         break;
01294     default:
01295         MesPrint("&Illegal object in tensor");
01296         error = -1;
01297         break;
01298     }}
01299     if ( inset >= 2 ) inset = 1;
01300 } while ( *s == TCOMMA );
01301 }
01302 else {
01303     dofunction: firstsumarg = 1;
01304     do {
01305         unsigned int ux2;
01306         s++;
01307         c = *s++;
01308         if ( c == TMINUS && ( *s == TVECTOR || *s == TNUMBER
01309         || *s == TNUMBER1 || *s == TSUBEXP ) ) {
01310             minus = 1; c = *s++;
01311         }
01312         else minus = 0;
01313         base = ( c == TNUMBER ) ? 100: 128;
01314         ux2 = 0; while ( *s >= 0 ) { ux2 = base*ux2 + *s++; }
01315         x2 = ux2; /* may cause an implementation-defined behaviour */
01316     /*
01317     !!!!!!! What if it does not fit?
01318     */
01319     if ( firstsumarg ) {
01320         firstsumarg = 0;
01321         if ( sumlevel > 0 ) {
01322             if ( c == TSYMBOL ) {

```

```

01323         sumlevel = x2; sumtype = SYMTOSYM;
01324     }
01325     else if ( c == TINDEX ) {
01326         sumlevel = x2+AM.OffsetIndex; sumtype = INDTOIND;
01327         if ( sumlevel > AM.IndDum ) {
01328             x2 = sumlevel - AM.IndDum;
01329             if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
01330         }
01331     }
01332 }
01333 }
01334 if ( *s == TWILDCARD ) {
01335     if ( c == TSYMBOL ) x2 += 2*MAXPOWER;
01336     else if ( c != TNUMBER ) x2 += WILDOFFSET;
01337     s++;
01338 }
01339 switch ( c ) {
01340     case TSYMBOL:
01341         *t++ = -SYMBOL; *t++ = x2; break;
01342     case TDOLLAR:
01343         *t++ = -DOLLAREXPRESSION; *t++ = x2; break;
01344     case TEXPRESSION:
01345         *t++ = -EXPRESSION; *t++ = x2;
01346 /*
01347
01348     The next code is added to facilitate parallel processing
01349     We need to call GetTable here to make sure all processors
01350     have the same numbering of all variables.
01351 */
01352     if ( Expressions[x2].status == STOREDEXPRESSION ) {
01353         TMproto[0] = EXPRESSION;
01354         TMproto[1] = SUBEXPSIZE;
01355         TMproto[2] = x2;
01356         TMproto[3] = 1;
01357         { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
01358         AT.TMaddr = TMproto;
01359         PUTZERO(position);
01360
01361         if ( (
01362             renumber =
01363             GetTable(x2,&position,0) ) == 0 ) {
01364             error = 1;
01365             MesPrint("&Problems getting information about stored
01366             expression %s(2) "
01367             ,EXPRNAME(x2));
01368         }
01369         M_free(renumber->symb.lo,"VarSpace");
01370         M_free(renumber,"Renummer");
01371
01372         if ( ( renumber = GetTable(x2,&position,0) ) == 0 ) {
01373             error = 1;
01374             MesPrint("&Problems getting information about stored
01375             expression %s(2) "
01376             ,EXPRNAME(x2));
01377         }
01378         if ( renumber->symb.lo != AN.dummyrenumlist )
01379             M_free(renumber->symb.lo,"VarSpace");
01380         M_free(renumber,"Renummer");
01381         AR.StoreData.dirtyflag = 1;
01382     }
01383     break;
01384     case TINDEX:
01385         *t++ = -INDEX; *t++ = x2 + AM.OffsetIndex;
01386         if ( t[-1] > AM.IndDum ) {
01387             x2 = t[-1] - AM.IndDum;
01388             if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
01389         }
01390     }
01391     break;
01392     case TGENINDEX:
01393         *t++ = -INDEX; *t++ = AC.DumNum + WILDOFFSET;
01394     }
01395     break;
01396     case TVECTOR:
01397         if ( minus ) *t++ = -MINVECTOR;
01398         else *t++ = -VECTOR;
01399         *t++ = x2 + AM.OffsetVector;
01400     }
01401     break;
01402     case TSGAMMA: /* Special gamma's */
01403         MesPrint("&5_,6_,7_ can only be used inside g_");
01404         error = -1;
01405         *t++ = -INDEX;
01406         *t++ = -x2;
01407     }
01408     break;
01409     case TDUBIOUS:
01410         *t++ = -SYMBOL; *t++ = x2; break;

```

```

01408         case TFUNCTION:
01409             *t++ = -x2-FUNCTION;
01410             break;
01411         case TSET:
01412             *t++ = -SETSET;
01413             *t++ = x2;
01414             break;
01415         case TWILDARG:
01416             *t++ = -ARGWILD; *t++ = x2; break;
01417         case TSETDOL:
01418             x2 = -x2;
01419             /* fall through */
01420         case TSETNUM:
01421             if ( inset == 0 ) {
01422                 w1 = t; t += 2; w2 = t;
01423                 while ( w1 > v ) *--w2 = *--w1;
01424                 tsize = v; relo = AT.WorkTop;
01425                 *v++ = SETSET; *v++ = 0;
01426                 inset = 1;
01427             }
01428             *--relo = x2; *--relo = t-v+1;
01429             c = *s++;
01430             x2 = 0; while ( *s >= 0 ) x2 = 128*x2 + *s++;
01431             switch ( c ) {
01432                 case TFUNCTION:
01433                     (*relo)--; *t++ = -x2-1; break;
01434                 case TSYMBOL:
01435                     *t++ = -SYMBOL; *t++ = x2; break;
01436                 case TINDEX:
01437                     *t++ = -INDEX; *t++ = x2;
01438                     if ( x2+AM.OffsetIndex > AM.IndDum ) {
01439                         x2 = x2+AM.OffsetIndex - AM.IndDum;
01440                         if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
01441                     }
01442                     break;
01443                 case TVECTOR:
01444                     *t++ = -VECTOR; *t++ = x2; break;
01445                 case TNUMBER1:
01446                     *t++ = -SNUMBER; *t++ = x2; break;
01447                 default:
01448                     MesPrint("&Internal error 435");
01449                     error = 1;
01450                     *t++ = -SYMBOL; *t++ = x2; break;
01451             }
01452             break;
01453         case TSUBEXP:
01454             w2 = AC.ProtoType; i = w2[1];
01455             w1 = t;
01456             *t++ = i+ARGHEAD+4;
01457             *t++ = 1;
01458             FILLARG(t);
01459             *t++ = i + 4;
01460             while ( --i >= 0 ) *t++ = *w2++;
01461             w1[ARGHEAD+3] = subexpbuffers[x2].subexpnum;
01462             w1[ARGHEAD+5] = subexpbuffers[x2].buffernum;
01463             if ( sumlevel > 0 ) {
01464                 w1[0] += 4;
01465                 w1[ARGHEAD] += 4;
01466                 w1[ARGHEAD+2] += 4;
01467                 *t++ = sumtype; *t++ = 4;
01468                 *t++ = sumlevel; *t++ = sumlevel;
01469             }
01470             *t++ = 1; *t++ = 1;
01471             if ( minus ) *t++ = -3;
01472             else *t++ = 3;
01473             break;
01474         case TNUMBER:
01475         case TNUMBER1:
01476             if ( minus ) x2 = UnsignedToInt(-IntAbs(x2));
01477             *t++ = -SNUMBER;
01478             *t++ = x2;
01479             break;
01480         default:
01481             MesPrint("&Illegal object in function");
01482             error = -1;
01483             break;
01484     }
01485     } while ( *s == TCOMMA );
01486 }
01487 if ( *s != TFUNCLOSE ) {
01488     MesPrint("&Illegal argument field for function. Expected ");
01489     return(-1);
01490 }
01491 s++; sumlevel = 0;
01492 v[1] = t-v;
01493 /*
01494     if ( *v == AM.termfunnum && ( v[1] != FUNHEAD+2 ||

```

```

01495         v[FUNHEAD] != -DOLLAREXPRESSION ) ) {
01496             MesPrint("&The function term_ can only have one argument with a single
$-expression");
01497             error = 1;
01498         }
01499     */
01500     goto fin;
01501 case TDUBIOUS:
01502     x1 = 0; while ( *s >= 0 ) x1 = 128*x1 + *s++;
01503     if ( *s == TWILDCARD ) s++;
01504     if ( *s == TDOT ) goto dovector;
01505     if ( *s == TFUNOPEN ) {
01506         x1 += FUNCTION;
01507         cc = 1;
01508         v = t; *t++ = x1; *t++ = FUNHEAD; *t++ = DIRTYFLAG;
01509         if ( x1 == AR.PolyFun && AR.PolyFunType == 2 && AR.Eside != LHSIDE )
01510             t[-1] |= MUSTCLEANPRF;
01511         FILLFUN3(t)
01512         needarg = -1; goto dofunction;
01513     }
01514     *t++ = SYMBOL; *t++ = 4; *t++ = 0;
01515     if ( inset ) *relo = 2;
01516     goto TryPower;
01517 case TSUBEXP:
01518     x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
01519     if ( *s == TPOWER ) {
01520         s++; c = *s++;
01521         base = ( c == TNUMBER ) ? 100: 128;
01522         x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
01523         if ( *s == TWILDCARD ) { x2 += 2*MAXPOWER; s++; }
01524         else if ( c == TSYMBOL ) x2 += 2*MAXPOWER;
01525     }
01526     else x2 = 1;
01527     r = AC.ProtoType; n = r[1] - 5; r += 5;
01528     *t++ = SUBEXPRESSION; *t++ = r[-4];
01529     *t++ = subexpbuffers[x1].subexpnum;
01530     *t++ = x2*deno;
01531     *t++ = subexpbuffers[x1].buffernum;
01532     NCOPY(t,r,n);
01533     if ( cbuf[subexpbuffers[x1].buffernum].CanCommu[subexpbuffers[x1].subexpnum] ) cc = 1;
01534     deno = 1;
01535     break;
01536 case TMULTIPLY:
01537     mulflag = 1;
01538     break;
01539 case TDIVIDE:
01540     mulflag = 1;
01541     deno = -deno;
01542     break;
01543 case TEXPRESSON:
01544     cc = 1;
01545     x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
01546     v = t;
01547     *t++ = EXPRESSION; *t++ = SUBEXPSIZE; *t++ = x1; *t++ = deno;
01548     *t++ = 0; FILLSUB(t)
01549     /*
01550     Here we had some erroneous code before. It should be after
01551     the reading of the parameters as it is now (after 15-jan-2007).
01552     Thomas Hahn noticed this and reported it.
01553     */
01554     if ( *s == TFUNOPEN ) {
01555         do {
01556             s++; c = *s++;
01557             base = ( c == TNUMBER ) ? 100: 128;
01558             x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
01559             switch ( c ) {
01560                 case TSYMBOL:
01561                     *t++ = SYMBOL; *t++ = 4; *t++ = x2; *t++ = 1;
01562                     break;
01563                 case TINDEX:
01564                     *t++ = INDEX; *t++ = 3; *t++ = x2+AM.OffsetIndex;
01565                     if ( t[-1] > AM.IndDum ) {
01566                         x2 = t[-1] - AM.IndDum;
01567                         if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
01568                     }
01569                     break;
01570                 case TVECTOR:
01571                     *t++ = INDEX; *t++ = 3; *t++ = x2+AM.OffsetVector;
01572                     break;
01573                 case TFUNCTION:
01574                     *t++ = x2+FUNCTION; *t++ = 2; break;
01575                 case TNUMBER:
01576                 case TNUMBER1:
01577                     if ( x2 >= AM.OffsetIndex || x2 < 0 ) {
01578                         MesPrint("&Index as argument of expression has illegal value");
01579                         error = -1;
01580                     }

```

```

01581         *t++ = INDEX; *t++ = 3; *t++ = x2; break;
01582     case TSETDOL:
01583         x2 = -x2;
01584         /* fall through */
01585     case TSETNUM:
01586         if ( inset == 0 ) {
01587             w1 = t; t += 2; w2 = t;
01588             while ( w1 > v ) *--w2 = *--w1;
01589             tsize = v; relo = AT.WorkTop;
01590             *v++ = SETSET; *v++ = 0;
01591             inset = 1;
01592         }
01593         *--relo = x2; *--relo = t-v+2;
01594         c = *s++;
01595         x2 = 0; while ( *s >= 0 ) x2 = 128*x2 + *s++;
01596         switch ( c ) {
01597             case TFUNCTION:
01598                 *relo -= 2; *t++ = -x2-1; break;
01599             case TSYMBOL:
01600                 *t++ = SYMBOL; *t++ = 4; *t++ = x2; *t++ = 1; break;
01601             case TINDEX:
01602                 *t++ = INDEX; *t++ = 3; *t++ = x2;
01603                 if ( x2+AM.OffsetIndex > AM.IndDum ) {
01604                     x2 = x2+AM.OffsetIndex - AM.IndDum;
01605                     if ( x2 > C->numdum[numexp] ) C->numdum[numexp] = x2;
01606                 }
01607                 break;
01608             case TVECTOR:
01609                 *t++ = VECTOR; *t++ = 3; *t++ = x2; break;
01610             case TNUMBER1:
01611                 *t++ = SNUMBER; *t++ = 4; *t++ = x2; *t++ = 1; break;
01612             default:
01613                 MesPrint("&Internal error 435");
01614                 error = 1;
01615                 *t++ = SYMBOL; *t++ = 4; *t++ = x2; *t++ = 1; break;
01616         }
01617         break;
01618     default:
01619         MesPrint("&Argument of expression can only be symbol, index, vector or
function");
01620         error = -1;
01621         break;
01622     }
01623     } while ( *s == TCOMMA );
01624     if ( *s != TFUNCLOSE ) {
01625         MesPrint("&Illegal object in argument field for expression");
01626         error = -1;
01627         while ( *s != TFUNCLOSE ) s++;
01628     }
01629     s++;
01630 }
01631 r = AC.ProtoType; n = r[1];
01632 if ( n > SUBEXPSIZE ) {
01633     *t++ = WILDCARDS; *t++ = n+2;
01634     NCOPY(t,r,n);
01635 }
01636 /*
01637     Code added for parallel processing.
01638     This is different from the other occurrences to test immediately
01639     for renumbering. Here we have to read the parameters first.
01640 */
01641 if ( Expressions[x1].status == STOREDEXPRESSION ) {
01642     v[1] = t-v;
01643     AT.TMaddr = v;
01644     PUTZERO(position);
01645     /*
01646     if ( (
01647     #ifdef WITHPTHREADS
01648         renumber =
01649     #endif
01650         GetTable(x1,&position,0) ) == 0 ) {
01651         error = 1;
01652         MesPrint("&Problems getting information about stored expression %s(3)"
01653             ,EXPRNAME(x1));
01654     }
01655     #ifdef WITHPTHREADS
01656     M_free(renumber->symb.lo,"VarSpace");
01657     M_free(renumber,"Renumber");
01658     #endif
01659     */
01660     if ( ( renumber = GetTable(x1,&position,0) ) == 0 ) {
01661         error = 1;
01662         MesPrint("&Problems getting information about stored expression %s(3)"
01663             ,EXPRNAME(x1));
01664     }
01665     if ( renumber->symb.lo != AN.dummyrenumlist )
01666         M_free(renumber->symb.lo,"VarSpace");

```

```

01667         M_free(renumber,"Renumber");
01668         AR.StoreData.dirtyflag = 1;
01669     }
01670     if ( *s == LBRACE ) {
01671         /*
01672         This should be one term that should be inserted
01673         FROMBRAC size+2 ( term )
01674         Because this term should have been translated
01675         already we can copy it from the 'subexpression'
01676         */
01677         s++;
01678         if ( *s != TSUBEXP ) {
01679             MesPrint("&Internal error 23");
01680             Terminate(-1);
01681         }
01682         s++; x2 = 0; while ( *s >= 0 ) { x2 = 128*x2 + *s++; }
01683         r = cbuf[subexpbuffers[x2].buffernum].rhs[subexpbuffers[x2].subexpnum];
01684         *t++ = FROMBRAC; *t++ = *r+2;
01685         n = *r;
01686         NCOPY(t,r,n);
01687         if ( *r != 0 ) {
01688             MesPrint("&Object between [] in expression should be a single term");
01689             error = -1;
01690         }
01691         if ( *s != RBRACE ) {
01692             MesPrint("&Internal error 23b");
01693             Terminate(-1);
01694         }
01695         s++;
01696     }
01697     if ( *s == TPOWER ) {
01698         s++; c = *s++;
01699         base = ( c == TNUMBER ) ? 100: 128;
01700         x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
01701         if ( *s == TWILDCARD || c == TSYMBOL ) { x2 += 2*MAXPOWER; s++; }
01702         v[3] = x2;
01703     }
01704     v[1] = t - v;
01705     deno = 1;
01706     break;
01707 case TNUMBER:
01708     if ( *s == 0 ) {
01709         s++;
01710         if ( *s == TPOWER ) {
01711             s++; if ( *s == TMINUS ) { s++; deno = -deno; }
01712             c = *s++; base = ( c == TNUMBER ) ? 100: 128;
01713             x2 = 0; while ( *s >= 0 ) { x2 = x2*base + *s++; }
01714             if ( x2 == 0 ) {
01715                 error = -1;
01716                 MesPrint("&Encountered 0^0 during compilation");
01717             }
01718             if ( deno < 0 ) {
01719                 error = -1;
01720                 MesPrint("&Division by zero during compilation (0 to the power negative
number)");
01721             }
01722         }
01723         else if ( deno < 0 ) {
01724             error = -1;
01725             MesPrint("&Division by zero during compilation");
01726         }
01727         sign = 0; break; /* term is zero */
01728     }
01729     y = *s++;
01730     if ( *s >= 0 ) { y = 100*y + *s++; }
01731     innum[0] = y; nin = 1;
01732     while ( *s >= 0 ) {
01733         y = *s++; x2 = 100;
01734         if ( *s >= 0 ) { y = 100*y + *s++; x2 = 10000; }
01735         Product(innum,&nin,(WORD)x2);
01736         if ( y ) AddLong(innum,nin,(UWORD *)(&y),(WORD)1,innum,&nin);
01737     }
01738 docoef:
01739     if ( *s == TPOWER ) {
01740         s++; if ( *s == TMINUS ) { s++; deno = -deno; }
01741         c = *s++; base = ( c == TNUMBER ) ? 100: 128;
01742         x2 = 0; while ( *s >= 0 ) { x2 = x2*base + *s++; }
01743         if ( x2 == 0 ) {
01744             innum[0] = 1; nin = 1;
01745         }
01746         else if ( RaisPow(BHEAD innum,&nin,x2) ) {
01747             error = -1; innum[0] = 1; nin = 1;
01748         }
01749     }
01750     if ( deno > 0 ) {
01751         Simplify(BHEAD innum,&nin,denominator,&ndenominator);
01752         for ( i = 0; i < nnumerator; i++ ) CGscrat7[i] = numerator[i];

```

```

01753         MulLong(innum,nin,CGscrat7,nnumerator,numerator,&nnumerator);
01754     }
01755     else if ( deno < 0 ) {
01756         Simplify(BHEAD innum,&nin,numerator,&nnumerator);
01757         for ( i = 0; i < ndenominator; i++ ) CGscrat7[i] = denominator[i];
01758         MulLong(innum,nin,CGscrat7,ndenominator,denominator,&ndenominator);
01759     }
01760     deno = 1;
01761     break;
01762 case TNUMBER1:
01763     if ( *s == 0 ) { s++; sign = 0; break; /* term is zero */ }
01764     y = *s++;
01765     if ( *s >= 0 ) { y = 128*y + *s++; }
01766     if ( inset == 0 ) {
01767         innum[0] = y; nin = 1;
01768         while ( *s >= 0 ) {
01769             y = *s++; x2 = 128;
01770             if ( *s >= 0 ) { y = 128*y + *s++; x2 = 16384; }
01771             Product(innum,&nin,(WORD)x2);
01772             if ( y ) AddLong(innum,nin,(UWORD *)&y,(WORD)1,innum,&nin);
01773         }
01774         goto docoef;
01775     }
01776     *relo = 2; *t++ = SNUMBER; *t++ = 4; *t++ = y;
01777     goto TryPower;
01778 #ifdef WITHFLOAT
01779 case TFLOAT:
01780     { WORD *w;
01781       s = ReadFloat(s);
01782       i = AT.WorkPointer[1];
01783       w = AT.WorkPointer;
01784       NCOPY(t,w,i);
01785     }
01786     /*
01787     Power?
01788     */
01789     break;
01790 #endif
01791 case TDOLLAR:
01792     {
01793         WORD *powplace;
01794         x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
01795         if ( AR.Eside != LHSIDE ) {
01796             *t++ = SUBEXPRESSION; *t++ = SUBEXPSIZE; *t++ = x1;
01797         }
01798         else {
01799             *t++ = DOLLAREXPRESSION; *t++ = SUBEXPSIZE; *t++ = x1;
01800         }
01801         powplace = t; t++;
01802         *t++ = AM.dbufnum; FILLSUB(t)
01803     }
01804     /*
01805     Now we have to test for factors of dollars with [ ] and [ [ ] ]
01806     */
01807     if ( *s == LBRACE ) {
01808         int bracelevel = 1;
01809         s++;
01810         while ( bracelevel > 0 ) {
01811             if ( *s == RBRACE ) {
01812                 bracelevel--; s++;
01813             }
01814             else if ( *s == TNUMBER ) {
01815                 s++;
01816                 x2 = 0; while ( *s >= 0 ) { x2 = 100*x2 + *s++; }
01817                 *t++ = DOLLAREXPR2; *t++ = 3; *t++ = -x2-1;
01818             }
01819             while ( bracelevel > 0 ) {
01820                 if ( *s != RBRACE ) {
01821                     error = -1;
01822                     MesPrint("&Improper use of [] in $-variable.");
01823                     return(error);
01824                 }
01825                 else {
01826                     s++; bracelevel--;
01827                 }
01828             }
01829         }
01830         else if ( *s == TDOLLAR ) {
01831             s++;
01832             x1 = 0; while ( *s >= 0 ) { x1 = x1*128 + *s++; }
01833             *t++ = DOLLAREXPR2; *t++ = 3; *t++ = x1;
01834             if ( *s == RBRACE ) goto CloseBraces;
01835             else if ( *s == LBRACE ) {
01836                 s++; bracelevel++;
01837             }
01838         }
01839         else goto ErrorBraces;

```



```

01840         }
01841     }
01842     /*
01843         Finally we can continue with the power
01844     */
01845     if ( *s == TPOWER ) {
01846         s++;
01847         if ( *s == TMINUS ) { s++; deno = -deno; }
01848         c = *s++;
01849         base = ( c == TNUMBER ) ? 100: 128;
01850         x2 = 0; while ( *s >= 0 ) { x2 = base*x2 + *s++; }
01851         if ( c == TSYMBOL ) {
01852             if ( *s == TWILDCARD ) s++;
01853             x2 += 2*MAXPOWER;
01854         }
01855         *powplace = deno*x2;
01856     }
01857     else *powplace = deno;
01858     deno = 1;
01859     /*
01860         if ( inset ) {
01861             while ( relo < AT.WorkTop ) *t++ = *relo++;
01862             inset = 0; tsize[1] = t - tsize;
01863         }
01864     */
01865     }
01866     break;
01867 case TSETNUM:
01868     inset = 1; tsize = t; relo = AT.WorkTop;
01869     *t++ = SETSET; *t++ = 0;
01870     x1 = 0; while ( *s >= 0 ) x1 = x1*128 + *s++;
01871     *--relo = x1; *--relo = 0;
01872     break;
01873 case TSETDOL:
01874     inset = 1; tsize = t; relo = AT.WorkTop;
01875     *t++ = SETSET; *t++ = 0;
01876     x1 = 0; while ( *s >= 0 ) x1 = x1*128 + *s++;
01877     *--relo = -x1; *--relo = 0;
01878     break;
01879 case TFUNOPEN:
01880     MesPrint("&Illegal use of function arguments");
01881     error = -1;
01882     funflag = 1;
01883     deno = 1;
01884     break;
01885 case TFUNCLOSE:
01886     if ( funflag == 0 )
01887         MesPrint("&Illegal use of function arguments");
01888     error = -1;
01889     funflag = 0;
01890     deno = 1;
01891     break;
01892 case TSGAMMA:
01893     MesPrint("&Illegal use special gamma symbols 5_, 6_, 7_");
01894     error = -1;
01895     funflag = 0;
01896     deno = 1;
01897     break;
01898 case TCONJUGATE:
01899     MesPrint("&Complex conjugate operator (%#) is not implemented");
01900     error = -1;
01901     deno = 1;
01902     break;
01903 default:
01904     MesPrint("&Internal error in code generator. Unknown object: %d",c);
01905     error = -1;
01906     deno = 1;
01907     break;
01908     }
01909 }
01910 }
01911 if ( mulflag ) {
01912     MesPrint("&Irregular end of statement.");
01913     error = 1;
01914 }
01915 if ( !first && error == 0 ) {
01916     *term = t-term;
01917     C->NumTerms[numexp]++;
01918     if ( cc && sign ) C->CanCommu[numexp]++;
01919     error = CompleteTerm(term,numerator,denominator,nnumerator,ndenominator,sign);
01920 }
01921 AT.WorkPointer = oldwork;
01922 if ( error ) return(-1);
01923 AddToCB(C,0)
01924 if ( AC.CompileLevel > 0 && AR.Eside != LHSIDE ) {
01925     /* See whether we have this one already */
01926     error = InsTree(AC.cbufnum,C->numrhs);

```

```

01927         if ( error < (C->numrhs) ) {
01928             C->Pointer = C->rhs[C->numrhs--];
01929             return(error);
01930         }
01931     }
01932     return(C->numrhs);
01933 OverWork:
01934     MLOCK(ErrorMessageLock);
01935     MesWork();
01936     MUNLOCK(ErrorMessageLock);
01937     return(-1);
01938 }
01939
01940 /*
01941     #] CodeGenerator :
01942     #[ CompleteTerm :
01943
01944     Completes the term
01945     Puts it in the buffer
01946 */
01947
01948 int CompleteTerm(WORD *term, UWORD *number, UWORD *denom, WORD nnum, WORD nden, int sign)
01949 {
01950     int nsize, i;
01951     WORD *t;
01952     if ( sign == 0 ) return(0); /* Term is zero */
01953     if ( nnum >= nden ) nsize = nnum;
01954     else nsize = nden;
01955     t = term + *term;
01956     for ( i = 0; i < nnum; i++ ) *t++ = number[i];
01957     for ( ; i < nsize; i++ ) *t++ = 0;
01958     for ( i = 0; i < nden; i++ ) *t++ = denom[i];
01959     for ( ; i < nsize; i++ ) *t++ = 0;
01960     *t++ = (2*nsize+1)*sign;
01961     *term = t - term;
01962     AddNtoC(AC.cbufnum,*term,term,7);
01963     return(0);
01964 }
01965
01966 /*
01967     #] CompleteTerm :
01968     #[ CodeFactors :
01969
01970     This routine does the part of reading in in terms of factors.
01971     If there is more than one term at this level we have only one
01972     factor. In that case any expression should first be unfactorized.
01973     Then the whole expression gets read as a new subexpression and finally
01974     we generate factor_*subexpression.
01975     If the whole has only multiplications we have factors. Then the
01976     nasty thing is powers of objects and in particular powers of
01977     factorized expressions or dollars.
01978     For a power we generate a new subexpression of the type
01979     1+factor_+...+factor_^(power-1)
01980     with which we multiply.
01981
01982     WE HAVE NOT YET WORRIED ABOUT SETS
01983 */
01984
01985 int CodeFactors(SBYTE *tokens)
01986 {
01987     GETIDENTITY
01988     EXPRESSIONS e = Expressions + AR.CurExpr;
01989     int nfactor = 1, nparenthesis, i, last = 0, error = 0;
01990     SBYTE *t, *startobject, *tt, *sl, *out, *outtokens;
01991     WORD nexp, subexp = 0, power, pow, x2, powfactor, first;
01992     /*
01993     First scan the number of factors
01994     */
01995     t = tokens;
01996     while ( *t != TENDOFIT ) {
01997         if ( *t >= 0 ) { while ( *t >= 0 ) t++; continue; }
01998         if ( *t == LPARENTHESIS || *t == LBRACE || *t == TSETOPEN || *t == TFUNOPEN ) {
01999             nparenthesis = 0; t++;
02000             while ( nparenthesis >= 0 ) {
02001                 if ( *t == LPARENTHESIS || *t == LBRACE || *t == TSETOPEN || *t == TFUNOPEN )
02002                     nparenthesis++;
02003                 else if ( *t == RPARENTHESIS || *t == RBRACE || *t == TSETCLOSE || *t == TFUNCLOSE )
02004                     nparenthesis--;
02005                 t++;
02006             }
02007             continue;
02008         }
02009         else if ( ( *t == TPLUS || *t == TMINUS ) && ( t > tokens )
02010             && ( t[-1] != TPLUS && t[-1] != TMINUS ) ) {
02011             if ( t[-1] >= 0 || t[-1] == RPARENTHESIS || t[-1] == RBRACE
02012                 || t[-1] == TSETCLOSE || t[-1] == TFUNCLOSE ) {
02013                 subexp = CodeGenerator(tokens);

```

```

02012         if ( subexp < 0 ) error = -1;
02013         if ( insubexpbuffers >= MAXSUBEXPRESSIONS ) {
02014             MesPrint("&More than %d subexpressions inside one
expression", (WORD)MAXSUBEXPRESSIONS);
02015             Terminate(-1);
02016         }
02017         if ( subexpbuffers+insubexpbuffers >= topsubexpbuffers ) {
02018             DoubleBuffer((void **)((void *)(&subexpbuffers))
02019                 , (void **)((void *)(&topsubexpbuffers)), sizeof(SUBBUF), "subexpbuffers");
02020         }
02021         subexpbuffers[insubexpbuffers].subexpnum = subexp;
02022         subexpbuffers[insubexpbuffers].buffernum = AC.cbufnum;
02023         subexp = insubexpbuffers++;
02024         t = tokens;
02025         *t++ = TSYMBOL; *t++ = FACTORSYMBOL;
02026         *t++ = TMULTIPLY; *t++ = TSUBEXP;
02027         PUTNUMBER128(t, subexp)
02028         *t++ = TENDOFIT;
02029         e->numfactors = 1;
02030         e->vflags |= ISFACTORIZED;
02031         return(subexp);
02032     }
02033 }
02034 else if ( ( *t == TMULTIPLY || *t == TDIVIDE ) && t > tokens ) {
02035     nfactor++;
02036 }
02037 else if ( *t == TEXPRESSION ) {
02038     t++;
02039     nex = 0; while ( *t >= 0 ) { nex = nex*128 + *t++; }
02040     if ( *t == LBRACE ) continue;
02041     if ( ( AS.Oldvflags[nex] & ISFACTORIZED ) != 0 ) {
02042         nfactor += AS.OldNumFactors[nex];
02043     }
02044     else { nfactor++; }
02045     continue;
02046 }
02047 else if ( *t == TDOLLAR ) {
02048     t++;
02049     nex = 0; while ( *t >= 0 ) { nex = nex*128 + *t++; }
02050     if ( *t == LBRACE ) continue;
02051     if ( Dollars[nex].nfactors > 0 ) {
02052         nfactor += Dollars[nex].nfactors;
02053     }
02054     else { nfactor++; }
02055     continue;
02056 }
02057     t++;
02058 }
02059 /*
02060     Now the real pass.
02061     nfactor is a not so reliable measure for the space we need.
02062 */
02063     outtokens = (SBYTE *)Malloc1(((t-tokens)+(nfactor+2)*25)*sizeof(SBYTE), "CodeFactors");
02064     out = outtokens;
02065     t = tokens; first = 1; powfactor = 1;
02066     while ( *t == TPLUS || *t == TMINUS ) { if ( *t == TMINUS ) first = -first; t++; }
02067     if ( first < 0 ) {
02068         *out++ = TMINUS; *out++ = TSYMBOL; *out++ = FACTORSYMBOL;
02069         *out++ = TPOWER; *out++ = TNUMBER; PUTNUMBER100(out, powfactor)
02070         powfactor++;
02071     }
02072     startobject = t; power = 1;
02073     while ( *t != TENDOFIT ) {
02074         if ( *t >= 0 ) { while ( *t >= 0 ) t++; continue; }
02075         if ( *t == LPARENTHESIS || *t == LBRACE || *t == TSETOPEN || *t == TFUNOPEN ) {
02076             nparenthesis = 0; t++;
02077             while ( nparenthesis >= 0 ) {
02078                 if ( *t == LPARENTHESIS || *t == LBRACE || *t == TSETOPEN || *t == TFUNOPEN )
02079                     nparenthesis++;
02080                 else if ( *t == RPARENTHESIS || *t == RBRACE || *t == TSETCLOSE || *t == TFUNCLOSE )
02081                     nparenthesis--;
02082                 t++;
02083             }
02084             continue;
02085         }
02086         else if ( ( *t == TMULTIPLY || *t == TDIVIDE ) && ( t > tokens ) ) {
02087             if ( t[-1] >= 0 || t[-1] == RPARENTHESIS || t[-1] == RBRACE
02088                 || t[-1] == TSETCLOSE || t[-1] == TFUNCLOSE ) {
02089                 dolast:
02090                 if ( startobject ) { /* apparently power is 1 or -1 */
02091                     *out++ = TPLUS;
02092                     if ( power < 0 ) { *out++ = TNUMBER; *out++ = 1; *out++ = TDIVIDE; }
02093                     s1 = startobject;
02094                     while ( s1 < t ) *out++ = *s1++;
02095                     *out++ = TMULTIPLY; *out++ = TSYMBOL; *out++ = FACTORSYMBOL;
02096                     *out++ = TPOWER; *out++ = TNUMBER; PUTNUMBER100(out, powfactor)
02097                     powfactor++;

```

```

02096         }
02097         if ( last ) { startobject = 0; break; }
02098         startobject = t+1;
02099         if ( *t == TDIVIDE ) power = -1;
02100         if ( *t == TMULTIPLY ) power = 1;
02101     }
02102 }
02103 else if ( *t == TPOWER ) {
02104     pow = 1;
02105     tt = t+1;
02106     while ( ( *tt == TMINUS ) || ( *tt == TPLUS ) ) {
02107         if ( *tt == TMINUS ) pow = -pow;
02108         tt++;
02109     }
02110     if ( *tt == TSYMBOL ) {
02111         tt++; while ( *tt >= 0 ) tt++;
02112         t = tt; continue;
02113     }
02114     tt++; x2 = 0; while ( *tt >= 0 ) { x2 = 100*x2 + *tt++; }
02115 /*
02116     We have an object in startobject till t. The power is
02117     power*pow*x2
02118 */
02119     power = power*pow*x2;
02120     if ( power < 0 ) { pow = -power; power = -1; }
02121     else if ( power == 0 ) { t = tt; startobject = tt; continue; }
02122     else { pow = power; power = 1; }
02123     *out++ = TPLUS;
02124     if ( pow > 1 ) {
02125         subexp = GenerateFactors(pow,1);
02126         if ( subexp < 0 ) { error = -1; subexp = 0; }
02127         *out++ = TSUBEXP; PUTNUMBER128(out,subexp);
02128     }
02129     *out++ = TSYMBOL; *out++ = FACTORSYMBOL;
02130     *out++ = TPOWER; *out++ = TNUMBER; PUTNUMBER100(out,powfactor)
02131     powfactor += pow;
02132     if ( power > 0 ) *out++ = TMULTIPLY;
02133     else *out++ = TDIVIDE;
02134     s1 = startobject; while ( s1 < t ) *out++ = *s1++;
02135     startobject = 0; t = tt; continue;
02136 }
02137 else if ( *t == TEXPRESSION ) {
02138     startobject = t;
02139     t++;
02140     nexpt = 0; while ( *t >= 0 ) { nexpt = nexpt*128 + *t++; }
02141     if ( *t == LBRACE ) continue;
02142     if ( *t == LPARENTHESIS ) {
02143         nparenthesis = 0; t++;
02144         while ( nparenthesis >= 0 ) {
02145             if ( *t == LPARENTHESIS ) nparenthesis++;
02146             else if ( *t == RPARENTHESIS ) nparenthesis--;
02147             t++;
02148         }
02149     }
02150     if ( ( AS.Oldvflags[nexpt] & ISFACTORIZED ) == 0 ) continue;
02151     if ( *t == TPOWER ) {
02152         pow = 1;
02153         tt = t+1;
02154         while ( ( *tt == TMINUS ) || ( *tt == TPLUS ) ) {
02155             if ( *tt == TMINUS ) pow = -pow;
02156             tt++;
02157         }
02158         if ( *tt != TNUMBER ) {
02159             MesPrint("Internal problems(1) in CodeFactors");
02160             return(-1);
02161         }
02162         tt++; x2 = 0; while ( *tt >= 0 ) { x2 = 100*x2 + *tt++; }
02163 /*
02164         We have an object in startobject till t. The power is
02165         power*pow*x2
02166 */
02167     dopower:
02168         power = power*pow*x2;
02169         if ( power < 0 ) { pow = -power; power = -1; }
02170         else if ( power == 0 ) { t = tt; startobject = tt; continue; }
02171         else { pow = power; power = 1; }
02172         *out++ = TPLUS;
02173         if ( pow > 1 ) {
02174             subexp = GenerateFactors(pow,AS.OldNumFactors[nexpt]);
02175             if ( subexp < 0 ) { error = -1; subexp = 0; }
02176             *out++ = TSUBEXP; PUTNUMBER128(out,subexp)
02177             *out++ = TMULTIPLY;
02178         }
02179         i = powfactor-1;
02180         if ( i > 0 ) {
02181             *out++ = TSYMBOL; *out++ = FACTORSYMBOL;
02182             if ( i > 1 ) {

```

```

02183         *out++ = TPOWER; *out++ = TNUMBER; PUTNUMBER100(out,i)
02184     }
02185     *out++ = TMULTIPLY;
02186 }
02187 powfactor += AS.OldNumFactors[nexp]*pow;
02188 s1 = startobject;
02189 while ( s1 < t ) *out++ = *s1++;
02190 startobject = 0; t = tt; continue;
02191 }
02192 else {
02193     tt = t; pow = 1; x2 = 1; goto dopower;
02194 }
02195 }
02196 else if ( *t == TDOLLAR ) {
02197     startobject = t;
02198     t++;
02199     nexp = 0; while ( *t >= 0 ) { nexp = nexp*128 + *t++; }
02200     if ( *t == LBRACE ) continue;
02201     if ( Dollars[nexp].nfactors == 0 ) continue;
02202     if ( *t == TPOWER ) {
02203         pow = 1;
02204         tt = t+1;
02205         while ( ( *tt == TMINUS ) || ( *tt == TPLUS ) ) {
02206             if ( *tt == TMINUS ) pow = -pow;
02207             tt++;
02208         }
02209         if ( *tt != TNUMBER ) {
02210             MesPrint("Internal problems(2) in CodeFactors");
02211             return(-1);
02212         }
02213         tt++; x2 = 0; while ( *tt >= 0 ) { x2 = 100*x2 + *tt++; }
02214     /*
02215     We have an object in startobject till t. The power is
02216     power*pow*x2
02217     */
02218     dopowerd:
02219         power = power*pow*x2;
02220         if ( power < 0 ) { pow = -power; power = -1; }
02221         else if ( power == 0 ) { t = tt; startobject = tt; continue; }
02222         else { pow = power; power = 1; }
02223         if ( pow > 1 ) {
02224             subexp = GenerateFactors(pow,1);
02225             if ( subexp < 0 ) { error = -1; subexp = 0; }
02226         }
02227         for ( i = 1; i <= Dollars[nexp].nfactors; i++ ) {
02228             s1 = startobject; *out++ = TPLUS;
02229             while ( s1 < t ) *out++ = *s1++;
02230             *out++ = LBRACE; *out++ = TNUMBER; PUTNUMBER128(out,i)
02231             *out++ = RBRACE;
02232             *out++ = TMULTIPLY;
02233             *out++ = TSYMBOL; *out++ = FACTORSYMBOL;
02234             *out++ = TPOWER; *out++ = TNUMBER; PUTNUMBER100(out,powfactor)
02235             powfactor += pow;
02236             if ( pow > 1 ) {
02237                 *out++ = TSUBEXP; PUTNUMBER128(out,subexp)
02238             }
02239         }
02240         startobject = 0; t = tt; continue;
02241     }
02242     else {
02243         tt = t; pow = 1; x2 = 1; goto dopowerd;
02244     }
02245 }
02246 t++;
02247 }
02248 if ( last == 0 ) { last = 1; goto dolast; }
02249 *out = TENDOFIT;
02250 e->numfactors = powfactor-1;
02251 e->vflags |= ISFACTORIZED;
02252 subexp = CodeGenerator(outtokens);
02253 if ( subexp < 0 ) error = -1;
02254 if ( insubexpbuffers >= MAXSUBEXPRESSIONS ) {
02255     MesPrint("&More than %d subexpressions inside one expression", (WORD)MAXSUBEXPRESSIONS);
02256     Terminate(-1);
02257 }
02258 if ( subexpbuffers+insubexpbuffers >= topsubexpbuffers ) {
02259     DoubleBuffer((void **)((void *)(&subexpbuffers))
02260     , (void **)((void *)(&topsubexpbuffers)), sizeof(SUBBUF), "subexpbuffers");
02261 }
02262 subexpbuffers[insubexpbuffers].subexpnum = subexp;
02263 subexpbuffers[insubexpbuffers].buffernum = AC.cbufnum;
02264 subexp = insubexpbuffers++;
02265 M_free(outtokens, "CodeFactors");
02266 s1 = tokens;
02267 *s1++ = TSUBEXP; PUTNUMBER128(s1,subexp); *s1++ = TENDOFIT;
02268 if ( error < 0 ) return(-1);
02269 else return(subexp);

```

```

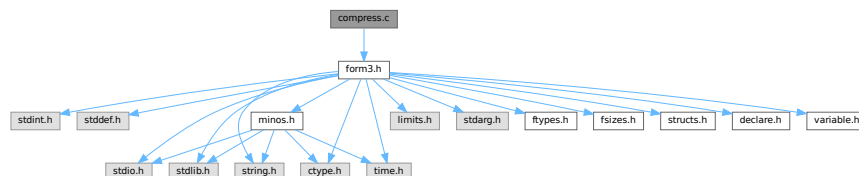
02270 }
02271
02272 /*
02273     #] CodeFactors :
02274     #[ GenerateFactors :
02275
02276     Generates an expression of the type
02277     1+factor_+factor_^2+...+factor_^(n-1)
02278     (this is if inc=1)
02279     Returns the subexpression pointer of it.
02280 */
02281
02282 WORD GenerateFactors(WORD n,WORD inc)
02283 {
02284     int subexp;
02285     int i, error = 0;
02286     SBYTE *s;
02287     SBYTE *tokenbuffer = (SBYTE *)Malloc1(8*n*sizeof(SBYTE), "GenerateFactors");
02288     s = tokenbuffer;
02289     *s++ = TNUMBER; *s++ = 1;
02290     for ( i = inc; i < n*inc; i += inc ) {
02291         *s++ = TPLUS; *s++ = TSYMBOL; *s++ = FACTORSYMBOL;
02292         if ( i > 1 ) {
02293             *s++ = TPOWER; *s++ = TNUMBER;
02294             PUTNUMBER100(s,i)
02295         }
02296     }
02297     *s++ = TENDOFIT;
02298     subexp = CodeGenerator(tokenbuffer);
02299     if ( subexp < 0 ) error = -1;
02300     M_free(tokenbuffer, "GenerateFactors");
02301     if ( insubexpbuffers >= MAXSUBEXPRESSIONS ) {
02302         MesPrint("&More than %d subexpressions inside one expression", (WORD)MAXSUBEXPRESSIONS);
02303         Terminate(-1);
02304     }
02305     if ( subexpbuffers+insubexpbuffers >= topsubexpbuffers ) {
02306         DoubleBuffer((void **) ((void *)(&subexpbuffers))
02307             , (void **) ((void *)(&topsubexpbuffers)), sizeof(SUBBUF), "subexpbuffers");
02308     }
02309     subexpbuffers[insubexpbuffers].subexpnum = subexp;
02310     subexpbuffers[insubexpbuffers].buffernum = AC.cbufnum;
02311     subexp = insubexpbuffers++;
02312     if ( error < 0 ) return(error);
02313     return(subexp);
02314 }
02315
02316 /*
02317     #] GenerateFactors :
02318     #] Compiler :
02319 */

```

4.13 compress.c File Reference

```
#include "form3.h"
```

Include dependency graph for compress.c:



4.13.1 Detailed Description

The routines for the use of gzip (de)compression of the information in the sort file.

Definition in file [compress.c](#).

4.14 compress.c

[Go to the documentation of this file.](#)

```

00001
00007 /* #[] License : */
00008 /*
00009  * Copyright (C) 1984-2026 J.A.M. Vermaseren
00010  * When using this file you are requested to refer to the publication
00011  * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00012  * This is considered a matter of courtesy as the development was paid
00013  * for by FOM the Dutch physics granting agency and we would like to
00014  * be able to track its scientific use to convince FOM of its value
00015  * for the community.
00016  *
00017  * This file is part of FORM.
00018  *
00019  * FORM is free software: you can redistribute it and/or modify it under the
00020  * terms of the GNU General Public License as published by the Free Software
00021  * Foundation, either version 3 of the License, or (at your option) any later
00022  * version.
00023  *
00024  * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00025  * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00026  * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00027  * details.
00028  *
00029  * You should have received a copy of the GNU General Public License along
00030  * with FORM. If not, see <http://www.gnu.org/licenses/>.
00031  */
00032 /* #[] License : */
00033
00034 #include "form3.h"
00035
00036 #ifndef WITHZLIB
00037 /*
00038 #define GZIPDEBUG
00039
00040 Low level routines for dealing with zlib during sorting and handling
00041 the scratch files. Work started 5-sep-2005.
00042 The .sor file handling was more or less completed on 8-sep-2005
00043 The handling of the scratch files still needs some thinking.
00044 Complications are:
00045     gzip compression should be per expression, not per buffer.
00046     No gzip compression for expressions with a bracket index.
00047     Separate decompression buffers for expressions in the rhs.
00048     This last one will involve more buffer work and organization.
00049     Information about compression should be stored for each expr.
00050     (including what method/program was used)
00051 Note: Be careful with compression. By far the most compact method
00052 is the original problem....
00053
00054 #[] Variables :
00055
00056 The following variables are to contain the intermediate buffers
00057 for the inflation of the various patches in the sort file.
00058 There can be up to MaxFpatches (FilePatches in the setup) and hence
00059 we can have that many streams simultaneously. We set this up once
00060 and only when needed.
00061     (in struct A.N or AB[threadnum].N)
00062 Bytef **AN.ziobufnum;
00063 Bytef *AN.ziobuffers;
00064 */
00065
00066 /*
00067 #[] Variables :
00068 #[] SetupOutputGZIP :
00069
00070 Routine prepares a gzip output stream for the given file.
00071 */
00072
00073 int SetupOutputGZIP(FILEHANDLE *f)
00074 {
00075     GETIDENTITY
00076
00077     if ( AT.SS != AT.S0 ) return(0);
00078     if ( AR.NoCompress == 1 ) return(0);
00079     if ( AR.gzipCompress <= 0 ) return(0);
00080
00081     if ( f->ziobuffer == 0 ) {
00082 /*
00083     1: Allocate a struct for the gzip stream:
00084 */
00085         f->zsp = Malloc1(sizeof(z_stream),"output zstream");
00086 /*
00087     2: Allocate the output buffer.

```

```

00088 */
00089     f->ziobuffer =
00090         (Bytef *)Malloc1(f->ziosize*sizeof(char),"output zbuffer");
00091     if ( f->zsp == 0 || f->ziobuffer == 0 ) {
00092         MLOCK(ErrorMessageLock);
00093         MesCall("SetupOutputGZIP");
00094         MUNLOCK(ErrorMessageLock);
00095         Terminate(-1);
00096     }
00097 }
00098 /*
00099 3: Set the default fields:
00100 */
00101 f->zsp->zalloc = Z_NULL;
00102 f->zsp->zfree  = Z_NULL;
00103 f->zsp->opaque = Z_NULL;
00104 /*
00105 4: Set the output space:
00106 */
00107 f->zsp->next_out = f->ziobuffer;
00108 f->zsp->avail_out = f->ziosize;
00109 f->zsp->total_out = 0;
00110 /*
00111 5: Set the input space:
00112 */
00113 f->zsp->next_in = (Bytef *) (f->PObuffer);
00114 f->zsp->avail_in = (Bytef *) (f->POfill) - (Bytef *) (f->PObuffer);
00115 f->zsp->total_in = 0;
00116 /*
00117 6: Initiate the deflation
00118 */
00119 if ( deflateInit(f->zsp,AR.gzipCompress) != Z_OK ) {
00120     MLOCK(ErrorMessageLock);
00121     MesPrint("Error from zlib: %s",f->zsp->msg);
00122     MesCall("SetupOutputGZIP");
00123     MUNLOCK(ErrorMessageLock);
00124     Terminate(-1);
00125 }
00126
00127 return(0);
00128 }
00129
00130 /*
00131 #] SetupOutputGZIP :
00132 #[ PutOutputGZIP :
00133
00134 Routine is called when the PObuffer of f is full.
00135 The contents of it will be compressed and whenever the output buffer
00136 f->ziobuffer is full it will be written and the output buffer
00137 will be reset.
00138 Upon exit the input buffer will be cleared.
00139 */
00140
00141 int PutOutputGZIP(FILEHANDLE *f)
00142 {
00143     GETIDENTITY
00144     int zerror;
00145     /*
00146     First set the number of bytes in the input
00147     */
00148     f->zsp->next_in = (Bytef *) (f->PObuffer);
00149     f->zsp->avail_in = (Bytef *) (f->POfill) - (Bytef *) (f->PObuffer);
00150     f->zsp->total_in = 0;
00151
00152     while ( ( zerror = deflate(f->zsp,Z_NO_FLUSH) ) == Z_OK ) {
00153         if ( f->zsp->avail_out == 0 ) {
00154             /*
00155             zibuffer is full. Write the output.
00156             */
00157             #ifdef GZIPDEBUG
00158             {
00159                 char *s = (char *) ((UBYTE *) (f->ziobuffer)+f->ziosize);
00160                 MLOCK(ErrorMessageLock);
00161                 MesPrint("%wWriting %l bytes at %10p: %d %d %d %d %d"
00162                     ,f->ziosize,&(f->POposition),s[-5],s[-4],s[-3],s[-2],s[-1]);
00163                 MUNLOCK(ErrorMessageLock);
00164             }
00165             #endif
00166             #ifdef ALLLOCK
00167             LOCK(f->pthreadlock);
00168             #endif
00169             if ( f == AR.hidefile ) {
00170                 LOCK(AS.inputslock);
00171             }
00172             SeekFile(f->handle,&(f->POposition),SEEK_SET);
00173             if ( WriteFile(f->handle,(UBYTE *) (f->ziobuffer),f->ziosize)
00174                 != f->ziosize ) {

```



```

00175         if ( f == AR.hidefile ) {
00176             UNLOCK(AS.inputslock);
00177         }
00178 #ifdef ALLLOCK
00179         UNLOCK(f->pthreadlock);
00180 #endif
00181         MLOCK(ErrorMessageLock);
00182         MesPrint("%wWrite error during compressed sort. Disk full?");
00183         MUNLOCK(ErrorMessageLock);
00184         return(-1);
00185     }
00186     if ( f == AR.hidefile ) {
00187         UNLOCK(AS.inputslock);
00188     }
00189 #ifdef ALLLOCK
00190     UNLOCK(f->pthreadlock);
00191 #endif
00192     ADDPOS(f->filesize,f->ziosize);
00193     ADDPOS(f->POposition,f->ziosize);
00194 #ifdef WITHPTHREADS
00195     if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
00196         if ( f->handle >= 0 ) SynchFile(f->handle);
00197     }
00198 #endif
00199 /*
00200     Reset the output
00201 */
00202     f->zsp->next_out = f->ziobuffer;
00203     f->zsp->avail_out = f->ziosize;
00204     f->zsp->total_out = 0;
00205 }
00206 else if ( f->zsp->avail_in == 0 ) {
00207 /*
00208     We compressed everything and it sits in ziobuffer. Finish
00209 */
00210     return(0);
00211 }
00212 else {
00213     MLOCK(ErrorMessageLock);
00214     MesPrint("%w avail_in = %d, avail_out = %d.",f->zsp->avail_in,f->zsp->avail_out);
00215     MUNLOCK(ErrorMessageLock);
00216     break;
00217 }
00218 }
00219 MLOCK(ErrorMessageLock);
00220 MesPrint("%wError in gzip handling of output. zerror = %d",zerror);
00221 MUNLOCK(ErrorMessageLock);
00222 return(-1);
00223 }
00224 /*
00225 #] PutOutputGZIP :
00226 #[ FlushOutputGZIP :
00227
00228 Routine is called to flush a stream. The compression of the input buffer
00229 will be completed and the contents of f->ziobuffer will be written.
00230 Both buffers will be cleared.
00231 */
00232 int FlushOutputGZIP(FILEHANDLE *f)
00233 {
00234     GETIDENTITY
00235     int zerror;
00236 /*
00237     Set the proper parameters
00238 */
00239     f->zsp->next_in = (Bytef *) (f->PObuffer);
00240     f->zsp->avail_in = (Bytef *) (f->POfill) - (Bytef *) (f->PObuffer);
00241     f->zsp->total_in = 0;
00242     while ( ( zerror = deflate(f->zsp,Z_FINISH) ) == Z_OK ) {
00243         if ( f->zsp->avail_out == 0 ) {
00244             /*
00245                 Write the output
00246             */
00247             #ifdef GZIPDEBUG
00248                 MLOCK(ErrorMessageLock);
00249                 MesPrint("%wWriting %l bytes at %l0p",f->ziosize,&(f->POposition));
00250                 MUNLOCK(ErrorMessageLock);
00251             #endif
00252             #ifdef ALLLOCK
00253                 LOCK(f->pthreadlock);
00254             #endif
00255             if ( f == AR.hidefile ) {
00256                 UNLOCK(AS.inputslock);
00257             }
00258             SeekFile(f->handle,&(f->POposition),SEEK_SET);

```

```

00262         if ( WriteFile(f->handle, (UBYTE *) (f->ziobuffer), f->ziosize)
00263             != f->ziosize ) {
00264             if ( f == AR.hidefile ) {
00265                 UNLOCK(AS.inputslock);
00266             }
00267 #ifdef ALLLOCK
00268         UNLOCK(f->pthreadlock);
00269 #endif
00270         MLOCK(ErrorMessageLock);
00271         MesPrint("%wWrite error during compressed sort. Disk full?");
00272         MUNLOCK(ErrorMessageLock);
00273         return(-1);
00274     }
00275     if ( f == AR.hidefile ) {
00276         UNLOCK(AS.inputslock);
00277     }
00278 #ifdef ALLLOCK
00279     UNLOCK(f->pthreadlock);
00280 #endif
00281     ADDPOS(f->filesize, f->ziosize);
00282     ADDPOS(f->PPosition, f->ziosize);
00283 #ifdef WITHPTHREADS
00284     if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
00285         if ( f->handle >= 0 ) SynchFile(f->handle);
00286     }
00287 #endif
00288 /*
00289     Reset the output
00290 */
00291     f->zsp->next_out = f->ziobuffer;
00292     f->zsp->avail_out = f->ziosize;
00293     f->zsp->total_out = 0;
00294 }
00295 }
00296 if ( zerror == Z_STREAM_END ) {
00297 /*
00298     Write the output
00299 */
00300 #ifdef GZIPDEBUG
00301     MLOCK(ErrorMessageLock);
00302     MesPrint("%wWriting %l bytes at %l0p", (LONG) (f->zsp->avail_out), &(f->PPosition));
00303     MUNLOCK(ErrorMessageLock);
00304 #endif
00305 #ifdef ALLLOCK
00306     LOCK(f->pthreadlock);
00307 #endif
00308     if ( f == AR.hidefile ) {
00309         LOCK(AS.inputslock);
00310     }
00311     SeekFile(f->handle, &(f->PPosition), SEEK_SET);
00312     if ( WriteFile(f->handle, (UBYTE *) (f->ziobuffer), f->zsp->total_out)
00313         != (LONG) (f->zsp->total_out) ) {
00314         if ( f == AR.hidefile ) {
00315             UNLOCK(AS.inputslock);
00316         }
00317 #ifdef ALLLOCK
00318         UNLOCK(f->pthreadlock);
00319 #endif
00320         MLOCK(ErrorMessageLock);
00321         MesPrint("%wWrite error during compressed sort. Disk full?");
00322         MUNLOCK(ErrorMessageLock);
00323         return(-1);
00324     }
00325     if ( f == AR.hidefile ) {
00326         LOCK(AS.inputslock);
00327     }
00328 #ifdef ALLLOCK
00329     UNLOCK(f->pthreadlock);
00330 #endif
00331     ADDPOS(f->filesize, f->zsp->total_out);
00332     ADDPOS(f->PPosition, f->zsp->total_out);
00333 #ifdef WITHPTHREADS
00334     if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
00335         if ( f->handle >= 0 ) SynchFile(f->handle);
00336     }
00337 #endif
00338 #ifdef GZIPDEBUG
00339     MLOCK(ErrorMessageLock);
00340     { char *s = f->ziobuffer+f->zsp->total_out;
00341       MesPrint("%w    Last bytes written: %d %d %d %d %d", s[-5], s[-4], s[-3], s[-2], s[-1]);
00342     }
00343     MesPrint("%w    Perceived position in FlushOutputGZIP is %l0p", &(f->PPosition));
00344     MUNLOCK(ErrorMessageLock);
00345 #endif
00346 /*
00347     Reset the output
00348 */

```

```

00349         f->zsp->next_out = f->ziobuffer;
00350         f->zsp->avail_out = f->ziosize;
00351         f->zsp->total_out = 0;
00352         if ( ( zerror = deflateEnd(f->zsp) ) == Z_OK ) return(0);
00353         MLOCK(ErrorMessageLock);
00354         if ( f->zsp->msg ) {
00355             MesPrint("%wError in finishing gzip handling of output: %s",f->zsp->msg);
00356         }
00357         else {
00358             MesPrint("%wError in finishing gzip handling of output.");
00359         }
00360         MUNLOCK(ErrorMessageLock);
00361     }
00362     else {
00363         MLOCK(ErrorMessageLock);
00364         MesPrint("%wError in gzip handling of output.");
00365         MUNLOCK(ErrorMessageLock);
00366     }
00367     return(-1);
00368 }
00369
00370 /*
00371  #] FlushOutputGZIP :
00372  #[ SetupAllInputGZIP :
00373
00374  Routine prepares all gzip input streams for a merge.
00375
00376  Problem (29-may-2008): If we never use GZIP compression, this routine
00377  will still allocate the array space. This is an enormous amount!
00378  It places an effective restriction on the value of SortIOSize
00379  */
00380
00381 int SetupAllInputGZIP(SORTING *S)
00382 {
00383     GETIDENTITY
00384     int i, NumberOpened = 0;
00385     z_stream zsp;
00386     /*
00387     This code was added 29-may-2008 by JV to prevent further processing if
00388     there is no compression at all (usually).
00389     */
00390     for ( i = 0; i < S->inNum; i++ ) {
00391         if ( S->fpincompressed[i] ) break;
00392     }
00393     if ( i >= S->inNum ) return(0);
00394
00395     if ( S->zsparray == 0 ) {
00396         S->zsparray = (z_stream*)Malloc1(sizeof(z_stream)*S->MaxFpatches,"input zstreams");
00397         if ( S->zsparray == 0 ) {
00398             MLOCK(ErrorMessageLock);
00399             MesCall("SetupAllInputGZIP");
00400             MUNLOCK(ErrorMessageLock);
00401             Terminate(-1);
00402         }
00403     }
00404     /*
00405     We add 128 bytes in the hope that if it can happen that it goes
00406     outside the buffer during decompression, it does not do damage.
00407     */
00408     AN.ziobuffers = (Bytef *)Malloc1(S->MaxFpatches*(S->file.ziosize+128)*sizeof(Bytef),"input raw
00409 buffers");
00410     /*
00411     This seems to be one of the really stupid errors:
00412     We allocate way too much space. Way way way too much.
00413     */
00414     AN.ziobufnum = (Bytef **)Malloc1(S->MaxFpatches*S->file.ziosize*sizeof(Bytef *),"input raw
00415 pointers");
00416     AN.ziobufnum = (Bytef **)Malloc1(S->MaxFpatches*sizeof(Bytef *),"input raw pointers");
00417     if ( AN.ziobuffers == 0 || AN.ziobufnum == 0 ) {
00418         MLOCK(ErrorMessageLock);
00419         MesCall("SetupAllInputGZIP");
00420         MUNLOCK(ErrorMessageLock);
00421         Terminate(-1);
00422     }
00423     for ( i = 0; i < S->MaxFpatches; i++ ) {
00424         AN.ziobufnum[i] = AN.ziobuffers + i * (S->file.ziosize+128);
00425     }
00426     for ( i = 0; i < S->inNum; i++ ) {
00427         #ifdef GZIPDEBUG
00428             MLOCK(ErrorMessageLock);
00429             MesPrint("%wPreparing z-stream %d with compression %d",i,S->fpincompressed[i]);
00430             MUNLOCK(ErrorMessageLock);
00431         #endif
00432         if ( S->fpincompressed[i] ) {
00433             zsp = &(S->zsparray[i]);
00434             /*
00435             1: Set the default fields:

```

```

00434 */
00435         zsp->zalloc = Z_NULL;
00436         zsp->zfree  = Z_NULL;
00437         zsp->opaque = Z_NULL;
00438 /*
00439         2: Set the output space:
00440 */
00441         zsp->next_out = Z_NULL;
00442         zsp->avail_out = 0;
00443         zsp->total_out = 0;
00444 /*
00445         3: Set the input space temporarily:
00446 */
00447         zsp->next_in = Z_NULL;
00448         zsp->avail_in = 0;
00449         zsp->total_in = 0;
00450 /*
00451         4: Initiate the inflation
00452 */
00453         if ( inflateInit(zsp) != Z_OK ) {
00454             MLOCK(ErrorMessageLock);
00455             if ( zsp->msg ) MesPrint("%wError from inflateInit: %s",zsp->msg);
00456             else           MesPrint("%wError from inflateInit");
00457             MesCall("SetupAllInputGZIP");
00458             MUNLOCK(ErrorMessageLock);
00459             Terminate(-1);
00460         }
00461         NumberOpened++;
00462     }
00463 }
00464 return (NumberOpened);
00465 }
00466
00467 /*
00468     #] SetupAllInputGZIP :
00469     #[ FillInputGZIP :
00470
00471     Routine is called when we need new input in the specified buffer.
00472     This buffer is used for the output and we keep reading and uncompressing
00473     input till either this buffer is full or the input stream is finished.
00474     The return value is the number of bytes in the buffer.
00475 */
00476
00477 LONG FillInputGZIP(FILEHANDLE *f, POSITION *position, UBYTE *buffer, LONG buffersize, int numstream)
00478 {
00479     GETIDENTITY
00480     int zerror;
00481     LONG readsize, toread = 0;
00482     SORTING *S = AT.SS;
00483     z_streamp zsp;
00484     POSITION pos;
00485     if ( S->fpincompressed[numstream] ) {
00486         zsp = &(S->zsparray[numstream]);
00487         zsp->next_out = (Bytef *)buffer;
00488         zsp->avail_out = buffersize;
00489         zsp->total_out = 0;
00490         if ( zsp->avail_in == 0 ) {
00491 /*
00492             First loading of the input
00493 */
00494             if ( ISGEPOSINC(S->fPatchesStop[numstream],*position,f->ziosize) ) {
00495                 toread = f->ziosize;
00496             }
00497             else {
00498                 DIFFPOS(pos,S->fPatchesStop[numstream],*position);
00499                 toread = (LONG) (BASEPOSITION(pos));
00500             }
00501             if ( toread > 0 ) {
00502 #ifdef GZIPDEBUG
00503                 MLOCK(ErrorMessageLock);
00504                 MesPrint("%w-+Reading %l bytes in stream %d at position %l0p; stop at
00505 %l0p",toread,numstream,position,&(S->fPatchesStop[numstream]));
00506                 MUNLOCK(ErrorMessageLock);
00507 #endif
00508 #ifdef ALLLOCK
00509                 LOCK(f->pthreadlock);
00510 #endif
00511                 SeekFile(f->handle,position,SEEK_SET);
00512                 readsize = ReadFile(f->handle,(UBYTE *) (AN.ziobufnum[numstream]),toread);
00513                 SeekFile(f->handle,position,SEEK_CUR);
00514 #ifdef ALLLOCK
00515                 UNLOCK(f->pthreadlock);
00516 #endif
00517 #ifdef GZIPDEBUG
00518                 MLOCK(ErrorMessageLock);
00519                 { char *s = AN.ziobufnum[numstream]+readsize;
00520                     MesPrint("%w read: %l +Last bytes read: %d %d %d %d %d in %s, newpos =

```

```

        %10p", readsize, s[-5], s[-4], s[-3], s[-2], s[-1], f->name, position);
00520     }
00521     MUNLOCK(ErrorMessageLock);
00522 #endif
00523     if ( readsize == 0 ) {
00524         zsp->next_in = AN.ziobufnum[numstream];
00525         zsp->avail_in = f->ziosize;
00526         zsp->total_in = 0;
00527         return(zsp->total_out);
00528     }
00529     if ( readsize < 0 ) {
00530         MLOCK(ErrorMessageLock);
00531         MesPrint("%wFillInputGZIP: Read error during compressed sort.");
00532         MUNLOCK(ErrorMessageLock);
00533         return(-1);
00534     }
00535     ADDPOS(f->filesize, readsize);
00536     ADDPOS(f->PPosition, readsize);
00537 /*
00538     Set the input
00539 */
00540     zsp->next_in = AN.ziobufnum[numstream];
00541     zsp->avail_in = readsize;
00542     zsp->total_in = 0;
00543 }
00544 }
00545 if ( toread > 0 || zsp->avail_in ) {
00546     while ( ( zerror = inflate(zsp, Z_NO_FLUSH) ) == Z_OK ) {
00547         if ( zsp->avail_out == 0 ) {
00548 /*
00549             Finish
00550 */
00551             return((LONG) (zsp->total_out));
00552         }
00553         if ( zsp->avail_in == 0 ) {
00554             if ( ISEQUALPOS(S->fPatchesStop[numstream], *position) ) {
00555 /*
00556                 We finished this stream. Try to terminate.
00557 */
00558                 zerror = Z_STREAM_END;
00559                 break;
00560             }
00561             if ( ( zerror = inflate(zsp, Z_SYNC_FLUSH) ) == Z_OK ) {
00562                 return((LONG) (zsp->total_out));
00563             }
00564             else {
00565                 break;
00566             }
00567 /*
00568 */
00569 #ifdef GZIPDEBUG
00570             MLOCK(ErrorMessageLock);
00571             MesPrint("%wClosing stream %d", numstream);
00572 #endif
00573             readsize = zsp->total_out;
00574 #ifdef GZIPDEBUG
00575             if ( readsize > 0 ) {
00576                 WORD *s = (WORD *) (buffer+zsp->total_out);
00577                 MesPrint("%w    Last words: %d %d %d %d %d", s[-5], s[-4], s[-3], s[-2], s[-1]);
00578             }
00579             else {
00580                 MesPrint("%w No words");
00581             }
00582             MUNLOCK(ErrorMessageLock);
00583 #endif
00584             if ( ( zerror = inflateEnd(zsp) ) == Z_OK ) return(readsize);
00585             break;
00586 /*
00587 */
00588             }
00589             Read more input
00590 */
00591 #ifdef GZIPDEBUG
00592             if ( numstream == 0 ) {
00593                 MLOCK(ErrorMessageLock);
00594                 MesPrint("%wWant to read in stream 0 at position %10p", position);
00595                 MUNLOCK(ErrorMessageLock);
00596             }
00597 #endif
00598             if ( ISGEPOSINC(S->fPatchesStop[numstream], *position, f->ziosize) ) {
00599                 toread = f->ziosize;
00600             }
00601             else {
00602                 DIFPOS(pos, S->fPatchesStop[numstream], *position);
00603                 toread = (LONG) (BASEPOSITION(pos));
00604             }
00605 #ifdef GZIPDEBUG

```

```

00606             MLOCK(ErrorMessageLock);
00607             MesPrint("%w--Reading %l bytes in stream %d at position
%10p",toread,numstream,position);
00608             MUNLOCK(ErrorMessageLock);
00609 #endif
00610 #ifdef ALLLOCK
00611             LOCK(f->pthreadlock);
00612 #endif
00613             SeekFile(f->handle,position,SEEK_SET);
00614             readsize = ReadFile(f->handle,(UBYTE *) (AN.ziobufnum[numstream]),toread);
00615             SeekFile(f->handle,position,SEEK_CUR);
00616 #ifdef ALLLOCK
00617             UNLOCK(f->pthreadlock);
00618 #endif
00619 #ifdef GZIPDEBUG
00620             MLOCK(ErrorMessageLock);
00621             { char *s = AN.ziobufnum[numstream]+readsize;
00622               MesPrint("%w Last bytes read: %d %d %d %d %d",s[-5],s[-4],s[-3],s[-2],s[-1]);
00623             }
00624             MUNLOCK(ErrorMessageLock);
00625 #endif
00626             if ( readsize == 0 ) {
00627                 zsp->next_in = AN.ziobufnum[numstream];
00628                 zsp->avail_in = f->ziosize;
00629                 zsp->total_in = 0;
00630                 return(zsp->total_out);
00631             }
00632             if ( readsize < 0 ) {
00633                 MLOCK(ErrorMessageLock);
00634                 MesPrint("%wFillInputGZIP: Read error during compressed sort.");
00635                 MUNLOCK(ErrorMessageLock);
00636                 return(-1);
00637             }
00638             ADDPOS(f->filesize,readsize);
00639             ADDPOS(f->POposition,readsize);
00640 /*
00641             Reset the input
00642 */
00643             zsp->next_in = AN.ziobufnum[numstream];
00644             zsp->avail_in = readsize;
00645             zsp->total_in = 0;
00646         }
00647         else {
00648             break;
00649         }
00650     }
00651 }
00652 else {
00653     zerror = Z_STREAM_END;
00654     zsp->total_out = 0;
00655 }
00656 #ifdef GZIPDEBUG
00657 MLOCK(ErrorMessageLock);
00658 MesPrint("%w zerror = %d in stream %d. At position %10p",zerror,numstream,position);
00659 MUNLOCK(ErrorMessageLock);
00660 #endif
00661 if ( zerror == Z_STREAM_END ) {
00662 /*
00663     Reset the input
00664 */
00665     zsp->next_in = Z_NULL;
00666     zsp->avail_in = 0;
00667     zsp->total_in = 0;
00668 /*
00669     Make the final call and finish
00670 */
00671 #ifdef GZIPDEBUG
00672 MLOCK(ErrorMessageLock);
00673 MesPrint("%wClosing stream %d",numstream);
00674 #endif
00675     readsize = zsp->total_out;
00676 #ifdef GZIPDEBUG
00677     if ( readsize > 0 ) {
00678         WORD *s = (WORD *) (buffer+zsp->total_out);
00679         MesPrint("%w -Last words: %d %d %d %d %d",s[-5],s[-4],s[-3],s[-2],s[-1]);
00680     }
00681     else {
00682         MesPrint("%w No words");
00683     }
00684     MUNLOCK(ErrorMessageLock);
00685 #endif
00686     if ( zsp->zalloc != Z_NULL ) {
00687         zerror = inflateEnd(zsp);
00688         zsp->zalloc = Z_NULL;
00689     }
00690     if ( zerror == Z_OK || zerror == Z_STREAM_END ) return(readsize);
00691 }

```

```

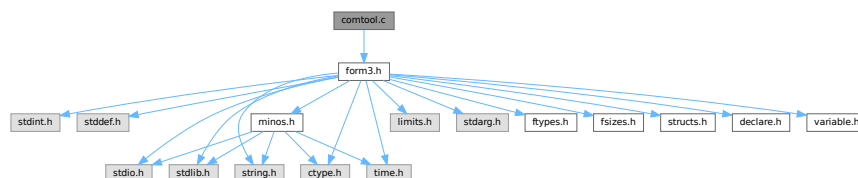
00692
00693     MLOCK(ErrorMessageLock);
00694     MesPrint("%wFillInputGZIP: Error in gzip handling of input. zerror = %d",zerror);
00695     MUNLOCK(ErrorMessageLock);
00696     return(-1);
00697 }
00698 else {
00699 #ifdef GZIPDEBUG
00700     MLOCK(ErrorMessageLock);
00701     MesPrint("%w++Reading %l bytes at position %l0p",bufferize,position);
00702     MUNLOCK(ErrorMessageLock);
00703 #endif
00704 #ifdef ALLLOCK
00705     LOCK(f->pthreadlock);
00706 #endif
00707     SeekFile(f->handle,position,SEEK_SET);
00708     readsize = ReadFile(f->handle,buffer,bufferize);
00709     SeekFile(f->handle,position,SEEK_CUR);
00710 #ifdef ALLLOCK
00711     UNLOCK(f->pthreadlock);
00712 #endif
00713     if ( readsize < 0 ) {
00714         MLOCK(ErrorMessageLock);
00715         MesPrint("%wFillInputGZIP: Read error during uncompressed sort.");
00716         MesPrint("%w++Reading %l bytes at position %l0p",bufferize,position);
00717         MUNLOCK(ErrorMessageLock);
00718     }
00719     return(readsize);
00720 }
00721 }
00722
00723 /*
00724 #] FillInputGZIP :
00725 #[ ClearSortGZIP :
00726 */
00727
00728 void ClearSortGZIP(FILEHANDLE *f)
00729 {
00730     if ( f->zibuffer ) {
00731         M_free(f->zibuffer,"output zbuffer");
00732         M_free(f->zsp,"output zstream");
00733         f->zibuffer = 0;
00734     }
00735 }
00736
00737 /*
00738 #] ClearSortGZIP :
00739 */
00740 #endif

```

4.15 comtool.c File Reference

```
#include "form3.h"
```

Include dependency graph for comtool.c:



Functions

- int [inicbufs](#) (void)
- void [finishcbuf](#) (WORD num)
- void [clearcbuf](#) (WORD num)

- WORD * [DoubleCbuffer](#) (int num, WORD *w, int par)
- WORD * [AddLHS](#) (int num)
- WORD * [AddRHS](#) (int num, int type)
- int [AddNtoL](#) (int n, WORD *array)
- int [AddNtoC](#) (int bufnum, int n, WORD *array, int par)
- int [InsTree](#) (int bufnum, int h)
- int [FindTree](#) (int bufnum, WORD *subexpr)
- void [RedoTree](#) (CBUF *C, int size)
- void [ClearTree](#) (int i)
- int [IniFbuffer](#) (WORD bufnum)
- LONG [numcommute](#) (WORD *terms, LONG *numterms)

4.15.1 Detailed Description

Utility routines for the compiler.

Definition in file [comtool.c](#).

4.15.2 Function Documentation

4.15.2.1 inicbufs()

```
int inicbufs (
    void )
```

Creates a new compiler buffer and returns its ID number.

Returns

The ID number for the new compiler buffer.

Definition at line 47 of file [comtool.c](#).

References [CbUf::boomlijst](#), [CbUf::Buffer](#), [CbUf::BufferSize](#), [CbUf::CanCommu](#), [CbUf::dimension](#), [CbUf::lhs](#), [CbUf::numdum](#), [CbUf::NumTerms](#), [CbUf::Pointer](#), [CbUf::rhs](#), and [CbUf::Top](#).

Referenced by [AddRHS\(\)](#), and [StartVariables\(\)](#).

4.15.2.2 finishcbuf()

```
void finishcbuf (
    WORD num )
```

Frees a compiler buffer.

Parameters

<i>num</i>	The ID number for the buffer to be freed.

Definition at line 89 of file [comtool.c](#).

References [CbUf::boomlijst](#), [CbUf::Buffer](#), [CbUf::BufferSize](#), [CbUf::CanCommu](#), [CbUf::lhs](#), [CbUf::NumTerms](#), [CbUf::Pointer](#), [CbUf::rhs](#), and [CbUf::Top](#).

Referenced by [PF_BroadcastCBuf\(\)](#).

4.15.2.3 clearcbuf()

```
void clearcbuf (
    WORD num )
```

Clears contents in a compiler buffer.

Parameters

<i>num</i>	The ID number for the buffer to be cleared.

Definition at line 116 of file [comtool.c](#).

References [CbUf::boomlijst](#), [CbUf::Buffer](#), [CbUf::Pointer](#), and [CbUf::rhs](#).

4.15.2.4 DoubleCbuffer()

```
WORD * DoubleCbuffer (
    int num,
    WORD * w,
    int par )
```

Doubles a compiler buffer.

Parameters

<i>num</i>	The ID number for the buffer to be doubled.
<i>w</i>	The pointer to the end (exclusive) of the current buffer. The contents in the range of [cbuff[num].Buffer,w) will be kept.

Definition at line 143 of file [comtool.c](#).

References [CbUf::Buffer](#), [CbUf::BufferSize](#), [CbUf::lhs](#), [CbUf::Pointer](#), [CbUf::rhs](#), and [CbUf::Top](#).

Referenced by [AddNtoC\(\)](#), and [AddNtoL\(\)](#).

4.15.2.5 AddLHS()

```
WORD * AddLHS (
    int num )
```

Adds an LHS to a compiler buffer and returns the pointer to a buffer for the new LHS.

Parameters

<i>num</i>	The ID number for the buffer to get another LHS.

Definition at line 188 of file [comtool.c](#).

References [CbUf::lhs](#), and [CbUf::Pointer](#).

Referenced by [AddNtoL\(\)](#).

4.15.2.6 AddRHS()

```
WORD * AddRHS (
    int num,
    int type )
```

Adds an RHS to a compiler buffer and returns the pointer to a buffer for the new RHS.

Parameters

<i>num</i>	The ID number for the buffer to get another RHS.
<i>type</i>	If 0, the subexpression tree will be reallocated.

Definition at line 214 of file [comtool.c](#).

References [TaBIEs::buffers](#), [TaBIEs::buffersfill](#), [TaBIEs::bufferssize](#), [TaBIEs::bufnum](#), [CbUf::CanCommu](#), [CbUf::dimension](#), [inicbufs\(\)](#), [CbUf::numdum](#), [CbUf::NumTerms](#), [CbUf::Pointer](#), and [CbUf::rhs](#).

Referenced by [InsertArg\(\)](#), [StartVariables\(\)](#), and [TestMatch\(\)](#).

Here is the call graph for this function:

**4.15.2.7 AddNtoL()**

```
int AddNtoL (
    int n,
    WORD * array )
```

Adds an LHS with the given data to the current compiler buffer.

Parameters

<i>n</i>	The length of the data.
<i>array</i>	The data to be added.

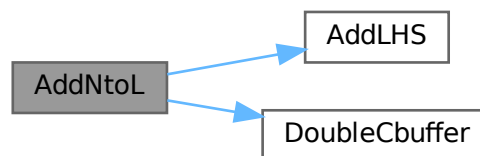
Returns

0 if succeeds.

Definition at line 288 of file [comtool.c](#).

References [AddLHS\(\)](#), [DoubleCbuffer\(\)](#), [CbUf::Pointer](#), and [CbUf::Top](#).

Here is the call graph for this function:



4.15.2.8 AddNtoC()

```

int AddNtoC (
    int bufnum,
    int n,
    WORD * array,
    int par )
  
```

Adds the given data to the last LHS/RHS in a compiler buffer.

Parameters

<i>bufnum</i>	The ID number for the buffer where the data will be added.
<i>n</i>	The length of the data.
<i>array</i>	The data to be added.

Returns

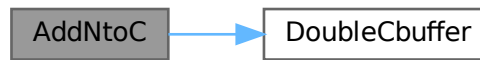
0 if succeeds.

Definition at line 317 of file [comtool.c](#).

References [DoubleCbuffer\(\)](#), [CbUf::Pointer](#), and [CbUf::Top](#).

Referenced by [InsertArg\(\)](#), and [StartVariables\(\)](#).

Here is the call graph for this function:



4.15.2.9 `InsTree()`

```
int InsTree (
    int bufnum,
    int h )
```

Definition at line [355](#) of file [comtool.c](#).

4.15.2.10 `FindTree()`

```
int FindTree (
    int bufnum,
    WORD * subexpr )
```

Definition at line [533](#) of file [comtool.c](#).

4.15.2.11 `RedoTree()`

```
void RedoTree (
    CBUF * C,
    int size )
```

Definition at line [569](#) of file [comtool.c](#).

4.15.2.12 `ClearTree()`

```
void ClearTree (
    int i )
```

Definition at line [588](#) of file [comtool.c](#).

4.15.2.13 IniFbuffer()

```
int IniFbuffer (
    WORD bufnum )
```

Initialize a factorization cache buffer. We set the size of the rhs and boomlijst buffers immediately to their final values.

Definition at line 614 of file [comtool.c](#).

References [tree::blnce](#), [CbUf::boomlijst](#), [CbUf::CanCommu](#), [CbUf::dimension](#), [tree::left](#), [CbUf::numdum](#), [CbUf::NumTerms](#), [tree::parent](#), [CbUf::rhs](#), [tree::right](#), [tree::usage](#), and [tree::value](#).

4.15.2.14 numcommute()

```
LONG numcommute (
    WORD * terms,
    LONG * numterms )
```

Definition at line 659 of file [comtool.c](#).

4.16 comtool.c

[Go to the documentation of this file.](#)

```
00001
00005 /* # [ License : */
00006 /*
00007 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00008 * When using this file you are requested to refer to the publication
00009 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010 * This is considered a matter of courtesy as the development was paid
00011 * for by FOM the Dutch physics granting agency and we would like to
00012 * be able to track its scientific use to convince FOM of its value
00013 * for the community.
00014 *
00015 * This file is part of FORM.
00016 *
00017 * FORM is free software: you can redistribute it and/or modify it under the
00018 * terms of the GNU General Public License as published by the Free Software
00019 * Foundation, either version 3 of the License, or (at your option) any later
00020 * version.
00021 *
00022 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00025 * details.
00026 *
00027 * You should have received a copy of the GNU General Public License along
00028 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00029 */
00030 /* # [ License : */
00031 /*
00032 * # [ Includes :
00033 */
00034
00035 #include "form3.h"
00036
00037 /*
00038 * # ] Includes :
00039 * # [ inicbufs :
00040 */
00041
00047 int inicbufs(void)
00048 {
00049     int i, num = AC.cbufList.num;
00050     CBUF *C = cbuf;
00051     for ( i = 0; i < num; i++, C++ ) {
00052         if ( C->Buffer == 0 ) break;
```

```

00053     }
00054     if ( i >= num ) C = (CBUF *)FromList(&AC.cbufList);
00055     else num = i;
00056     C->BufferSize = 2000;
00057     C->Buffer = (WORD *)Malloca1(C->BufferSize*sizeof(WORD),"compiler buffer-1");
00058     C->Pointer = C->Buffer;
00059     C->Top = C->Buffer + C->BufferSize;
00060     C->maxlhs = 10;
00061     C->lhs = (WORD **)Malloca1(C->maxlhs*sizeof(WORD *),"compiler buffer-2");
00062     C->numlhs = 0;
00063     C->mnumlhs = 0;
00064     C->maxrhs = 25;
00065     C->rhs = (WORD **)Malloca1(C->maxrhs*(sizeof(WORD *)+2*sizeof(LONG)+2*sizeof(WORD)),"compiler
buffer-3");
00066     C->CanCommu = (LONG *) (C->rhs+C->maxrhs);
00067     C->NumTerms = C->CanCommu+C->maxrhs;
00068     C->numdum = (WORD *) (C->NumTerms+C->maxrhs);
00069     C->dimension = C->numdum + C->maxrhs;
00070     C->numrhs = 0;
00071     C->mnumrhs = 0;
00072     C->rhs[0] = C->rhs[1] = C->Pointer;
00073     C->boomlijst = 0;
00074     RedoTree(C,C->maxrhs);
00075     ClearTree(num);
00076     return(num);
00077 }
00078
00079 /*
00080     #] inicbufs :
00081     #[ finishcbuf :
00082 */
00083
00089 void finishcbuf(WORD num)
00090 {
00091     CBUF *C = cbuf+num;
00092     if ( C->Buffer ) M_free(C->Buffer,"compiler buffer-1");
00093     if ( C->rhs ) M_free(C->rhs,"compiler buffer-3");
00094     if ( C->lhs ) M_free(C->lhs,"compiler buffer-2");
00095     if ( C->boomlijst ) M_free(C->boomlijst,"boomlijst");
00096     C->Top = C->Pointer = C->Buffer = 0;
00097     C->rhs = C->lhs = 0;
00098     C->CanCommu = 0;
00099     C->NumTerms = 0;
00100     C->BufferSize = 0;
00101     C->boomlijst = 0;
00102     C->numlhs = C->numrhs = C->maxlhs = C->maxrhs = C->mnumlhs =
00103     C->mnumrhs = C->numtree = C->rootnum = C->MaxTreeSize = 0;
00104 }
00105
00106 /*
00107     #] finishcbuf :
00108     #[ clearcbuf :
00109 */
00110
00116 void clearcbuf(WORD num)
00117 {
00118     CBUF *C = cbuf+num;
00119     if ( C->boomlijst ) M_free(C->boomlijst,"boomlijst");
00120     C->Pointer = C->Buffer;
00121     C->numrhs = C->numlhs = 0;
00122     C->mnumlhs = 0;
00123     C->boomlijst = 0;
00124     C->mnumrhs = 0;
00125     C->rhs[0] = C->rhs[1] = C->Pointer;
00126     C->numtree = C->rootnum = C->MaxTreeSize = 0;
00127     RedoTree(C,C->maxrhs);
00128     ClearTree(num);
00129 }
00130
00131 /*
00132     #] clearcbuf :
00133     #[ DoubleCbuffer :
00134 */
00135
00143 WORD *DoubleCbuffer(int num, WORD *w,int par)
00144 {
00145     CBUF *C = cbuf + num;
00146     LONG newsize = C->BufferSize*2;
00147     WORD *newbuffer = (WORD *)Malloca1(newsize*sizeof(WORD),"compiler buffer-4");
00148     WORD *wl, *w2;
00149     LONG offset, j, i;
00150     DUMMYUSE(par)
00151 /*
00152     MLOCK(ErrorMessageLock);
00153     MesPrint(" doubleCbuffer: par = %d",par);
00154     MUNLOCK(ErrorMessageLock);
00155 */

```

```

00156     w1 = C->Buffer; w2 = newbuffer;
00157     i = w - w1;
00158     j = i & 7;
00159     while ( --j >= 0 ) *w2++ = *w1++;
00160     i >= 3;
00161     while ( --i >= 0 ) {
00162         *w2++ = *w1++; *w2++ = *w1++; *w2++ = *w1++; *w2++ = *w1++;
00163         *w2++ = *w1++; *w2++ = *w1++; *w2++ = *w1++; *w2++ = *w1++;
00164     }
00165     offset = newbuffer - C->Buffer;
00166     for ( i = 0; i <= C->numlhs; i++ ) C->lhs[i] += offset;
00167     for ( i = 1; i <= C->numrhs; i++ ) C->rhs[i] += offset;
00168     w1 = C->Buffer;
00169     C->Pointer += offset;
00170     C->Top = newbuffer + newsize;
00171     C->BufferSize = newsize;
00172     C->Buffer = newbuffer;
00173     M_free(w1,"DoubleCbuffer");
00174     return(w2);
00175 }
00176
00177 /*
00178     #] DoubleCbuffer :
00179     #[ AddLHS :
00180 */
00181
00182 WORD *AddLHS(int num)
00183 {
00184     CBUF *C = cbuf + num;
00185     C->numlhs++;
00186     if ( C->numlhs >= (C->maxlhs-2) ) {
00187         WORD ***ppp = &(C->lhs); /* to avoid compiler warning */
00188         if ( DoubleList((void ***)ppp,&(C->maxlhs),sizeof(WORD *),
00189             "statement lists") ) Terminate(-1);
00190     }
00191     C->lhs[C->numlhs] = C->Pointer;
00192     C->lhs[C->numlhs+1] = 0;
00193     return(C->Pointer);
00194 }
00195
00196 /*
00197     #] AddLHS :
00198     #[ AddRHS :
00199 */
00200
00201 WORD *AddRHS(int num, int type)
00202 {
00203     LONG fullsize, *lold, newsize;
00204     int i;
00205     WORD *old, *wold;
00206     CBUF *C;
00207 restart:;
00208     C = cbuf + num;
00209     if ( C->numrhs >= (C->maxrhs-2) ) {
00210         if ( C->maxrhs == 0 ) newsize = 100;
00211         else newsize = C->maxrhs * 2;
00212         if ( newsize > MAXCOMBUFRHS ) newsize = MAXCOMBUFRHS;
00213         if ( newsize == C->maxrhs ) {
00214             if ( AC.tablefilling ) {
00215                 TABLES T = functions[AC.tablefilling].tabl;
00216                 /*
00217                     We add a compiler buffer, change a few settings and continue.
00218                 */
00219                 if ( T->buffersfill >= T->buffersize ) {
00220                     int new1 = 2*T->buffersize;
00221                     WORD *nbufs = (WORD *)Malloc1(new1*sizeof(WORD),"Table compile buffers");
00222                     for ( i = 0; i < T->buffersfill; i++ )
00223                         nbufs[i] = T->buffers[i];
00224                     for ( ; i < new1; i++ ) nbufs[i] = 0;
00225                     M_free(T->buffers,"Table compile buffers");
00226                     T->buffers = nbufs;
00227                     T->buffersize = new1;
00228                 }
00229                 T->buffers[T->buffersfill++] = T->bufnum = inicbufs();
00230                 AC.cbufnum = num = T->bufnum;
00231                 goto restart;
00232             }
00233             else {
00234                 MesPrint("@Compiler buffer overflow. Try to make modules smaller");
00235                 Terminate(-1);
00236             }
00237         }
00238     }
00239     old = C->rhs;
00240     fullsize = newsize * (sizeof(WORD *) + 2*sizeof(LONG) + 2*sizeof(WORD));
00241     C->rhs = (WORD **)Malloc1(fullsize,"subexpression lists");
00242     for ( i = 0; i < C->maxrhs; i++ ) C->rhs[i] = old[i];
00243     lold = C->CanCommu; C->CanCommu = (LONG *) (C->rhs+newsize);

```

```

00256         for ( i = 0; i < C->maxrhs; i++ ) C->CanCommu[i] = lold[i];
00257         lold = C->NumTerms; C->NumTerms = (LONG *) (C->rhs+2*newsize);
00258         for ( i = 0; i < C->maxrhs; i++ ) C->NumTerms[i] = lold[i];
00259         wold = C->numdum; C->numdum = (WORD *) (C->NumTerms+newsize);
00260         for ( i = 0; i < C->maxrhs; i++ ) C->numdum[i] = wold[i];
00261         wold = C->dimension; C->dimension = (WORD *) (C->numdum+newsize);
00262         for ( i = 0; i < C->maxrhs; i++ ) C->dimension[i] = wold[i];
00263         if ( old ) M_free(old,"subexpression lists");
00264         C->maxrhs = newsize;
00265         if ( type == 0 ) RedoTree(C,C->maxrhs);
00266     }
00267     C->numrhs++;
00268     C->CanCommu[C->numrhs] = 0;
00269     C->NumTerms[C->numrhs] = 0;
00270     C->numdum[C->numrhs] = 0;
00271     C->dimension[C->numrhs] = 0;
00272     C->rhs[C->numrhs] = C->Pointer;
00273     return(C->Pointer);
00274 }
00275
00276 /*
00277  #] AddRHS :
00278  #[ AddNtoL :
00279  */
00280
00281 int AddNtoL(int n, WORD *array)
00282 {
00283     int i;
00284     CBUF *C = cbuf+AC.cbunum;
00285     #ifdef COMPUFDEBBUG
00286     MesPrint("LH: %a",n,array);
00287     #endif
00288     AddLHS(AC.cbunum);
00289     while ( C->Pointer+n >= C->Top ) DoubleCbuffer(AC.cbunum,C->Pointer,1);
00290     for ( i = 0; i < n; i++ ) *(C->Pointer)++ = *array++;
00291     return(0);
00292 }
00293
00294 /*
00295  #] AddNtoL :
00296  #[ AddNtoC :
00297
00298     Commentary: added the bufnum on 14-sep-2010 to make the whole a bit
00299     more flexible (JV). Still to do with AddNtoL.
00300  */
00301
00302 int AddNtoC(int bufnum, int n, WORD *array,int par)
00303 {
00304     int i;
00305     WORD *w;
00306     CBUF *C = cbuf+bufnum;
00307     #ifdef COMPUFDEBBUG
00308     MesPrint("RH: %a",n,array);
00309     #endif
00310     while ( C->Pointer+n+1 >= C->Top ) DoubleCbuffer(bufnum,C->Pointer,50+par);
00311     w = C->Pointer;
00312     for ( i = 0; i < n; i++ ) *w++ = *array++;
00313     C->Pointer = w;
00314     return(0);
00315 }
00316
00317 /*
00318  #] AddNtoC :
00319  #[ InsTree :
00320
00321     Routines for balanced tree searching and insertion.
00322     Compared to Knuth we have a parent link. This minimizes the
00323     number of compares. That is better for anything that is more
00324     complicated than just single numbers.
00325     There are no provisions for removing elements from the tree.
00326     The routines are:
00327     void RedoTree(size) Re-allocates the tree space. There will
00328                        be MaxTreeSize = size elements.
00329     void ClearTree() Prunes the tree down to the root element.
00330     int InsTree(int,int) Searches for the requested element. If not found it
00331                        will allocate a new element, balance the tree if
00332                        necessary and return the called number.
00333                        If it was in the tree, it returns the tree 'value'.
00334
00335     Commentary: added the bufnum on 14-sep-2010 to make the whole a bit
00336     more flexible (JV).
00337  */
00338 static COMPTREE comptreezero = {0,0,0,0,0,0};
00339
00340 int InsTree(int bufnum, int h)
00341 {
00342     CBUF *C = cbuf + bufnum;

```



```

00358     COMPTREE *boomlijst = C->boomlijst, *q = boomlijst + C->rootnum, *p, *s;
00359     WORD *v1, *v2, *v3;
00360     int ip, iq, is;
00361
00362     if ( C->numtree + 1 >= C->MaxTreeSize ) {
00363         if ( C->MaxTreeSize == 0 ) {
00364             COMPTREE *root;
00365             C->MaxTreeSize = 125;
00366             C->boomlijst = (COMPTREE *)Malloc1((C->MaxTreeSize+1)*sizeof(COMPTREE),
00367                 "ClearInsTree");
00368             root = C->boomlijst;
00369             C->numtree = 0;
00370             C->rootnum = 0;
00371             root->left = -1;
00372             root->right = -1;
00373             root->parent = -1;
00374             root->blnce = 0;
00375             root->value = -1;
00376             root->usage = 0;
00377             for ( ip = 1; ip < C->MaxTreeSize; ip++ ) { C->boomlijst[ip] = comptreezero; }
00378         }
00379         else {
00380             is = C->MaxTreeSize * 2;
00381             s = (COMPTREE *)Malloc1((is+1)*sizeof(COMPTREE), "InsTree");
00382             for ( ip = 0; ip < C->MaxTreeSize; ip++ ) { s[ip] = C->boomlijst[ip]; }
00383             for ( ip = C->MaxTreeSize; ip <= is; ip++ ) { s[ip] = comptreezero; }
00384             if ( C->boomlijst ) M_free(C->boomlijst, "InsTree");
00385             C->boomlijst = s;
00386             C->MaxTreeSize = is;
00387         }
00388         boomlijst = C->boomlijst;
00389         q = boomlijst + C->rootnum;
00390     }
00391
00392     if ( q->right == -1 ) { /* First element */
00393         C->numtree++;
00394         s = boomlijst+C->numtree;
00395         q->right = C->numtree;
00396         s->parent = C->rootnum;
00397         s->left = s->right = -1;
00398         s->blnce = 0;
00399         s->value = h;
00400         s->usage = 1;
00401         return(h);
00402     }
00403     ip = q->right;
00404     while ( ip >= 0 ) {
00405         p = boomlijst + ip;
00406         v1 = C->rhs[p->value]; v2 = v3 = C->rhs[h];
00407         while ( *v3 ) v3 += *v3; /* find the 0 that indicates end-of-expr */
00408         while ( *v1 == *v2 && v2 < v3 ) { v1++; v2++; }
00409         if ( *v1 > *v2 ) {
00410             iq = p->right;
00411             if ( iq >= 0 ) { ip = iq; }
00412             else {
00413                 C->numtree++;
00414                 is = C->numtree;
00415                 p->right = is;
00416                 s = boomlijst + is;
00417                 s->parent = ip; s->left = s->right = -1;
00418                 s->blnce = 0; s->value = h; s->usage = 1;
00419                 p->blnce++;
00420                 if ( p->blnce == 0 ) return(h);
00421                 goto balance;
00422             }
00423         }
00424         else if ( *v1 < *v2 ) {
00425             iq = p->left;
00426             if ( iq >= 0 ) { ip = iq; }
00427             else {
00428                 C->numtree++;
00429                 is = C->numtree;
00430                 s = boomlijst+is;
00431                 p->left = is;
00432                 s->parent = ip; s->left = s->right = -1;
00433                 s->blnce = 0; s->value = h; s->usage = 1;
00434                 p->blnce--;
00435                 if ( p->blnce == 0 ) return(h);
00436                 goto balance;
00437             }
00438         }
00439         else {
00440             p->usage++;
00441             return(p->value);
00442         }
00443     }
00444     MesPrint("We vallen uit de boom!");

```

```

00445     Terminate(-1);
00446     return(h);
00447 balance;;
00448     for (;;) {
00449         p = boomlijst + ip;
00450         iq = p->parent;
00451         if ( iq == C->rootnum ) break;
00452         q = boomlijst + iq;
00453         if ( ip == q->left ) q->blnce--;
00454         else q->blnce++;
00455         if ( q->blnce == 0 ) break;
00456         if ( q->blnce == -2 ) {
00457             if ( p->blnce == -1 ) { /* single rotation */
00458                 q->left = p->right;
00459                 p->right = iq;
00460                 p->parent = q->parent;
00461                 q->parent = ip;
00462                 if ( boomlijst[p->parent].left == iq ) boomlijst[p->parent].left = ip;
00463                 else boomlijst[p->parent].right = ip;
00464                 if ( q->left >= 0 ) boomlijst[q->left].parent = iq;
00465                 q->blnce = p->blnce = 0;
00466             }
00467             else { /* double rotation */
00468                 s = boomlijst + is;
00469                 q->left = s->right;
00470                 p->right = s->left;
00471                 s->right = iq;
00472                 s->left = ip;
00473                 if ( p->right >= 0 ) boomlijst[p->right].parent = ip;
00474                 if ( q->left >= 0 ) boomlijst[q->left].parent = iq;
00475                 s->parent = q->parent;
00476                 q->parent = is;
00477                 p->parent = is;
00478                 if ( boomlijst[s->parent].left == iq )
00479                     boomlijst[s->parent].left = is;
00480                 else boomlijst[s->parent].right = is;
00481                 if ( s->blnce > 0 ) { q->blnce = s->blnce = 0; p->blnce = -1; }
00482                 else if ( s->blnce < 0 ) { p->blnce = s->blnce = 0; q->blnce = 1; }
00483                 else { p->blnce = s->blnce = q->blnce = 0; }
00484             }
00485             break;
00486         }
00487         else if ( q->blnce == 2 ) {
00488             if ( p->blnce == 1 ) { /* single rotation */
00489                 q->right = p->left;
00490                 p->left = iq;
00491                 p->parent = q->parent;
00492                 q->parent = ip;
00493                 if ( boomlijst[p->parent].left == iq ) boomlijst[p->parent].left = ip;
00494                 else boomlijst[p->parent].right = ip;
00495                 if ( q->right >= 0 ) boomlijst[q->right].parent = iq;
00496                 q->blnce = p->blnce = 0;
00497             }
00498             else { /* double rotation */
00499                 s = boomlijst + is;
00500                 q->right = s->left;
00501                 p->left = s->right;
00502                 s->left = iq;
00503                 s->right = ip;
00504                 if ( p->left >= 0 ) boomlijst[p->left].parent = ip;
00505                 if ( q->right >= 0 ) boomlijst[q->right].parent = iq;
00506                 s->parent = q->parent;
00507                 q->parent = is;
00508                 p->parent = is;
00509                 if ( boomlijst[s->parent].left == iq ) boomlijst[s->parent].left = is;
00510                 else boomlijst[s->parent].right = is;
00511                 if ( s->blnce < 0 ) { q->blnce = s->blnce = 0; p->blnce = 1; }
00512                 else if ( s->blnce > 0 ) { p->blnce = s->blnce = 0; q->blnce = -1; }
00513                 else { p->blnce = s->blnce = q->blnce = 0; }
00514             }
00515             break;
00516         }
00517         is = ip; ip = iq;
00518     }
00519     return(h);
00520 }
00521
00522 /*
00523 #] InsTree :
00524 #[ FindTree :
00525
00526 Routines for balanced tree searching.
00527 Is like InsTree but without the insertions.
00528 Returns -1 if the element is not in the tree.
00529 The advantage of this routine over InsTree is that this routine
00530 can be run in parallel.
00531 */

```

```

00532
00533 int FindTree(int bufnum, WORD *subexpr)
00534 {
00535     CBUF *C = cbuf + bufnum;
00536     COMPTREE *boomlijst = C->boomlijst, *q = boomlijst + C->rootnum, *p;
00537     WORD *v1, *v2, *v3;
00538     int ip, iq;
00539
00540     ip = q->right;
00541     while ( ip >= 0 ) {
00542         p = boomlijst + ip;
00543         v1 = C->rhs[p->value]; v2 = v3 = subexpr;
00544         while ( *v3 ) v3 += *v3; /* find the 0 that indicates end-of-expr */
00545         while ( *v1 == *v2 && v2 < v3 ) { v1++; v2++; }
00546         if ( *v1 > *v2 ) {
00547             iq = p->right;
00548             if ( iq >= 0 ) { ip = iq; }
00549             else { return(-1); }
00550         }
00551         else if ( *v1 < *v2 ) {
00552             iq = p->left;
00553             if ( iq >= 0 ) { ip = iq; }
00554             else { return(-1); }
00555         }
00556         else {
00557             p->usage++;
00558             return(p->value);
00559         }
00560     }
00561     return(-1);
00562 }
00563
00564 /*
00565 #] FindTree :
00566 #[ RedoTree :
00567 */
00568
00569 void RedoTree(CBUF *C, int size)
00570 {
00571     COMPTREE *newboomlijst;
00572     int i;
00573     newboomlijst = (COMPTREE *)Malloc1((size+1)*sizeof(COMPTREE), "newboomlijst");
00574     if ( C->boomlijst ) {
00575         if ( C->MaxTreeSize > size ) C->MaxTreeSize = size;
00576         for ( i = 0; i < C->MaxTreeSize; i++ ) newboomlijst[i] = C->boomlijst[i];
00577         M_free(C->boomlijst, "boomlijst");
00578     }
00579     C->boomlijst = newboomlijst;
00580     C->MaxTreeSize = size;
00581 }
00582
00583 /*
00584 #] RedoTree :
00585 #[ ClearTree :
00586 */
00587
00588 void ClearTree(int i)
00589 {
00590     CBUF *C = cbuf + i;
00591     COMPTREE *root = C->boomlijst;
00592     if ( root ) {
00593         C->numtree = 0;
00594         C->rootnum = 0;
00595         root->left = -1;
00596         root->right = -1;
00597         root->parent = -1;
00598         root->blnce = 0;
00599         root->value = -1;
00600         root->usage = 0;
00601     }
00602 }
00603
00604 /*
00605 #] ClearTree :
00606 #[ IniFbuffer :
00607 */
00614 int IniFbuffer(WORD bufnum)
00615 {
00616     CBUF *C = cbuf + bufnum;
00617     COMPTREE *root;
00618     int i;
00619     LONG fullsize;
00620     C->maxrhs = AM.fbuffersize;
00621     C->MaxTreeSize = AM.fbuffersize;
00622
00623     /*
00624     * Note that bufnum is a return value of inicbufs(). So C has been already

```

```

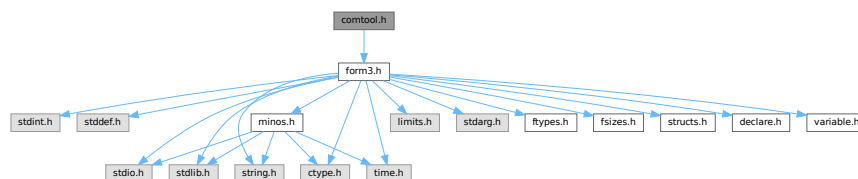
00625     * initialized. (TU 20 Dec 2011)
00626     */
00627     if ( C->boomlijst ) M_free(C->boomlijst, "IniFbuffer-tree");
00628     if ( C->rhs ) M_free(C->rhs, "IniFbuffer-rhs");
00629
00630     C->boomlijst = (COMPTREE *)Malloc1((C->MaxTreeSize+1)*sizeof(COMPTREE),"IniFbuffer-tree");
00631     root = C->boomlijst;
00632     C->numtree = 0;
00633     C->rootnum = 0;
00634     root->left = -1;
00635     root->right = -1;
00636     root->parent = -1;
00637     root->blnce = 0;
00638     root->value = -1;
00639     root->usage = 0;
00640     for ( i = 1; i < C->MaxTreeSize; i++ ) { C->boomlijst[i] = comptreezero; }
00641
00642     fullsize = (C->maxrhs+1) * (sizeof(WORD *) + 2*sizeof(LONG) + 2*sizeof(WORD));
00643     C->rhs = (WORD **)Malloc1(fullsize,"IniFbuffer-rhs");
00644     C->CanCommu = (LONG *) (C->rhs+C->maxrhs);
00645     C->NumTerms = (LONG *) (C->rhs+2*C->maxrhs);
00646     C->numdum = (WORD *) (C->NumTerms+C->maxrhs);
00647     C->dimension = (WORD *) (C->numdum+C->maxrhs);
00648
00649     return(0);
00650 }
00651
00652 /*
00653  #] IniFbuffer :
00654  #[ numcommute :
00655
00656  Returns the number of non-commuting terms in the expression
00657  */
00658
00659 LONG numcommute(WORD *terms, LONG *numterms)
00660 {
00661     LONG num = 0;
00662     WORD *t, *m;
00663     *numterms = 0;
00664     while ( *terms ) {
00665         *numterms += 1;
00666         t = terms + 1;
00667         GETSTOP(terms,m);
00668         while ( t < m ) {
00669             if ( *t >= FUNCTION ) {
00670                 if ( functions[*t-FUNCTION].commute ) { num++; break; }
00671             }
00672             t += t[1];
00673         }
00674         terms = terms + *terms;
00675     }
00676     return(num);
00677 }
00678
00679 /*
00680  #] numcommute :
00681  */

```

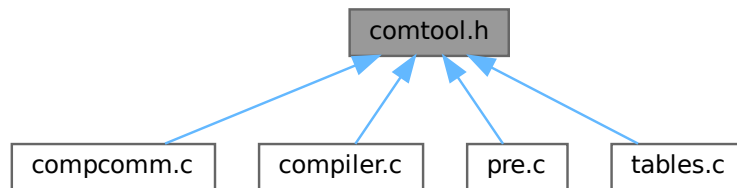
4.17 comtool.h File Reference

```
#include "form3.h"
```

Include dependency graph for comtool.h:



This graph shows which files directly or indirectly include this file:



4.17.1 Detailed Description

Utility routines for the compiler.

Definition in file [comtool.h](#).

4.18 comtool.h

[Go to the documentation of this file.](#)

```

00001
00005 /* #[] License : */
00006 /*
00007 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008 *   When using this file you are requested to refer to the publication
00009 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010 *   This is considered a matter of courtesy as the development was paid
00011 *   for by FOM the Dutch physics granting agency and we would like to
00012 *   be able to track its scientific use to convince FOM of its value
00013 *   for the community.
00014 *
00015 *   This file is part of FORM.
00016 *
00017 *   FORM is free software: you can redistribute it and/or modify it under the
00018 *   terms of the GNU General Public License as published by the Free Software
00019 *   Foundation, either version 3 of the License, or (at your option) any later
00020 *   version.
00021 *
00022 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00025 *   details.
00026 *
00027 *   You should have received a copy of the GNU General Public License along
00028 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00029 */
00030 /* #[] License : */
00031 #ifndef FORM_COMTOOL_H_
00032 #define FORM_COMTOOL_H_
00033 /*
00034 *   #[] Includes :
00035 */
00036
00037 #include "form3.h"
00038
00039 /*
00040 *   #[] Includes :
00041 *   #[] Inline functions :
00042 */
00043
00055 static inline void SkipSpaces(UBYTE **s)
00056 {
00057     const char *p = (const char *)*s;

```

```

00058     while ( *p == ' ' || *p == ',' || *p == '\t' ) p++;
00059     *s = (UBYTE *)p;
00060 }
00061
00073 static inline int ConsumeOption(UBYTE **s, const char *opt)
00074 {
00075     const char *p = (const char *)*s;
00076     while ( *p && *opt && tolower(*p) == tolower(*opt) ) {
00077         p++;
00078         opt++;
00079     }
00080     /* Check if `opt` ended. */
00081     if ( !*opt ) {
00082         /* Check if `*p` is a word boundary. */
00083         UINT c = FG.cTable[(unsigned char)*p];
00084         if ( c != 0 && c != 1 && *p != '_' && *p != '$' ) {
00085             /* Consume the option. Skip the trailing whitespace. */
00086             *s = (UBYTE *)p;
00087             SkipSpaces(s);
00088             return(1);
00089         }
00090     }
00091     return(0);
00092 }
00093
00094 /*
00095 #] Inline functions :
00096 */
00097 #endif /* FORM_COMTOOL_H_ */

```

4.19 declare.h File Reference

This graph shows which files directly or indirectly include this file:



Macros

- #define **MaX**(x, y) ((x) > (y) ? (x): (y))
- #define **MiN**(x, y) ((x) < (y) ? (x): (y))
- #define **ABS**(x) ((x) < 0 ? -(x): (x))
- #define **SGN**(x) ((x) > 0 ? 1 : (x) < 0 ? -1 : 0)
- #define **REDLENG**(x) (((x)<0)?((x)+1):((x)-1))/2)
- #define **INCLENG**(x) (((x)<0)?(((x)*2)-1):(((x)*2)+1))
- #define **GETCOEF**(x, y) x += *x;y = x[-1];x -= ABS(y);y=REDLENG(y)
- #define **GETSTOP**(x, y) y=x+(*x)-1;y -= ABS(*y)-1
- #define **StuffAdd**(x, y) (((x)<0?-1:1)*y)+((y)<0?-1:1)*(x))
- #define **EXCHN**(t1, t2, n) { WORD a,i; for(i=0;i<n;i++){a=t1[i];t1[i]=t2[i];t2[i]=a;} }
- #define **EXCH**(x, y) { WORD a = (x); (x) = (y); (y) = a; }
- #define **TOKENTOLINE**(x, y)
- #define **UngetFromStream**(stream, c) ((stream)->nextchar[(stream)->isnextchar++]=c)
- #define **AddLineFeed**(s, n) { (s)[(n)++] = LINEFEED; }
- #define **TryRecover**(x) Terminate(-1)
- #define **UngetChar**(c) { pushbackchar = c; }
- #define **ParseNumber**(x, s) {(x)=0;while(*s)>='0'&&*s<='9')(x)=10*(x)+*(s)++ -'0';}
- #define **ParseSign**(sgn, s)
- #define **ParseSignedNumber**(x, s)
- #define **NCOPY**(s, t, n) { while ((n)-- > 0) { *s++ = *t++; } }
- #define **NCOPYI**(s, t, n) { while ((n)-- > 0) { *s++ = *t++; } }
- #define **NCOPYB**(s, t, n) { while ((n)-- > 0) { *s++ = *t++; } }

- #define [NCOPYI32](#)(s, t, n) { while ((n)-- > 0) { *s++ = *t++; } }
- #define [WCOPY](#)(s, t, n) { int nn=n; WORD *ss=(WORD *)s, *tt=(WORD *)t; while ((nn)-- > 0) { *ss++ = *tt++; } }
- #define [NeedNumber](#)(x, s, err)
- #define [SKIPBLANKS](#)(s) { while (*(s) == ' ' || *(s) == '\t') (s)++; }
- #define [FLUSHCONSOLE](#) if (AP.InOutBuf > 0) CharOut(LINEFEED)
- #define [SKIPBRA1](#)(s)
- #define [SKIPBRA2](#)(s)
- #define [SKIPBRA3](#)(s)
- #define [SKIPBRA4](#)(s)
- #define [SKIPBRA5](#)(s)
- #define [CYCLE1](#)(t, a, i) { t iX=*a; WORD jX; for(jX=1;jX<i;jX++)a[jX-1]=a[jX]; a[i-1]=iX; }
- #define [AddToCB](#)(c, wx)
- #define [EXCHINOUT](#)
- #define [BACKINOUT](#)
- #define [CopyArg](#)(to, from)
- #define [FILLARG](#)(w)
- #define [COPYARG](#)(w, t)
- #define [ZEROARG](#)(w)
- #define [FILLFUN](#)(w)
- #define [COPYFUN](#)(w, t)
- #define [COPYFUN3](#)(w, t)
- #define [FILLFUN3](#)(w)
- #define [FILLSUB](#)(w)
- #define [COPYSUB](#)(w, ww)
- #define [FILLEXP](#)(w)
- #define [NEXTARG](#)(x) if(*x>0) x += *x; else if(*x <= -FUNCTION)x++; else x += 2;
- #define [COPY1ARG](#)(s1, t1)
- #define [ZeroFillRange](#)(w, begin, end)
- #define [TABLESIZE](#)(a, b) (((WORD)sizeof(a))/((WORD)sizeof(b)))
- #define [WORDDIF](#)(x, y) (WORD)(x-y)
- #define [wsizeof](#)(a) ((WORD)sizeof(a))
- #define [VARNAME](#)(type, num) (AC.varnames->namebuffer+type[num].name)
- #define [DOLLARNAME](#)(type, num) (AC.dollarnames->namebuffer+type[num].name)
- #define [EXPRNAME](#)(num) (AC.exprnames->namebuffer+Expressions[num].name)
- #define [PREV](#)(x) prevorder?prevorder:x
- #define [Terminate](#)(x) do { TerminateImpl(x, __FILE__, __LINE__, __FUNCTION__); } while(0)
- #define [SETERROR](#)(x) { Terminate(-1); return(-1); }
- #define [DUMMYUSE](#)(x) (void)(x);
- #define [ADDPOS](#)(pp, x) (pp).p1 = ((pp).p1+(LONG)(x))
- #define [SETBASELENGTH](#)(ss, x) (ss).p1 = (LONG)(x)
- #define [SETBASEPOSITION](#)(pp, x) (pp).p1 = (LONG)(x)
- #define [ISEQUALPOSINC](#)(pp1, pp2, x) ((pp1).p1 == ((pp2).p1+(LONG)(x)))
- #define [ISGEPOSINC](#)(pp1, pp2, x) ((pp1).p1 >= ((pp2).p1+(LONG)(x)))
- #define [DIVPOS](#)(pp, n) ((pp).p1/(LONG)(n))
- #define [MULPOS](#)(pp, n) (pp).p1 *= (LONG)(n)
- #define [DIFPOS](#)(ss, pp1, pp2) (ss).p1 = ((pp1).p1-(pp2).p1)
- #define [DIFBASE](#)(pp1, pp2) ((pp1).p1-(pp2).p1)
- #define [ADD2POS](#)(pp1, pp2) (pp1).p1 += (pp2).p1
- #define [PUTZERO](#)(pp) (pp).p1 = 0
- #define [BASEPOSITION](#)(pp) ((pp).p1)
- #define [SETSTARTPOS](#)(pp) (pp).p1 = -2
- #define [NOTSTARTPOS](#)(pp) ((pp).p1 > -2)
- #define [ISMINPOS](#)(pp) ((pp).p1 == -1)
- #define [ISEQUALPOS](#)(pp1, pp2) ((pp1).p1 == (pp2).p1)

- #define [ISNOTEQUALPOS](#)(pp1, pp2) ((pp1).p1 != (pp2).p1)
- #define [ISLESSPOS](#)(pp1, pp2) ((pp1).p1 < (pp2).p1)
- #define [ISGEPOS](#)(pp1, pp2) ((pp1).p1 >= (pp2).p1)
- #define [ISNOTZEROPOS](#)(pp) ((pp).p1 != 0)
- #define [ISZEROPOS](#)(pp) ((pp).p1 == 0)
- #define [ISPOSP](#)(pp) ((pp).p1 > 0)
- #define [ISNEGPOS](#)(pp) ((pp).p1 < 0)
- #define [TOLONG](#)(x) ((LONG)(x))
- #define [Add2Com](#)(x) { WORD cod[2]; cod[0] = x; cod[1] = 2; [AddNtoL](#)(2,cod); }
- #define [Add3Com](#)(x1, x2) { WORD cod[3]; cod[0] = x1; cod[1] = 3; cod[2] = x2; [AddNtoL](#)(3,cod); }
- #define [Add4Com](#)(x1, x2, x3)
- #define [Add5Com](#)(x1, x2, x3, x4)
- #define [WantAddPointers](#)(x)
- #define [WantAddLongs](#)(x)
- #define [WantAddPositions](#)(x)
- #define [FORM_INLINE](#) inline
- #define [MEMORYMACROS](#)
- #define [TermMalloc](#)(x) ((AT.TermMemTop <= 0) ? [TermMallocAddMemory](#)(BHEAD0), AT.TermMemHeap[--AT.TermMemTop]: AT.TermMemHeap[--AT.TermMemTop])
- #define [NumberMalloc](#)(x) ((AT.NumberMemTop <= 0) ? [NumberMallocAddMemory](#)(BHEAD0), AT.NumberMemHeap[--AT.NumberMemTop]: AT.NumberMemHeap[--AT.NumberMemTop])
- #define [CacheNumberMalloc](#)(x) ((AT.CacheNumberMemTop <= 0) ? [CacheNumberMallocAddMemory](#)(BHEAD0), AT.CacheNumberMemHeap[--AT.CacheNumberMemTop]: AT.CacheNumberMemHeap[--AT.CacheNumberMemTop])
- #define [TermFree](#)(TermMem, x) AT.TermMemHeap[AT.TermMemTop++] = (WORD *) (TermMem)
- #define [NumberFree](#)(NumberMem, x) AT.NumberMemHeap[AT.NumberMemTop++] = (UWORD *) (NumberMem)
- #define [CacheNumberFree](#)(NumberMem, x) AT.CacheNumberMemHeap[AT.CacheNumberMemTop++] = (UWORD *) (NumberMem)
- #define [NestingChecksum](#)() (AC.IfLevel + AC.RepLevel + AC.arglevel + AC.insidelevel + AC.termlevel + AC.inexprlevel + AC.dolooplevel + AC.SwitchLevel)
- #define [MesNesting](#)() MesPrint("&Illegal nesting of if, repeat, argument, inside, term, inexpression and do")
- #define [MarkPolyRatFunDirty](#)(T)
- #define [PUSHPREASSIGNLEVEL](#)
- #define [POPPREASSIGNLEVEL](#)
- #define [EXTERNLOCK](#)(x)
- #define [INILOCK](#)(x)
- #define [LOCK](#)(x)
- #define [UNLOCK](#)(x)
- #define [EXTERNRWLOCK](#)(x)
- #define [INIRWLOCK](#)(x)
- #define [RWLOCKR](#)(x)
- #define [RWLOCKW](#)(x)
- #define [UNRWLOCK](#)(x)
- #define [MLOCK](#)(x)
- #define [MUNLOCK](#)(x)
- #define [GETIDENTITY](#)
- #define [GETBIDENTITY](#)
- #define [M_alloc](#)(x) malloc((size_t)(x))
- #define [CompareTerms](#) ((COMPARE)AR.CompareRoutine)
- #define [FiniShuffle](#) AN.SHvar.finishuf
- #define [DoShuffle](#) ((DO_UFFLE)AN.SHvar.do_uffle)

Typedefs

- typedef int(* [WRITEBUFTOEXTCHANNEL](#)) (char *, size_t)
- typedef int(* [GETCFROMEXTCHANNEL](#)) (void)
- typedef int(* [SETTERMINATORFOREXTERNALCHANNEL](#)) (char *)
- typedef int(* [SETKILLMODEFOREXTERNALCHANNEL](#)) (int, int)
- typedef LONG(* [WRITEFILE](#)) (int, UBYTE *, LONG)
- typedef WORD(* [GETTERM](#)) (PHEAD WORD *)

Functions

- void [TELLFILE](#) (int, [POSITION](#) *)
- void [StartVariables](#) (void)
- void [setSignalHandlers](#) (void)
- UBYTE * [CodeToLine](#) (WORD, UBYTE *)
- UBYTE * [AddArrayIndex](#) (WORD, UBYTE *)
- [INDEXENTRY](#) * [FindInIndex](#) (WORD, [FILEDATA](#) *, WORD, WORD)
- [INDEXENTRY](#) * [NextFileIndex](#) ([POSITION](#) *)
- WORD * [PasteTerm](#) (PHEAD WORD, WORD *, WORD *, WORD, WORD)
- UBYTE * [StrCopy](#) (UBYTE *, UBYTE *)
- UBYTE * [WrtPower](#) (UBYTE *, WORD)
- int [AccumGCD](#) (PHEAD UWORD *, WORD *, UWORD *, WORD)
- void [AddArgs](#) (PHEAD WORD *, WORD *, WORD *)
- int [AddCoef](#) (PHEAD WORD **, WORD **)
- int [AddLong](#) (UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- int [AddPLon](#) (UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- int [AddPoly](#) (PHEAD WORD **, WORD **)
- int [AddRat](#) (PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- void [AddToLine](#) (UBYTE *)
- int [AddWild](#) (PHEAD WORD, WORD, WORD)
- int [BigLong](#) (UWORD *, WORD, UWORD *, WORD)
- int [BinomGen](#) (PHEAD WORD *, WORD, WORD **, WORD, WORD, WORD, WORD, WORD, UWORD *, WORD)
- int [CheckWild](#) (PHEAD WORD, WORD, WORD, WORD *)
- int [Chisholm](#) (PHEAD WORD *, WORD)
- int [CleanExpr](#) (WORD)
- void [CleanUp](#) (WORD)
- void [ClearWild](#) (PHEAD0)
- int [CompareFunctions](#) (WORD *, WORD *)
- int [Commute](#) (WORD *, WORD *)
- WORD [DetCommu](#) (WORD *)
- WORD [DoesCommu](#) (WORD *)
- int [CompArg](#) (WORD *, WORD *)
- WORD [CompCoef](#) (WORD *, WORD *)
- int [CompGroup](#) (PHEAD WORD, WORD **, WORD *, WORD *, WORD)
- WORD [Compare1](#) (PHEAD WORD *, WORD *, WORD)
- WORD [CountDo](#) (WORD *, WORD *)
- WORD [CountFun](#) (WORD *, WORD *)
- WORD [DimensionSubterm](#) (WORD *)
- WORD [DimensionTerm](#) (WORD *)
- WORD [DimensionExpression](#) (PHEAD WORD *)
- int [Deferred](#) (PHEAD WORD *, WORD)
- int [DeleteStore](#) (WORD)
- WORD [DetCurDum](#) (PHEAD WORD *)

- void [DetVars](#) (WORD *, WORD)
- int [Distribute](#) (DISTRIBUTE *, WORD)
- int [DivLong](#) (UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *, UWORD *, WORD *)
- int [DivRat](#) (PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- int [Divvy](#) (PHEAD UWORD *, WORD *, UWORD *, WORD)
- int [DoDelta](#) (WORD *)
- int [DoDelta3](#) (PHEAD WORD *, WORD)
- int [TestPartitions](#) (WORD *, [PARTI](#) *)
- int [DoPartitions](#) (PHEAD WORD *, WORD)
- int [CoCanonicalize](#) (UBYTE *)
- int [DoCanonicalize](#) (PHEAD WORD *, WORD *)
- int [GenDiagrams](#) (PHEAD WORD *, WORD)
- int [DoTopologyCanonicalize](#) (PHEAD WORD *, WORD, WORD, WORD *)
- int [DoShattering](#) (PHEAD WORD *, WORD *, WORD *, WORD)
- int [DoTableExpansion](#) (WORD *, WORD)
- int [DoDistrib](#) (PHEAD WORD *, WORD)
- int [DoShuffle](#) (WORD *, WORD, WORD, WORD)
- int [DoPermutations](#) (PHEAD WORD *, WORD)
- int [Shuffle](#) (WORD *, WORD *, WORD *)
- int [FinishShuffle](#) (WORD *)
- int [DoStuffle](#) (WORD *, WORD, WORD, WORD)
- int [Stuffle](#) (WORD *, WORD *, WORD *)
- int [FinishStuffle](#) (WORD *)
- WORD * [StuffRootAdd](#) (WORD *, WORD *, WORD *)
- int [TestUse](#) (WORD *, WORD)
- DBASE * [FindTB](#) (UBYTE *)
- int [CheckTableDeclarations](#) (DBASE *)
- void [Apply](#) (WORD *, WORD)
- int [ApplyExec](#) (WORD *, int, WORD)
- void [ApplyReset](#) (WORD)
- void [TableReset](#) (void)
- void [ReWorkT](#) (WORD *, WORD *, WORD)
- WORD [GetIfDollarNum](#) (WORD *, WORD *)
- int [FindVar](#) (WORD *, WORD *)
- int [DolfStatement](#) (PHEAD WORD *, WORD *)
- int [DoOnePow](#) (PHEAD WORD *, WORD, WORD, WORD *, WORD *, WORD, WORD *)
- void [DoRevert](#) (WORD *, WORD *)
- int [DoSumF1](#) (PHEAD WORD *, WORD *, WORD, WORD)
- int [DoSumF2](#) (PHEAD WORD *, WORD *, WORD, WORD)
- int [DoTheta](#) (PHEAD WORD *)
- LONG [EndSort](#) (PHEAD WORD *, int)
- int [EntVar](#) (WORD, UBYTE *, WORD, WORD, WORD, WORD)
- int [EpfCon](#) (PHEAD WORD *, WORD *, WORD, WORD)
- int [EpfFind](#) (PHEAD WORD *, WORD *)
- WORD [EpfGen](#) (WORD, WORD *, WORD *, WORD *, WORD)
- int [EqualArg](#) (WORD *, WORD, WORD)
- int [Factorial](#) (PHEAD WORD, UWORD *, WORD *)
- int [Bernoulli](#) (WORD, UWORD *, WORD *)
- int [FactorIn](#) (PHEAD WORD *, WORD)
- int [FactorInExpr](#) (PHEAD WORD *, WORD)
- int [FindAll](#) (PHEAD WORD *, WORD *, WORD, WORD *)
- WORD [FindMulti](#) (PHEAD WORD *, WORD *)
- int [FindOnce](#) (PHEAD WORD *, WORD *)
- int [FindOnly](#) (PHEAD WORD *, WORD *)
- int [FindRest](#) (PHEAD WORD *, WORD *)

- void **FindSpecial** (WORD *)
- WORD **FindrNumber** (WORD, VARRENUM *)
- void **FiniLine** (void)
- int **FiniTerm** (PHEAD WORD *, WORD *, WORD *, WORD, WORD)
- int **FlushOut** (POSITION *, FILEHANDLE *, int)
- void **FunLevel** (PHEAD WORD *)
- void **AdjustRenumScratch** (PHEAD0)
- void **GarbHand** (void)
- int **GcdLong** (PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- int **LcmLong** (PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- void **GCD** (UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- ULONG **GCD2** (ULONG, ULONG)
- int **Generator** (PHEAD WORD *, WORD)
- int **GetBinom** (UWORD *, WORD *, WORD, WORD)
- WORD **GetFromStore** (WORD *, POSITION *, RENUMBER, WORD *, WORD)
- int **GetLong** (UBYTE *, UWORD *, WORD *)
- WORD **GetMoreTerms** (WORD *)
- int **GetMoreFromMem** (WORD *, WORD **)
- WORD **GetOneTerm** (PHEAD WORD *, FILEHANDLE *, POSITION *, int)
- RENUMBER **GetTable** (WORD, POSITION *, WORD)
- WORD **GetTerm** (PHEAD WORD *)
- int **Glue** (PHEAD WORD *, WORD *, WORD *, WORD)
- int **InFunction** (PHEAD WORD *, WORD *)
- void **IniLine** (WORD)
- void **IniVars** (void)
- int **InsertTerm** (PHEAD WORD *, WORD, WORD, WORD *, WORD *, WORD)
- void **LongToLine** (UWORD *, WORD)
- int **MakeDirty** (WORD *, WORD *, WORD)
- void **MarkDirty** (WORD *, WORD)
- void **PolyFunDirty** (PHEAD WORD *)
- void **PolyFunClean** (PHEAD WORD *)
- int **MakeModTable** (void)
- int **MatchE** (PHEAD WORD *, WORD *, WORD *, WORD)
- int **MatchCy** (PHEAD WORD *, WORD *, WORD *, WORD)
- int **FunMatchCy** (PHEAD WORD *, WORD *, WORD *, WORD)
- int **FunMatchSy** (PHEAD WORD *, WORD *, WORD *, WORD)
- int **MatchArgument** (PHEAD WORD *, WORD *)
- int **MatchFunction** (PHEAD WORD *, WORD *, WORD *)
- int **MergePatches** (WORD)
- int **MesCerr** (char *, UBYTE *)
- int **MesComp** (char *, UBYTE *, UBYTE *)
- int **Modulus** (WORD *)
- void **MoveDummies** (PHEAD WORD *, WORD)
- int **MulLong** (UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- int **MulRat** (PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- int **Mully** (PHEAD UWORD *, WORD *, UWORD *, WORD)
- int **MultDo** (PHEAD WORD *, WORD *)
- int **NewSort** (PHEAD0)
- int **ExtraSymbol** (WORD, WORD, WORD, WORD *, WORD *)
- int **Normalize** (PHEAD WORD *)
- int **BracketNormalize** (PHEAD WORD *)
- void **DropCoefficient** (PHEAD WORD *)
- void **DropSymbols** (PHEAD WORD *)
- int **PutInside** (PHEAD WORD *, WORD *)
- void **OpenTemp** (void)

- void [Pack](#) (UWORD *, WORD *, UWORD *, WORD)
- LONG [PasteFile](#) (PHEAD WORD, WORD *, [POSITION](#) *, WORD **, [RENUMBER](#), WORD *, WORD)
- int [Permute](#) ([PERM](#) *, WORD)
- int [PermuteP](#) ([PERMP](#) *, WORD)
- int [PolyFunMul](#) (PHEAD WORD *)
- int [PopVariables](#) (void)
- int [PrepPoly](#) (PHEAD WORD *, WORD)
- int [Processor](#) (void)
- int [Product](#) (UWORD *, WORD *, WORD)
- void [PrtLong](#) (UWORD *, WORD, UBYTE *)
- void [PrtTerms](#) (void)
- void [PrintDeprecation](#) (const char *, const char *)
- void [PrintRunningTime](#) (void)
- LONG [GetRunningTime](#) (void)
- int [PutBracket](#) (PHEAD WORD *)
- LONG [PutIn](#) ([FILEHANDLE](#) *, [POSITION](#) *, WORD *, WORD **, int)
- int [PutInStore](#) ([INDEXENTRY](#) *, WORD)
- WORD [PutOut](#) (PHEAD WORD *, [POSITION](#) *, [FILEHANDLE](#) *, WORD)
- UWORD [Quotient](#) (UWORD *, WORD *, WORD)
- int [RaisPow](#) (PHEAD UWORD *, WORD *, UWORD)
- void [RaisPowCached](#) (PHEAD WORD, WORD, UWORD **, WORD *)
- WORD [RaisPowMod](#) (WORD, WORD, WORD)
- int [NormalModulus](#) (UWORD *, WORD *)
- int [MakeInverses](#) (void)
- int [GetModInverses](#) (WORD, WORD, WORD *, WORD *)
- int [GetLongModInverses](#) (PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *, UWORD *, WORD *)
- void [RatToLine](#) (UWORD *, WORD)
- int [RatioFind](#) (PHEAD WORD *, WORD *)
- int [RatioGen](#) (PHEAD WORD *, WORD *, WORD, WORD)
- WORD [ReNumber](#) (PHEAD WORD *)
- WORD [ReadSnum](#) (UBYTE **)
- WORD [Remain10](#) (UWORD *, WORD *)
- WORD [Remain4](#) (UWORD *, WORD *)
- int [ResetScratch](#) (void)
- int [ResolveSet](#) (PHEAD WORD *, WORD *, WORD *)
- int [RevertScratch](#) (void)
- int [ScanFunctions](#) (PHEAD WORD *, WORD *, WORD)
- void [SeekScratch](#) ([FILEHANDLE](#) *, [POSITION](#) *)
- void [SetEndScratch](#) ([FILEHANDLE](#) *, [POSITION](#) *)
- void [SetEndHScratch](#) ([FILEHANDLE](#) *, [POSITION](#) *)
- int [SetFileIndex](#) (void)
- int [Sflush](#) ([FILEHANDLE](#) *)
- int [Simplify](#) (PHEAD UWORD *, WORD *, UWORD *, WORD *)
- int [SortWild](#) (WORD *, WORD)
- FILE * [LocateBase](#) (char **, char **, char *)
- LONG [SplitMerge](#) (PHEAD WORD **, LONG)
- int [StoreTerm](#) (PHEAD WORD *)
- void [SubPLon](#) (UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)
- void [Substitute](#) (PHEAD WORD *, WORD *, WORD)
- int [SymFind](#) (PHEAD WORD *, WORD *)
- int [SymGen](#) (PHEAD WORD *, WORD *, WORD, WORD)
- WORD [Symmetrize](#) (PHEAD WORD *, WORD *, WORD, WORD, WORD)
- int [FullSymmetrize](#) (PHEAD WORD *, int)
- int [TakeModulus](#) (UWORD *, WORD *, UWORD *, WORD, WORD)

- int [TakeNormalModulus](#) (UWORD *, WORD *, UWORD *, WORD, WORD)
- void [TailToLine](#) (UWORD)
- int [TenVec](#) (PHEAD WORD *, WORD *, WORD, WORD)
- int [TenVecFind](#) (PHEAD WORD *, WORD *)
- int [TermRenummer](#) (WORD *, [RENUMBER](#), WORD)
- void [TestDrop](#) (void)
- void [PutInVflags](#) (WORD)
- int [TestMatch](#) (PHEAD WORD *, WORD *)
- WORD [TestSub](#) (PHEAD WORD *, WORD)
- LONG [TimeCPU](#) (WORD)
- LONG [TimeChildren](#) (WORD)
- LONG [TimeWallClock](#) (WORD)
- LONG [Timer](#) (int)
- int [GetTimerInfo](#) (LONG **, LONG **)
- void [WriteTimerInfo](#) (LONG *, LONG *)
- LONG [GetWorkerTimes](#) (void)
- int [ToStorage](#) ([EXPRESSIONS](#), [POSITION](#) *)
- void [TokenToLine](#) (UBYTE *)
- int [Trace4](#) (PHEAD WORD *, WORD *, WORD, WORD)
- int [Trace4Gen](#) (PHEAD [TRACES](#) *, WORD)
- int [Trace4no](#) (WORD, WORD *, [TRACES](#) *)
- int [TraceFind](#) (PHEAD WORD *, WORD *)
- int [TraceN](#) (PHEAD WORD *, WORD *, WORD, WORD)
- int [TraceNgen](#) (PHEAD [TRACES](#) *, WORD)
- WORD [TraceNno](#) (WORD, WORD *, [TRACES](#) *)
- int [Traces](#) (PHEAD WORD *, WORD *, WORD, WORD)
- WORD [Trick](#) (WORD *, [TRACES](#) *)
- int [TryDo](#) (PHEAD WORD *, WORD *, WORD)
- void [UnPack](#) (UWORD *, WORD, WORD *, WORD *)
- int [VarStore](#) (UBYTE *, WORD, WORD, WORD)
- WORD [WildFill](#) (PHEAD WORD *, WORD *, WORD *)
- int [WriteAll](#) (void)
- int [WriteOne](#) (UBYTE *, int, int, WORD)
- void [WriteArgument](#) (WORD *)
- int [WriteExpression](#) (WORD *, LONG)
- int [WriteInnerTerm](#) (WORD *, WORD)
- void [WriteLists](#) (void)
- void [WriteSetup](#) (void)
- void [WriteStats](#) ([POSITION](#) *, WORD, WORD)
- int [WriteSubTerm](#) (WORD *, WORD)
- int [WriteTerm](#) (WORD *, WORD *, WORD, WORD, WORD)
- int [execarg](#) (PHEAD WORD *, WORD)
- int [execterm](#) (PHEAD WORD *, WORD)
- void [SpecialCleanup](#) (PHEAD0)
- void [SetMods](#) (void)
- void [UnSetMods](#) (void)
- int [DoExecute](#) (WORD, WORD)
- void [SetScratch](#) ([FILEHANDLE](#) *, [POSITION](#) *)
- void [Warning](#) (char *)
- void [HighWarning](#) (char *)
- int [SpareTable](#) ([TABLES](#))
- UBYTE * [strDup1](#) (UBYTE *, char *)
- void * [Malloc1](#) (LONG, const char *)
- int [DoTail](#) (int, UBYTE **)
- int [OpenInput](#) (void)

- int [PutPreVar](#) (UBYTE *, UBYTE *, UBYTE *, int)
- void [Error0](#) (char *)
- void [Error1](#) (char *, UBYTE *)
- void [Error2](#) (char *, char *, UBYTE *)
- UBYTE [ReadFromStream](#) (STREAM *)
- UBYTE [GetFromStream](#) (STREAM *)
- UBYTE [LookInStream](#) (STREAM *)
- STREAM * [OpenStream](#) (UBYTE *, int, int, int)
- int [LocateFile](#) (UBYTE **, int)
- STREAM * [CloseStream](#) (STREAM *)
- void [PositionStream](#) (STREAM *, LONG)
- int [ReverseStatements](#) (STREAM *)
- int [ProcessOption](#) (UBYTE *, UBYTE *, int)
- int [DoSetups](#) (void)
- void [TerminateImpl](#) (int, const char *, int, const char *) NORETURN
- NAMETREE * [GetNode](#) (NAMETREE *, UBYTE *)
- int [AddName](#) (NAMETREE *, UBYTE *, WORD, WORD, int *)
- int [GetName](#) (NAMETREE *, UBYTE *, WORD *, int)
- UBYTE * [GetFunction](#) (UBYTE *, WORD *)
- UBYTE * [GetNumber](#) (UBYTE *, WORD *)
- int [GetLastExprName](#) (UBYTE *, WORD *)
- int [GetAutoName](#) (UBYTE *, WORD *)
- int [GetVar](#) (UBYTE *, WORD *, WORD *, int, int)
- int [MakeDubious](#) (NAMETREE *, UBYTE *, WORD *)
- int [GetOName](#) (NAMETREE *, UBYTE *, WORD *, int)
- void [DumpTree](#) (NAMETREE *)
- void [DumpNode](#) (NAMETREE *, WORD, WORD)
- void [LinkTree](#) (NAMETREE *, WORD, WORD)
- void [CopyTree](#) (NAMETREE *, NAMETREE *, WORD, WORD)
- int [CompactifyTree](#) (NAMETREE *, WORD)
- NAMETREE * [MakeNameTree](#) (void)
- void [FreeNameTree](#) (NAMETREE *)
- int [AddExpression](#) (UBYTE *, int, int)
- int [AddSymbol](#) (UBYTE *, int, int, int, int)
- int [AddDollar](#) (UBYTE *, WORD, WORD *, LONG)
- int [ReplaceDollar](#) (WORD, WORD, WORD *, LONG)
- int [DollarRaiseLow](#) (UBYTE *, LONG)
- int [AddVector](#) (UBYTE *, int, int)
- int [AddDubious](#) (UBYTE *)
- int [AddIndex](#) (UBYTE *, int, int)
- UBYTE * [DoDimension](#) (UBYTE *, int *, int *)
- int [AddFunction](#) (UBYTE *, int, int, int, int, int, int, int)
- int [CoCommutelnSet](#) (UBYTE *)
- int [CoFunction](#) (UBYTE *, int, int)
- int [TestName](#) (UBYTE *)
- int [AddSet](#) (UBYTE *, WORD)
- int [DoElements](#) (UBYTE *, SETS, UBYTE *)
- int [DoTempSet](#) (UBYTE *, UBYTE *)
- int [NameConflict](#) (int, UBYTE *)
- int [OpenFile](#) (char *)
- int [OpenAddFile](#) (char *)
- int [ReOpenFile](#) (char *)
- int [CreateFile](#) (char *)
- int [CreateLogFile](#) (char *)
- void [CloseFile](#) (int)

- int [CopyFile](#) (char *, char *)
- int [CreateHandle](#) (void)
- LONG [ReadFile](#) (int, UBYTE *, LONG)
- LONG [ReadPosFile](#) (PHEAD [FILEHANDLE](#) *, UBYTE *, LONG, [POSITION](#) *)
- LONG [WriteFileToFile](#) (int, UBYTE *, LONG)
- void [SeekFile](#) (int, [POSITION](#) *, int)
- LONG [TellFile](#) (int)
- void [FlushFile](#) (int)
- int [GetPosFile](#) (int, fpos_t *)
- int [SetPosFile](#) (int, fpos_t *)
- void [SynchFile](#) (int)
- void [TruncateFile](#) (int)
- int [GetChannel](#) (char *, int)
- int [GetAppendChannel](#) (char *)
- int [CloseChannel](#) (char *)
- void [inictable](#) (void)
- [KEYWORD](#) * [findcommand](#) (UBYTE *)
- int [inicbufs](#) (void)
- void [StartFiles](#) (void)
- UBYTE * [MakeDate](#) (void)
- void [PreProcessor](#) (void)
- void * [FromList](#) ([LIST](#) *)
- void * [From0List](#) ([LIST](#) *)
- void * [FromVarList](#) ([LIST](#) *)
- int [DoubleList](#) (void ***, int *, int, char *)
- int [DoubleLList](#) (void ***, LONG *, int, char *)
- void [DoubleBuffer](#) (void **, void **, int, char *)
- void [ExpandBuffer](#) (void **, LONG *, int)
- LONG [iexp](#) (LONG, int)
- int [IsLikeVector](#) (WORD *)
- int [AreArgsEqual](#) (WORD *, WORD *)
- int [CompareArgs](#) (WORD *, WORD *)
- UBYTE * [SkipField](#) (UBYTE *, int)
- int [StrCmp](#) (UBYTE *, UBYTE *)
- int [StrLCmp](#) (UBYTE *, UBYTE *)
- int [StrHICmp](#) (UBYTE *, UBYTE *)
- int [StrLCont](#) (UBYTE *, UBYTE *)
- int [CmpArray](#) (WORD *, WORD *, WORD)
- int [ConWord](#) (UBYTE *, UBYTE *)
- int [StrLen](#) (UBYTE *)
- UBYTE * [GetPreVar](#) (UBYTE *, int)
- void [ToGeneral](#) (WORD *, WORD *, WORD)
- WORD [ToPolyFunGeneral](#) (PHEAD WORD *)
- int [ToFast](#) (WORD *, WORD *)
- [SETUPPARAMETERS](#) * [GetSetupPar](#) (UBYTE *)
- int [RecalcSetups](#) (void)
- int [AllocSetups](#) (void)
- [SORTING](#) * [AllocSort](#) (LONG, LONG, LONG, LONG, int, int, LONG, int)
- void [AllocSortFileName](#) ([SORTING](#) *)
- UBYTE * [LoadInputFile](#) (UBYTE *, int)
- UBYTE [GetInput](#) (void)
- void [ClearPushback](#) (void)
- UBYTE [GetChar](#) (int)
- void [CharOut](#) (UBYTE)
- void [UnsetAllowDelay](#) (void)

- void `PopPreVars` (int)
- void `IniModule` (int)
- void `IniSpecialModule` (int)
- int `ModuleInstruction` (int *, int *)
- int `PreProInstruction` (void)
- int `LoadInstruction` (int)
- int `LoadStatement` (int)
- `KEYWORD` * `FindKeyWord` (UBYTE *, `KEYWORD` *, int)
- `KEYWORD` * `FindInKeyWord` (UBYTE *, `KEYWORD` *, int)
- int `DoDefine` (UBYTE *)
- int `DoRedefine` (UBYTE *)
- int `TheDefine` (UBYTE *, int)
- int `TheUndefine` (UBYTE *)
- int `ClearMacro` (UBYTE *)
- int `DoUndefine` (UBYTE *)
- int `DoInclude` (UBYTE *)
- int `DoReverseInclude` (UBYTE *)
- int `Include` (UBYTE *, int)
- int `DoExternal` (UBYTE *)
- int `DoToExternal` (UBYTE *)
- int `DoFromExternal` (UBYTE *)
- int `DoPrompt` (UBYTE *)
- int `DoSetExternal` (UBYTE *)
- int `DoSetExternalAttr` (UBYTE *)
- int `DoRmExternal` (UBYTE *)
- int `DoFactDollar` (UBYTE *)
- WORD `GetDollarNumber` (UBYTE **, `DOLLARS`)
- int `DoSetRandom` (UBYTE *)
- int `DoOptimize` (UBYTE *)
- int `DoClearOptimize` (UBYTE *)
- int `DoSkipExtraSymbols` (UBYTE *)
- int `DoTimeOutAfter` (UBYTE *)
- int `DoMessage` (UBYTE *)
- int `DoPreOut` (UBYTE *)
- int `DoPreAppend` (UBYTE *)
- int `DoPreCreate` (UBYTE *)
- int `DoPreAssign` (UBYTE *)
- int `DoPreBreak` (UBYTE *)
- int `DoPreDefault` (UBYTE *)
- int `DoPreSwitch` (UBYTE *)
- int `DoPreEndSwitch` (UBYTE *)
- int `DoPreCase` (UBYTE *)
- int `DoPreShow` (UBYTE *)
- int `DoPreExchange` (UBYTE *)
- int `DoSystem` (UBYTE *)
- int `DoPipe` (UBYTE *)
- void `StartPrepro` (void)
- int `Dofldef` (UBYTE *, int)
- int `Doflydef` (UBYTE *)
- int `Dofnndef` (UBYTE *)
- int `DoElse` (UBYTE *)
- int `DoElseif` (UBYTE *)
- int `DoEndif` (UBYTE *)
- int `DoTerminate` (UBYTE *)
- int `Dolf` (UBYTE *)

- int [DoCall](#) (UBYTE *)
- int [DoDebug](#) (UBYTE *)
- int [DoContinueDo](#) (UBYTE *)
- int [DoDo](#) (UBYTE *)
- int [DoBreakDo](#) (UBYTE *)
- int [DoEnddo](#) (UBYTE *)
- int [DoEndprocedure](#) (UBYTE *)
- int [DoInside](#) (UBYTE *)
- int [DoEndInside](#) (UBYTE *)
- int [DoProcedure](#) (UBYTE *)
- int [DoPrePrintTimes](#) (UBYTE *)
- int [DoPreWrite](#) (UBYTE *)
- int [DoPreClose](#) (UBYTE *)
- int [DoPreRemove](#) (UBYTE *)
- int [DoPreSortReallocate](#) (UBYTE *)
- int [DoCommentChar](#) (UBYTE *)
- int [DoProcExtension](#) (UBYTE *)
- int [DoPreReset](#) (UBYTE *)
- void [WriteString](#) (int, UBYTE *, int)
- void [WriteUnfinString](#) (int, UBYTE *, int)
- UBYTE * [AddToString](#) (UBYTE *, UBYTE *, int)
- UBYTE * [PreCalc](#) (void)
- UBYTE * [PreEval](#) (UBYTE *, LONG *)
- void [NumToStr](#) (UBYTE *, LONG)
- int [PreCmp](#) (int, int, UBYTE *, int, int, UBYTE *, int)
- int [PreEq](#) (int, int, UBYTE *, int, int, UBYTE *, int)
- UBYTE * [pParseObject](#) (UBYTE *, int *, LONG *)
- UBYTE * [PrelfEval](#) (UBYTE *, int *)
- int [EvalPrelf](#) (UBYTE *)
- int [PreLoad](#) (PRELOAD *, UBYTE *, UBYTE *, int, char *)
- int [PreSkip](#) (UBYTE *, UBYTE *, int)
- UBYTE * [EndOfToken](#) (UBYTE *)
- void [SetSpecialMode](#) (int, int)
- void [MakeGlobal](#) (void)
- int [ExecModule](#) (int)
- int [ExecStore](#) (void)
- void [FullCleanUp](#) (void)
- int [DoExecStatement](#) (void)
- int [DoPipeStatement](#) (void)
- int [DoPolyfun](#) (UBYTE *)
- int [DoPolyratfun](#) (UBYTE *)
- int [CompileStatement](#) (UBYTE *)
- UBYTE * [ToToken](#) (UBYTE *)
- int [GetDollar](#) (UBYTE *)
- void [MesWork](#) (void) NORETURN
- int [MesPrint](#) (const char *,...)
- int [MesCall](#) (char *)
- UBYTE * [NumCopy](#) (WORD, UBYTE *)
- char * [LongCopy](#) (LONG, char *)
- char * [LongLongCopy](#) (off_t *, char *)
- void [ReserveTempFiles](#) (int)
- void [PrintTerm](#) (WORD *, char *)
- void [PrintTermC](#) (WORD *, char *)
- void [PrintSubTerm](#) (WORD *, char *)
- void [PrintWords](#) (WORD *, LONG)

- void [PrintSeq](#) (WORD *, char *)
- int [ExpandTripleDots](#) (int)
- LONG [ComPress](#) (WORD **, LONG *)
- void [StageSort](#) (FILEHANDLE *)
- void [M_free](#) (void *, const char *)
- void [ClearWildcardNames](#) (void)
- int [AddWildcardName](#) (UBYTE *)
- int [GetWildcardName](#) (UBYTE *)
- void [Globalize](#) (int)
- void [ResetVariables](#) (int)
- void [AddToPreTypes](#) (int)
- void [MessPreNesting](#) (int)
- LONG [GetStreamPosition](#) (STREAM *)
- WORD * [DoubleCbuffer](#) (int, WORD *, int)
- WORD * [AddLHS](#) (int)
- WORD * [AddRHS](#) (int, int)
- int [AddNtoL](#) (int, WORD *)
- int [AddNtoC](#) (int, int, WORD *, int)
- void [DoubleIfBuffers](#) (void)
- STREAM * [CreateStream](#) (UBYTE *)
- int [setonoff](#) (UBYTE *, int *, int, int)
- int [DoPrint](#) (UBYTE *, int)
- int [SetExpr](#) (UBYTE *, int, int)
- void [AddToCom](#) (int, WORD *)
- int [Add2ComStrings](#) (int, WORD *, UBYTE *, UBYTE *)
- int [DoSymmetrize](#) (UBYTE *, int)
- int [DoArgument](#) (UBYTE *, int)
- int [ArgFactorize](#) (PHEAD WORD *, WORD *)
- WORD * [TakeArgContent](#) (PHEAD WORD *, WORD *)
- WORD * [MakeInteger](#) (PHEAD WORD *, WORD *, WORD *)
- WORD * [MakeMod](#) (PHEAD WORD *, WORD *, WORD *)
- WORD [FindArg](#) (PHEAD WORD *)
- int [InsertArg](#) (PHEAD WORD *, WORD *, int)
- int [CleanupArgCache](#) (PHEAD WORD)
- int [ArgSymbolMerge](#) (WORD *, WORD *)
- int [ArgDotproductMerge](#) (WORD *, WORD *)
- void [SortWeights](#) (LONG *, LONG *, WORD)
- int [DoBrackets](#) (UBYTE *, int)
- int [DoPutInside](#) (UBYTE *, int)
- WORD * [CountComp](#) (UBYTE *, WORD *)
- int [CoAntiBracket](#) (UBYTE *)
- int [CoAntiSymmetrize](#) (UBYTE *)
- int [DoArgPlode](#) (UBYTE *, int)
- int [CoArgExplode](#) (UBYTE *)
- int [CoArgImplode](#) (UBYTE *)
- int [CoArgument](#) (UBYTE *)
- int [CoInside](#) (UBYTE *)
- int [ExeclnInside](#) (UBYTE *)
- int [CoInExpression](#) (UBYTE *)
- int [CoInParallel](#) (UBYTE *)
- int [CoNotInParallel](#) (UBYTE *)
- int [DoInParallel](#) (UBYTE *, int)
- int [CoEndInExpression](#) (UBYTE *)
- int [CoBracket](#) (UBYTE *)
- int [CoPutInside](#) (UBYTE *)

- int [CoAntiPutInside](#) (UBYTE *)
- int [CoMultiBracket](#) (UBYTE *)
- int [CoCFunction](#) (UBYTE *)
- int [CoCTensor](#) (UBYTE *)
- int [CoCollect](#) (UBYTE *)
- int [CoCompress](#) (UBYTE *)
- int [CoContract](#) (UBYTE *)
- int [CoCycleSymmetrize](#) (UBYTE *)
- int [CoDelete](#) (UBYTE *)
- int [CoTableBase](#) (UBYTE *)
- int [CoApply](#) (UBYTE *)
- int [CoDenominators](#) (UBYTE *)
- int [CoDimension](#) (UBYTE *)
- int [CoDiscard](#) (UBYTE *)
- int [CoDisorder](#) (UBYTE *)
- int [CoDrop](#) (UBYTE *)
- int [CoDropCoefficient](#) (UBYTE *)
- int [CoDropSymbols](#) (UBYTE *)
- int [CoElse](#) (UBYTE *)
- int [CoElseif](#) (UBYTE *)
- int [CoEndArgument](#) (UBYTE *)
- int [CoEndInside](#) (UBYTE *)
- int [CoEndIf](#) (UBYTE *)
- int [CoEndRepeat](#) (UBYTE *)
- int [CoEndTerm](#) (UBYTE *)
- int [CoEndWhile](#) (UBYTE *)
- int [CoExit](#) (UBYTE *)
- int [CoFactArg](#) (UBYTE *)
- int [CoFactDollar](#) (UBYTE *)
- int [CoFactorize](#) (UBYTE *)
- int [CoNFactorize](#) (UBYTE *)
- int [CoUnFactorize](#) (UBYTE *)
- int [CoNUnFactorize](#) (UBYTE *)
- int [DoFactorize](#) (UBYTE *, int)
- int [CoFill](#) (UBYTE *)
- int [CoFillExpression](#) (UBYTE *)
- int [CoFixIndex](#) (UBYTE *)
- int [CoFormat](#) (UBYTE *)
- int [CoGlobal](#) (UBYTE *)
- int [CoGlobalFactorized](#) (UBYTE *)
- int [CoGoTo](#) (UBYTE *)
- int [Cold](#) (UBYTE *)
- int [ColdNew](#) (UBYTE *)
- int [ColdOld](#) (UBYTE *)
- int [Colf](#) (UBYTE *)
- int [ColfMatch](#) (UBYTE *)
- int [ColfNoMatch](#) (UBYTE *)
- int [ColIndex](#) (UBYTE *)
- int [ColInsideFirst](#) (UBYTE *)
- int [CoKeep](#) (UBYTE *)
- int [CoLabel](#) (UBYTE *)
- int [CoLoad](#) (UBYTE *)
- int [CoLocal](#) (UBYTE *)
- int [CoLocalFactorized](#) (UBYTE *)
- int [CoMany](#) (UBYTE *)

- int [CoMerge](#) (UBYTE *)
- int [CoStuffle](#) (UBYTE *)
- int [CoMetric](#) (UBYTE *)
- int [CoModOption](#) (UBYTE *)
- int [CoModuleOption](#) (UBYTE *)
- int [CoModulus](#) (UBYTE *)
- int [CoMulti](#) (UBYTE *)
- int [CoMultiply](#) (UBYTE *)
- int [CoNFunction](#) (UBYTE *)
- int [CoNPrint](#) (UBYTE *)
- int [CoNTensor](#) (UBYTE *)
- int [CoNWrite](#) (UBYTE *)
- int [CoNoDrop](#) (UBYTE *)
- int [CoNoSkip](#) (UBYTE *)
- int [CoNormalize](#) (UBYTE *)
- int [CoMakeInteger](#) (UBYTE *)
- int [CoFlags](#) (UBYTE *, int)
- int [CoOff](#) (UBYTE *)
- int [CoOn](#) (UBYTE *)
- int [CoOnce](#) (UBYTE *)
- int [CoOnly](#) (UBYTE *)
- int [CoOptimizeOption](#) (UBYTE *)
- int [CoPolyFun](#) (UBYTE *)
- int [CoPolyRatFun](#) (UBYTE *)
- int [CoPrint](#) (UBYTE *)
- int [CoPrintB](#) (UBYTE *)
- int [CoProperCount](#) (UBYTE *)
- int [CoUnitTrace](#) (UBYTE *)
- int [CoRCycleSymmetrize](#) (UBYTE *)
- int [CoRatio](#) (UBYTE *)
- int [CoRedefine](#) (UBYTE *)
- int [CoRenumber](#) (UBYTE *)
- int [CoRepeat](#) (UBYTE *)
- int [CoSave](#) (UBYTE *)
- int [CoSelect](#) (UBYTE *)
- int [CoSet](#) (UBYTE *)
- int [CoSetExitFlag](#) (UBYTE *)
- int [CoSkip](#) (UBYTE *)
- int [CoProcessBucket](#) (UBYTE *)
- int [CoPushHide](#) (UBYTE *)
- int [CoPopHide](#) (UBYTE *)
- int [CoHide](#) (UBYTE *)
- int [CoIntoHide](#) (UBYTE *)
- int [CoNoIntoHide](#) (UBYTE *)
- int [CoNoHide](#) (UBYTE *)
- int [CoUnHide](#) (UBYTE *)
- int [CoNoUnHide](#) (UBYTE *)
- int [CoSort](#) (UBYTE *)
- int [CoSplitArg](#) (UBYTE *)
- int [CoSplitFirstArg](#) (UBYTE *)
- int [CoSplitLastArg](#) (UBYTE *)
- int [CoSum](#) (UBYTE *)
- int [CoSymbol](#) (UBYTE *)
- int [CoSymmetrize](#) (UBYTE *)
- int [DoTable](#) (UBYTE *, int)

- int [CoTable](#) (UBYTE *)
- int [CoTerm](#) (UBYTE *)
- int [CoNTable](#) (UBYTE *)
- int [CoCTable](#) (UBYTE *)
- void [EmptyTable](#) ([TABLES](#))
- int [CoToTensor](#) (UBYTE *)
- int [CoToVector](#) (UBYTE *)
- int [CoTrace4](#) (UBYTE *)
- int [CoTraceN](#) (UBYTE *)
- int [CoChisholm](#) (UBYTE *)
- int [CoTransform](#) (UBYTE *)
- int [CoClearTable](#) (UBYTE *)
- int [DoChain](#) (UBYTE *, int)
- int [CoChainin](#) (UBYTE *)
- int [CoChainout](#) (UBYTE *)
- int [CoTryReplace](#) (UBYTE *)
- int [CoVector](#) (UBYTE *)
- int [CoWhile](#) (UBYTE *)
- int [CoWrite](#) (UBYTE *)
- int [CoAuto](#) (UBYTE *)
- int [CoSwitch](#) (UBYTE *)
- int [CoCase](#) (UBYTE *)
- int [CoBreak](#) (UBYTE *)
- int [CoDefault](#) (UBYTE *)
- int [CoEndSwitch](#) (UBYTE *)
- int [CoTBaddto](#) (UBYTE *)
- int [CoTBaudit](#) (UBYTE *)
- int [CoTbcleanup](#) (UBYTE *)
- int [CoTBcreate](#) (UBYTE *)
- int [CoTBenter](#) (UBYTE *)
- int [CoTBhelp](#) (UBYTE *)
- int [CoTBload](#) (UBYTE *)
- int [CoTBoff](#) (UBYTE *)
- int [CoTBon](#) (UBYTE *)
- int [CoTBopen](#) (UBYTE *)
- int [CoTBreplace](#) (UBYTE *)
- int [CoTBuse](#) (UBYTE *)
- int [CoTestUse](#) (UBYTE *)
- int [CoThreadBucket](#) (UBYTE *)
- int [AddComString](#) (int, WORD *, UBYTE *, int)
- int [CompileAlgebra](#) (UBYTE *, int, WORD *)
- int [IsIdStatement](#) (UBYTE *)
- UBYTE * [IsRHS](#) (UBYTE *, UBYTE)
- int [ParenthesesTest](#) (UBYTE *)
- int [tokenize](#) (UBYTE *, WORD)
- void [WriteTokens](#) (SBYTE *)
- int [simp1token](#) (SBYTE *)
- int [simpwtoken](#) (SBYTE *)
- int [simp2token](#) (SBYTE *)
- int [simp3atoken](#) (SBYTE *, int)
- int [simp3btoken](#) (SBYTE *, int)
- int [simp4token](#) (SBYTE *)
- int [simp5token](#) (SBYTE *, int)
- int [simp6token](#) (SBYTE *, int)
- UBYTE * [SkipAName](#) (UBYTE *)

- int [TestTables](#) (void)
- int [GetLabel](#) (UBYTE *)
- int [ColdExpression](#) (UBYTE *, int)
- int [CoAssign](#) (UBYTE *)
- int [DoExpr](#) (UBYTE *, int, int)
- int [CompileSubExpressions](#) (SBYTE *)
- int [CodeGenerator](#) (SBYTE *)
- int [CompleteTerm](#) (WORD *, UWORD *, UWORD *, WORD, WORD, int)
- int [CodeFactors](#) (SBYTE *s)
- WORD [GenerateFactors](#) (WORD, WORD)
- int [InsTree](#) (int, int)
- int [FindTree](#) (int, WORD *)
- void [RedoTree](#) (CBUF *, int)
- void [ClearTree](#) (int)
- int [CatchDollar](#) (int)
- int [AssignDollar](#) (PHEAD WORD *, WORD)
- UBYTE * [WriteDollarToBuffer](#) (WORD, WORD)
- UBYTE * [WriteDollarFactorToBuffer](#) (WORD, WORD, WORD)
- void [AddToDollarBuffer](#) (UBYTE *)
- int [PutTermInDollar](#) (WORD *, WORD)
- void [TermAssign](#) (WORD *)
- void [WildDollars](#) (PHEAD WORD *)
- LONG [numcommute](#) (WORD *, LONG *)
- int [FullRenummer](#) (PHEAD WORD *, WORD)
- int [Lus](#) (WORD *, WORD, WORD, WORD, WORD, WORD)
- int [FindLus](#) (int, int, int)
- int [CoReplaceLoop](#) (UBYTE *)
- int [CoFindLoop](#) (UBYTE *)
- int [DoFindLoop](#) (UBYTE *, int)
- int [CoFunPowers](#) (UBYTE *)
- int [SortTheList](#) (int *, int)
- int [MatchIsPossible](#) (WORD *, WORD *)
- int [StudyPattern](#) (WORD *)
- WORD [DolToTensor](#) (PHEAD WORD)
- WORD [DolToFunction](#) (PHEAD WORD)
- WORD [DolToVector](#) (PHEAD WORD)
- WORD [DolToNumber](#) (PHEAD WORD)
- WORD [DolToSymbol](#) (PHEAD WORD)
- WORD [DolToIndex](#) (PHEAD WORD)
- LONG [DolToLong](#) (PHEAD WORD)
- int [DollarFactorize](#) (PHEAD WORD)
- int [CoPrintTable](#) (UBYTE *)
- int [CoDeallocateTable](#) (UBYTE *)
- void [CleanDollarFactors](#) (DOLLARS)
- WORD * [TakeDollarContent](#) (PHEAD WORD *, WORD **)
- WORD * [MakeDollarInteger](#) (PHEAD WORD *, WORD **)
- WORD * [MakeDollarMod](#) (PHEAD WORD *, WORD **)
- int [GetDolNum](#) (PHEAD WORD *, WORD *)
- void [AddPotModdollar](#) (WORD)
- int [Optimize](#) (WORD, int)
- int [ClearOptimize](#) (void)
- int [TakeLongRoot](#) (UWORD *, WORD *, WORD)
- int [TakeRatRoot](#) (UWORD *, WORD *, WORD)
- int [MakeRational](#) (WORD, WORD, WORD *, WORD *)
- int [MakeLongRational](#) (PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *)

- void [ClearTableTree](#) ([TABLES](#))
- int [InsTableTree](#) ([TABLES](#), WORD *)
- void [RedoTableTree](#) ([TABLES](#), int)
- int [FindTableTree](#) ([TABLES](#), WORD *, int)
- void [finishcbuf](#) (WORD)
- void [clearcbuf](#) (WORD)
- void [CleanUpSort](#) (int)
- FILEHANDLE * [AllocFileHandle](#) (WORD, char *)
- void [DeAllocFileHandle](#) (FILEHANDLE *)
- void [LowerSortLevel](#) (void)
- WORD * [PolyRatFunSpecial](#) (PHEAD WORD *, WORD *)
- void [SimpleSplitMergeRec](#) (WORD *, WORD, WORD *)
- void [SimpleSplitMerge](#) (WORD *, WORD)
- WORD [BinarySearch](#) (WORD *, WORD, WORD)
- int [InsideDollar](#) (PHEAD WORD *, WORD)
- DOLLARS [DoToTerms](#) (PHEAD WORD)
- WORD [EvalDoLoopArg](#) (PHEAD WORD *, WORD)
- int [SetExprCases](#) (int, int, int)
- int [TestSelect](#) (WORD *, WORD *)
- void [SubsInAll](#) (PHEAD0)
- void [TransferBuffer](#) (int, int, int)
- int [TakeIDfunction](#) (PHEAD WORD *)
- int [MakeSetupAllocs](#) (void)
- int [TryFileSetups](#) (void)
- void [ExchangeExpressions](#) (int, int)
- void [ExchangeDollars](#) (int, int)
- int [GetFirstBracket](#) (WORD *, int)
- int [GetFirstTerm](#) (WORD *, int, int)
- int [GetContent](#) (WORD *, int)
- int [CleanupTerm](#) (WORD *)
- WORD [ContentMerge](#) (PHEAD WORD *, WORD *)
- UBYTE * [PrelfDollarEval](#) (UBYTE *, int *)
- LONG [TermsInDollar](#) (WORD)
- LONG [SizeOfDollar](#) (WORD)
- LONG [TermsInExpression](#) (WORD)
- LONG [SizeOfExpression](#) (WORD)
- WORD * [TranslateExpression](#) (UBYTE *)
- int [IsSetMember](#) (WORD *, WORD)
- int [IsMultipleOf](#) (WORD *, WORD *)
- int [TwoExprCompare](#) (WORD *, WORD *, int)
- void [UpdatePositions](#) (void)
- void [M_check](#) (void)
- void [M_print](#) (void)
- void [M_check1](#) (void)
- void [PrintTime](#) (UBYTE *)
- POSITION * [FindBracket](#) (WORD, WORD *)
- void [PutBracketInIndex](#) (PHEAD WORD *, [POSITION](#) *)
- void [ClearBracketIndex](#) (WORD)
- void [OpenBracketIndex](#) (WORD)
- int [DoNoParallel](#) (UBYTE *)
- int [DoParallel](#) (UBYTE *)
- int [DoModSum](#) (UBYTE *)
- int [DoModMax](#) (UBYTE *)
- int [DoModMin](#) (UBYTE *)
- int [DoModLocal](#) (UBYTE *)

- UBYTE * [DoModDollar](#) (UBYTE *, int)
- int [DoProcessBucket](#) (UBYTE *)
- int [DoinParallel](#) (UBYTE *)
- int [DonotinParallel](#) (UBYTE *)
- int [FlipTable](#) (FUNCTIONS, int)
- int [ChainIn](#) (PHEAD WORD *, WORD)
- int [ChainOut](#) (PHEAD WORD *, WORD)
- int [ArgumentImplode](#) (PHEAD WORD *, WORD *)
- int [ArgumentExplode](#) (PHEAD WORD *, WORD *)
- int [DenToFunction](#) (WORD *, WORD)
- WORD [HowMany](#) (PHEAD WORD *, WORD *)
- void [RemoveDollars](#) (void)
- LONG [CountTerms1](#) (PHEAD0)
- LONG [TermsInBracket](#) (PHEAD WORD *, WORD)
- int [Crash](#) (void)
- char * [str_dup](#) (char *)
- void [convertblock](#) (INDEXBLOCK *, INDEXBLOCK *, int)
- void [convertnamesblock](#) (NAMESBLOCK *, NAMESBLOCK *, int)
- void [convertiniinfo](#) (INIINFO *, INIINFO *, int)
- int [ReadIndex](#) (DBASE *)
- int [WriteIndexBlock](#) (DBASE *, MLONG)
- int [WriteNamesBlock](#) (DBASE *, MLONG)
- int [WriteIndex](#) (DBASE *)
- int [WriteIniInfo](#) (DBASE *)
- int [ReadIniInfo](#) (DBASE *)
- int [AddToIndex](#) (DBASE *, MLONG)
- DBASE * [GetDbase](#) (char *, MLONG)
- DBASE * [OpenDbase](#) (char *)
- char * [ReadObject](#) (DBASE *, MLONG, char *)
- char * [ReadijObject](#) (DBASE *, MLONG, MLONG, char *)
- int [ExistsObject](#) (DBASE *, MLONG, char *)
- int [DeleteObject](#) (DBASE *, MLONG, char *)
- int [WriteObject](#) (DBASE *, MLONG, char *, char *, MLONG)
- MLONG [AddObject](#) (DBASE *, MLONG, char *, char *)
- DBASE * [NewDbase](#) (char *, MLONG)
- void [FreeTableBase](#) (DBASE *)
- int [ComposeTableNames](#) (DBASE *)
- int [PutTableNames](#) (DBASE *)
- MLONG [AddTableName](#) (DBASE *, char *, TABLES)
- MLONG [GetTableName](#) (DBASE *, char *)
- MLONG [FindTableNumber](#) (DBASE *, char *)
- int [TryEnvironment](#) (void)
- int [CopyExpression](#) (FILEHANDLE *, FILEHANDLE *)
- int [set_in](#) (UBYTE, set_of_char)
- one_byte [set_set](#) (UBYTE, set_of_char)
- one_byte [set_del](#) (UBYTE, set_of_char)
- one_byte [set_sub](#) (set_of_char, set_of_char, set_of_char)
- int [DoPreAddSeparator](#) (UBYTE *)
- int [DoPreRmSeparator](#) (UBYTE *)
- int [openExternalChannel](#) (UBYTE *, int, UBYTE *, UBYTE *)
- int [initPresetExternalChannels](#) (UBYTE *, int)
- int [closeExternalChannel](#) (int)
- int [selectExternalChannel](#) (int)
- int [getCurrentExternalChannel](#) (void)
- void [closeAllExternalChannels](#) (void)

- UBYTE * [defineChannel](#) (UBYTE *, [HANDLERS](#) *)
- int [writeToChannel](#) (int, UBYTE *, [HANDLERS](#) *)
- int [writeBufToExtChannelOk](#) (char *, size_t)
- int [getcFromExtChannelOk](#) (void)
- int [setKillModeForExternalChannelOk](#) (int, int)
- int [setTerminatorForExternalChannelOk](#) (char *)
- int [getcFromExtChannelFailure](#) (void)
- int [setKillModeForExternalChannelFailure](#) (int, int)
- int [setTerminatorForExternalChannelFailure](#) (char *)
- int [writeBufToExtChannelFailure](#) (char *, size_t)
- int [ReleaseTB](#) (void)
- int [SymbolNormalize](#) (WORD *)
- int [TestFunFlag](#) (PHEAD WORD *)
- WORD [CompareSymbols](#) (PHEAD WORD *, WORD *, WORD)
- WORD [CompareHSymbols](#) (PHEAD WORD *, WORD *, WORD)
- WORD [NextPrime](#) (PHEAD WORD)
- WORD [Moebius](#) (PHEAD WORD)
- UWORD [wranf](#) (PHEAD0)
- UWORD [iranf](#) (PHEAD UWORD)
- void [iniwranf](#) (PHEAD0)
- UBYTE * [PreRandom](#) (UBYTE *)
- WORD * [PolyNormPoly](#) (PHEAD WORD)
- WORD * [EvaluateGcd](#) (PHEAD WORD *)
- int [TreatPolyRatFun](#) (PHEAD WORD *)
- int [ReadSaveHeader](#) (void)
- LONG [ReadSaveIndex](#) ([FILEINDEX](#) *)
- LONG [ReadSaveExpression](#) (UBYTE *, UBYTE *, LONG *, LONG *)
- UBYTE * [ReadSaveTerm32](#) (UBYTE *, UBYTE *, UBYTE **, UBYTE *, UBYTE *, int)
- LONG [ReadSaveVariables](#) (UBYTE *, UBYTE *, LONG *, LONG *, [INDEXENTRY](#) *, LONG *)
- LONG [WriteStoreHeader](#) (WORD)
- void [InitRecovery](#) (void)
- int [CheckRecoveryFile](#) (void)
- void [DeleteRecoveryFile](#) (void)
- char * [RecoveryFilename](#) (void)
- int [DoRecovery](#) (int *)
- void [DoCheckpoint](#) (int)
- void [NumberMallocAddMemory](#) (PHEAD0)
- void [CacheNumberMallocAddMemory](#) (PHEAD0)
- void [TermMallocAddMemory](#) (PHEAD0)
- void [ExprStatus](#) ([EXPRESSIONS](#))
- void [iniTools](#) (void)
- int [TestTerm](#) (WORD *)
- int [RunTransform](#) (PHEAD WORD *term, WORD *params)
- int [RunEncode](#) (PHEAD WORD *fun, WORD *args, WORD *info)
- int [RunDecode](#) (PHEAD WORD *fun, WORD *args, WORD *info)
- int [RunReplace](#) (PHEAD WORD *fun, WORD *args, WORD *info)
- int [RunImplode](#) (WORD *fun, WORD *args)
- int [RunExplode](#) (PHEAD WORD *fun, WORD *args)
- int [TestArgNum](#) (int n, int totarg, WORD *args)
- WORD [PutArgInScratch](#) (WORD *arg, UWORD *scrat)
- UBYTE * [ReadRange](#) (UBYTE *s, WORD *out, int par)
- int [FindRange](#) (PHEAD WORD *, WORD *, WORD *, WORD)
- int [RunPermute](#) (PHEAD WORD *fun, WORD *args, WORD *info)
- int [RunReverse](#) (PHEAD WORD *fun, WORD *args)
- int [RunCycle](#) (PHEAD WORD *fun, WORD *args, WORD *info)

- int [RunAddArg](#) (PHEAD WORD *fun, WORD *args)
- int [RunMulArg](#) (PHEAD WORD *fun, WORD *args)
- int [RunIsLyndon](#) (PHEAD WORD *fun, WORD *args, int par)
- WORD [RunToLyndon](#) (PHEAD WORD *fun, WORD *args, int par)
- int [RunDropArg](#) (PHEAD WORD *fun, WORD *args)
- int [RunSelectArg](#) (PHEAD WORD *fun, WORD *args)
- int [RunDedup](#) (PHEAD WORD *fun, WORD *args)
- int [RunZtoHArg](#) (PHEAD WORD *fun, WORD *args)
- int [RunHtoZArg](#) (PHEAD WORD *fun, WORD *args)
- int [NormPolyTerm](#) (PHEAD WORD *)
- WORD **ComparePoly** (WORD *, WORD *, WORD)
- int [ConvertToPoly](#) (PHEAD WORD *, WORD *, WORD *, WORD)
- int [LocalConvertToPoly](#) (PHEAD WORD *, WORD *, WORD, WORD)
- int [ConvertFromPoly](#) (PHEAD WORD *, WORD *, WORD, WORD, WORD, WORD)
- int [FindSubterm](#) (WORD *)
- int [FindLocalSubterm](#) (PHEAD WORD *, WORD)
- void [PrintSubtermList](#) (int, int)
- void [PrintExtraSymbol](#) (int, WORD *, int)
- int [FindSubexpression](#) (WORD *)
- void [UpdateMaxSize](#) (void)
- int [CoToPolynomial](#) (UBYTE *)
- int [CoFromPolynomial](#) (UBYTE *)
- int [CoArgToExtraSymbol](#) (UBYTE *)
- int [CoExtraSymbols](#) (UBYTE *)
- UBYTE * [GetDoParam](#) (UBYTE *, WORD **, int)
- WORD * [GetIfDollarFactor](#) (UBYTE **, WORD *)
- int [CoDo](#) (UBYTE *)
- int [CoEndDo](#) (UBYTE *)
- int [ExtraSymFun](#) (PHEAD WORD *, WORD)
- int [PruneExtraSymbols](#) (WORD)
- int [IniFbuffer](#) (WORD)
- void **IniFbufs** (void)
- int [GCDfunction](#) (PHEAD WORD *, WORD)
- WORD * [GCDfunction3](#) (PHEAD WORD *, WORD *)
- int [ReadPolyRatFun](#) (PHEAD WORD *)
- int [FromPolyRatFun](#) (PHEAD WORD *, WORD **, WORD **)
- WORD * [PolyDiv](#) (PHEAD WORD *, WORD *, char *)
- void [GCDclean](#) (PHEAD WORD *, WORD *)
- WORD * [TakeSymbolContent](#) (PHEAD WORD *, WORD *)
- int [GCDterms](#) (PHEAD WORD *, WORD *, WORD *)
- WORD * [PutExtraSymbols](#) (PHEAD WORD *, WORD, int *)
- WORD * [TakeExtraSymbols](#) (PHEAD WORD *, WORD)
- WORD * [MultiplyWithTerm](#) (PHEAD WORD *, WORD *, WORD)
- WORD * [TakeContent](#) (PHEAD WORD *, WORD *)
- int [MergeSymbolLists](#) (PHEAD WORD *, WORD *, int)
- int [MergeDotproductLists](#) (PHEAD WORD *, WORD *, int)
- WORD * [CreateExpression](#) (PHEAD WORD)
- int [DIVfunction](#) (PHEAD WORD *, WORD, int)
- WORD * [MULfunc](#) (PHEAD WORD *, WORD *)
- WORD * [ConvertArgument](#) (PHEAD WORD *, int *)
- int [ExpandRat](#) (PHEAD WORD *)
- int [InvPoly](#) (PHEAD WORD *, WORD, WORD)
- WORD [TestDoLoop](#) (PHEAD WORD *, WORD)
- WORD [TestEndDoLoop](#) (PHEAD WORD *, WORD)
- WORD * [poly_gcd](#) (PHEAD WORD *, WORD *, WORD)

- WORD * [poly_div](#) (PHEAD WORD *, WORD *, WORD)
- WORD * [poly_rem](#) (PHEAD WORD *, WORD *, WORD)
- WORD * [poly_inverse](#) (PHEAD WORD *, WORD *)
- WORD * [poly_mul](#) (PHEAD WORD *, WORD *)
- WORD * [poly_ratfun_add](#) (PHEAD WORD *, WORD *)
- int [poly_ratfun_normalize](#) (PHEAD WORD *)
- int [poly_factorize_argument](#) (PHEAD WORD *, WORD *)
- WORD * [poly_factorize_dollar](#) (PHEAD WORD *)
- int [poly_factorize_expression](#) (EXPRESSIONS)
- int [poly_unfactorize_expression](#) (EXPRESSIONS)
- void [poly_free_poly_vars](#) (PHEAD const char *)
- void [optimize_print_code](#) (int)
- int [DoPreAdd](#) (UBYTE *s)
- int [DoPreUseDictionary](#) (UBYTE *s)
- int [DoPreCloseDictionary](#) (UBYTE *s)
- int [DoPreOpenDictionary](#) (UBYTE *s)
- void [RemoveDictionary](#) (DICTIONARY *dict)
- void [UnSetDictionary](#) (void)
- int [SetDictionaryOptions](#) (UBYTE *options)
- int [AddToDictionary](#) (DICTIONARY *dict, UBYTE *left, UBYTE *right)
- int [AddDictionary](#) (UBYTE *name)
- int [FindDictionary](#) (UBYTE *name)
- UBYTE * [IsExponentSign](#) (void)
- UBYTE * [IsMultiplySign](#) (void)
- void [TransformRational](#) (UWORD *a, WORD na)
- void [WriteDictionary](#) (DICTIONARY *)
- void [ShrinkDictionary](#) (DICTIONARY *)
- void [MultiplyToLine](#) (void)
- UBYTE * [FindSymbol](#) (WORD num)
- UBYTE * [FindVector](#) (WORD num)
- UBYTE * [FindIndex](#) (WORD num)
- UBYTE * [FindFunction](#) (WORD num)
- UBYTE * [FindFunWithArgs](#) (WORD *t)
- UBYTE * [FindExtraSymbol](#) (WORD num)
- LONG [DictToBytes](#) (DICTIONARY *dict, UBYTE *buf)
- DICTIONARY * [DictFromBytes](#) (UBYTE *buf)
- int [CoCreateSpectator](#) (UBYTE *inp)
- int [CoToSpectator](#) (UBYTE *inp)
- int [CoRemoveSpectator](#) (UBYTE *inp)
- int [CoEmptySpectator](#) (UBYTE *inp)
- int [CoCopySpectator](#) (UBYTE *inp)
- int [PutInSpectator](#) (WORD *, WORD)
- void [ClearSpectators](#) (WORD)
- WORD [GetFromSpectator](#) (WORD *, WORD)
- void [FlushSpectators](#) (void)
- WORD * [PreGCD](#) (PHEAD WORD *, WORD *, int)
- int [ReadFromScratch](#) (FILEHANDLE *, POSITION *, UBYTE *, POSITION *)
- int [AddToScratch](#) (FILEHANDLE *, POSITION *, UBYTE *, POSITION *, int)
- int [DoPreAppendPath](#) (UBYTE *)
- int [DoPrePrependPath](#) (UBYTE *)
- int [DoSwitch](#) (PHEAD WORD *, WORD *)
- int [DoEndSwitch](#) (PHEAD WORD *, WORD *)
- SWITCHTABLE * [FindCase](#) (WORD, WORD)
- void [SwitchSplitMergeRec](#) (SWITCHTABLE *, WORD, SWITCHTABLE *)
- void [SwitchSplitMerge](#) (SWITCHTABLE *, WORD)

- int [DoubleSwitchBuffers](#) (void)
- int [DistrN](#) (int, int *, int, int *)
- int [DoNamespace](#) (UBYTE *)
- int [DoEndNamespace](#) (UBYTE *)
- int [DoUse](#) (UBYTE *)
- UBYTE * [SkipName](#) (UBYTE *)
- UBYTE * [ConstructName](#) (UBYTE *, UBYTE)
- int [DoSetUserFlag](#) (UBYTE *)
- int [DoClearUserFlag](#) (UBYTE *)
- VERTEX * [CreateVertex](#) (MODEL *)
- UBYTE * [ReadParticle](#) (UBYTE *, VERTEX *, MODEL *, int)
- int [CoModel](#) (UBYTE *)
- int [CoParticle](#) (UBYTE *)
- int [CoVertex](#) (UBYTE *)
- int [CoEndModel](#) (UBYTE *)
- int [LoadModel](#) (MODEL *)
- int [CoSetUserFlag](#) (UBYTE *)
- int [CoClearUserFlag](#) (UBYTE *)
- int [CoCreateAllLoops](#) (UBYTE *)
- int [CoCreateAllPaths](#) (UBYTE *)
- int [CoCreateAll](#) (UBYTE *)
- int [AllLoops](#) (PHEAD WORD *, WORD)
- LONG [StartLoops](#) (PHEAD WORD *, WORD, LONG, WORD, WORD *, WORD, WORD *, WORD)
- LONG [GenLoops](#) (PHEAD WORD *, WORD, LONG, WORD, WORD *, WORD, WORD *, WORD)
- void [LoopOutput](#) (PHEAD WORD *, WORD, WORD *, WORD)
- int [AllPaths](#) (PHEAD WORD *, WORD)
- LONG [GenPaths](#) (PHEAD WORD *, WORD, LONG, WORD, WORD *, WORD, WORD *, WORD)
- void [PathOutput](#) (PHEAD WORD *, WORD, WORD *, WORD)

4.19.1 Detailed Description

Contains macros and function declarations.

Definition in file [declare.h](#).

4.19.2 Macro Definition Documentation

4.19.2.1 MaX

```
#define MaX(
    x,
    y ) ((x) > (y) ? (x) : (y))
```

Definition at line 40 of file [declare.h](#).

4.19.2.2 MiN

```
#define MiN(
    x,
    y ) ((x) < (y) ? (x) : (y))
```

Definition at line 41 of file [declare.h](#).

4.19.2.3 ABS

```
#define ABS(  
    x ) ( (x) < 0 ? -(x) : (x) )
```

Definition at line 42 of file [declare.h](#).

4.19.2.4 SGN

```
#define SGN(  
    x ) ( (x) > 0 ? 1 : (x) < 0 ? -1 : 0 )
```

Definition at line 43 of file [declare.h](#).

4.19.2.5 REDLENG

```
#define REDLENG(  
    x ) (((x)<0)?((x)+1):((x)-1))/2)
```

Definition at line 44 of file [declare.h](#).

4.19.2.6 INCLENG

```
#define INCLENG(  
    x ) (((x)<0)?((x)*2)-1):((x)*2)+1)
```

Definition at line 45 of file [declare.h](#).

4.19.2.7 GETCOEF

```
#define GETCOEF(  
    x,  
    y ) x += *x; y = x[-1]; x -= ABS(y); y=REDLENG(y)
```

Definition at line 46 of file [declare.h](#).

4.19.2.8 GETSTOP

```
#define GETSTOP(  
    x,  
    y ) y=x+(*x)-1; y -= ABS(*y)-1
```

Definition at line 47 of file [declare.h](#).

4.19.2.9 StuffAdd

```
#define StuffAdd(
    x,
    y ) ( ( (x)<0?-1:1)*(y)+( (y)<0?-1:1)*(x) )
```

Definition at line 48 of file [declare.h](#).

4.19.2.10 EXCHN

```
#define EXCHN(
    t1,
    t2,
    n ) { WORD a,i; for(i=0;i<n;i++){a=t1[i];t1[i]=t2[i];t2[i]=a;} }
```

Definition at line 50 of file [declare.h](#).

4.19.2.11 EXCH

```
#define EXCH(
    x,
    y ) { WORD a = (x); (x) = (y); (y) = a; }
```

Definition at line 51 of file [declare.h](#).

4.19.2.12 TOKENTOLINE

```
#define TOKENTOLINE(
    x,
    y )
```

Value:

```
if ( AC.OutputSpaces == NOSPACEFORMAT ) { \
    TokenToLine((UBYTE *) (y)); } else { TokenToLine((UBYTE *) (x)); }
```

Definition at line 53 of file [declare.h](#).

4.19.2.13 UngetFromStream

```
#define UngetFromStream(
    stream,
    c ) ((stream)->nextchar[(stream)->isnextchar++]=c)
```

Definition at line 56 of file [declare.h](#).

4.19.2.14 AddLineFeed

```
#define AddLineFeed(
    s,
    n ) { (s)[(n)++] = LINEFEED; }
```

Definition at line 60 of file [declare.h](#).

4.19.2.15 TryRecover

```
#define TryRecover(  
    x ) Terminate(-1)
```

Definition at line 62 of file [declare.h](#).

4.19.2.16 UngetChar

```
#define UngetChar(  
    c ) { pushbackchar = c; }
```

Definition at line 63 of file [declare.h](#).

4.19.2.17 ParseNumber

```
#define ParseNumber(  
    x,  
    s ) { (x)=0;while(*(s)>='0'&&*(s)<='9') (x)=10*(x)+*(s)++ -'0'; }
```

Definition at line 64 of file [declare.h](#).

4.19.2.18 ParseSign

```
#define ParseSign(  
    sgn,  
    s )
```

Value:

```
{ (sgn)=0;while(*(s)=='-' || *(s)=='+') {\n  if ( *(s)++ == '-' ) sgn ^= 1;}}
```

Definition at line 65 of file [declare.h](#).

4.19.2.19 ParseSignedNumber

```
#define ParseSignedNumber(  
    x,  
    s )
```

Value:

```
{ int sgn; ParseSign(sgn,s)\n  ParseNumber(x,s) if ( sgn ) x = -x; }
```

Definition at line 67 of file [declare.h](#).

4.19.2.20 NCOPY

```
#define NCOPY(  
    s,  
    t,  
    n ) { while ( (n)-- > 0 ) { *s++ = *t++; } }
```

Definition at line 71 of file [declare.h](#).

4.19.2.21 NCOPYI

```
#define NCOPYI(
    s,
    t,
    n ) { while ( (n)-- > 0 ) { *s++ = *t++; } }
```

Definition at line 73 of file [declare.h](#).

4.19.2.22 NCOPYB

```
#define NCOPYB(
    s,
    t,
    n ) { while ( (n)-- > 0 ) { *s++ = *t++; } }
```

Definition at line 74 of file [declare.h](#).

4.19.2.23 NCOPYI32

```
#define NCOPYI32(
    s,
    t,
    n ) { while ( (n)-- > 0 ) { *s++ = *t++; } }
```

Definition at line 75 of file [declare.h](#).

4.19.2.24 WCOPY

```
#define WCOPY(
    s,
    t,
    n ) { int nn=n; WORD *ss=(WORD *)s, *tt=(WORD *)t; while ( (nn)-- > 0 ) { *ss++ =
    *tt++; } }
```

Definition at line 76 of file [declare.h](#).

4.19.2.25 NeedNumber

```
#define NeedNumber(
    x,
    s,
    err )
```

Value:

```
{ int sgn = 1;
  while ( *s == ' ' || *s == '\t' || *s == '-' || *s == '+' ) {
    if ( *s == '-' ) {sgn = -sgn;} s++; }
  if ( chartype[*s] != 1 ) goto err;
  ParseNumber(x,s)
  if ( sgn < 0 ) {(x) = -(x);} while ( *s == ' ' || *s == '\t' ) s++;
}
```

Definition at line 78 of file [declare.h](#).

4.19.2.26 SKIPBLANKS

```
#define SKIPBLANKS(  
    s ) { while ( *(s) == ' ' || *(s) == '\\t' ) (s)++; }
```

Definition at line 85 of file [declare.h](#).

4.19.2.27 FLUSHCONSOLE

```
#define FLUSHCONSOLE if ( AP.InOutBuf > 0 ) CharOut(LINEFEED)
```

Definition at line 86 of file [declare.h](#).

4.19.2.28 SKIPBRA1

```
#define SKIPBRA1(  
    s )
```

Value:

```
{ int lev1=0; s++; while(*s) { if(*s=='['){lev1++;} \\  
else {if(*s=='{'&&--lev1<0){break;}} s++;} }
```

Definition at line 88 of file [declare.h](#).

4.19.2.29 SKIPBRA2

```
#define SKIPBRA2(  
    s )
```

Value:

```
{ int lev2=0; s++; while(*s) { if(*s=='{'){lev2++;} \\  
else {if(*s=='['&&--lev2<0){break;}} \\  
else {if(*s=='{'){SKIPBRA1(s)}} s++;} }
```

Definition at line 90 of file [declare.h](#).

4.19.2.30 SKIPBRA3

```
#define SKIPBRA3(  
    s )
```

Value:

```
{ int lev3=0; s++; while(*s) { if(*s=='('){lev3++;} \\  
else {if(*s=='['&&--lev3<0){break;}} \\  
else {if(*s=='{'){SKIPBRA2(s)} \\  
else {if(*s=='('){SKIPBRA1(s)}}} s++;} }
```

Definition at line 93 of file [declare.h](#).

4.19.2.31 SKIPBRA4

```
#define SKIPBRA4(
    s )
```

Value:

```
{ int lev4=0; s++; while(*s) { if(*s=='('){lev4++;} \
else {if(*s=='&--lev4<0){break;} \
else {if(*s=='('){SKIPBRA1(s)}} s++;} }
```

Definition at line 97 of file [declare.h](#).

4.19.2.32 SKIPBRA5

```
#define SKIPBRA5(
    s )
```

Value:

```
{ int lev5=0; s++; while(*s) { if(*s=='('){lev5++;} \
else {if(*s=='&--lev5<0){break;} \
else {if(*s=='('){SKIPBRA4(s)} \
else {if(*s=='('){SKIPBRA1(s)}}} s++;} }
```

Definition at line 100 of file [declare.h](#).

4.19.2.33 CYCLE1

```
#define CYCLE1(
    t,
    a,
    i ) {t iX=*a; WORD jX; for(jX=1;jX<i;jX++)a[jX-1]=a[jX]; a[i-1]=iX;}
```

Definition at line 108 of file [declare.h](#).

4.19.2.34 AddToCB

```
#define AddToCB(
    c,
    wx )
```

Value:

```
if(c->Pointer>=c->Top) \
{DoubleCbuffer(c-cbuf,c->Pointer,21);} \
*(c->Pointer)++ = wx;
```

Definition at line 110 of file [declare.h](#).

4.19.2.35 EXCHINOUT

```
#define EXCHINOUT
```

Value:

```
{ FILEHANDLE *ffFi = AR.outfile; \
AR.outfile = AR.infile; AR.infile = ffFi; }
```

Definition at line 114 of file [declare.h](#).

4.19.2.36 BACKINOUT

```
#define BACKINOUT
```

Value:

```
{ FILEHANDLE *ffFi = AR.outfile; POSITION posi; \
  AR.outfile = AR.infile; AR.infile = ffFi; \
  SetEndScratch(AR.infile,&posi); }
```

Definition at line 116 of file [declare.h](#).

4.19.2.37 CopyArg

```
#define CopyArg(  
    to,  
    from )
```

Value:

```
{ if ( *from > 0 ) { int ica = *from; NCOPY(to,from,ica) } \
  else if ( *from <= -FUNCTION ) *to++ = *from++; \
  else { *to++ = *from++; *to++ = *from++; } }
```

Definition at line 120 of file [declare.h](#).

4.19.2.38 FILLARG

```
#define FILLARG(  
    w )
```

Definition at line 129 of file [declare.h](#).

4.19.2.39 COPYARG

```
#define COPYARG(  
    w,  
    t )
```

Definition at line 130 of file [declare.h](#).

4.19.2.40 ZEROARG

```
#define ZEROARG(  
    w )
```

Definition at line 131 of file [declare.h](#).

4.19.2.41 FILLFUN

```
#define FILLFUN(  
    w )
```

Definition at line 138 of file [declare.h](#).

4.19.2.42 COPYFUN

```
#define COPYFUN(  
    w,  
    t )
```

Definition at line 139 of file [declare.h](#).

4.19.2.43 COPYFUN3

```
#define COPYFUN3(  
    w,  
    t )
```

Definition at line 146 of file [declare.h](#).

4.19.2.44 FILLFUN3

```
#define FILLFUN3(  
    w )
```

Definition at line 147 of file [declare.h](#).

4.19.2.45 FILLSUB

```
#define FILLSUB(  
    w )
```

Definition at line 154 of file [declare.h](#).

4.19.2.46 COPYSUB

```
#define COPYSUB(  
    w,  
    ww )
```

Definition at line 155 of file [declare.h](#).

4.19.2.47 FILLEXP

```
#define FILLEXP(  
    w )
```

Definition at line 161 of file [declare.h](#).

4.19.2.48 NEXTARG

```
#define NEXTARG(  
    x ) if(*x>0) x += *x; else if(*x <= -FUNCTION)x++; else x += 2;
```

Definition at line 164 of file [declare.h](#).

4.19.2.49 COPY1ARG

```
#define COPY1ARG(  
    s1,  
    t1 )
```

Value:

```
{ int ica; if ( (ica=t1) > 0 ) { NCOPY(s1,t1,ica) } \  
else if(*t1<=-FUNCTION){*s1++=*t1++;} else{*s1++=*t1++;*s1++=*t1++;} }
```

Definition at line 165 of file [declare.h](#).

4.19.2.50 ZeroFillRange

```
#define ZeroFillRange(  
    w,  
    begin,  
    end )
```

Value:

```
do { \  
    int tmp_i; \  
    for ( tmp_i = begin; tmp_i < end; tmp_i++ ) { (w)[tmp_i] = 0; } \  
} while (0)
```

Fills a buffer by zero in the range [begin,end).

Parameters

<i>w</i>	The buffer.
<i>begin</i>	The index for the beginning of the range.
<i>end</i>	The index for the end of the range (exclusive).

Definition at line 175 of file [declare.h](#).

4.19.2.51 TABLESIZE

```
#define TABLESIZE(  
    a,  
    b ) ((WORD)sizeof(a))/((WORD)sizeof(b))
```

Definition at line 180 of file [declare.h](#).

4.19.2.52 WORDDIF

```
#define WORDDIF(  
    x,  
    y ) (WORD) (x-y)
```

Definition at line 181 of file [declare.h](#).

4.19.2.53 sizeof

```
#define sizeof(  
    a ) ((WORD)sizeof(a))
```

Definition at line 182 of file [declare.h](#).

4.19.2.54 VARNAME

```
#define VARNAME(  
    type,  
    num ) (AC.varnames->namebuffer+type[num].name)
```

Definition at line 183 of file [declare.h](#).

4.19.2.55 DOLLARNAME

```
#define DOLLARNAME(  
    type,  
    num ) (AC.dollarnames->namebuffer+type[num].name)
```

Definition at line 184 of file [declare.h](#).

4.19.2.56 EXPRNAME

```
#define EXPRNAME(  
    num ) (AC.exprnames->namebuffer+Expressions[num].name)
```

Definition at line 185 of file [declare.h](#).

4.19.2.57 PREV

```
#define PREV(  
    x ) prevorder?prevorder:x
```

Definition at line 187 of file [declare.h](#).

4.19.2.58 Terminate

```
#define Terminate(  
    x ) do { TerminateImpl(x, __FILE__, __LINE__, __FUNCTION__); } while(0)
```

Definition at line 189 of file [declare.h](#).

4.19.2.59 SETERROR

```
#define SETERROR(  
    x ) { Terminate(-1); return(-1); }
```

Definition at line 190 of file [declare.h](#).

4.19.2.60 DUMMYUSE

```
#define DUMMYUSE(  
    x ) (void)(x);
```

Definition at line 193 of file [declare.h](#).

4.19.2.61 ADDPOS

```
#define ADDPOS(  
    pp,  
    x ) (pp).p1 = ((pp).p1+(LONG)(x))
```

Definition at line 219 of file [declare.h](#).

4.19.2.62 SETBASELENGTH

```
#define SETBASELENGTH(  
    ss,  
    x ) (ss).p1 = (LONG)(x)
```

Definition at line 220 of file [declare.h](#).

4.19.2.63 SETBASEPOSITION

```
#define SETBASEPOSITION(  
    pp,  
    x ) (pp).p1 = (LONG)(x)
```

Definition at line 221 of file [declare.h](#).

4.19.2.64 ISEQUALPOSINC

```
#define ISEQUALPOSINC(  
    pp1,  
    pp2,  
    x ) ( (pp1).p1 == ((pp2).p1+(LONG)(x)) )
```

Definition at line 222 of file [declare.h](#).

4.19.2.65 ISGEPOSINC

```
#define ISGEPOSINC(  
    pp1,  
    pp2,  
    x ) ( (pp1).p1 >= ((pp2).p1+(LONG)(x)) )
```

Definition at line 223 of file [declare.h](#).

4.19.2.66 DIVPOS

```
#define DIVPOS(  
    pp,  
    n ) ( (pp).p1/(LONG)(n) )
```

Definition at line 224 of file [declare.h](#).

4.19.2.67 MULPOS

```
#define MULPOS(  
    pp,  
    n ) (pp).p1 *= (LONG)(n)
```

Definition at line 225 of file [declare.h](#).

4.19.2.68 DIFPOS

```
#define DIFPOS(  
    ss,  
    pp1,  
    pp2 ) (ss).p1 = ((pp1).p1-(pp2).p1)
```

Definition at line 228 of file [declare.h](#).

4.19.2.69 DIFBASE

```
#define DIFBASE(  
    pp1,  
    pp2 ) ((pp1).p1-(pp2).p1)
```

Definition at line 229 of file [declare.h](#).

4.19.2.70 ADD2POS

```
#define ADD2POS(  
    pp1,  
    pp2 ) (pp1).p1 += (pp2).p1
```

Definition at line 230 of file [declare.h](#).

4.19.2.71 PUTZERO

```
#define PUTZERO(  
    pp ) (pp).p1 = 0
```

Definition at line 231 of file [declare.h](#).

4.19.2.72 BASEPOSITION

```
#define BASEPOSITION(  
    pp ) ((pp).p1)
```

Definition at line 232 of file [declare.h](#).

4.19.2.73 SETSTARTPOS

```
#define SETSTARTPOS(  
    pp ) (pp).p1 = -2
```

Definition at line 233 of file [declare.h](#).

4.19.2.74 NOTSTARTPOS

```
#define NOTSTARTPOS(  
    pp ) ( (pp).p1 > -2 )
```

Definition at line 234 of file [declare.h](#).

4.19.2.75 ISMINPOS

```
#define ISMINPOS(  
    pp ) ( (pp).p1 == -1 )
```

Definition at line 235 of file [declare.h](#).

4.19.2.76 ISEQUALPOS

```
#define ISEQUALPOS(  
    pp1,  
    pp2 ) ( (pp1).p1 == (pp2).p1 )
```

Definition at line 236 of file [declare.h](#).

4.19.2.77 ISNOTEQUALPOS

```
#define ISNOTEQUALPOS(  
    pp1,  
    pp2 ) ( (pp1).p1 != (pp2).p1 )
```

Definition at line 237 of file [declare.h](#).

4.19.2.78 ISLESSPOS

```
#define ISLESSPOS(  
    pp1,  
    pp2 ) ( (pp1).p1 < (pp2).p1 )
```

Definition at line 238 of file [declare.h](#).

4.19.2.79 ISGEPOS

```
#define ISGEPOS(  
    pp1,  
    pp2 ) ( (pp1).p1 >= (pp2).p1 )
```

Definition at line 239 of file [declare.h](#).

4.19.2.80 ISNOTZEROPOS

```
#define ISNOTZEROPOS(  
    pp ) ( (pp).p1 != 0 )
```

Definition at line 240 of file [declare.h](#).

4.19.2.81 ISZEROPOS

```
#define ISZEROPOS(  
    pp ) ( (pp).p1 == 0 )
```

Definition at line 241 of file [declare.h](#).

4.19.2.82 ISPOSPOS

```
#define ISPOSPOS(  
    pp ) ( (pp).p1 > 0 )
```

Definition at line 242 of file [declare.h](#).

4.19.2.83 ISNEGPOS

```
#define ISNEGPOS(  
    pp ) ( (pp).p1 < 0 )
```

Definition at line 243 of file [declare.h](#).

4.19.2.84 TOLONG

```
#define TOLONG(  
    x ) ((LONG)(x))
```

Definition at line 246 of file [declare.h](#).

4.19.2.85 Add2Com

```
#define Add2Com(  
    x ) { WORD cod[2]; cod[0] = x; cod[1] = 2; AddNtoL(2,cod); }
```

Definition at line 248 of file [declare.h](#).

4.19.2.86 Add3Com

```
#define Add3Com(  
    x1,  
    x2 ) { WORD cod[3]; cod[0] = x1; cod[1] = 3; cod[2] = x2; AddNtoL(3,cod); }
```

Definition at line 249 of file [declare.h](#).

4.19.2.87 Add4Com

```
#define Add4Com(  
    x1,  
    x2,  
    x3 )
```

Value:

```
{ WORD cod[4]; cod[0] = x1; cod[1] = 4; \  
cod[2] = x2; cod[3] = x3; AddNtoL(4,cod); }
```

Definition at line 250 of file [declare.h](#).

4.19.2.88 Add5Com

```
#define Add5Com(  
    x1,  
    x2,  
    x3,  
    x4 )
```

Value:

```
{ WORD cod[5]; cod[0] = x1; cod[1] = 5; \  
cod[2] = x2; cod[3] = x3; cod[4] = x4; AddNtoL(5,cod); }
```

Definition at line 252 of file [declare.h](#).

4.19.2.89 WantAddPointers

```
#define WantAddPointers(
    x )
```

Value:

```
while ( (AT.pWorkPointer+(x))>AR.pWorkSize) {WORD **ppp=&AT.pWorkSpace;\
ExpandBuffer ( (void **)ppp, &AR.pWorkSize, (int) (sizeof (WORD *))) ;}
```

Definition at line 258 of file [declare.h](#).

4.19.2.90 WantAddLongs

```
#define WantAddLongs(
    x )
```

Value:

```
while ( (AT.lWorkPointer+(x))>AR.lWorkSize) {LONG **ppp=&AT.lWorkSpace;\
ExpandBuffer ( (void **)ppp, &AR.lWorkSize, sizeof (LONG)) ;}
```

Definition at line 260 of file [declare.h](#).

4.19.2.91 WantAddPositions

```
#define WantAddPositions(
    x )
```

Value:

```
while ( (AT.posWorkPointer+(x))>AR.posWorkSize) {POSITION **ppp=&AT.posWorkSpace;\
ExpandBuffer ( (void **)ppp, &AR.posWorkSize, sizeof (POSITION)) ;}
```

Definition at line 262 of file [declare.h](#).

4.19.2.92 FORM_INLINE

```
#define FORM_INLINE inline
```

Definition at line 266 of file [declare.h](#).

4.19.2.93 MEMORYMACROS

```
#define MEMORYMACROS
```

Definition at line 274 of file [declare.h](#).

4.19.2.94 TermMalloc

```
#define TermMalloc(
    x ) ( (AT.TermMemTop <= 0 ) ? TermMallocAddMemory (BHEAD0), AT.TermMemHeap[--AT.↵
TermMemTop]: AT.TermMemHeap[--AT.TermMemTop] )
```

Definition at line 278 of file [declare.h](#).

4.19.2.95 NumberMalloc

```
#define NumberMalloc(  
    x ) ( (AT.NumberMemTop <= 0 ) ? NumberMallocAddMemory(BHEAD0), AT.NumberMem↵  
Heap[--AT.NumberMemTop]: AT.NumberMemHeap[--AT.NumberMemTop] )
```

Definition at line 279 of file [declare.h](#).

4.19.2.96 CacheNumberMalloc

```
#define CacheNumberMalloc(  
    x ) ( (AT.CacheNumberMemTop <= 0 ) ? CacheNumberMallocAddMemory(BHEAD0), AT.↵  
CacheNumberMemHeap[--AT.CacheNumberMemTop]: AT.CacheNumberMemHeap[--AT.CacheNumberMemTop] )
```

Definition at line 280 of file [declare.h](#).

4.19.2.97 TermFree

```
#define TermFree(  
    TermMem,  
    x ) AT.TermMemHeap[AT.TermMemTop++] = (WORD *) (TermMem)
```

Definition at line 281 of file [declare.h](#).

4.19.2.98 NumberFree

```
#define NumberFree(  
    NumberMem,  
    x ) AT.NumberMemHeap[AT.NumberMemTop++] = (UWORD *) (NumberMem)
```

Definition at line 282 of file [declare.h](#).

4.19.2.99 CacheNumberFree

```
#define CacheNumberFree(  
    NumberMem,  
    x ) AT.CacheNumberMemHeap[AT.CacheNumberMemTop++] = (UWORD *) (NumberMem)
```

Definition at line 283 of file [declare.h](#).

4.19.2.100 NestingChecksum

```
#define NestingChecksum( ) (AC.IfLevel + AC.RepLevel + AC.arglevel + AC.insidelevel + AC.↵  
termlevel + AC.inexprlevel + AC.dolooplevel + AC.SwitchLevel)
```

Definition at line 311 of file [declare.h](#).

4.19.2.101 MesNesting

```
#define MesNesting( ) MesPrint("&Illegal nesting of if, repeat, argument, inside, term, inexpression  
and do")
```

Definition at line 312 of file [declare.h](#).

4.19.2.102 MarkPolyRatFunDirty

```
#define MarkPolyRatFunDirty(  
    T )
```

Value:

```
{if (*T&&AR.PolyFunType==2) {WORD *TP,*TT;TT=T+*T;TT-=ABS(TT[-1]);\  
TP=T+1;while (TP<TT) {if (*TP==AR.PolyFun) {TP[2]|=(DIRTYFLAG|MUSTCLEANPRF);}TP+=TP[1];}}
```

Definition at line 314 of file [declare.h](#).

4.19.2.103 PUSHPREASSIGNLEVEL

```
#define PUSHPREASSIGNLEVEL
```

Value:

```
AP.PreAssignLevel++; { GETIDENTITY \  
if ( AP.PreAssignLevel >= AP.MaxPreAssignLevel ) { int i; \  
    LONG *ap = (LONG *)Malloca(2*AP.MaxPreAssignLevel*sizeof(LONG *),"PreAssignStack"); \  
    for ( i = 0; i < AP.MaxPreAssignLevel; i++ ) ap[i] = AP.PreAssignStack[i]; \  
    M_free(AP.PreAssignStack,"PreAssignStack"); \  
    AP.MaxPreAssignLevel *= 2; AP.PreAssignStack = ap; \  
} \  
*AT.WorkPointer++ = AP.PreContinuation; AP.PreContinuation = 0; \  
AP.PreAssignStack[AP.PreAssignLevel] = AC.iPointer - AC.iBuffer; }
```

Definition at line 321 of file [declare.h](#).

4.19.2.104 POPPREASSIGNLEVEL

```
#define POPPREASSIGNLEVEL
```

Value:

```
if ( AP.PreAssignLevel > 0 ) { GETIDENTITY \  
AC.iPointer = AC.iBuffer + AP.PreAssignStack[AP.PreAssignLevel--]; \  
AP.PreContinuation = *--AT.WorkPointer; \  
*AC.iPointer = 0; }
```

Definition at line 331 of file [declare.h](#).

4.19.2.105 EXTERNLOCK

```
#define EXTERNLOCK(  
    x )
```

NOTE: We have replaced LOCK(ErrorMessageLock) and UNLOCK(ErrorMessageLock) by MLOCK(ErrorMessageLock) and MUNLOCK(ErrorMessageLock). They are used for the synchronised output in ParFORM. (TU 28 May 2011)

Definition at line 477 of file [declare.h](#).

4.19.2.106 INILOCK

```
#define INILOCK(  
    x )
```

Definition at line 478 of file [declare.h](#).

4.19.2.107 LOCK

```
#define LOCK(  
    x )
```

Definition at line 479 of file [declare.h](#).

4.19.2.108 UNLOCK

```
#define UNLOCK(  
    x )
```

Definition at line 480 of file [declare.h](#).

4.19.2.109 EXTERNRWLOCK

```
#define EXTERNRWLOCK(  
    x )
```

Definition at line 481 of file [declare.h](#).

4.19.2.110 INIRWLOCK

```
#define INIRWLOCK(  
    x )
```

Definition at line 482 of file [declare.h](#).

4.19.2.111 RWLOCKR

```
#define RWLOCKR(  
    x )
```

Definition at line 483 of file [declare.h](#).

4.19.2.112 RWLOCKW

```
#define RWLOCKW(  
    x )
```

Definition at line 484 of file [declare.h](#).

4.19.2.113 UNRWLOCK

```
#define UNRWLOCK(  
    x )
```

Definition at line 485 of file [declare.h](#).

4.19.2.114 MLOCK

```
#define MLOCK(  
    x )
```

Definition at line 490 of file [declare.h](#).

4.19.2.115 MUNLOCK

```
#define MUNLOCK(  
    x )
```

Definition at line 491 of file [declare.h](#).

4.19.2.116 GETIDENTITY

```
#define GETIDENTITY
```

Definition at line 493 of file [declare.h](#).

4.19.2.117 GETBIDENTITY

```
#define GETBIDENTITY
```

Definition at line 494 of file [declare.h](#).

4.19.2.118 M_alloc

```
#define M_alloc(  
    x ) malloc((size_t)(x))
```

Definition at line 1004 of file [declare.h](#).

4.19.2.119 CompareTerms

```
#define CompareTerms ((COMPARE)AR.CompareRoutine)
```

Definition at line 1465 of file [declare.h](#).

4.19.2.120 FiniShuffle

```
#define FiniShuffle AN.SHvar.finishuf
```

Definition at line 1466 of file [declare.h](#).

4.19.2.121 DoShtuffle

```
#define DoShtuffle ((DO_UFFLE)AN.SHvar.do_uffle)
```

Definition at line 1467 of file [declare.h](#).

4.19.3 Typedef Documentation

4.19.3.1 WRITEBUFTOEXTCHANNEL

```
typedef int(* WRITEBUFTOEXTCHANNEL) (char *, size_t)
```

Definition at line 1458 of file [declare.h](#).

4.19.3.2 GETCFROMEXTCHANNEL

```
typedef int(* GETCFROMEXTCHANNEL) (void)
```

Definition at line 1459 of file [declare.h](#).

4.19.3.3 SETTERMINATORFOREXTERNALCHANNEL

```
typedef int(* SETTERMINATORFOREXTERNALCHANNEL) (char *)
```

Definition at line 1460 of file [declare.h](#).

4.19.3.4 SETKILLMODEFOREXTERNALCHANNEL

```
typedef int(* SETKILLMODEFOREXTERNALCHANNEL) (int, int)
```

Definition at line 1461 of file [declare.h](#).

4.19.3.5 WRITEFILE

```
typedef LONG(* WRITEFILE) (int, UBYTE *, LONG)
```

Definition at line 1462 of file [declare.h](#).

4.19.3.6 GETTERM

```
typedef WORD(* GETTERM) (PHEAD WORD *)
```

Definition at line 1463 of file [declare.h](#).

4.19.4 Function Documentation

4.19.4.1 TELLFILE()

```
void TELLFILE (
    int handle,
    POSITION * position ) [extern]
```

Definition at line 1408 of file [tools.c](#).

4.19.4.2 StartVariables()

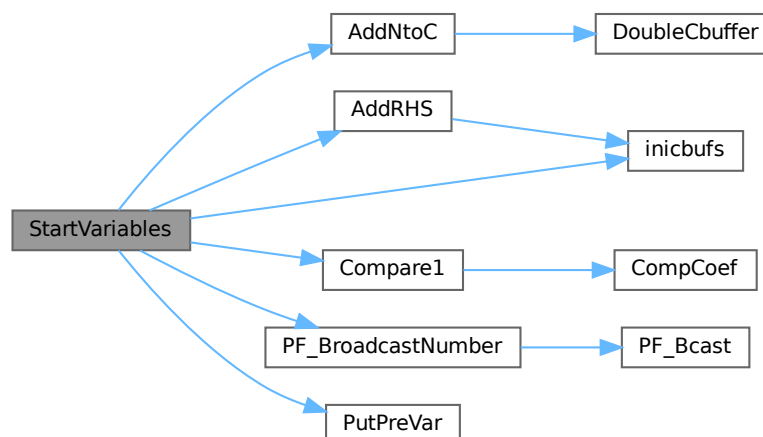
```
void StartVariables (
    void ) [extern]
```

All functions (well, nearly all) are declared here.

Definition at line 905 of file [startup.c](#).

References [AddNtoC\(\)](#), [AddRHS\(\)](#), [Compare1\(\)](#), [inicbufs\(\)](#), [FuNcTiOn::name](#), [PF_BroadcastNumber\(\)](#), [PutPreVar\(\)](#), and [FuNcTiOn::symmetric](#).

Here is the call graph for this function:



4.19.4.3 CodeToLine()

```

UBYTE * CodeToLine (
    WORD number,
    UBYTE * Out ) [extern]

```

Definition at line 640 of file [sch.c](#).

4.19.4.4 AddArrayIndex()

```

UBYTE * AddArrayIndex (
    WORD num,
    UBYTE * out ) [extern]

```

Definition at line 678 of file [sch.c](#).

4.19.4.5 FindInIndex()

```

INDEXENTRY * FindInIndex (
    WORD expr,
    FILEDATA * f,
    WORD par,
    WORD mode ) [extern]

```

Definition at line 2664 of file [store.c](#).

4.19.4.6 NextFileIndex()

```

INDEXENTRY * NextFileIndex (
    POSITION * indexpos ) [extern]

```

Definition at line 2257 of file [store.c](#).

4.19.4.7 PasteTerm()

```

WORD * PasteTerm (
    PHEAD WORD number,
    WORD * accum,
    WORD * position,
    WORD times,
    WORD divby ) [extern]

```

Puts the term at position in the accumulator *accum* at position '*number*+1'. if *times* > 0 the coefficient of this term is multiplied by *times*/*divby*.

Parameters

<i>number</i>	The number of term fragments in <i>accum</i> that should be skipped
<i>accum</i>	The accumulator of term fragments
<i>position</i>	A position in (typically) a compiler buffer from where a (piece of a) term comes.
<i>times</i>	Multiply the result by this
<i>divby</i>	Divide the result by this.

This routine is typically used when we have to replace a (sub)expression pointer by a power of a (sub)expression. This uses mostly a binomial expansion and the new term is the old term multiplied one by one by terms of the new expression. The factors times and divby keep track of the binomial coefficient. Once this is complete, the routine FiniTerm will make the contents of the accumulator into a proper term that still needs to be normalized.

Definition at line 2988 of file [proces.c](#).

Referenced by [Generator\(\)](#).

4.19.4.8 StrCopy()

```
UBYTE * StrCopy (
    UBYTE * from,
    UBYTE * to ) [extern]
```

Definition at line 49 of file [sch.c](#).

4.19.4.9 WrtPower()

```
UBYTE * WrtPower (
    UBYTE * Out,
    WORD Power ) [extern]
```

Definition at line 720 of file [sch.c](#).

4.19.4.10 AccumGCD()

```
int AccumGCD (
    PHEAD UWORD * a,
    WORD * na,
    UWORD * b,
    WORD nb ) [extern]
```

Definition at line 648 of file [reken.c](#).

4.19.4.11 AddArgs()

```
void AddArgs (
    PHEAD WORD * s1,
    WORD * s2,
    WORD * m ) [extern]
```

Adds the arguments of two occurrences of the PolyFun.

Parameters

<i>s1</i>	Pointer to the first occurrence.
<i>s2</i>	Pointer to the second occurrence.
<i>m</i>	Pointer to where the answer should be.

Definition at line 2347 of file [sort.c](#).

Referenced by [AddPoly\(\)](#), and [MergePatches\(\)](#).

4.19.4.12 AddCoef()

```
int AddCoef (  
    PHEAD WORD ** ps1,  
    WORD ** ps2 ) [extern]
```

Adds the coefficients of the terms *ps1 and *ps2. The problem comes when there is not enough space for a new longer coefficient. First a local solution is tried. If this is not successful we need to move terms around. The possibility of a garbage collection should not be ignored, as avoiding this costs very much extra space which is nearly wasted otherwise.

If the return value is zero the terms cancelled.

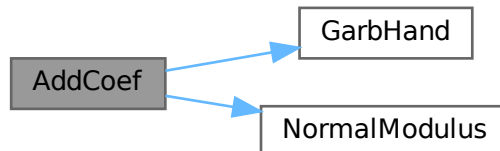
The resulting term is left in *ps1.

Definition at line 2046 of file [sort.c](#).

References [GarbHand\(\)](#), and [NormalModulus\(\)](#).

Referenced by [SplitMerge\(\)](#).

Here is the call graph for this function:



4.19.4.13 AddLong()

```
int AddLong (  
    UWORD * a,  
    WORD na,  
    UWORD * b,  
    WORD nb,  
    UWORD * c,  
    WORD * nc ) [extern]
```

Definition at line 706 of file [reken.c](#).

4.19.4.14 AddPLon()

```
int AddPLon (
    UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc ) [extern]
```

Definition at line 751 of file [reken.c](#).

4.19.4.15 AddPoly()

```
int AddPoly (
    PHEAD WORD ** ps1,
    WORD ** ps2 ) [extern]
```

Routine should be called when $S \rightarrow \text{PolyWise} \neq 0$. It points then to the position of AR.PolyFun in both terms.

We add the contents of the arguments of the two polynomials. Special attention has to be given to special arguments. We have to reserve a space equal to the size of one term + the size of the argument of the other. The addition has to be done in this routine because not all objects are reentrant.

Newer addition (12-nov-2007). The PolyFun can have two arguments. In that case $S \rightarrow \text{PolyFlag}$ is 2 and we have to call the routine for adding rational polynomials. We have to be rather careful what happens with: The location of the output The order of the terms in the arguments At first we allow only univariate polynomials in the PolyFun. This restriction will be lifted a.s.a.p.

Parameters

<i>ps1</i>	A pointer to the position of the first term
<i>ps2</i>	A pointer to the position of the second term

Returns

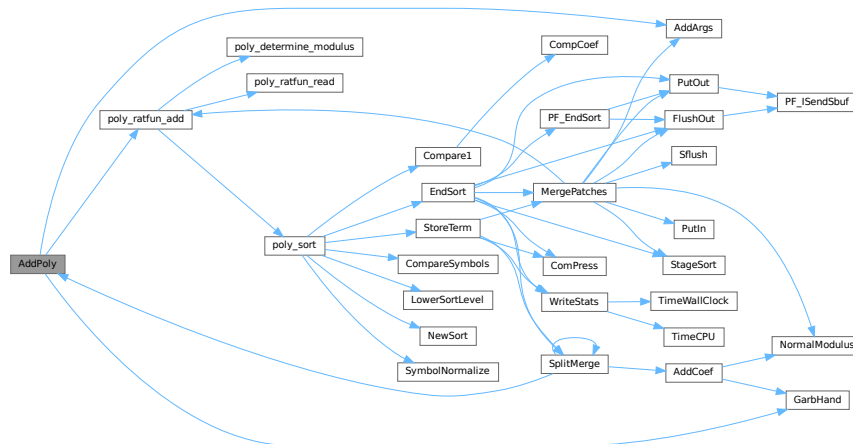
If zero the terms cancel. Otherwise the new term is in *ps1.

Definition at line 2175 of file [sort.c](#).

References [AddArgs\(\)](#), [GarbHand\(\)](#), and [poly_ratfun_add\(\)](#).

Referenced by [SplitMerge\(\)](#).

Here is the call graph for this function:



4.19.4.16 AddRat()

```

int AddRat (
    PHEAD UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc ) [extern]
  
```

Definition at line 210 of file [reken.c](#).

4.19.4.17 AddToLine()

```

void AddToLine (
    UBYTE * s ) [extern]
  
```

Definition at line 64 of file [sch.c](#).

4.19.4.18 AddWild()

```

int AddWild (
    PHEAD WORD,
    WORD type,
    WORD newnumber ) [extern]
  
```

Definition at line 1544 of file [wildcard.c](#).

4.19.4.19 BigLong()

```
int BigLong (
    UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb ) [extern]
```

Definition at line 945 of file [reken.c](#).

4.19.4.20 BinomGen()

```
int BinomGen (
    PHEAD WORD * term,
    WORD level,
    WORD ** tstops,
    WORD x1,
    WORD x2,
    WORD pow1,
    WORD pow2,
    WORD sign,
    UWORD * coef,
    WORD ncoef ) [extern]
```

Definition at line 320 of file [ratio.c](#).

4.19.4.21 CheckWild()

```
int CheckWild (
    PHEAD WORD,
    WORD type,
    WORD newnumber,
    WORD * newval ) [extern]
```

Definition at line 1791 of file [wildcard.c](#).

4.19.4.22 Chisholm()

```
int Chisholm (
    PHEAD WORD * term,
    WORD level ) [extern]
```

Definition at line 1966 of file [opera.c](#).

4.19.4.23 CleanExpr()

```
int CleanExpr (
    WORD par ) [extern]
```

Definition at line 47 of file [execute.c](#).

4.19.4.24 CleanUp()

```
void CleanUp (  
    WORD par ) [extern]
```

Definition at line 1780 of file [startup.c](#).

4.19.4.25 ClearWild()

```
void ClearWild (  
    PHEAD0 ) [extern]
```

Definition at line 1518 of file [wildcard.c](#).

4.19.4.26 CompareFunctions()

```
int CompareFunctions (  
    WORD * fleft,  
    WORD * fright ) [extern]
```

Definition at line 54 of file [normal.c](#).

4.19.4.27 Commute()

```
int Commute (  
    WORD * fleft,  
    WORD * fright ) [extern]
```

Definition at line 118 of file [normal.c](#).

4.19.4.28 DetCommu()

```
WORD DetCommu (  
    WORD * terms ) [extern]
```

Definition at line 4530 of file [normal.c](#).

4.19.4.29 DoesCommu()

```
WORD DoesCommu (  
    WORD * term ) [extern]
```

Definition at line 4589 of file [normal.c](#).

4.19.4.30 CompArg()

```
int CompArg (
    WORD * s1,
    WORD * s2 ) [extern]
```

Definition at line 3243 of file [tools.c](#).

4.19.4.31 CompCoef()

```
WORD CompCoef (
    WORD * term1,
    WORD * term2 ) [extern]
```

Routine takes $a_1 \bmod m_1$ and $a_2 \bmod m_2$ and returns $a \bmod m_1*m_2$ with
 $a \bmod m_1 = a_1$ and $a \bmod m_2 = a_2$

Chinese remainder: $a \bmod (m_1*m_2) = q_1*m_1 + a_1$ $a \bmod (m_1*m_2) = q_2*m_2 + a_2$ Compute n_1 such that $(n_1*m_1) \bmod m_2$ is one
 Compute n_2 such that $(n_2*m_2) \bmod m_1$ is one Then $(a_1*n_2*m_2 + a_2*n_1*m_1) \bmod (m_1*m_2)$ is $a \bmod (m_1*m_2)$

Definition at line 3048 of file [reken.c](#).

Referenced by [Compare1\(\)](#).

4.19.4.32 CompGroup()

```
int CompGroup (
    PHEAD WORD,
    WORD ** args,
    WORD * a1,
    WORD * a2,
    WORD num ) [extern]
```

Definition at line 373 of file [function.c](#).

4.19.4.33 Compare1()

```
WORD Compare1 (
    PHEAD WORD * term1,
    WORD * term2,
    WORD level ) [extern]
```

Compares two terms. The answer is: 0 equal (with exception of the coefficient if level == 0.) >0 term1 comes first.
 <0 term2 comes first. Some special precautions may be needed to keep the CompCoef routine from generating overflows, although this is very unlikely in subterms. This routine should not return an error condition.

Originally this routine was called Compare. With the treatment of special polynomials with terms that contain only symbols and the need for extreme speed for the polynomial routines we made a special compare routine and now we store the address of the current compare routine in AR.CompareRoutine and have a macro Compare which makes all existing code work properly and we can just replace the routine on a thread by thread basis (each thread has its own AR struct).

Parameters

<i>term1</i>	First input term
<i>term2</i>	Second input term
<i>level</i>	The sorting level (may influence on the result)

Returns

0 equal (with exception of the coefficient if level == 0.) >0 term1 comes first. <0 term2 comes first.

When there are floating point numbers active (float_ = FLOATFUN) the presence of one or more float_ functions is returned in AT.SortFloatMode: 0: no float_ 1: float_ in term1 only 2: float_ in term2 only 3: float_ in both terms

Definition at line 2640 of file [sort.c](#).

References [CompCoef\(\)](#).

Referenced by [InFunction\(\)](#), [poly_sort\(\)](#), and [StartVariables\(\)](#).

Here is the call graph for this function:



4.19.4.34 CountDo()

```
WORD CountDo (
    WORD * term,
    WORD * instruct ) [extern]
```

Definition at line 552 of file [reshuf.c](#).

4.19.4.35 CountFun()

```
WORD CountFun (
    WORD * term,
    WORD * countfun ) [extern]
```

Definition at line 706 of file [reshuf.c](#).

4.19.4.36 DimensionSubterm()

```
WORD DimensionSubterm (
    WORD * subterm ) [extern]
```

Definition at line 867 of file [reshuf.c](#).

4.19.4.37 DimensionTerm()

```
WORD DimensionTerm (
    WORD * term ) [extern]
```

Definition at line [968](#) of file [reshuf.c](#).

4.19.4.38 DimensionExpression()

```
WORD DimensionExpression (
    PHEAD WORD * expr ) [extern]
```

Definition at line [1000](#) of file [reshuf.c](#).

4.19.4.39 Deferred()

```
int Deferred (
    PHEAD WORD * term,
    WORD level ) [extern]
```

Picks up the deferred brackets. These are the bracket contents of which we postpone the reading when we use the 'Keep Brackets' statement. These contents are multiplying the terms just before they are sent to the sorting system. Special attention goes to having it thread-safe We have to lock positioning the file and reading it in a thread specific buffer.

Parameters

<i>term</i>	The term that must be multiplied by the contents of the current bracket
<i>level</i>	The compiler level. This is needed because after multiplying term by term we call Generator again.

Definition at line [4840](#) of file [proces.c](#).

References [Generator\(\)](#), and [InsertTerm\(\)](#).

Referenced by [Generator\(\)](#).

```
int DeleteStore (
                WORD par ) [extern]
```

```
WORD DetCurDum (
    PHEAD WORD * t ) [extern]
```

```
void DetVars (
    WORD * term,
    WORD par ) [extern]
```

Generated by Doxygen

4.19.4.43 Distribute()

```
int Distribute (
    DISTRIBUTE * d,
    WORD first ) [extern]
```

Definition at line 510 of file [symmetr.c](#).

4.19.4.44 DivLong()

```
int DivLong (
    UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc,
    UWORD * d,
    WORD * nd ) [extern]
```

Definition at line 973 of file [reken.c](#).

4.19.4.45 DivRat()

```
int DivRat (
    PHEAD UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc ) [extern]
```

Definition at line 500 of file [reken.c](#).

4.19.4.46 Divvy()

```
int Divvy (
    PHEAD UWORD * a,
    WORD * na,
    UWORD * b,
    WORD nb ) [extern]
```

Definition at line 172 of file [reken.c](#).

4.19.4.47 DoDelta()

```
int DoDelta (
    WORD * t ) [extern]
```

Definition at line 4375 of file [normal.c](#).

4.19.4.48 DoDelta3()

```
int DoDelta3 (
    PHEAD WORD * term,
    WORD level ) [extern]
```

Definition at line 1405 of file [reshuf.c](#).

4.19.4.49 TestPartitions()

```
int TestPartitions (
    WORD * tfun,
    PARTI * parti ) [extern]
```

Definition at line 1607 of file [reshuf.c](#).

4.19.4.50 DoPartitions()

```
int DoPartitions (
    PHEAD WORD * term,
    WORD level ) [extern]
```

Definition at line 1722 of file [reshuf.c](#).

4.19.4.51 CoCanonicalize()

```
int CoCanonicalize (
    UBYTE * s ) [extern]
```

Definition at line 64 of file [diagrams.c](#).

4.19.4.52 DoCanonicalize()

```
int DoCanonicalize (
    PHEAD WORD * term,
    WORD * params ) [extern]
```

Definition at line 185 of file [diagrams.c](#).

4.19.4.53 GenDiagrams()

```
int GenDiagrams (
    PHEAD WORD * term,
    WORD level ) [extern]
```

Definition at line 805 of file [diawrap.cc](#).

4.19.4.54 DoTopologyCanonicalize()

```
int DoTopologyCanonicalize (
    PHEAD WORD * term,
    WORD vert,
    WORD edge,
    WORD * args ) [extern]
```

Definition at line [346](#) of file [diagrams.c](#).

4.19.4.55 DoShattering()

```
int DoShattering (
    PHEAD WORD * connect,
    WORD * environ,
    WORD * partitions,
    WORD nvert ) [extern]
```

Definition at line [555](#) of file [diagrams.c](#).

4.19.4.56 DoTableExpansion()

```
int DoTableExpansion (
    WORD * term,
    WORD level ) [extern]
```

Definition at line [334](#) of file [tables.c](#).

4.19.4.57 DoDistrib()

```
int DoDistrib (
    PHEAD WORD * term,
    WORD level ) [extern]
```

Definition at line [1119](#) of file [reshuf.c](#).

4.19.4.58 DoShuffle()

```
int DoShuffle (
    WORD * term,
    WORD level,
    WORD fun,
    WORD option ) [extern]
```

Definition at line [2095](#) of file [reshuf.c](#).

4.19.4.59 DoPermutations()

```
int DoPermutations (
    PHEAD WORD * term,
    WORD level ) [extern]
```

Definition at line 2007 of file [reshuf.c](#).

4.19.4.60 Shuffle()

```
int Shuffle (
    WORD * from1,
    WORD * from2,
    WORD * to ) [extern]
```

Definition at line 2229 of file [reshuf.c](#).

4.19.4.61 FinishShuffle()

```
int FinishShuffle (
    WORD * fini ) [extern]
```

Definition at line 2421 of file [reshuf.c](#).

4.19.4.62 DoStuffle()

```
int DoStuffle (
    WORD * term,
    WORD level,
    WORD fun,
    WORD option ) [extern]
```

Definition at line 2487 of file [reshuf.c](#).

4.19.4.63 Stuffle()

```
int Stuffle (
    WORD * from1,
    WORD * from2,
    WORD * to ) [extern]
```

Definition at line 2702 of file [reshuf.c](#).

4.19.4.64 FinishStuffle()

```
int FinishStuffle (
    WORD * fini ) [extern]
```

Definition at line 2829 of file [reshuf.c](#).

4.19.4.65 StuffRootAdd()

```
WORD * StuffRootAdd (  
    WORD * t1,  
    WORD * t2,  
    WORD * to ) [extern]
```

Definition at line 2876 of file [reshuf.c](#).

4.19.4.66 TestUse()

```
int TestUse (  
    WORD * term,  
    WORD level ) [extern]
```

Definition at line 1414 of file [tables.c](#).

4.19.4.67 FindTB()

```
DBASE * FindTB (  
    UBYTE * name ) [extern]
```

Definition at line 734 of file [tables.c](#).

4.19.4.68 CheckTableDeclarations()

```
int CheckTableDeclarations (  
    DBASE * d ) [extern]
```

Definition at line 1178 of file [tables.c](#).

4.19.4.69 Apply()

```
void Apply (  
    WORD * term,  
    WORD level ) [extern]
```

Definition at line 1981 of file [tables.c](#).

4.19.4.70 ApplyExec()

```
int ApplyExec (  
    WORD * term,  
    int maxtogo,  
    WORD level ) [extern]
```

Definition at line 2039 of file [tables.c](#).

4.19.4.71 ApplyReset()

```
void ApplyReset (
    WORD level ) [extern]
```

Definition at line 2256 of file [tables.c](#).

4.19.4.72 TableReset()

```
void TableReset (
    void ) [extern]
```

Definition at line 2291 of file [tables.c](#).

4.19.4.73 ReWorkT()

```
void ReWorkT (
    WORD * term,
    WORD * funcs,
    WORD numfuncs ) [extern]
```

Definition at line 1900 of file [tables.c](#).

4.19.4.74 GetIfDollarNum()

```
WORD GetIfDollarNum (
    WORD * ifp,
    WORD * ifstop ) [extern]
```

Definition at line 106 of file [if.c](#).

4.19.4.75 FindVar()

```
int FindVar (
    WORD * v,
    WORD * term ) [extern]
```

Definition at line 173 of file [if.c](#).

4.19.4.76 DolfStatement()

```
int DoIfStatement (
    PHEAD WORD * ifcode,
    WORD * term ) [extern]
```

Definition at line 280 of file [if.c](#).

4.19.4.77 DoOnePow()

```
int DoOnePow (
    PHEAD WORD * term,
    WORD power,
    WORD nexp,
    WORD * accum,
    WORD * aa,
    WORD level,
    WORD * freeze ) [extern]
```

Routine gets one power of an expression in the scratch system. If there are more powers needed there will be a recursion.

No attempt is made to use binomials because we have no information about commuting properties.

There is a searching for the contents of brackets if needed. This searching may be rather slow because of the single links.

Parameters

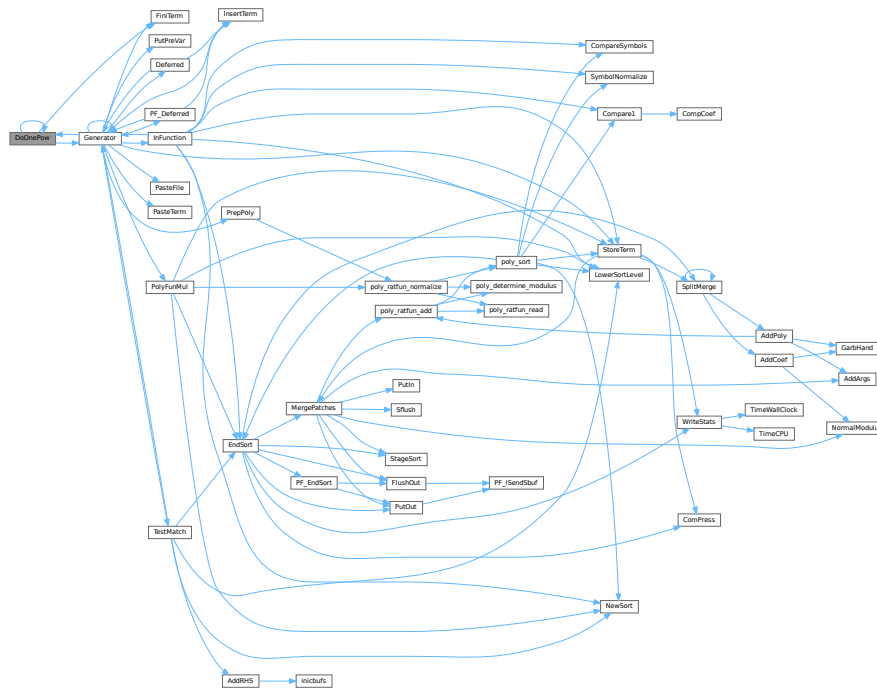
<i>term</i>	is the term we are adding to.
<i>power</i>	is the power of the expression that we need.
<i>nexp</i>	is the number of the expression.
<i>accum</i>	is the accumulator of terms. It accepts the termfragments that are made into a proper term in FiniTerm
<i>aa</i>	points to the start of the entire accumulator. In *aa we store the number of term fragments that are in the accumulator.
<i>level</i>	is the current depth in the tree of statements. It is needed to continue to the next operation/substitution with each generated term
<i>freeze</i>	is the pointer to the bracket information that should be matched.

Definition at line 4619 of file [proces.c](#).

References [DoOnePow\(\)](#), [FiniTerm\(\)](#), [Generator\(\)](#), and [FiLe::handle](#).

Referenced by [DoOnePow\(\)](#), and [Generator\(\)](#).

Here is the call graph for this function:



4.19.4.78 DoRevert()

```
void DoRevert (
    WORD * fun,
    WORD * tmp ) [extern]
```

Definition at line 4442 of file [normal.c](#).

4.19.4.79 DoSumF1()

```
int DoSumF1 (
    PHEAD WORD * term,
    WORD * params,
    WORD * replac,
    WORD level ) [extern]
```

Definition at line 395 of file [ratio.c](#).

4.19.4.80 DoSumF2()

```
int DoSumF2 (
    PHEAD WORD * term,
    WORD * params,
    WORD * replac,
    WORD level ) [extern]
```

Definition at line 520 of file [ratio.c](#).

4.19.4.81 DoTheta()

```
int DoTheta (
    PHEAD WORD * t ) [extern]
```

Definition at line 4278 of file [normal.c](#).

4.19.4.82 EndSort()

```
LONG EndSort (
    PHEAD WORD * buffer,
    int par ) [extern]
```

Finishes a sort. At AR.sLevel == 0 the output is to the regular output stream. When AR.sLevel > 0, the parameter par determines the actual output. The AR.sLevel will be popped. All ongoing stages are finished and if the sortfile is open it is closed. The statistics are printed when AR.sLevel == 0 par == 0 [Output](#) to the buffer. par == 1 Sort for function arguments. The output will be copied into the buffer. It is assumed that this is in the WorkSpace. par == 2 Sort for \$-variable. We return the address of the buffer that contains the output in buffer (treated like WORD **). We first catch the output in a file (unless we can intercept things after the small buffer has been sorted) Then we read from the file into a buffer. Only when par == 0 data compression can be attempted at AT.SS==AT.S0.

Parameters

<i>buffer</i>	buffer for output when needed
<i>par</i>	See above

Returns

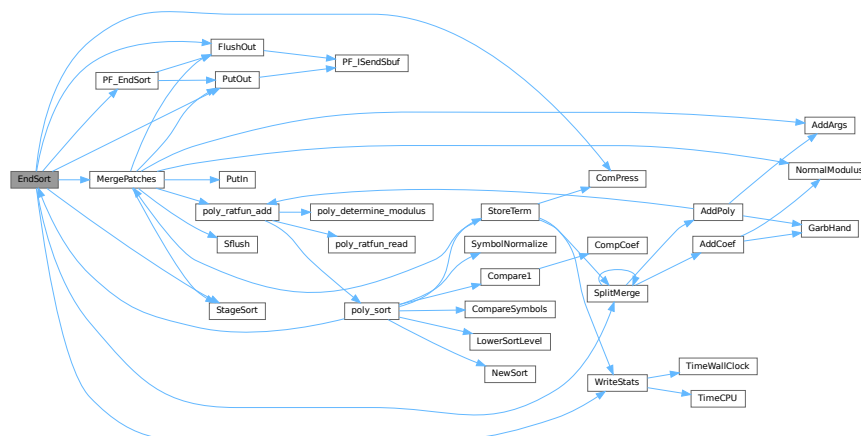
If negative: error. If positive: number of words in output.

Definition at line 745 of file [sort.c](#).

References [ComPress\(\)](#), [FlushOut\(\)](#), [MergePatches\(\)](#), [PF_EndSort\(\)](#), [PutOut\(\)](#), [SplitMerge\(\)](#), [StageSort\(\)](#), and [WriteStats\(\)](#).

Referenced by [generate_expression\(\)](#), [get_expression\(\)](#), [InFunction\(\)](#), [MakeDollarInteger\(\)](#), [MakeDollarMod\(\)](#), [PF_CollectModifiedDollars\(\)](#), [PF_Processor\(\)](#), [poly_factorize_expression\(\)](#), [poly_sort\(\)](#), [poly_unfactorize_expression\(\)](#), [PolyFunMul\(\)](#), [Processor\(\)](#), [TakeArgContent\(\)](#), and [TestMatch\(\)](#).

Here is the call graph for this function:



4.19.4.83 EntVar()

```
int EntVar (
    WORD type,
    UBYTE * name,
    WORD x,
    WORD y,
    WORD z,
    WORD d ) [extern]
```

Definition at line 557 of file [names.c](#).

4.19.4.84 EpfCon()

```
int EpfCon (
    PHEAD WORD * term,
    WORD * params,
    WORD num,
    WORD level ) [extern]
```

Definition at line 212 of file [opera.c](#).

4.19.4.85 EpfFind()

```
int EpfFind (
    PHEAD WORD * term,
    WORD * params ) [extern]
```

Definition at line 58 of file [opera.c](#).

4.19.4.86 EpfGen()

```
WORD EpfGen (
    WORD number,
    WORD * inlist,
    WORD * kron,
    WORD * perm,
    WORD sgn ) [extern]
```

Definition at line 282 of file [opera.c](#).

4.19.4.87 EqualArg()

```
int EqualArg (
    WORD * parms,
    WORD num1,
    WORD num2 ) [extern]
```

Definition at line 1379 of file [reshuf.c](#).

4.19.4.88 Factorial()

```
int Factorial (
    PHEAD WORD,
    UWORD * a,
    WORD * na ) [extern]
```

Definition at line [3450](#) of file [reken.c](#).

4.19.4.89 Bernoulli()

```
int Bernoulli (
    WORD n,
    UWORD * a,
    WORD * na ) [extern]
```

Definition at line [3530](#) of file [reken.c](#).

4.19.4.90 FactorIn()

```
int FactorIn (
    PHEAD WORD * term,
    WORD level ) [extern]
```

Definition at line [46](#) of file [factor.c](#).

4.19.4.91 FactorInExpr()

```
int FactorInExpr (
    PHEAD WORD * term,
    WORD level ) [extern]
```

Definition at line [421](#) of file [factor.c](#).

4.19.4.92 FindAll()

```
int FindAll (
    PHEAD WORD * term,
    WORD * pattern,
    WORD level,
    WORD * par ) [extern]
```

Definition at line [1197](#) of file [pattern.c](#).

4.19.4.93 FindMulti()

```
WORD FindMulti (
    PHEAD WORD * term,
    WORD * pattern ) [extern]
```

Definition at line [954](#) of file [findpat.c](#).

4.19.4.94 FindOnce()

```
int FindOnce (
    PHEAD WORD * term,
    WORD * pattern ) [extern]
```

Definition at line 419 of file [findpat.c](#).

4.19.4.95 FindOnly()

```
int FindOnly (
    PHEAD WORD * term,
    WORD * pattern ) [extern]
```

Definition at line 61 of file [findpat.c](#).

4.19.4.96 FindRest()

```
int FindRest (
    PHEAD WORD * term,
    WORD * pattern ) [extern]
```

Definition at line 1070 of file [findpat.c](#).

4.19.4.97 FindrNumber()

```
WORD FindrNumber (
    WORD n,
    VARRENUM * v ) [extern]
```

Definition at line 2587 of file [store.c](#).

4.19.4.98 FiniLine()

```
void FiniLine (
    void ) [extern]
```

Definition at line 164 of file [sch.c](#).

4.19.4.99 FiniTerm()

```
int FiniTerm (
    PHEAD WORD * term,
    WORD * accum,
    WORD * termout,
    WORD number,
    WORD tepos ) [extern]
```

Concatenates the contents of the accumulator into a single legal term, which replaces the subexpression pointer

Parameters

<i>term</i>	the input term with the (sub)expression subterm
<i>accum</i>	the accumulator with the term fragments
<i>termout</i>	the location where the output should be written
<i>number</i>	the number of term fragments in the accumulator
<i>tepos</i>	the position of the subterm in term to be replaced

Definition at line 3053 of file [proces.c](#).

Referenced by [DoOnePow\(\)](#), and [Generator\(\)](#).

4.19.4.100 FlushOut()

```
int FlushOut (
    POSITION * position,
    FILEHANDLE * fi,
    int compr ) [extern]
```

Completes output to an output file and writes the trailing zero.

Parameters

<i>position</i>	The position in the file after writing
<i>fi</i>	The file (or its cache)
<i>compr</i>	Indicates whether there should be compression with gzip.

Returns

Regular conventions (OK -> 0).

Definition at line 1832 of file [sort.c](#).

References [FiLe::handle](#), [BrAcKeTiNfO::indexbuffer](#), and [PF_ISendSbuf\(\)](#).

Referenced by [EndSort\(\)](#), [MergePatches\(\)](#), [PF_EndSort\(\)](#), and [Processor\(\)](#).

Here is the call graph for this function:



4.19.4.101 FunLevel()

```
void FunLevel (
    PHEAD WORD * term ) [extern]
```

Definition at line 130 of file [reshuf.c](#).

4.19.4.102 AdjustRenumScratch()

```
void AdjustRenumScratch (
    PHEAD0 ) [extern]
```

Definition at line 506 of file [reshuf.c](#).

4.19.4.103 GarbHand()

```
void GarbHand (
    void ) [extern]
```

Garbage collection that takes place when the small extension is full and we need to place more terms there. When this is the case there are many holes in the small buffer and the whole can be compactified. The major complication is the buffer for SplitMerge. There are two options for temporary memory: 1: find some buffer that has enough space (maybe in the large buffer). 2: allocate a buffer. Give it back afterwards of course. If the small extension is properly dimensioned this routine should be called very rarely. Most of the time it will be called when the polyfun or polyratfun is active.

Definition at line 3652 of file [sort.c](#).

Referenced by [AddCoef\(\)](#), and [AddPoly\(\)](#).

4.19.4.104 GcdLong()

```
int GcdLong (
    PHEAD UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc ) [extern]
```

Definition at line 2277 of file [reken.c](#).

4.19.4.105 LcmLong()

```
int LcmLong (
    PHEAD UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc ) [extern]
```

Definition at line 2702 of file [reken.c](#).

4.19.4.106 Generator()

```
int Generator (
    PHEAD WORD * term,
    WORD level ) [extern]
```

The heart of the program. Here the expansion tree is set up in one giant recursion

Parameters

<i>term</i>	the input term. may be overwritten
<i>level</i>	the level in the compiler buffer (number of statement)

Returns

Normal conventions (OK = 0).

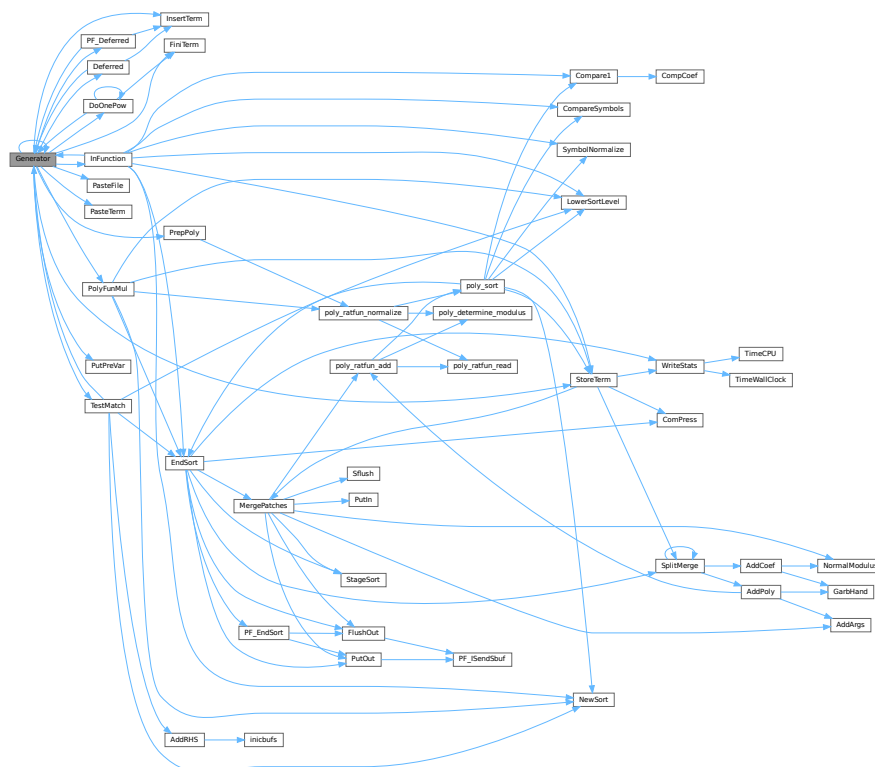
The routine looks first whether there are unsubstituted (sub)expressions. If so, one of them gets inserted term by term and the new term is used in a renewed call to Generator. If there are no (sub)expressions, the term is normalized, the compiler level is raised (next statement) and the program looks what type of statement this is. If this is a special statement it is either treated on the spot or the appropriate routine is called. If it is a substitution, the pattern matcher is called (TestMatch) which tells whether there was a match. If so we need to call TestSub again to test for (sub)expressions. If we run out of levels, the term receives a final treatment for modulus calculus and/or brackets and is then sent off to the sorting routines.

Definition at line 3252 of file [proces.c](#).

References [Deferred\(\)](#), [DoOnePow\(\)](#), [FiniTerm\(\)](#), [Generator\(\)](#), [InFunction\(\)](#), [InsertTerm\(\)](#), [CbUf::lhs](#), [VaRrEnUm::lo](#), [PasteFile\(\)](#), [PasteTerm\(\)](#), [PF_Deferred\(\)](#), [CbUf::Pointer](#), [PolyFunMul\(\)](#), [PrepPoly\(\)](#), [PutPreVar\(\)](#), [CbUf::rhs](#), [StoreTerm\(\)](#), [ReNuMbEr::symb](#), and [TestMatch\(\)](#).

Referenced by [Deferred\(\)](#), [DoOnePow\(\)](#), [generate_expression\(\)](#), [Generator\(\)](#), [InFunction\(\)](#), [MakeDollarInteger\(\)](#), [MakeDollarMod\(\)](#), [PF_CollectModifiedDollars\(\)](#), [PF_Deferred\(\)](#), [PF_Processor\(\)](#), [poly_factorize_expression\(\)](#), [poly_unfactorize_expression\(\)](#), [Processor\(\)](#), and [TestMatch\(\)](#).

Here is the call graph for this function:



4.19.4.107 GetBinom()

```
int GetBinom (
    UWORD * a,
    WORD * na,
    WORD i1,
    WORD i2 ) [extern]
```

Definition at line [2668](#) of file [reken.c](#).

4.19.4.108 GetFromStore()

```
WORD GetFromStore (
    WORD * to,
    POSITION * position,
    RENUMBER renumber,
    WORD * InCompState,
    WORD nexpr ) [extern]
```

Definition at line [1628](#) of file [store.c](#).

4.19.4.109 GetLong()

```
int GetLong (
    UBYTE * s,
    UWORD * a,
    WORD * na ) [extern]
```

Definition at line [1775](#) of file [reken.c](#).

4.19.4.110 GetMoreTerms()

```
WORD GetMoreTerms (
    WORD * term ) [extern]
```

Definition at line [1425](#) of file [store.c](#).

4.19.4.111 GetMoreFromMem()

```
int GetMoreFromMem (
    WORD * term,
    WORD ** tpoin ) [extern]
```

Definition at line [1521](#) of file [store.c](#).

4.19.4.112 GetOneTerm()

```
WORD GetOneTerm (
    PHEAD WORD * term,
    FILEHANDLE * fi,
    POSITION * pos,
    int par ) [extern]
```

Definition at line 1267 of file [store.c](#).

4.19.4.113 GetTable()

```
RENUMBER GetTable (
    WORD expr,
    POSITION * position,
    WORD mode ) [extern]
```

Definition at line 2825 of file [store.c](#).

4.19.4.114 GetTerm()

```
WORD GetTerm (
    PHEAD WORD * term ) [extern]
```

Definition at line 992 of file [store.c](#).

4.19.4.115 Glue()

```
int Glue (
    PHEAD WORD * term1,
    WORD * term2,
    WORD * sub,
    WORD insert ) [extern]
```

Definition at line 461 of file [ratio.c](#).

4.19.4.116 InFunction()

```
int InFunction (
    PHEAD WORD * term,
    WORD * termout ) [extern]
```

Makes the replacement of the subexpression with the number 'replac' in a function argument. Additional information is passed in some of the AR, AN, AT variables.

Parameters

<i>term</i>	The input term
<i>termout</i>	The output term

Returns

0: everything is fine, Negative: fatal, Positive: error.

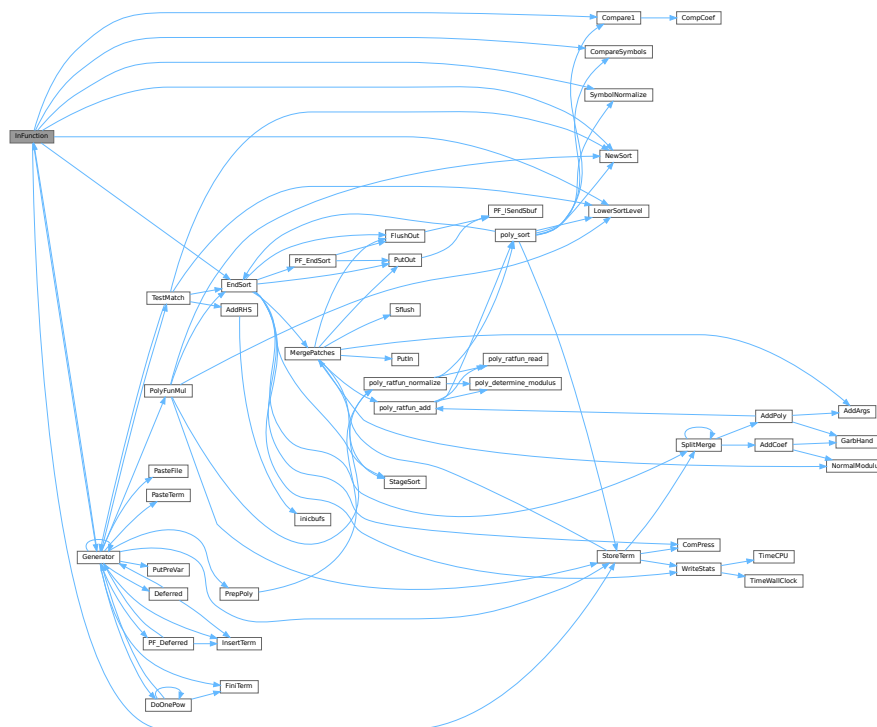
Special attention should be given to nested functions!

Definition at line 2160 of file [proces.c](#).

References [Compare1\(\)](#), [CompareSymbols\(\)](#), [EndSort\(\)](#), [Generator\(\)](#), [LowerSortLevel\(\)](#), [NewSort\(\)](#), [StoreTerm\(\)](#), and [SymbolNormalize\(\)](#).

Referenced by [Generator\(\)](#).

Here is the call graph for this function:



4.19.4.117 IniLine()

```
void IniLine (
    WORD extrablank ) [extern]
```

Definition at line 249 of file [sch.c](#).

4.19.4.118 IniVars()

```
void IniVars (
    void ) [extern]
```

Definition at line 1371 of file [startup.c](#).

4.19.4.119 InsertTerm()

```
int InsertTerm (
    PHEAD WORD * term,
    WORD replac,
    WORD extractbuff,
    WORD * position,
    WORD * termout,
    WORD tepos ) [extern]
```

Puts the terms 'term' and 'position' together into a single legal term in termout. replac is the number of the subexpression that should be replaced. It must be a positive term. When action is needed in the argument of a function all terms in that argument are dealt with recursively. The subexpression is sorted. Only one subexpression is done at a time this way.

Parameters

<i>term</i>	the input term
<i>replac</i>	number of the subexpression pointer to replace
<i>extractbuff</i>	number of the compiler buffer replac refers to
<i>position</i>	position from where to take the term in the compiler buffer
<i>termout</i>	the output term
<i>tepos</i>	offset in term where the subexpression is.

Returns

Normal conventions (OK = 0).

Definition at line [2730](#) of file [proces.c](#).

Referenced by [Deferred\(\)](#), [Generator\(\)](#), and [PF_Deferred\(\)](#).

4.19.4.120 LongToLine()

```
void LongToLine (
    UWORD * a,
    WORD na ) [extern]
```

Definition at line [285](#) of file [sch.c](#).

4.19.4.121 MakeDirty()

```
int MakeDirty (
    WORD * term,
    WORD * x,
    WORD par ) [extern]
```

Definition at line [51](#) of file [function.c](#).

4.19.4.122 MarkDirty()

```
void MarkDirty (
    WORD * term,
    WORD flags ) [extern]
```

Definition at line 97 of file [function.c](#).

4.19.4.123 PolyFunDirty()

```
void PolyFunDirty (
    PHEAD WORD * term ) [extern]
```

Definition at line 139 of file [function.c](#).

4.19.4.124 PolyFunClean()

```
void PolyFunClean (
    PHEAD WORD * term ) [extern]
```

Definition at line 174 of file [function.c](#).

4.19.4.125 MakeModTable()

```
int MakeModTable (
    void ) [extern]
```

Definition at line 3364 of file [reken.c](#).

4.19.4.126 MatchE()

```
int MatchE (
    PHEAD WORD * pattern,
    WORD * fun,
    WORD * inter,
    WORD par ) [extern]
```

Definition at line 56 of file [symmetr.c](#).

4.19.4.127 MatchCy()

```
int MatchCy (
    PHEAD WORD * pattern,
    WORD * fun,
    WORD * inter,
    WORD par ) [extern]
```

Definition at line 571 of file [symmetr.c](#).

4.19.4.128 FunMatchCy()

```
int FunMatchCy (
    PHEAD WORD * pattern,
    WORD * fun,
    WORD * inter,
    WORD par ) [extern]
```

Definition at line 1067 of file [symmetr.c](#).

4.19.4.129 FunMatchSy()

```
int FunMatchSy (
    PHEAD WORD * pattern,
    WORD * fun,
    WORD * inter,
    WORD par ) [extern]
```

Definition at line 1525 of file [symmetr.c](#).

4.19.4.130 MatchArgument()

```
int MatchArgument (
    PHEAD WORD * arg,
    WORD * pat ) [extern]
```

Definition at line 2052 of file [symmetr.c](#).

4.19.4.131 MatchFunction()

```
int MatchFunction (
    PHEAD WORD * pattern,
    WORD * interm,
    WORD * wilds ) [extern]
```

Definition at line 826 of file [function.c](#).

4.19.4.132 MergePatches()

```
int MergePatches (
    WORD par ) [extern]
```

The general merge routine. Can be used for the large buffer and the file merging. The array S->Patches tells where the patches start S->pStop tells where they end (has to be computed first). The end of a 'line to be merged' is indicated by a zero. If the end is reached without running into a zero or a term runs over the boundary of a patch it is a file merging operation and a new piece from the file is read in.

Parameters

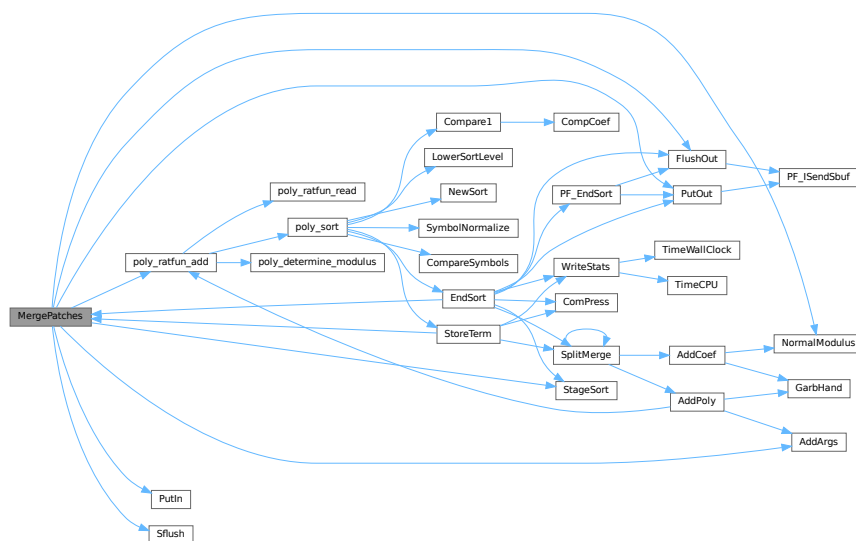
<i>par</i>	If par == 0 the sort is for file -> outfile. If par == 1 the sort is for large buffer -> sortfile. If par == 2 the sort is for large buffer -> outfile.

Definition at line 3767 of file [sort.c](#).

References [AddArgs\(\)](#), [FlushOut\(\)](#), [FiLe::handle](#), [NormalModulus\(\)](#), [poly_ratfun_add\(\)](#), [PutIn\(\)](#), [PutOut\(\)](#), [Sflush\(\)](#), and [StageSort\(\)](#).

Referenced by [EndSort\(\)](#), and [StoreTerm\(\)](#).

Here is the call graph for this function:



4.19.4.133 MesCerr()

```
int MesCerr (
    char * s,
    UBYTE * t ) [extern]
```

Definition at line 812 of file [message.c](#).

4.19.4.134 MesComp()

```
int MesComp (
    char * s,
    UBYTE * p,
    UBYTE * q ) [extern]
```

Definition at line 831 of file [message.c](#).

4.19.4.135 Modulus()

```
int Modulus (
    WORD * term ) [extern]
```

Definition at line [3103](#) of file [reken.c](#).

4.19.4.136 MoveDummies()

```
void MoveDummies (
    PHEAD WORD * term,
    WORD shift ) [extern]
```

Definition at line [436](#) of file [reshuf.c](#).

4.19.4.137 MulLong()

```
int MulLong (
    UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc ) [extern]
```

Definition at line [842](#) of file [reken.c](#).

4.19.4.138 MulRat()

```
int MulRat (
    PHEAD UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc ) [extern]
```

Definition at line [378](#) of file [reken.c](#).

4.19.4.139 Mully()

```
int Mully (
    PHEAD UWORD * a,
    WORD * na,
    UWORD * b,
    WORD nb ) [extern]
```

Definition at line [126](#) of file [reken.c](#).

4.19.4.140 MultDo()

```
int MultDo (
    PHEAD WORD * term,
    WORD * pattern ) [extern]
```

Definition at line [1044](#) of file [reshuf.c](#).

4.19.4.141 NewSort()

```
int NewSort (
    PHEAD0 ) [extern]
```

Starts a new sort. At the lowest level this is a 'main sort' with the struct according to the parameters in S0. At higher levels this is a sort for functions, subroutines or dollars. We prepare the arrays and structs.

Returns

Regular convention (OK -> 0)

Definition at line [649](#) of file [sort.c](#).

Referenced by [generate_expression\(\)](#), [get_expression\(\)](#), [InFunction\(\)](#), [MakeDollarInteger\(\)](#), [MakeDollarMod\(\)](#), [PF_CollectModifiedDollars\(\)](#), [PF_Processor\(\)](#), [poly_factorize_expression\(\)](#), [poly_sort\(\)](#), [poly_unfactorize_expression\(\)](#), [PolyFunMul\(\)](#), [Processor\(\)](#), [TakeArgContent\(\)](#), and [TestMatch\(\)](#).

4.19.4.142 ExtraSymbol()

```
int ExtraSymbol (
    WORD sym,
    WORD pow,
    WORD nsym,
    WORD * ppsym,
    WORD * ncoef ) [extern]
```

Definition at line [4216](#) of file [normal.c](#).

4.19.4.143 Normalize()

```
int Normalize (
    PHEAD WORD * term ) [extern]
```

Definition at line [193](#) of file [normal.c](#).

4.19.4.144 BracketNormalize()

```
int BracketNormalize (
    PHEAD WORD * term ) [extern]
```

Definition at line [5291](#) of file [normal.c](#).

4.19.4.145 DropCoefficient()

```
void DropCoefficient (
    PHEAD WORD * term ) [extern]
```

Definition at line 5116 of file [normal.c](#).

4.19.4.146 DropSymbols()

```
void DropSymbols (
    PHEAD WORD * term ) [extern]
```

Definition at line 5134 of file [normal.c](#).

4.19.4.147 PutInside()

```
int PutInside (
    PHEAD WORD * term,
    WORD * code ) [extern]
```

Definition at line 609 of file [index.c](#).

4.19.4.148 OpenTemp()

```
void OpenTemp (
    void ) [extern]
```

Definition at line 48 of file [store.c](#).

4.19.4.149 Pack()

```
void Pack (
    UWORD * a,
    WORD * na,
    UWORD * b,
    WORD nb ) [extern]
```

Definition at line 62 of file [reken.c](#).

4.19.4.150 PasteFile()

```
LONG PasteFile (
    PHEAD WORD number,
    WORD * accum,
    POSITION * position,
    WORD ** accfill,
    RENUMBER renumber,
    WORD * freeze,
    WORD nexpr ) [extern]
```

Gets a term from stored expression *expr* and puts it in the accumulator at position number. It returns the length of the term that came from file.

Parameters

<i>number</i>	number of partial terms to skip in accum
<i>accum</i>	the accumulator
<i>position</i>	file position from where to get the stored term
<i>accfill</i>	returns tail position in accum
<i>renumber</i>	the renumber struct for the variables in the stored expression
<i>freeze</i>	information about if we need only the contents of a bracket
<i>nexpr</i>	the number of the stored expression

Returns

Normal conventions (OK = 0).

Definition at line [2866](#) of file [proces.c](#).

Referenced by [Generator\(\)](#).

4.19.4.151 Permute()

```
int Permute (
    PERM * perm,
    WORD first ) [extern]
```

Definition at line [444](#) of file [symmetr.c](#).

4.19.4.152 PermuteP()

```
int PermuteP (
    PERMP * perm,
    WORD first ) [extern]
```

Definition at line [478](#) of file [symmetr.c](#).

4.19.4.153 PolyFunMul()

```
int PolyFunMul (
    PHEAD WORD * term ) [extern]
```

Multiplies the arguments of multiple occurrences of the polyfun. In this routine we do the original PolyFun with one argument only. The PolyRatFun (PolyFunType = 2) is done in a dedicated routine in the file [polywrap.cc](#) The new result is written over the old result.

Parameters

<i>term</i>	It contains the input term and later the output.

Returns

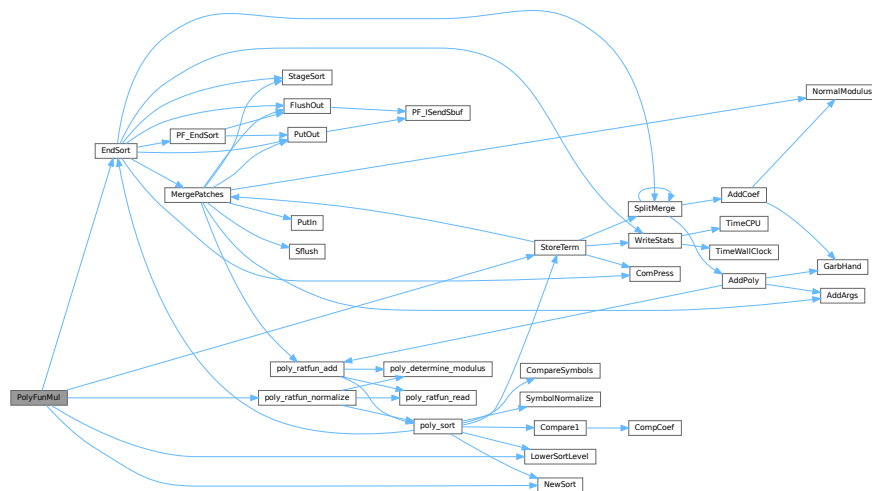
Normal conventions (OK = 0).

Definition at line 5356 of file [proces.c](#).

References [EndSort\(\)](#), [LowerSortLevel\(\)](#), [NewSort\(\)](#), [poly_ratfun_normalize\(\)](#), and [StoreTerm\(\)](#).

Referenced by [Generator\(\)](#).

Here is the call graph for this function:



4.19.4.154 PopVariables()

```
int PopVariables (
    void ) [extern]
```

Definition at line 211 of file [execute.c](#).

4.19.4.155 PrepPoly()

```
int PrepPoly (
    PHEAD WORD * term,
    WORD par ) [extern]
```

Routine checks whether the count of function AR.PolyFun is zero or one. If it is one and it has one scalarlike argument the coefficient of the term is pulled inside the argument. If the count is zero a new function is made with the coefficient as its only argument. The function should be placed at its proper position.

When this function is active it places the PolyFun as last object before the coefficient. This is needed because otherwise the compress algorithm has problems in MergePatches.

The bracket routine should also place the PolyFun at a comparable spot. The compression should then stop at the PolyFun. It doesn't really have to stop when writing the final result but this may be too complicated.

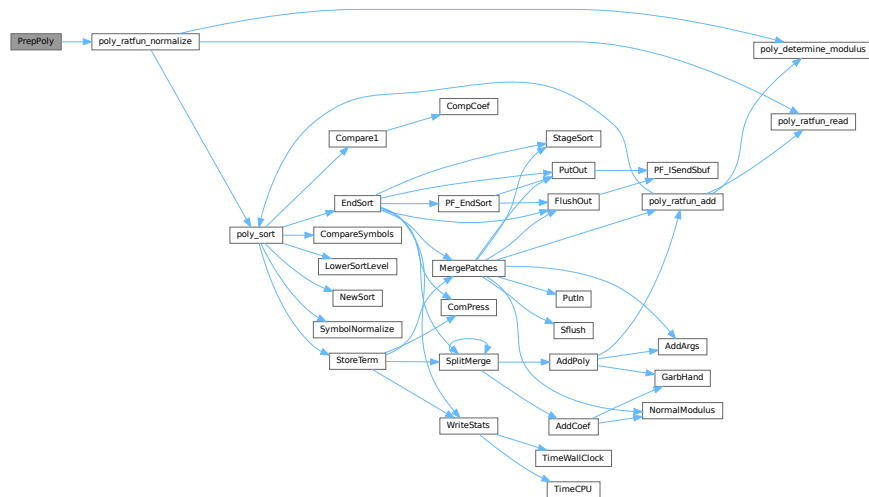
The parameter `par` tells whether we are at groundlevel or inside a function or dollar variable.

Definition at line 4968 of file `proces.c`.

References `poly_ratfun_normalize()`.

Referenced by `Generator()`.

Here is the call graph for this function:



4.19.4.156 Processor()

```
int Processor (
    void ) [extern]
```

This is the central processor. It accepts a stream of Expressions which is accessed by calls to `GetTerm`. The expressions reside either in `AR.infile` or `AR.hidefile`. The definitions of an expression are seen as an id-statement, so the primary Expressions should be written to the system of scratch files as single terms with an expression pointer. Each expression is terminated with a zero and the whole is terminated by two zeroes.

The routine `DoExecute` should determine whether results are to be printed, should revert the scratch I/O directions etc. In principle it is `DoExecute` that calls `Processor`.

Returns

if everything OK: 0. Otherwise error. The preprocessor may continue with compilation though. Really fatal errors should return on the spot by calling `Terminate`.

Definition at line 64 of file `proces.c`.

References `CbUf::Buffer`, `EndSort()`, `FlushOut()`, `Generator()`, `CbUf::lhs`, `LowerSortLevel()`, `NewSort()`, `PF_BroadcastRHS()`, `PF_InParallelProcessor()`, `PF_Processor()`, `CbUf::Pointer`, `poly_factorize_expression()`, `poly_unfactorize_expression()`, `PutOut()`, `CbUf::rhs`, and `StoreTerm()`.

[illegible]

```
int Product (
    UWORD * a,
    WORD * na,
    WORD b ) [extern]
```

```
void PrtLong (
    UWORD * a,
    WORD na,
    UBYTE * s ) [extern]
```

Generated by Doxygen

4.19.4.159 PrtTerms()

```
void PrtTerms (
    void ) [extern]
```

Definition at line 698 of file [sch.c](#).

4.19.4.160 PrintDeprecation()

```
void PrintDeprecation (
    const char * feature,
    const char * issue ) [extern]
```

Prints a deprecation warning for a given feature.

Parameters

<i>feature</i>	The name of the deprecated feature.
<i>issue</i>	The associated issue, e.g., "issues/700".

Definition at line 2082 of file [startup.c](#).

4.19.4.161 PrintRunningTime()

```
void PrintRunningTime (
    void ) [extern]
```

Definition at line 2106 of file [startup.c](#).

4.19.4.162 GetRunningTime()

```
LONG GetRunningTime (
    void ) [extern]
```

Definition at line 2147 of file [startup.c](#).

4.19.4.163 PutBracket()

```
int PutBracket (
    PHEAD WORD * termin ) [extern]
```

Definition at line 1112 of file [execute.c](#).

4.19.4.164 PutIn()

```
LONG PutIn (
    FILEHANDLE * file,
    POSITION * position,
    WORD * buffer,
    WORD ** take,
    int npat ) [extern]
```

Reads a new patch from position in file handle. It is put at buffer, anything after take is moved forward. This would be part of a term that hasn't been used yet. Because of this there should be some space before the start of the buffer

Parameters

<i>file</i>	The file system from which to read
<i>position</i>	The position from which to read
<i>buffer</i>	The buffer into which to read
<i>take</i>	The unused tail should be moved before the buffer
<i>npat</i>	The number of the patch. Is needed if the information was compressed with gzip, because each patch has its own independent gzip encoding.

Definition at line 1324 of file [sort.c](#).

References [FiLe::handle](#).

Referenced by [MergePatches\(\)](#).

4.19.4.165 PutInStore()

```
int PutInStore (
    INDEXENTRY * ind,
    WORD num ) [extern]
```

Definition at line 873 of file [store.c](#).

4.19.4.166 PutOut()

```
WORD PutOut (
    PHEAD WORD * term,
    POSITION * position,
    FILEHANDLE * fi,
    WORD ncomp ) [extern]
```

Routine writes one term to file handle at position. It returns the new value of the position.

NOTE: For 'final output' we may have to index the brackets. See the struct BRACKETINDEX. We should maintain:
 1: a list with brackets array with the brackets
 2: a list of objects of type BRACKETINDEX. It contains array with either pointers or offsets to the list of brackets. starting positions in the file. The index may be tied to a maximum size. In that case we may have to prune the list occasionally.

Parameters

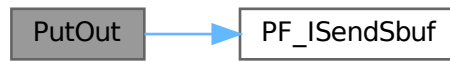
<i>term</i>	The term to be written
<i>position</i>	The position in the file. Afterwards it is updated
<i>fi</i>	The file (or its cache) to which should be written
<i>ncomp</i>	Information about what type of compression should be used

Definition at line 1470 of file [sort.c](#).

References [FiLe::handle](#), and [PF_ISendSbuf\(\)](#).

Referenced by [EndSort\(\)](#), [generate_expression\(\)](#), [MergePatches\(\)](#), [PF_EndSort\(\)](#), [PF_Processor\(\)](#), and [Processor\(\)](#).

Here is the call graph for this function:



4.19.4.167 Quotient()

```
UWORD Quotient (  
    UWORD * a,  
    WORD * na,  
    WORD b ) [extern]
```

Definition at line 1621 of file [reken.c](#).

4.19.4.168 RaisPow()

```
int RaisPow (  
    PHEAD UWORD * a,  
    WORD * na,  
    UWORD b ) [extern]
```

Definition at line 1229 of file [reken.c](#).

4.19.4.169 RaisPowCached()

```
void RaisPowCached (  
    PHEAD WORD x,  
    WORD n,  
    UWORD ** c,  
    WORD * nc ) [extern]
```

Computes power x^n and caches the value

4.19.5 Description

Calculates the power x^n and stores the results for caching purposes. The pointer c (i.e., the pointer, and not what it points to) is overwritten. What it points to should not be overwritten in the calling function.

4.19.6 Notes

- Caching is done in `AT.small_power[]`. This array is extended if necessary.

Definition at line 1297 of file [reken.c](#).

Referenced by [poly::divmod_heap\(\)](#), [poly::divmod_univar\(\)](#), and [poly::mul_heap\(\)](#).

4.19.6.1 RaisPowMod()

```
WORD RaisPowMod (
    WORD x,
    WORD n,
    WORD m ) [extern]
```

Definition at line 1384 of file [reken.c](#).

4.19.6.2 NormalModulus()

```
int NormalModulus (
    UWORD * a,
    WORD * na ) [extern]
```

Brings a modular representation in the range $-p/2$ to $+p/2$ The return value tells whether anything was done. Routine made in the general modulus revamp of July 2008 (JV).

Definition at line 1404 of file [reken.c](#).

Referenced by [AddCoef\(\)](#), and [MergePatches\(\)](#).

4.19.6.3 MakeInverses()

```
int MakeInverses (
    void ) [extern]
```

Makes a table of inverses in modular calculus The modulus is in `AC.cmod` and `AC.ncmod` One should notice that the table of inverses can only be made if the modulus fits inside a single FORM word. Otherwise the table lookup becomes too difficult and the table too long.

Definition at line 1441 of file [reken.c](#).

References [GetModInverses\(\)](#).

Here is the call graph for this function:



4.19.6.4 GetModInverses()

```
int GetModInverses (
    WORD m1,
    WORD m2,
    WORD * im1,
    WORD * im2 ) [extern]
```

Input m1 and m2, which are relative prime. determines $a*m1+b*m2 = 1$ (and 1 is the gcd of m1 and m2) then $a*m1 = 1 \bmod m2$ and hence $im1 = a$. and $b*m2 = 1 \bmod m1$ and hence $im2 = b$. Set $m1 = 0*m1+1*m2 = a1*m1+b1*m2$
 $m2 = 1*m1+0*m2 = a2*m1+b2*m2$ If everything is OK, the return value is zero

Definition at line [1477](#) of file [reken.c](#).

Referenced by [poly::divmod_heap\(\)](#), [poly::divmod_univar\(\)](#), [MakeDollarMod\(\)](#), [MakeInverses\(\)](#), [MakeMod\(\)](#), [TakeContent\(\)](#), and [TakeSymbolContent\(\)](#).

4.19.6.5 GetLongModInverses()

```
int GetLongModInverses (
    PHEAD UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * ia,
    WORD * nia,
    UWORD * ib,
    WORD * nib ) [extern]
```

Definition at line [1509](#) of file [reken.c](#).

4.19.6.6 RatToLine()

```
void RatToLine (
    UWORD * a,
    WORD na ) [extern]
```

Definition at line [319](#) of file [sch.c](#).

4.19.6.7 RatioFind()

```
int RatioFind (
    PHEAD WORD * term,
    WORD * params ) [extern]
```

Definition at line [62](#) of file [ratio.c](#).

4.19.6.8 RatioGen()

```
int RatioGen (
    PHEAD WORD * term,
    WORD * params,
    WORD num,
    WORD level ) [extern]
```

Definition at line 170 of file [ratio.c](#).

4.19.6.9 ReNumber()

```
WORD ReNumber (
    PHEAD WORD * term ) [extern]
```

Definition at line 80 of file [reshuf.c](#).

4.19.6.10 ReadSnum()

```
WORD ReadSnum (
    UBYTE ** p ) [extern]
```

Definition at line 2003 of file [tools.c](#).

4.19.6.11 Remain10()

```
WORD Remain10 (
    UWORD * a,
    WORD * na ) [extern]
```

Definition at line 1660 of file [reken.c](#).

4.19.6.12 Remain4()

```
WORD Remain4 (
    UWORD * a,
    WORD * na ) [extern]
```

Definition at line 1687 of file [reken.c](#).

4.19.6.13 ResetScratch()

```
int ResetScratch (
    void ) [extern]
```

Definition at line 234 of file [store.c](#).

4.19.6.14 ResolveSet()

```
int ResolveSet (
    PHEAD WORD * from,
    WORD * to,
    WORD * subs ) [extern]
```

Definition at line [1361](#) of file [wildcard.c](#).

4.19.6.15 RevertScratch()

```
int RevertScratch (
    void ) [extern]
```

Definition at line [189](#) of file [store.c](#).

4.19.6.16 ScanFunctions()

```
int ScanFunctions (
    PHEAD WORD * inpat,
    WORD * inter,
    WORD par ) [extern]
```

Definition at line [1618](#) of file [function.c](#).

4.19.6.17 SeekScratch()

```
void SeekScratch (
    FILEHANDLE * fi,
    POSITION * pos ) [extern]
```

Definition at line [63](#) of file [store.c](#).

4.19.6.18 SetEndScratch()

```
void SetEndScratch (
    FILEHANDLE * f,
    POSITION * position ) [extern]
```

Definition at line [74](#) of file [store.c](#).

4.19.6.19 SetEndHScratch()

```
void SetEndHScratch (
    FILEHANDLE * f,
    POSITION * position ) [extern]
```

Definition at line [88](#) of file [store.c](#).

4.19.6.20 SetFileIndex()

```
int SetFileIndex (
    void ) [extern]
```

Reads the next file index and puts it into AR.StoreData.Index. TODO

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 2320 of file [store.c](#).

References [WriteStoreHeader\(\)](#).

Here is the call graph for this function:



4.19.6.21 Sflush()

```
int Sflush (
    FILEHANDLE * fi ) [extern]
```

Puts the contents of a buffer to output Only to be used when there is a single patch in the large buffer.

Parameters

<i>fi</i>	The filesystem (or its cache) to which the patch should be written

Definition at line 1384 of file [sort.c](#).

References [FiLe::handle](#).

Referenced by [MergePatches\(\)](#).

4.19.6.22 Simplify()

```
int Simplify (
    PHEAD UWORD * a,
    WORD * na,
    UWORD * b,
    WORD * nb ) [extern]
```

Definition at line 531 of file [reken.c](#).

4.19.6.23 SortWild()

```
int SortWild (
    WORD * w,
    WORD nw ) [extern]
```

Sorts the wildcard entries in the parameter w. Double entries are removed. Full space taken is nw words. Routine serves for the reading of wildcards in the compiler. The entries come in the format: (type,4,number,0) in which the zero is reserved for the future replacement of 'number'.

Parameters

<i>w</i>	buffer with wildcard entries.
<i>nw</i>	number of wildcard entries.

Returns

Normal conventions (OK -> 0)

Definition at line [4763](#) of file [sort.c](#).

4.19.6.24 LocateBase()

```
FILE * LocateBase (
    char ** name,
    char ** newname,
    char * iomode ) [extern]
```

Definition at line [225](#) of file [minos.c](#).

4.19.6.25 SplitMerge()

```
LONG SplitMerge (
    PHEAD WORD ** Pointer,
    LONG number ) [extern]
```

Algorithm by J.A.M.Vermaseren (31-7-1988)

Note that AN.SplitScratch and AN.InScratch are used also in GarbHand

Merge sort in memory. The input is an array of pointers. Sorting is done recursively by dividing the array in two equal parts and calling SplitMerge for each. When the parts are small enough we can do the compare and take the appropriate action. An addition is that we look for 'runs'. Sequences that are already ordered. This happens a lot when there is very little action in a module. This made FORM faster by a few percent.

Parameters

<i>Pointer</i>	The array of pointers to the terms to be sorted.
<i>number</i>	The number of pointers in Pointer.

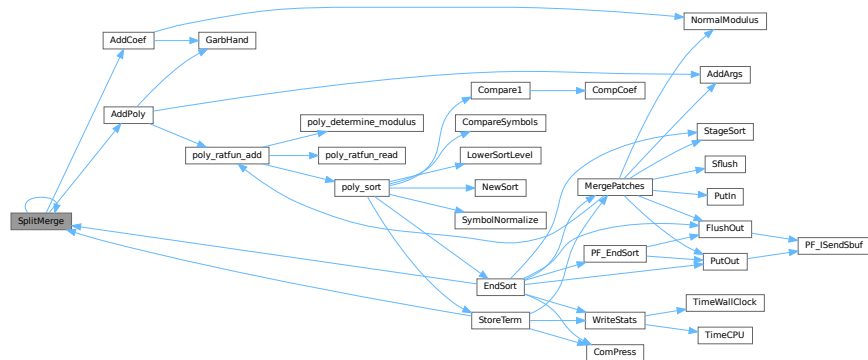
The terms are supposed to be sitting in the small buffer and there is supposed to be an extension to this buffer for when there are two terms that should be added and the result takes more space than each of the original terms. The notation guarantees that the result never needs more space than the sum of the spaces of the original terms.

Definition at line 3372 of file [sort.c](#).

References [AddCoef\(\)](#), [AddPoly\(\)](#), and [SplitMerge\(\)](#).

Referenced by [EndSort\(\)](#), [SplitMerge\(\)](#), and [StoreTerm\(\)](#).

Here is the call graph for this function:



4.19.6.26 StoreTerm()

```
int StoreTerm (
    PHEAD WORD * term ) [extern]
```

The central routine to accept terms, store them and keep things at least partially sorted. A call to EndSort will then complete storing and sorting.

Parameters

<i>term</i>	The term to be stored

Returns

Regular return conventions (OK -> 0)

Definition at line 4550 of file [sort.c](#).

References [ComPress\(\)](#), [MergePatches\(\)](#), [SplitMerge\(\)](#), and [WriteStats\(\)](#).

Referenced by [Generator\(\)](#), [get_expression\(\)](#), [InFunction\(\)](#), [PF_Processor\(\)](#), [poly_sort\(\)](#), [PolyFunMul\(\)](#), [Processor\(\)](#), and [TakeArgContent\(\)](#).

4.19.6.30 SymGen()

```
int SymGen (
    PHEAD WORD * term,
    WORD * params,
    WORD num,
    WORD level ) [extern]
```

Definition at line 518 of file [function.c](#).

4.19.6.31 Symmetrize()

```
WORD Symmetrize (
    PHEAD WORD * func,
    WORD * Lijst,
    WORD ngroups,
    WORD gsize,
    WORD type ) [extern]
```

Definition at line 213 of file [function.c](#).

4.19.6.32 FullSymmetrize()

```
int FullSymmetrize (
    PHEAD WORD * fun,
    int type ) [extern]
```

Definition at line 473 of file [function.c](#).

4.19.6.33 TakeModulus()

```
int TakeModulus (
    UWORD * a,
    WORD * na,
    UWORD * cmodvec,
    WORD ncmod,
    WORD par ) [extern]
```

Definition at line 3150 of file [reken.c](#).

4.19.6.34 TakeNormalModulus()

```
int TakeNormalModulus (
    UWORD * a,
    WORD * na,
    UWORD * c,
    WORD nc,
    WORD par ) [extern]
```

Definition at line 3322 of file [reken.c](#).

4.19.6.35 TalToLine()

```
void TalToLine (
    UWORD x ) [extern]
```

Definition at line 504 of file [sch.c](#).

4.19.6.36 TenVec()

```
int TenVec (
    PHEAD WORD * term,
    WORD * params,
    WORD num,
    WORD level ) [extern]
```

Definition at line 2315 of file [opera.c](#).

4.19.6.37 TenVecFind()

```
int TenVecFind (
    PHEAD WORD * term,
    WORD * params ) [extern]
```

Definition at line 2181 of file [opera.c](#).

4.19.6.38 TermRenumber()

```
int TermRenumber (
    WORD * term,
    RENUMBER renumber,
    WORD nexpr ) [extern]
```

```
!! WORD *memterm=term; static LONG ctrap=0; !!!
```

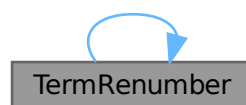
```
!! ctrap++; !!!
```

Definition at line 2427 of file [store.c](#).

References [ReNuMbEr::func](#), [ReNuMbEr::funnum](#), [ReNuMbEr::indi](#), [ReNuMbEr::indnum](#), [ReNuMbEr::symb](#), [ReNuMbEr::symnum](#), [TermRenumber\(\)](#), [ReNuMbEr::vecnum](#), and [ReNuMbEr::vect](#).

Referenced by [TermRenumber\(\)](#).

Here is the call graph for this function:



4.19.6.39 TestDrop()

```
void TestDrop (
    void ) [extern]
```

Definition at line 484 of file [execute.c](#).

4.19.6.40 PutInVflags()

```
void PutInVflags (
    WORD nexpr ) [extern]
```

Definition at line 562 of file [execute.c](#).

4.19.6.41 TestMatch()

```
int TestMatch (
    PHEAD WORD * term,
    WORD * level ) [extern]
```

This routine governs the pattern matching. If it decides that a substitution should be made, this can be either the insertion of a right hand side (C->rhs) or the automatic generation of terms as a result of an operation (like trace). The object to be replaced is removed from term and a subexpression pointer is inserted. If the substitution is made more than once there can be more subexpression pointers. Its number is positive as it corresponds to the level at which the C->rhs can be found in the compiler output. The subexpression pointer contains the wildcard substitution information. The power is found in *AT.TMout. For operations the subexpression pointer is negative and corresponds to an address in the array AT.TMout. In this array are then the instructions for the routine to be called and its number in the array 'Operations' The format is here: length,functionnumber,length-2 parameters

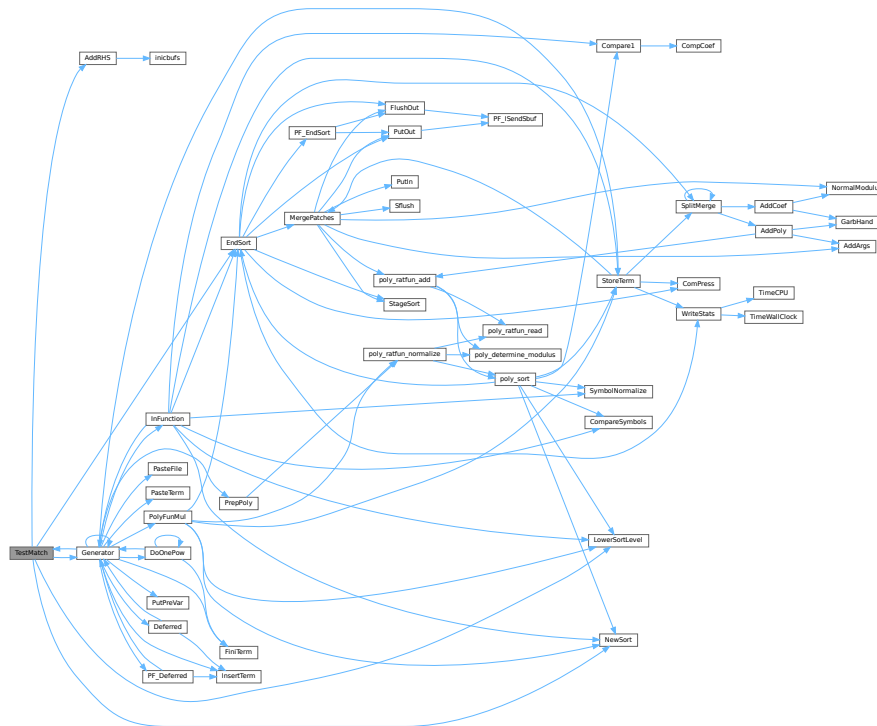
There is a certain complexity wrt repeat levels. Another complication is the poking of the wildcard values in the subexpression prototype in the compiler buffer. This was how things were done in the past with sequential FORM, but with the advent of TFORM this cannot be maintained. Now, for TFORM we make a copy of it. 7-may-2008 (JV): We cannot yet guarantee that this has been done 100% correctly. There are errors that occur in TFORM only and that may indicate problems.

Definition at line 97 of file [pattern.c](#).

References [AddRHS\(\)](#), [EndSort\(\)](#), [Generator\(\)](#), [CbUf::lhs](#), [LowerSortLevel\(\)](#), and [NewSort\(\)](#).

Referenced by [Generator\(\)](#).

Here is the call graph for this function:



4.19.6.42 TestSub()

```
WORD TestSub (
    PHEAD WORD * term,
    WORD level ) [extern]
```

Definition at line 687 of file [proces.c](#).

4.19.6.43 TimeCPU()

```
LONG TimeCPU (
    WORD par ) [extern]
```

Returns the CPU time.

Parameters

<i>par</i>	If zero, the CPU time will be reset to 0.

Returns

The CPU time in milliseconds.

Definition at line 3487 of file [tools.c](#).

Referenced by [PF_GetSlaveTimes\(\)](#), [PF_Processor\(\)](#), and [WriteStats\(\)](#).

4.19.6.44 TimeChildren()

```
LONG TimeChildren (
    WORD par ) [extern]
```

Definition at line [3469](#) of file [tools.c](#).

4.19.6.45 TimeWallClock()

```
LONG TimeWallClock (
    WORD par ) [extern]
```

Returns the wall-clock time.

Parameters

<i>par</i>	If zero, the wall-clock time will be reset to 0.

Returns

The wall-clock time in centiseconds.

Definition at line [3413](#) of file [tools.c](#).

Referenced by [DoCheckpoint\(\)](#), [DoRecovery\(\)](#), [find_Horner_MCTS\(\)](#), [optimize_expression_given_Horner\(\)](#), [optimize_greedy\(\)](#), and [WriteStats\(\)](#).

4.19.6.46 ToStorage()

```
int ToStorage (
    EXPRESSIONS e,
    POSITION * length ) [extern]
```

Definition at line [2016](#) of file [store.c](#).

4.19.6.47 TokenToLine()

```
void TokenToLine (
    UBYTE * s ) [extern]
```

Definition at line [533](#) of file [sch.c](#).

4.19.6.48 Trace4()

```
int Trace4 (
    PHEAD WORD * term,
    WORD * params,
    WORD num,
    WORD level ) [extern]
```

Definition at line [695](#) of file [opera.c](#).

4.19.6.49 Trace4Gen()

```
int Trace4Gen (
    PHEAD TRACES * t,
    WORD number ) [extern]
```

Definition at line 858 of file [opera.c](#).

4.19.6.50 Trace4no()

```
int Trace4no (
    WORD number,
    WORD * kron,
    TRACES * t ) [extern]
```

Definition at line 418 of file [opera.c](#).

4.19.6.51 TraceFind()

```
int TraceFind (
    PHEAD WORD * term,
    WORD * params ) [extern]
```

Definition at line 1856 of file [opera.c](#).

4.19.6.52 TraceN()

```
int TraceN (
    PHEAD WORD * term,
    WORD * params,
    WORD num,
    WORD level ) [extern]
```

Definition at line 1384 of file [opera.c](#).

4.19.6.53 TraceNgen()

```
int TraceNgen (
    PHEAD TRACES * t,
    WORD number ) [extern]
```

Definition at line 1449 of file [opera.c](#).

4.19.6.54 TraceNno()

```
WORD TraceNno (
    WORD number,
    WORD * kron,
    TRACES * t ) [extern]
```

Definition at line 1343 of file [opera.c](#).

4.19.6.55 Traces()

```
int Traces (
    PHEAD WORD * term,
    WORD * params,
    WORD num,
    WORD level ) [extern]
```

Definition at line 1834 of file [opera.c](#).

4.19.6.56 Trick()

```
WORD Trick (
    WORD * in,
    TRACES * t ) [extern]
```

Definition at line 331 of file [opera.c](#).

4.19.6.57 TryDo()

```
int TryDo (
    PHEAD WORD * term,
    WORD * pattern,
    WORD level ) [extern]
```

Definition at line 1072 of file [reshuf.c](#).

4.19.6.58 UnPack()

```
void UnPack (
    UWORD * a,
    WORD na,
    WORD * denom,
    WORD * numer ) [extern]
```

Definition at line 100 of file [reken.c](#).

4.19.6.59 VarStore()

```
int VarStore (
    UBYTE * s,
    WORD n,
    WORD name,
    WORD namesize ) [extern]
```

Definition at line 2369 of file [store.c](#).

4.19.6.60 WildFill()

```
WORD WildFill (
    PHEAD WORD * to,
    WORD * from,
    WORD * sub ) [extern]
```

Definition at line 65 of file [wildcard.c](#).

4.19.6.61 WriteAll()

```
int WriteAll (
    void ) [extern]
```

Definition at line 2486 of file [sch.c](#).

4.19.6.62 WriteOne()

```
int WriteOne (
    UBYTE * name,
    int alreadyinline,
    int nosemi,
    WORD plus ) [extern]
```

Definition at line 2712 of file [sch.c](#).

4.19.6.63 WriteArgument()

```
void WriteArgument (
    WORD * t ) [extern]
```

Definition at line 1406 of file [sch.c](#).

4.19.6.64 WriteExpression()

```
int WriteExpression (
    WORD * terms,
    LONG ltot ) [extern]
```

Definition at line 2455 of file [sch.c](#).

4.19.6.65 WriteInnerTerm()

```
int WriteInnerTerm (
    WORD * term,
    WORD first ) [extern]
```

Definition at line 1933 of file [sch.c](#).

4.19.6.66 WriteLists()

```
void WriteLists (
    void ) [extern]
```

Definition at line 792 of file [sch.c](#).

4.19.6.67 WriteSetup()

```
void WriteSetup (
    void ) [extern]
```

Definition at line 800 of file [setfile.c](#).

4.19.6.68 WriteStats()

```
void WriteStats (
    POSITION * plspace,
    WORD par,
    WORD checkLogType ) [extern]
```

Writes the statistics.

Parameters

<i>plspace</i>	The size in bytes currently occupied
<i>par</i>	par = 0 = STATSSPLITMERGE after a splitmerge. par = 1 = STATSMERGETOFILE after merge to sortfile. par = 2 = STATSPOSTSORT after the sort
<i>checkLogType</i>	== CHECKLOGTYPE: The output should not go to the log file if AM.LogType is 1.

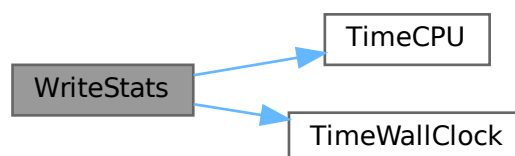
current expression is to be found in AR.CurExpr. terms are in S->TermsLeft. S->GenTerms.

Definition at line 127 of file [sort.c](#).

References [TimeCPU\(\)](#), and [TimeWallClock\(\)](#).

Referenced by [EndSort\(\)](#), [PF_Processor\(\)](#), and [StoreTerm\(\)](#).

Here is the call graph for this function:



4.19.6.69 WriteSubTerm()

```
int WriteSubTerm (
    WORD * sterm,
    WORD first ) [extern]
```

Definition at line 1524 of file [sch.c](#).

4.19.6.70 WriteTerm()

```
int WriteTerm (
    WORD * term,
    WORD * lbrac,
    WORD first,
    WORD prtf,
    WORD br ) [extern]
```

Definition at line 2133 of file [sch.c](#).

4.19.6.71 execarg()

```
int execarg (
    PHEAD WORD * term,
    WORD level ) [extern]
```

Definition at line 56 of file [argument.c](#).

4.19.6.72 execterm()

```
int execterm (
    PHEAD WORD * term,
    WORD level ) [extern]
```

Definition at line 1775 of file [argument.c](#).

4.19.6.73 SpecialCleanup()

```
void SpecialCleanup (
    PHEAD0 ) [extern]
```

Definition at line 1626 of file [execute.c](#).

4.19.6.74 SetMods()

```
void SetMods (
    void ) [extern]
```

Definition at line 1640 of file [execute.c](#).

4.19.6.75 UnSetMods()

```
void UnSetMods (
    void ) [extern]
```

Definition at line 1658 of file [execute.c](#).

4.19.6.76 DoExecute()

```
int DoExecute (
    WORD par,
    WORD skip ) [extern]
```

Definition at line 616 of file [execute.c](#).

4.19.6.77 SetScratch()

```
void SetScratch (
    FILEHANDLE * f,
    POSITION * position ) [extern]
```

Definition at line 114 of file [store.c](#).

4.19.6.78 Warning()

```
void Warning (
    char * s ) [extern]
```

Definition at line 778 of file [message.c](#).

4.19.6.79 HighWarning()

```
void HighWarning (
    char * s ) [extern]
```

Definition at line 790 of file [message.c](#).

4.19.6.80 SpareTable()

```
int SpareTable (
    TABLES TT ) [extern]
```

Definition at line 694 of file [tables.c](#).

4.19.6.81 strDup1()

```
UBYTE * strDup1 (
    UBYTE * instring,
    char * ifwrong ) [extern]
```

Definition at line 1906 of file [tools.c](#).

4.19.6.82 Malloc1()

```
void * Malloc1 (
    LONG size,
    const char * messageifwrong ) [extern]
```

Definition at line 2243 of file [tools.c](#).

4.19.6.83 DoTail()

```
int DoTail (
    int argc,
    UBYTE ** argv ) [extern]
```

Definition at line 237 of file [startup.c](#).

4.19.6.84 OpenInput()

```
int OpenInput (
    void ) [extern]
```

Definition at line 542 of file [startup.c](#).

4.19.6.85 PutPreVar()

```
int PutPreVar (
    UBYTE * name,
    UBYTE * value,
    UBYTE * args,
    int mode ) [extern]
```

Inserts/Updates a preprocessor variable in the name administration.

Parameters

<i>name</i>	Character string with the variable name.
<i>value</i>	Character string with a possible value. Special case: if this argument is zero, then we have no value. Note: This is different from having an empty argument! This should only occur when the name starts with a ?
<i>args</i>	Character string with possible arguments.
<i>mode</i>	=0: always create a new name entry, =1: try to do a redefinition if possible.

Returns

Index of used entry in name list.

Definition at line 724 of file [pre.c](#).

Referenced by [ClearOptimize\(\)](#), [Generator\(\)](#), [Optimize\(\)](#), [PF_BroadcastRedefinedPreVars\(\)](#), [StartVariables\(\)](#), and [TheDefine\(\)](#).

4.19.6.86 Error0()

```
void Error0 (
    char * s ) [extern]
```

Definition at line 56 of file [message.c](#).

4.19.6.87 Error1()

```
void Error1 (
    char * s,
    UBYTE * t ) [extern]
```

Definition at line 67 of file [message.c](#).

4.19.6.88 Error2()

```
void Error2 (
    char * s1,
    char * s2,
    UBYTE * t ) [extern]
```

Definition at line 78 of file [message.c](#).

4.19.6.89 ReadFromStream()

```
UBYTE ReadFromStream (
    STREAM * stream ) [extern]
```

Definition at line 151 of file [tools.c](#).

4.19.6.90 GetFromStream()

```
UBYTE GetFromStream (
    STREAM * stream ) [extern]
```

Definition at line 286 of file [tools.c](#).

4.19.6.91 LookInStream()

```
UBYTE LookInStream (  
    STREAM * stream ) [extern]
```

Definition at line 309 of file [tools.c](#).

4.19.6.92 OpenStream()

```
STREAM * OpenStream (  
    UBYTE * name,  
    int type,  
    int prevarmode,  
    int raiselow ) [extern]
```

Definition at line 321 of file [tools.c](#).

4.19.6.93 LocateFile()

```
int LocateFile (  
    UBYTE ** name,  
    int type ) [extern]
```

Definition at line 576 of file [tools.c](#).

4.19.6.94 CloseStream()

```
STREAM * CloseStream (  
    STREAM * stream ) [extern]
```

Definition at line 663 of file [tools.c](#).

4.19.6.95 PositionStream()

```
void PositionStream (  
    STREAM * stream,  
    LONG position ) [extern]
```

Definition at line 825 of file [tools.c](#).

4.19.6.96 ReverseStatements()

```
int ReverseStatements (  
    STREAM * stream ) [extern]
```

Definition at line 857 of file [tools.c](#).

4.19.6.97 ProcessOption()

```
int ProcessOption (
    UBYTE * s1,
    UBYTE * s2,
    int filetype ) [extern]
```

Definition at line [182](#) of file [setfile.c](#).

4.19.6.98 DoSetups()

```
int DoSetups (
    void ) [extern]
```

Definition at line [132](#) of file [setfile.c](#).

4.19.6.99 TerminateImpl()

```
void TerminateImpl (
    int errorcode,
    const char * file,
    int line,
    const char * function ) [extern]
```

Definition at line [1869](#) of file [startup.c](#).

4.19.6.100 GetNode()

```
NAMENODE * GetNode (
    NAMETREE * nametree,
    UBYTE * name ) [extern]
```

Definition at line [49](#) of file [names.c](#).

4.19.6.101 AddName()

```
int AddName (
    NAMETREE * nametree,
    UBYTE * name,
    WORD type,
    WORD number,
    int * nodenum ) [extern]
```

Definition at line [71](#) of file [names.c](#).

4.19.6.102 GetName()

```
int GetName (
    NAMETREE * nametree,
    UBYTE * namein,
    WORD * number,
    int par ) [extern]
```

Definition at line 252 of file [names.c](#).

4.19.6.103 GetFunction()

```
UBYTE * GetFunction (
    UBYTE * s,
    WORD * funnum ) [extern]
```

Definition at line 342 of file [names.c](#).

4.19.6.104 GetNumber()

```
UBYTE * GetNumber (
    UBYTE * s,
    WORD * num ) [extern]
```

Definition at line 388 of file [names.c](#).

4.19.6.105 GetLastExprName()

```
int GetLastExprName (
    UBYTE * name,
    WORD * number ) [extern]
```

Definition at line 438 of file [names.c](#).

4.19.6.106 GetAutoName()

```
int GetAutoName (
    UBYTE * name,
    WORD * number ) [extern]
```

Definition at line 479 of file [names.c](#).

4.19.6.107 GetVar()

```
int GetVar (
    UBYTE * name,
    WORD * type,
    WORD * number,
    int wantedtype,
    int par ) [extern]
```

Definition at line 524 of file [names.c](#).

4.19.6.108 MakeDubious()

```
int MakeDubious (
    NAMETREE * nametree,
    UBYTE * name,
    WORD * number ) [extern]
```

Definition at line 2695 of file [names.c](#).

4.19.6.109 GetOName()

```
int GetOName (
    NAMETREE * nametree,
    UBYTE * name,
    WORD * number,
    int par ) [extern]
```

Definition at line 460 of file [names.c](#).

4.19.6.110 DumpTree()

```
void DumpTree (
    NAMETREE * nametree ) [extern]
```

Definition at line 602 of file [names.c](#).

4.19.6.111 DumpNode()

```
void DumpNode (
    NAMETREE * nametree,
    WORD node,
    WORD depth ) [extern]
```

Definition at line 615 of file [names.c](#).

4.19.6.112 LinkTree()

```
void LinkTree (
    NAMETREE * tree,
    WORD offset,
    WORD numnodes ) [extern]
```

Definition at line 784 of file [names.c](#).

4.19.6.113 CopyTree()

```
void CopyTree (
    NAMETREE * newtree,
    NAMETREE * oldtree,
    WORD node,
    WORD par ) [extern]
```

Definition at line 696 of file [names.c](#).

4.19.6.114 CompactifyTree()

```
int CompactifyTree (
    NAMETREE * nametree,
    WORD par ) [extern]
```

Definition at line 634 of file [names.c](#).

4.19.6.115 MakeNameTree()

```
NAMETREE * MakeNameTree (
    void ) [extern]
```

Definition at line 815 of file [names.c](#).

4.19.6.116 FreeNameTree()

```
void FreeNameTree (
    NAMETREE * n ) [extern]
```

Definition at line 834 of file [names.c](#).

4.19.6.117 AddExpression()

```
int AddExpression (
    UBYTE * name,
    int x,
    int y ) [extern]
```

Definition at line 2746 of file [names.c](#).

4.19.6.118 AddSymbol()

```
int AddSymbol (
    UBYTE * name,
    int minpow,
    int maxpow,
    int cplx,
    int dim ) [extern]
```

Definition at line 920 of file [names.c](#).

4.19.6.119 AddDollar()

```
int AddDollar (
    UBYTE * name,
    WORD type,
    WORD * start,
    LONG size ) [extern]
```

Definition at line 2604 of file [names.c](#).

4.19.6.120 ReplaceDollar()

```
int ReplaceDollar (
    WORD number,
    WORD newtype,
    WORD * newstart,
    LONG newsize ) [extern]
```

Definition at line [2648](#) of file [names.c](#).

4.19.6.121 DollarRaiseLow()

```
int DollarRaiseLow (
    UBYTE * name,
    LONG value ) [extern]
```

Definition at line [2552](#) of file [dollar.c](#).

4.19.6.122 AddVector()

```
int AddVector (
    UBYTE * name,
    int cplx,
    int dim ) [extern]
```

Definition at line [1244](#) of file [names.c](#).

4.19.6.123 AddDubious()

```
int AddDubious (
    UBYTE * name ) [extern]
```

Definition at line [2681](#) of file [names.c](#).

4.19.6.124 AddIndex()

```
int AddIndex (
    UBYTE * name,
    int dim,
    int dim4 ) [extern]
```

Definition at line [1088](#) of file [names.c](#).

4.19.6.125 DoDimension()

```
UBYTE * DoDimension (
    UBYTE * s,
    int * dim,
    int * dim4 ) [extern]
```

Definition at line [1160](#) of file [names.c](#).

4.19.6.126 AddFunction()

```
int AddFunction (
    UBYTE * name,
    int comm,
    int istensor,
    int cplx,
    int symprop,
    int dim,
    int argmax,
    int argmin ) [extern]
```

Definition at line [1326](#) of file [names.c](#).

4.19.6.127 CoCommuteInSet()

```
int CoCommuteInSet (
    UBYTE * s ) [extern]
```

Definition at line [1356](#) of file [names.c](#).

4.19.6.128 CoFunction()

```
int CoFunction (
    UBYTE * s,
    int comm,
    int istensor ) [extern]
```

Definition at line [1446](#) of file [names.c](#).

4.19.6.129 TestName()

```
int TestName (
    UBYTE * name ) [extern]
```

Definition at line [3256](#) of file [names.c](#).

4.19.6.130 AddSet()

```
int AddSet (
    UBYTE * name,
    WORD dim ) [extern]
```

Definition at line [2124](#) of file [names.c](#).

4.19.6.131 DoElements()

```
int DoElements (
    UBYTE * s,
    SETS set,
    UBYTE * name ) [extern]
```

Definition at line [2157](#) of file [names.c](#).

4.19.6.132 DoTempSet()

```
int DoTempSet (
    UBYTE * from,
    UBYTE * to ) [extern]
```

Definition at line [2482](#) of file [names.c](#).

4.19.6.133 NameConflict()

```
int NameConflict (
    int type,
    UBYTE * name ) [extern]
```

Definition at line [2730](#) of file [names.c](#).

4.19.6.134 OpenFile()

```
int OpenFile (
    char * name ) [extern]
```

Definition at line [985](#) of file [tools.c](#).

4.19.6.135 OpenAddFile()

```
int OpenAddFile (
    char * name ) [extern]
```

Definition at line [1004](#) of file [tools.c](#).

4.19.6.136 ReOpenFile()

```
int ReOpenFile (
    char * name ) [extern]
```

Definition at line [1025](#) of file [tools.c](#).

4.19.6.137 CreateFile()

```
int CreateFile (
    char * name ) [extern]
```

Definition at line [1045](#) of file [tools.c](#).

4.19.6.138 CreateLogFile()

```
int CreateLogFile (
    char * name ) [extern]
```

Definition at line [1062](#) of file [tools.c](#).

4.19.6.139 CloseFile()

```
void CloseFile (
    int handle ) [extern]
```

Definition at line 1080 of file [tools.c](#).

4.19.6.140 CopyFile()

```
int CopyFile (
    char * source,
    char * dest ) [extern]
```

Copies a file with name *source to a file named *dest. The involved files must not be open. Returns non-zero if an error occurred. Uses if possible the combined large and small sorting buffers as cache.

Definition at line 1103 of file [tools.c](#).

Referenced by [DoRecovery\(\)](#).

4.19.6.141 CreateHandle()

```
int CreateHandle (
    void ) [extern]
```

Definition at line 1164 of file [tools.c](#).

4.19.6.142 ReadFile()

```
LONG ReadFile (
    int handle,
    UBYTE * buffer,
    LONG size ) [extern]
```

Definition at line 1219 of file [tools.c](#).

4.19.6.143 ReadPosFile()

```
LONG ReadPosFile (
    PHEAD FILEHANDLE * fi,
    UBYTE * buffer,
    LONG size,
    POSITION * pos ) [extern]
```

Definition at line 1269 of file [tools.c](#).

4.19.6.144 WriteFileToFile()

```
LONG WriteFileToFile (
    int handle,
    UBYTE * buffer,
    LONG size ) [extern]
```

Definition at line [1335](#) of file [tools.c](#).

4.19.6.145 SeekFile()

```
void SeekFile (
    int handle,
    POSITION * offset,
    int origin ) [extern]
```

Definition at line [1373](#) of file [tools.c](#).

4.19.6.146 TellFile()

```
LONG TellFile (
    int handle ) [extern]
```

Definition at line [1398](#) of file [tools.c](#).

4.19.6.147 FlushFile()

```
void FlushFile (
    int handle ) [extern]
```

Definition at line [1422](#) of file [tools.c](#).

4.19.6.148 GetPosFile()

```
int GetPosFile (
    int handle,
    fpos_t * pospointer ) [extern]
```

Definition at line [1436](#) of file [tools.c](#).

4.19.6.149 SetPosFile()

```
int SetPosFile (
    int handle,
    fpos_t * pospointer ) [extern]
```

Definition at line [1450](#) of file [tools.c](#).

4.19.6.150 SynchFile()

```
void SynchFile (
    int handle ) [extern]
```

Definition at line 1470 of file [tools.c](#).

4.19.6.151 TruncateFile()

```
void TruncateFile (
    int handle ) [extern]
```

Definition at line 1492 of file [tools.c](#).

4.19.6.152 GetChannel()

```
int GetChannel (
    char * name,
    int mode ) [extern]
```

Definition at line 1512 of file [tools.c](#).

4.19.6.153 GetAppendChannel()

```
int GetAppendChannel (
    char * name ) [extern]
```

Definition at line 1548 of file [tools.c](#).

4.19.6.154 CloseChannel()

```
int CloseChannel (
    char * name ) [extern]
```

Definition at line 1578 of file [tools.c](#).

4.19.6.155 inictable()

```
void inictable (
    void ) [extern]
```

Definition at line 315 of file [compiler.c](#).

4.19.6.156 findcommand()

```
KEYWORD * findcommand (
    UBYTE * in ) [extern]
```

Definition at line 341 of file [compiler.c](#).

4.19.6.157 inicbufs()

```
int inicbufs (
    void ) [extern]
```

Creates a new compiler buffer and returns its ID number.

Returns

The ID number for the new compiler buffer.

Definition at line 47 of file [comtool.c](#).

References [CbUf::boomlijst](#), [CbUf::Buffer](#), [CbUf::BufferSize](#), [CbUf::CanCommu](#), [CbUf::dimension](#), [CbUf::lhs](#), [CbUf::numdum](#), [CbUf::NumTerms](#), [CbUf::Pointer](#), [CbUf::rhs](#), and [CbUf::Top](#).

Referenced by [AddRHS\(\)](#), and [StartVariables\(\)](#).

4.19.6.158 StartFiles()

```
void StartFiles (
    void ) [extern]
```

Definition at line 958 of file [tools.c](#).

4.19.6.159 MakeDate()

```
UBYTE * MakeDate (
    void ) [extern]
```

Definition at line 2117 of file [tools.c](#).

4.19.6.160 PreProcessor()

```
void PreProcessor (
    void ) [extern]
```

Definition at line 953 of file [pre.c](#).

4.19.6.161 FromList()

```
void * FromList (
    LIST * L ) [extern]
```

Definition at line 2716 of file [tools.c](#).

4.19.6.162 From0List()

```
void * From0List (
    LIST * L ) [extern]
```

Definition at line 2742 of file [tools.c](#).

4.19.6.163 FromVarList()

```
void * FromVarList (
    LIST * L ) [extern]
```

Definition at line 2771 of file [tools.c](#).

4.19.6.164 DoubleList()

```
int DoubleList (
    void *** lijst,
    int * oldsize,
    int objectsize,
    char * nameoftype ) [extern]
```

Definition at line 2813 of file [tools.c](#).

4.19.6.165 DoubleLList()

```
int DoubleLList (
    void *** lijst,
    LONG * oldsize,
    int objectsize,
    char * nameoftype ) [extern]
```

Definition at line 2865 of file [tools.c](#).

4.19.6.166 DoubleBuffer()

```
void DoubleBuffer (
    void ** start,
    void ** stop,
    int size,
    char * text ) [extern]
```

Definition at line 2913 of file [tools.c](#).

4.19.6.167 ExpandBuffer()

```
void ExpandBuffer (
    void ** buffer,
    LONG * oldsize,
    int type ) [extern]
```

Definition at line 2938 of file [tools.c](#).

4.19.6.168 iexp()

```
LONG iexp (
    LONG x,
    int p ) [extern]
```

Definition at line 2962 of file [tools.c](#).

4.19.6.169 IsLikeVector()

```
int IsLikeVector (
    WORD * arg ) [extern]
```

Definition at line 3165 of file [tools.c](#).

4.19.6.170 AreArgsEqual()

```
int AreArgsEqual (
    WORD * arg1,
    WORD * arg2 ) [extern]
```

Definition at line 3193 of file [tools.c](#).

4.19.6.171 CompareArgs()

```
int CompareArgs (
    WORD * arg1,
    WORD * arg2 ) [extern]
```

Definition at line 3212 of file [tools.c](#).

4.19.6.172 SkipField()

```
UBYTE * SkipField (
    UBYTE * s,
    int level ) [extern]
```

Skips from *s* to the end of a declaration field (e.g., *x*, *1+2*[x+a]*, or *f ({x, [x+a] }, 1)*).

Parameters

in	<i>s</i>	Pointer to a null-terminated input buffer.
in	<i>level</i>	Number of parentheses that still have to be closed.

Returns

Pointer to the character after the field, which is either a comma at level 0 or the null terminator.

Definition at line 1976 of file [tools.c](#).

4.19.6.173 StrCmp()

```
int StrCmp (
    UBYTE * s1,
    UBYTE * s2 ) [extern]
```

Definition at line 1702 of file [tools.c](#).

4.19.6.174 StrICmp()

```
int StrICmp (
    UBYTE * s1,
    UBYTE * s2 ) [extern]
```

Definition at line 1713 of file [tools.c](#).

4.19.6.175 StrHICmp()

```
int StrHICmp (
    UBYTE * s1,
    UBYTE * s2 ) [extern]
```

Definition at line 1724 of file [tools.c](#).

4.19.6.176 StrICont()

```
int StrICont (
    UBYTE * s1,
    UBYTE * s2 ) [extern]
```

Definition at line 1735 of file [tools.c](#).

4.19.6.177 CmpArray()

```
int CmpArray (
    WORD * t1,
    WORD * t2,
    WORD n ) [extern]
```

Definition at line 1747 of file [tools.c](#).

4.19.6.178 ConWord()

```
int ConWord (
    UBYTE * s1,
    UBYTE * s2 ) [extern]
```

Definition at line 1761 of file [tools.c](#).

4.19.6.179 StrLen()

```
int StrLen (
    UBYTE * s ) [extern]
```

Definition at line 1773 of file [tools.c](#).

4.19.6.180 GetPreVar()

```
UBYTE * GetPreVar (
    UBYTE * name,
    int flag ) [extern]
```

Definition at line 542 of file [pre.c](#).

4.19.6.181 ToGeneral()

```
void ToGeneral (
    WORD * r,
    WORD * m,
    WORD par ) [extern]
```

Definition at line 2993 of file [tools.c](#).

4.19.6.182 ToPolyFunGeneral()

```
WORD ToPolyFunGeneral (
    PHEAD WORD * term ) [extern]
```

Definition at line 3090 of file [tools.c](#).

4.19.6.183 ToFast()

```
int ToFast (
    WORD * r,
    WORD * m ) [extern]
```

Definition at line 3037 of file [tools.c](#).

4.19.6.184 GetSetupPar()

```
SETUPPARAMETERS * GetSetupPar (
    UBYTE * s ) [extern]
```

Definition at line 337 of file [setfile.c](#).

4.19.6.185 RecalcSetups()

```
int RecalcSetups (
    void ) [extern]
```

Definition at line [358](#) of file [setfile.c](#).

4.19.6.186 AllocSetups()

```
int AllocSetups (
    void ) [extern]
```

Definition at line [405](#) of file [setfile.c](#).

4.19.6.187 AllocSort()

```
SORTING * AllocSort (
    LONG inLargeSize,
    LONG inSmallSize,
    LONG inSmallEsize,
    LONG inTermsInSmall,
    int inMaxPatches,
    int inMaxFpatches,
    LONG inIOSize,
    int level ) [extern]
```

Definition at line [858](#) of file [setfile.c](#).

4.19.6.188 AllocSortFileName()

```
void AllocSortFileName (
    SORTING * sort ) [extern]
```

Definition at line [1008](#) of file [setfile.c](#).

4.19.6.189 LoadInputFile()

```
UBYTE * LoadInputFile (
    UBYTE * filename,
    int type ) [extern]
```

Definition at line [112](#) of file [tools.c](#).

4.19.6.190 GetInput()

```
UBYTE GetInput (
    void ) [extern]
```

Definition at line [138](#) of file [pre.c](#).

4.19.6.191 ClearPushback()

```
void ClearPushback (
    void ) [extern]
```

Definition at line 167 of file [pre.c](#).

4.19.6.192 GetChar()

```
UBYTE GetChar (
    int level ) [extern]
```

Definition at line 185 of file [pre.c](#).

4.19.6.193 CharOut()

```
void CharOut (
    UBYTE c ) [extern]
```

Definition at line 495 of file [pre.c](#).

4.19.6.194 UnsetAllowDelay()

```
void UnsetAllowDelay (
    void ) [extern]
```

Definition at line 516 of file [pre.c](#).

4.19.6.195 PopPreVars()

```
void PopPreVars (
    int tonumber ) [extern]
```

Definition at line 826 of file [pre.c](#).

4.19.6.196 IniModule()

```
void IniModule (
    int type ) [extern]
```

Definition at line 841 of file [pre.c](#).

4.19.6.197 IniSpecialModule()

```
void IniSpecialModule (
    int type ) [extern]
```

Definition at line 943 of file [pre.c](#).

4.19.6.198 ModuleInstruction()

```
int ModuleInstruction (
    int * moduletype,
    int * specialtype ) [extern]
```

Definition at line 76 of file [module.c](#).

4.19.6.199 PreProInstruction()

```
int PreProInstruction (
    void ) [extern]
```

Definition at line 1172 of file [pre.c](#).

4.19.6.200 LoadInstruction()

```
int LoadInstruction (
    int mode ) [extern]
```

Definition at line 1260 of file [pre.c](#).

4.19.6.201 LoadStatement()

```
int LoadStatement (
    int type ) [extern]
```

Definition at line 1463 of file [pre.c](#).

4.19.6.202 FindKeyWord()

```
KEYWORD * FindKeyWord (
    UBYTE * theword,
    KEYWORD * table,
    int size ) [extern]
```

Definition at line 1969 of file [pre.c](#).

4.19.6.203 FindInKeyWord()

```
KEYWORD * FindInKeyWord (
    UBYTE * theword,
    KEYWORD * table,
    int size ) [extern]
```

Definition at line 2000 of file [pre.c](#).

4.19.6.204 DoDefine()

```
int DoDefine (
    UBYTE * s ) [extern]
```

Definition at line 2165 of file [pre.c](#).

4.19.6.205 DoRedefine()

```
int DoRedefine (
    UBYTE * s ) [extern]
```

Definition at line 2175 of file [pre.c](#).

4.19.6.206 TheDefine()

```
int TheDefine (
    UBYTE * s,
    int mode ) [extern]
```

Preprocessor assignment. Possible arguments and values are treated and the new preprocessor variable is put into the name administration.

Parameters

<i>s</i>	Pointer to the character string following the preprocessor command.
<i>mode</i>	Bitmask. 0-bit clear: always create a new name entry, 0-bit set: try to redefine an existing name, 1-bit set: ignore preprocessor if/switch status.

Returns

zero: no errors, negative number: errors.

Definition at line 2030 of file [pre.c](#).

References [PutPreVar\(\)](#).

Here is the call graph for this function:



4.19.6.207 TheUndefine()

```
int TheUndefine (
    UBYTE * name ) [extern]
```

Definition at line 2215 of file [pre.c](#).

4.19.6.208 ClearMacro()

```
int ClearMacro (
    UBYTE * name ) [extern]
```

Definition at line 2187 of file [pre.c](#).

4.19.6.209 DoUndefine()

```
int DoUndefine (
    UBYTE * s ) [extern]
```

Definition at line 2269 of file [pre.c](#).

4.19.6.210 DoInclude()

```
int DoInclude (
    UBYTE * s ) [extern]
```

Definition at line 2321 of file [pre.c](#).

4.19.6.211 DoReverseInclude()

```
int DoReverseInclude (
    UBYTE * s ) [extern]
```

Definition at line 2328 of file [pre.c](#).

4.19.6.212 Include()

```
int Include (
    UBYTE * s,
    int type ) [extern]
```

Definition at line 2335 of file [pre.c](#).

4.19.6.213 DoExternal()

```
int DoExternal (
    UBYTE * s ) [extern]
```

Definition at line 5502 of file [pre.c](#).

4.19.6.214 DoToExternal()

```
int DoToExternal (
    UBYTE * s ) [extern]
```

Definition at line 6064 of file [pre.c](#).

4.19.6.215 DoFromExternal()

```
int DoFromExternal (
    UBYTE * s ) [extern]
```

Definition at line 5869 of file [pre.c](#).

4.19.6.216 DoPrompt()

```
int DoPrompt (
    UBYTE * s ) [extern]
```

Definition at line 5584 of file [pre.c](#).

4.19.6.217 DoSetExternal()

```
int DoSetExternal (
    UBYTE * s ) [extern]
```

Definition at line 5617 of file [pre.c](#).

4.19.6.218 DoSetExternalAttr()

```
int DoSetExternalAttr (
    UBYTE * s ) [extern]
```

Definition at line 5687 of file [pre.c](#).

4.19.6.219 DoRmExternal()

```
int DoRmExternal (
    UBYTE * s ) [extern]
```

Definition at line 5804 of file [pre.c](#).

4.19.6.220 DoFactDollar()

```
int DoFactDollar (
    UBYTE * s ) [extern]
```

Definition at line 6711 of file [pre.c](#).

4.19.6.221 GetDollarNumber()

```
WORD GetDollarNumber (
    UBYTE ** inp,
    DOLLARS d ) [extern]
```

Definition at line 6748 of file [pre.c](#).

4.19.6.222 DoSetRandom()

```
int DoSetRandom (
    UBYTE * s ) [extern]
```

Definition at line 6838 of file [pre.c](#).

4.19.6.223 DoOptimize()

```
int DoOptimize (
    UBYTE * s ) [extern]
```

Definition at line 6881 of file [pre.c](#).

4.19.6.224 DoClearOptimize()

```
int DoClearOptimize (
    UBYTE * s ) [extern]
```

Definition at line 7014 of file [pre.c](#).

4.19.6.225 DoSkipExtraSymbols()

```
int DoSkipExtraSymbols (
    UBYTE * s ) [extern]
```

Definition at line 7034 of file [pre.c](#).

4.19.6.226 DoTimeOutAfter()

```
int DoTimeOutAfter (
    UBYTE * s ) [extern]
```

Definition at line 7260 of file [pre.c](#).

4.19.6.227 DoMessage()

```
int DoMessage (
    UBYTE * s ) [extern]
```

Definition at line 3739 of file [pre.c](#).

4.19.6.228 DoPreOut()

```
int DoPreOut (
    UBYTE * s ) [extern]
```

Definition at line [3810](#) of file [pre.c](#).

4.19.6.229 DoPreAppend()

```
int DoPreAppend (
    UBYTE * s ) [extern]
```

Definition at line [3867](#) of file [pre.c](#).

4.19.6.230 DoPreCreate()

```
int DoPreCreate (
    UBYTE * s ) [extern]
```

Definition at line [3910](#) of file [pre.c](#).

4.19.6.231 DoPreAssign()

```
int DoPreAssign (
    UBYTE * s ) [extern]
```

Definition at line [2134](#) of file [pre.c](#).

4.19.6.232 DoPreBreak()

```
int DoPreBreak (
    UBYTE * s ) [extern]
```

Definition at line [4125](#) of file [pre.c](#).

4.19.6.233 DoPreDefault()

```
int DoPreDefault (
    UBYTE * s ) [extern]
```

Definition at line [4188](#) of file [pre.c](#).

4.19.6.234 DoPreSwitch()

```
int DoPreSwitch (
    UBYTE * s ) [extern]
```

Definition at line [4239](#) of file [pre.c](#).

4.19.6.235 DoPreEndSwitch()

```
int DoPreEndSwitch (
    UBYTE * s ) [extern]
```

Definition at line [4212](#) of file [pre.c](#).

4.19.6.236 DoPreCase()

```
int DoPreCase (
    UBYTE * s ) [extern]
```

Definition at line [4149](#) of file [pre.c](#).

4.19.6.237 DoPreShow()

```
int DoPreShow (
    UBYTE * s ) [extern]
```

Definition at line [4293](#) of file [pre.c](#).

4.19.6.238 DoPreExchange()

```
int DoPreExchange (
    UBYTE * s ) [extern]
```

Definition at line [2496](#) of file [pre.c](#).

4.19.6.239 DoSystem()

```
int DoSystem (
    UBYTE * s ) [extern]
```

Definition at line [4332](#) of file [pre.c](#).

4.19.6.240 DoPipe()

```
int DoPipe (
    UBYTE * s ) [extern]
```

Definition at line [3753](#) of file [pre.c](#).

4.19.6.241 StartPrepro()

```
void StartPrepro (
    void ) [extern]
```

Definition at line [4510](#) of file [pre.c](#).

4.19.6.242 DoIfdef()

```
int DoIfdef (
    UBYTE * s,
    int par ) [extern]
```

Definition at line 3454 of file [pre.c](#).

4.19.6.243 DoIfydef()

```
int DoIfydef (
    UBYTE * s ) [extern]
```

Definition at line 3479 of file [pre.c](#).

4.19.6.244 DoIfndef()

```
int DoIfndef (
    UBYTE * s ) [extern]
```

Definition at line 3489 of file [pre.c](#).

4.19.6.245 DoElse()

```
int DoElse (
    UBYTE * s ) [extern]
```

Definition at line 3157 of file [pre.c](#).

4.19.6.246 DoElseif()

```
int DoElseif (
    UBYTE * s ) [extern]
```

Definition at line 3194 of file [pre.c](#).

4.19.6.247 DoEndif()

```
int DoEndif (
    UBYTE * s ) [extern]
```

Definition at line 3375 of file [pre.c](#).

4.19.6.248 DoTerminate()

```
int DoTerminate (
    UBYTE * s ) [extern]
```

Definition at line 2766 of file [pre.c](#).

4.19.6.249 DoIf()

```
int DoIf (
    UBYTE * s ) [extern]
```

Definition at line [3430](#) of file [pre.c](#).

4.19.6.250 DoCall()

```
int DoCall (
    UBYTE * s ) [extern]
```

Definition at line [2557](#) of file [pre.c](#).

4.19.6.251 DoDebug()

```
int DoDebug (
    UBYTE * s ) [extern]
```

Definition at line [2729](#) of file [pre.c](#).

4.19.6.252 DoContinueDo()

```
int DoContinueDo (
    UBYTE * s ) [extern]
```

Jumps forward to the corresponding `#enddo` of the specified number of outer `#do` loops.

Syntax:

```
#continuedo [<number>=1]
```

If `number` is omitted, it defaults to 1. If `number` is zero then the instruction has no effect.

Definition at line [2800](#) of file [pre.c](#).

4.19.6.253 DoDo()

```
int DoDo (
    UBYTE * s ) [extern]
```

Definition at line [2864](#) of file [pre.c](#).

4.19.6.254 DoBreakDo()

```
int DoBreakDo (
    UBYTE * s ) [extern]
```

Definition at line [3079](#) of file [pre.c](#).

4.19.6.255 DoEnddo()

```
int DoEnddo (
    UBYTE * s ) [extern]
```

Definition at line [3229](#) of file [pre.c](#).

4.19.6.256 DoEndprocedure()

```
int DoEndprocedure (
    UBYTE * s ) [extern]
```

Definition at line [3403](#) of file [pre.c](#).

4.19.6.257 DoInside()

```
int DoInside (
    UBYTE * s ) [extern]
```

Definition at line [3514](#) of file [pre.c](#).

4.19.6.258 DoEndInside()

```
int DoEndInside (
    UBYTE * s ) [extern]
```

Definition at line [3607](#) of file [pre.c](#).

4.19.6.259 DoProcedure()

```
int DoProcedure (
    UBYTE * s ) [extern]
```

Definition at line [4068](#) of file [pre.c](#).

4.19.6.260 DoPrePrintTimes()

```
int DoPrePrintTimes (
    UBYTE * s ) [extern]
```

Definition at line [3833](#) of file [pre.c](#).

4.19.6.261 DoPreWrite()

```
int DoPreWrite (
    UBYTE * s ) [extern]
```

Definition at line [4027](#) of file [pre.c](#).

4.19.6.262 DoPreClose()

```
int DoPreClose (
    UBYTE * s ) [extern]
```

Definition at line [3983](#) of file [pre.c](#).

4.19.6.263 DoPreRemove()

```
int DoPreRemove (
    UBYTE * s ) [extern]
```

Definition at line [3950](#) of file [pre.c](#).

4.19.6.264 DoPreSortReallocate()

```
int DoPreSortReallocate (
    UBYTE * s ) [extern]
```

Definition at line [3847](#) of file [pre.c](#).

4.19.6.265 DoCommentChar()

```
int DoCommentChar (
    UBYTE * s ) [extern]
```

Definition at line [2103](#) of file [pre.c](#).

4.19.6.266 DoPrCExtension()

```
int DoPrCExtension (
    UBYTE * s ) [extern]
```

Definition at line [3776](#) of file [pre.c](#).

4.19.6.267 DoPreReset()

```
int DoPreReset (
    UBYTE * s ) [extern]
```

Definition at line [7073](#) of file [pre.c](#).

4.19.6.268 WriteString()

```
void WriteString (
    int type,
    UBYTE * str,
    int num ) [extern]
```

Definition at line [1809](#) of file [tools.c](#).

4.19.6.269 WriteUnfinString()

```
void WriteUnfinString (
    int type,
    UBYTE * str,
    int num ) [extern]
```

Definition at line 1843 of file [tools.c](#).

4.19.6.270 AddToString()

```
UBYTE * AddToString (
    UBYTE * outstring,
    UBYTE * extrastring,
    int par ) [extern]
```

Definition at line 1868 of file [tools.c](#).

4.19.6.271 PreCalc()

```
UBYTE * PreCalc (
    void ) [extern]
```

Definition at line 5191 of file [pre.c](#).

4.19.6.272 PreEval()

```
UBYTE * PreEval (
    UBYTE * s,
    LONG * x ) [extern]
```

Definition at line 5269 of file [pre.c](#).

4.19.6.273 NumToStr()

```
void NumToStr (
    UBYTE * s,
    LONG x ) [extern]
```

Definition at line 1785 of file [tools.c](#).

4.19.6.274 PreCmp()

```
int PreCmp (
    int type,
    int val,
    UBYTE * t,
    int type2,
    int val2,
    UBYTE * t2,
    int cmpop ) [extern]
```

Definition at line 4702 of file [pre.c](#).

4.19.6.275 PreEq()

```
int PreEq (
    int type,
    int val,
    UBYTE * t,
    int type2,
    int val2,
    UBYTE * t2,
    int egop ) [extern]
```

Definition at line [4726](#) of file [pre.c](#).

4.19.6.276 pParseObject()

```
UBYTE * pParseObject (
    UBYTE * s,
    int * type,
    LONG * val2 ) [extern]
```

Definition at line [4755](#) of file [pre.c](#).

4.19.6.277 PreIfEval()

```
UBYTE * PreIfEval (
    UBYTE * s,
    int * value ) [extern]
```

Definition at line [4573](#) of file [pre.c](#).

4.19.6.278 EvalPreIf()

```
int EvalPreIf (
    UBYTE * s ) [extern]
```

Definition at line [4538](#) of file [pre.c](#).

4.19.6.279 PreLoad()

```
int PreLoad (
    PRELOAD * p,
    UBYTE * start,
    UBYTE * stop,
    int mode,
    char * message ) [extern]
```

Definition at line [4372](#) of file [pre.c](#).

4.19.6.280 PreSkip()

```
int PreSkip (
    UBYTE * start,
    UBYTE * stop,
    int mode ) [extern]
```

Definition at line 4455 of file [pre.c](#).

4.19.6.281 EndOfToken()

```
UBYTE * EndOfToken (
    UBYTE * s ) [extern]
```

Skips over alphanumeric characters to find the end of the token.

Note

This function does not handle formal names (e.g., `[x+a]`). To handle them, use `SkipAName` instead.

Parameters

in	s	Pointer to a null-terminated input buffer.

Returns

Pointer to the first non-alphanumeric character (i.e., the character immediately after the token), or the null terminator if the token reaches the end of the string.

Definition at line 1932 of file [tools.c](#).

4.19.6.282 SetSpecialMode()

```
void SetSpecialMode (
    int moduletype,
    int specialtype ) [extern]
```

Definition at line 257 of file [module.c](#).

4.19.6.283 MakeGlobal()

```
void MakeGlobal (
    void ) [extern]
```

Definition at line 383 of file [execute.c](#).

4.19.6.284 ExecModule()

```
int ExecModule (
    int moduletype ) [extern]
```

Definition at line 274 of file [module.c](#).

4.19.6.285 ExecStore()

```
int ExecStore (
    void ) [extern]
```

Definition at line 299 of file [module.c](#).

4.19.6.286 FullCleanUp()

```
void FullCleanUp (
    void ) [extern]
```

Definition at line 314 of file [module.c](#).

4.19.6.287 DoExecStatement()

```
int DoExecStatement (
    void ) [extern]
```

Definition at line 780 of file [module.c](#).

4.19.6.288 DoPipeStatement()

```
int DoPipeStatement (
    void ) [extern]
```

Definition at line 797 of file [module.c](#).

4.19.6.289 DoPolyfun()

```
int DoPolyfun (
    UBYTE * s ) [extern]
```

Definition at line 395 of file [module.c](#).

4.19.6.290 DoPolyratfun()

```
int DoPolyratfun (
    UBYTE * s ) [extern]
```

Definition at line 458 of file [module.c](#).

4.19.6.291 CompileStatement()

```
int CompileStatement (
    UBYTE * in ) [extern]
```

Definition at line 572 of file [compiler.c](#).

4.19.6.292 ToToken()

```
UBYTE * ToToken (
    UBYTE * s ) [extern]
```

Skips over non-alphanumeric characters to find the start of a token.

Note

This function does not handle formal names (e.g., `[x+a]`). To handle them, consider simply skipping whitespace, e.g., by using `SkipSpaces`.

Parameters

<code>in</code>	<code>s</code>	Pointer to a null-terminated input buffer.

Returns

Pointer to the first alphanumeric character (i.e., the start of the token), or the null terminator if none is found.

Definition at line 1955 of file [tools.c](#).

4.19.6.293 GetDollar()

```
int GetDollar (
    UBYTE * name ) [extern]
```

Definition at line 590 of file [names.c](#).

4.19.6.294 MesWork()

```
void MesWork (
    void ) [extern]
```

Definition at line 89 of file [message.c](#).

4.19.6.295 MesPrint()

```
int MesPrint (
    const char * fmt,
    ... ) [extern]
```

Definition at line 135 of file [message.c](#).

4.19.6.296 MesCall()

```
int MesCall (
    char * s ) [extern]
```

Definition at line 802 of file [message.c](#).

4.19.6.297 NumCopy()

```
UBYTE * NumCopy (
    WORD y,
    UBYTE * to ) [extern]
```

Definition at line 2027 of file [tools.c](#).

4.19.6.298 LongCopy()

```
char * LongCopy (
    LONG y,
    char * to ) [extern]
```

Definition at line 2054 of file [tools.c](#).

4.19.6.299 LongLongCopy()

```
char * LongLongCopy (
    off_t * y,
    char * to ) [extern]
```

Definition at line 2081 of file [tools.c](#).

4.19.6.300 ReserveTempFiles()

```
void ReserveTempFiles (
    int par ) [extern]
```

Definition at line 666 of file [startup.c](#).

4.19.6.301 PrintTerm()

```
void PrintTerm (
    WORD * term,
    char * where ) [extern]
```

Definition at line 845 of file [message.c](#).

4.19.6.302 PrintTermC()

```
void PrintTermC (
    WORD * term,
    char * where ) [extern]
```

Definition at line 875 of file [message.c](#).

4.19.6.303 PrintSubTerm()

```
void PrintSubTerm (
    WORD * term,
    char * where ) [extern]
```

Definition at line 909 of file [message.c](#).

4.19.6.304 PrintWords()

```
void PrintWords (
    WORD * buffer,
    LONG number ) [extern]
```

Definition at line 931 of file [message.c](#).

4.19.6.305 PrintSeq()

```
void PrintSeq (
    WORD * a,
    char * text ) [extern]
```

Definition at line 949 of file [message.c](#).

4.19.6.306 ExpandTripleDots()

```
int ExpandTripleDots (
    int par ) [extern]
```

Definition at line 1601 of file [pre.c](#).

4.19.6.307 ComPress()

```
LONG ComPress (
    WORD ** ss,
    LONG * n ) [extern]
```

Gets a list of pointers to terms and compresses the terms. In *n* it collects the number of terms and the return value of the function is the space that is occupied.

We have to pay some special attention to the compression of terms with a PolyFun. This PolyFun should occur only straight before the coefficient, so we can use the same trick as for the coefficient to sabotage compression of this object (Replace in the history the function pointer by zero. This is safe, because terms that would be identical otherwise would have been added).

Parameters

<i>ss</i>	Array of pointers to terms to be compressed.
<i>n</i>	Number of pointers in <i>ss</i> .

Returns

Total number of words needed for the compressed result.

Definition at line [3206](#) of file [sort.c](#).

Referenced by [EndSort\(\)](#), and [StoreTerm\(\)](#).

4.19.6.308 StageSort()

```
void StageSort (
    FILEHANDLE * fout ) [extern]
```

Prepares a stage 4 or higher sort. Stage 4 sorts occur when the sort file contains more patches than can be merged in one pass.

Definition at line [4664](#) of file [sort.c](#).

References [FiLe::handle](#).

Referenced by [EndSort\(\)](#), and [MergePatches\(\)](#).

4.19.6.309 M_free()

```
void M_free (
    void * x,
    const char * where ) [extern]
```

Definition at line [2319](#) of file [tools.c](#).

4.19.6.310 ClearWildcardNames()

```
void ClearWildcardNames (
    void ) [extern]
```

Definition at line [849](#) of file [names.c](#).

4.19.6.311 AddWildcardName()

```
int AddWildcardName (
    UBYTE * name ) [extern]
```

Definition at line [854](#) of file [names.c](#).

4.19.6.312 GetWildcardName()

```
int GetWildcardName (
    UBYTE * name ) [extern]
```

Definition at line 898 of file [names.c](#).

4.19.6.313 Globalize()

```
void Globalize (
    int par ) [extern]
```

Definition at line 3157 of file [names.c](#).

4.19.6.314 ResetVariables()

```
void ResetVariables (
    int par ) [extern]
```

Definition at line 2834 of file [names.c](#).

4.19.6.315 AddToPreTypes()

```
void AddToPreTypes (
    int type ) [extern]
```

Definition at line 5419 of file [pre.c](#).

4.19.6.316 MessPreNesting()

```
void MessPreNesting (
    int par ) [extern]
```

Definition at line 5437 of file [pre.c](#).

4.19.6.317 GetStreamPosition()

```
LONG GetStreamPosition (
    STREAM * stream ) [extern]
```

Definition at line 815 of file [tools.c](#).

4.19.6.318 DoubleCbuffer()

```
WORD * DoubleCbuffer (
    int num,
    WORD * w,
    int par ) [extern]
```

Doubles a compiler buffer.

Parameters

<i>num</i>	The ID number for the buffer to be doubled.
<i>w</i>	The pointer to the end (exclusive) of the current buffer. The contents in the range of [cbuff[num].Buffer,w) will be kept.

Definition at line 143 of file [comtool.c](#).

References [CbUf::Buffer](#), [CbUf::BufferSize](#), [CbUf::lhs](#), [CbUf::Pointer](#), [CbUf::rhs](#), and [CbUf::Top](#).

Referenced by [AddNtoC\(\)](#), and [AddNtoL\(\)](#).

4.19.6.319 AddLHS()

```
WORD * AddLHS (
    int num ) [extern]
```

Adds an LHS to a compiler buffer and returns the pointer to a buffer for the new LHS.

Parameters

<i>num</i>	The ID number for the buffer to get another LHS.

Definition at line 188 of file [comtool.c](#).

References [CbUf::lhs](#), and [CbUf::Pointer](#).

Referenced by [AddNtoL\(\)](#).

4.19.6.320 AddRHS()

```
WORD * AddRHS (
    int num,
    int type ) [extern]
```

Adds an RHS to a compiler buffer and returns the pointer to a buffer for the new RHS.

Parameters

<i>num</i>	The ID number for the buffer to get another RHS.
<i>type</i>	If 0, the subexpression tree will be reallocated.

Definition at line 214 of file [comtool.c](#).

References [TaBIEs::buffers](#), [TaBIEs::buffersfill](#), [TaBIEs::bufferssize](#), [TaBIEs::bufnum](#), [CbUf::CanCommu](#), [CbUf::dimension](#), [inicbufs\(\)](#), [CbUf::numdum](#), [CbUf::NumTerms](#), [CbUf::Pointer](#), and [CbUf::rhs](#).

Referenced by [InsertArg\(\)](#), [StartVariables\(\)](#), and [TestMatch\(\)](#).

Here is the call graph for this function:



4.19.6.321 AddNtoL()

```
int AddNtoL (  
    int n,  
    WORD * array ) [extern]
```

Adds an LHS with the given data to the current compiler buffer.

Parameters

<i>n</i>	The length of the data.
<i>array</i>	The data to be added.

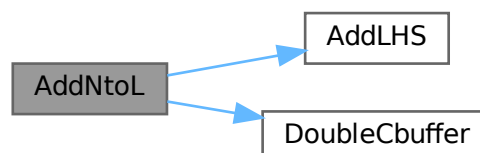
Returns

0 if succeeds.

Definition at line 288 of file [comtool.c](#).

References [AddLHS\(\)](#), [DoubleCbuffer\(\)](#), [CbUf::Pointer](#), and [CbUf::Top](#).

Here is the call graph for this function:



4.19.6.322 AddNtoC()

```
int AddNtoC (
    int bufnum,
    int n,
    WORD * array,
    int par ) [extern]
```

Adds the given data to the last LHS/RHS in a compiler buffer.

Parameters

<i>bufnum</i>	The ID number for the buffer where the data will be added.
<i>n</i>	The length of the data.
<i>array</i>	The data to be added.

Returns

0 if succeeds.

Definition at line 317 of file [comtool.c](#).

References [DoubleCbuffer\(\)](#), [CbUf::Pointer](#), and [CbUf::Top](#).

Referenced by [InsertArg\(\)](#), and [StartVariables\(\)](#).

Here is the call graph for this function:

**4.19.6.323 DoubleIfBuffers()**

```
void DoubleIfBuffers (
    void ) [extern]
```

Definition at line 1033 of file [if.c](#).

4.19.6.324 CreateStream()

```
STREAM * CreateStream (
    UBYTE * where ) [extern]
```

Definition at line 784 of file [tools.c](#).

4.19.6.325 setonoff()

```
int setonoff (
    UBYTE * s,
    int * flag,
    int onvalue,
    int offvalue ) [extern]
```

Definition at line 496 of file [compcomm.c](#).

4.19.6.326 DoPrint()

```
int DoPrint (
    UBYTE * s,
    int par ) [extern]
```

Definition at line 1047 of file [compcomm.c](#).

4.19.6.327 SetExpr()

```
int SetExpr (
    UBYTE * s,
    int setunset,
    int par ) [extern]
```

Definition at line 1518 of file [compcomm.c](#).

4.19.6.328 AddToCom()

```
void AddToCom (
    int n,
    WORD * array ) [extern]
```

Definition at line 1655 of file [compcomm.c](#).

4.19.6.329 Add2ComStrings()

```
int Add2ComStrings (
    int n,
    WORD * array,
    UBYTE * string1,
    UBYTE * string2 ) [extern]
```

Definition at line 1729 of file [compcomm.c](#).

4.19.6.330 DoSymmetrize()

```
int DoSymmetrize (
    UBYTE * s,
    int par ) [extern]
```

Definition at line 2240 of file [compcomm.c](#).

4.19.6.334 MakeInteger()

```
WORD * MakeInteger (
    PHEAD WORD * argin,
    WORD * argout,
    WORD * argfree ) [extern]
```

For normalizing everything to integers we have to determine for all elements of this argument the LCM of the denominators and the GCD of the numerators. The input argument is in argin. The number that comes out should go to argout. The new pointer in the argout buffer is the return value. The normalized argument is in argfree.

Definition at line 3318 of file [argument.c](#).

Referenced by [TakeArgContent\(\)](#).

4.19.6.335 MakeMod()

```
WORD * MakeMod (
    PHEAD WORD * argin,
    WORD * argout,
    WORD * argfree ) [extern]
```

Similar to MakeInteger but now with modulus arithmetic using only a one WORD 'prime'. We make the coefficient of the first term in the argument equal to one. Already the coefficients are taken modulus AN.cmod and AN.ncmod == 1

Definition at line 3489 of file [argument.c](#).

References [GetModInverses\(\)](#).

Referenced by [TakeArgContent\(\)](#).

Here is the call graph for this function:



4.19.6.336 FindArg()

```
WORD FindArg (
    PHEAD WORD * a ) [extern]
```

Looks the argument up in the (workers) table. If it is found the number in the table is returned (plus one to make it positive). If it is not found we look in the compiler provided table. If it is found - the number in the table is returned (minus one to make it negative). If in neither table we return zero.

Definition at line 2519 of file [argument.c](#).

4.19.6.337 InsertArg()

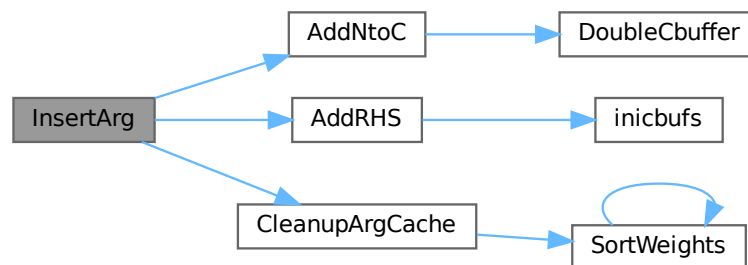
```
int InsertArg (
    PHEAD WORD * argin,
    WORD * argout,
    int par ) [extern]
```

Inserts the argument into the (workers) table. If the table is too full we eliminate half of it. The eliminated elements are the ones that have not been used most recently, weighted by their total use and age(?). If `par == 0` it inserts in the regular factorization cache If `par == 1` it inserts in the cache defined with the `FactorCache` statement

Definition at line 2543 of file [argument.c](#).

References [AddNtoC\(\)](#), [AddRHS\(\)](#), and [CleanupArgCache\(\)](#).

Here is the call graph for this function:



4.19.6.338 CleanupArgCache()

```
int CleanupArgCache (
    PHEAD WORD bufnum ) [extern]
```

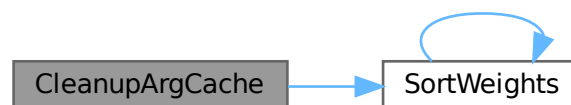
Cleans up the argument factorization cache. We throw half the elements. For a weight of what we want to keep we use the product of usage and the number in the buffer.

Definition at line 2578 of file [argument.c](#).

References [CbUf::boomlijst](#), [CbUf::Buffer](#), [CbUf::Pointer](#), [CbUf::rhs](#), [SortWeights\(\)](#), and [tree::usage](#).

Referenced by [InsertArg\(\)](#).

Here is the call graph for this function:



4.19.6.339 ArgSymbolMerge()

```
int ArgSymbolMerge (
    WORD * t1,
    WORD * t2 ) [extern]
```

Definition at line 2649 of file [argument.c](#).

4.19.6.340 ArgDotproductMerge()

```
int ArgDotproductMerge (
    WORD * t1,
    WORD * t2 ) [extern]
```

Definition at line 2704 of file [argument.c](#).

4.19.6.341 SortWeights()

```
void SortWeights (
    LONG * weights,
    LONG * extraspace,
    WORD number ) [extern]
```

Sorts an array of LONGS in the same way SplitMerge (in [sort.c](#)) works We use gradual division in two.

Definition at line 3534 of file [argument.c](#).

References [SortWeights\(\)](#).

Referenced by [CleanupArgCache\(\)](#), and [SortWeights\(\)](#).

Here is the call graph for this function:



4.19.6.342 DoBrackets()

```
int DoBrackets (
    UBYTE * inp,
    int par ) [extern]
```

Definition at line 3775 of file [compcomm.c](#).

4.19.6.343 DoPutInside()

```
int DoPutInside (
    UBYTE * inp,
    int par ) [extern]
```

Definition at line [7068](#) of file [compcomm.c](#).

4.19.6.344 CountComp()

```
WORD * CountComp (
    UBYTE * inp,
    WORD * to ) [extern]
```

Definition at line [4053](#) of file [compcomm.c](#).

4.19.6.345 CoAntiBracket()

```
int CoAntiBracket (
    UBYTE * inp ) [extern]
```

Definition at line [3910](#) of file [compcomm.c](#).

4.19.6.346 CoAntiSymmetrize()

```
int CoAntiSymmetrize (
    UBYTE * s ) [extern]
```

Definition at line [2367](#) of file [compcomm.c](#).

4.19.6.347 DoArgPlode()

```
int DoArgPlode (
    UBYTE * s,
    int par ) [extern]
```

Definition at line [5728](#) of file [compcomm.c](#).

4.19.6.348 CoArgExplode()

```
int CoArgExplode (
    UBYTE * s ) [extern]
```

Definition at line [5784](#) of file [compcomm.c](#).

4.19.6.349 CoArgImplode()

```
int CoArgImplode (  
    UBYTE * s ) [extern]
```

Definition at line 5791 of file [compcomm.c](#).

4.19.6.350 CoArgument()

```
int CoArgument (  
    UBYTE * s ) [extern]
```

Definition at line 2119 of file [compcomm.c](#).

4.19.6.351 CoInside()

```
int CoInside (  
    UBYTE * s ) [extern]
```

Definition at line 2152 of file [compcomm.c](#).

4.19.6.352 ExecInside()

```
int ExecInside (  
    UBYTE * s ) [extern]
```

Definition at line 1681 of file [dollar.c](#).

4.19.6.353 CoInExpression()

```
int CoInExpression (  
    UBYTE * s ) [extern]
```

Definition at line 3380 of file [compcomm.c](#).

4.19.6.354 CoInParallel()

```
int CoInParallel (  
    UBYTE * s ) [extern]
```

Definition at line 3286 of file [compcomm.c](#).

4.19.6.355 CoNotInParallel()

```
int CoNotInParallel (  
    UBYTE * s ) [extern]
```

Definition at line 3296 of file [compcomm.c](#).

4.19.6.356 DoInParallel()

```
int DoInParallel (
    UBYTE * s,
    int par ) [extern]
```

Definition at line [3311](#) of file [compcomm.c](#).

4.19.6.357 CoEndInExpression()

```
int CoEndInExpression (
    UBYTE * s ) [extern]
```

Definition at line [3433](#) of file [compcomm.c](#).

4.19.6.358 CoBracket()

```
int CoBracket (
    UBYTE * inp ) [extern]
```

Definition at line [3902](#) of file [compcomm.c](#).

4.19.6.359 CoPutInside()

```
int CoPutInside (
    UBYTE * inp ) [extern]
```

Definition at line [7065](#) of file [compcomm.c](#).

4.19.6.360 CoAntiPutInside()

```
int CoAntiPutInside (
    UBYTE * inp ) [extern]
```

Definition at line [7066](#) of file [compcomm.c](#).

4.19.6.361 CoMultiBracket()

```
int CoMultiBracket (
    UBYTE * inp ) [extern]
```

Definition at line [3921](#) of file [compcomm.c](#).

4.19.6.362 CoCFunction()

```
int CoCFunction (
    UBYTE * s ) [extern]
```

Definition at line [1627](#) of file [names.c](#).

4.19.6.363 CoCTensor()

```
int CoCTensor (
    UBYTE * s ) [extern]
```

Definition at line 1629 of file [names.c](#).

4.19.6.364 CoCollect()

```
int CoCollect (
    UBYTE * s ) [extern]
```

Definition at line 434 of file [compcomm.c](#).

4.19.6.365 CoCompress()

```
int CoCompress (
    UBYTE * s ) [extern]
```

Definition at line 512 of file [compcomm.c](#).

4.19.6.366 CoContract()

```
int CoContract (
    UBYTE * s ) [extern]
```

Definition at line 1793 of file [compcomm.c](#).

4.19.6.367 CoCycleSymmetrize()

```
int CoCycleSymmetrize (
    UBYTE * s ) [extern]
```

Definition at line 2374 of file [compcomm.c](#).

4.19.6.368 CoDelete()

```
int CoDelete (
    UBYTE * s ) [extern]
```

Definition at line 948 of file [compcomm.c](#).

4.19.6.369 CoTableBase()

```
int CoTableBase (
    UBYTE * s ) [extern]
```

Definition at line 627 of file [tables.c](#).

4.19.6.370 CoApply()

```
int CoApply (
    UBYTE * s ) [extern]
```

Definition at line 1797 of file [tables.c](#).

4.19.6.371 CoDenominators()

```
int CoDenominators (
    UBYTE * s ) [extern]
```

Definition at line 5880 of file [compcomm.c](#).

4.19.6.372 CoDimension()

```
int CoDimension (
    UBYTE * s ) [extern]
```

Definition at line 1226 of file [names.c](#).

4.19.6.373 CoDiscard()

```
int CoDiscard (
    UBYTE * s ) [extern]
```

Definition at line 1770 of file [compcomm.c](#).

4.19.6.374 CoDisorder()

```
int CoDisorder (
    UBYTE * inp ) [extern]
```

Definition at line 417 of file [comexpr.c](#).

4.19.6.375 CoDrop()

```
int CoDrop (
    UBYTE * s ) [extern]
```

Definition at line 1563 of file [compcomm.c](#).

4.19.6.376 CoDropCoefficient()

```
int CoDropCoefficient (
    UBYTE * s ) [extern]
```

Definition at line 5909 of file [compcomm.c](#).

4.19.6.377 CoDropSymbols()

```
int CoDropSymbols (
    UBYTE * s ) [extern]
```

Definition at line [5923](#) of file [compcomm.c](#).

4.19.6.378 CoElse()

```
int CoElse (
    UBYTE * p ) [extern]
```

Definition at line [4883](#) of file [compcomm.c](#).

4.19.6.379 CoElseIf()

```
int CoElseIf (
    UBYTE * inp ) [extern]
```

Definition at line [4910](#) of file [compcomm.c](#).

4.19.6.380 CoEndArgument()

```
int CoEndArgument (
    UBYTE * s ) [extern]
```

Definition at line [2126](#) of file [compcomm.c](#).

4.19.6.381 CoEndInside()

```
int CoEndInside (
    UBYTE * s ) [extern]
```

Definition at line [2159](#) of file [compcomm.c](#).

4.19.6.382 CoEndIf()

```
int CoEndIf (
    UBYTE * inp ) [extern]
```

Definition at line [4937](#) of file [compcomm.c](#).

4.19.6.383 CoEndRepeat()

```
int CoEndRepeat (
    UBYTE * inp ) [extern]
```

Definition at line [3737](#) of file [compcomm.c](#).

4.19.6.384 CoEndTerm()

```
int CoEndTerm (
    UBYTE * s ) [extern]
```

Definition at line 5286 of file [compcomm.c](#).

4.19.6.385 CoEndWhile()

```
int CoEndWhile (
    UBYTE * inp ) [extern]
```

Definition at line 5003 of file [compcomm.c](#).

4.19.6.386 CoExit()

```
int CoExit (
    UBYTE * s ) [extern]
```

Definition at line 3259 of file [compcomm.c](#).

4.19.6.387 CoFactArg()

```
int CoFactArg (
    UBYTE * s ) [extern]
```

Definition at line 2220 of file [compcomm.c](#).

4.19.6.388 CoFactDollar()

```
int CoFactDollar (
    UBYTE * inp ) [extern]
```

Definition at line 6439 of file [compcomm.c](#).

4.19.6.389 CoFactorize()

```
int CoFactorize (
    UBYTE * s ) [extern]
```

Definition at line 6468 of file [compcomm.c](#).

4.19.6.390 CoNFactorize()

```
int CoNFactorize (
    UBYTE * s ) [extern]
```

Definition at line 6475 of file [compcomm.c](#).

4.19.6.391 CoUnFactorize()

```
int CoUnFactorize (
    UBYTE * s ) [extern]
```

Definition at line [6482](#) of file [compcomm.c](#).

4.19.6.392 CoNUnFactorize()

```
int CoNUnFactorize (
    UBYTE * s ) [extern]
```

Definition at line [6489](#) of file [compcomm.c](#).

4.19.6.393 DoFactorize()

```
int DoFactorize (
    UBYTE * s,
    int par ) [extern]
```

Definition at line [6496](#) of file [compcomm.c](#).

4.19.6.394 CoFill()

```
int CoFill (
    UBYTE * inp ) [extern]
```

Definition at line [1070](#) of file [comexpr.c](#).

4.19.6.395 CoFillExpression()

```
int CoFillExpression (
    UBYTE * inp ) [extern]
```

Definition at line [1356](#) of file [comexpr.c](#).

4.19.6.396 CoFixIndex()

```
int CoFixIndex (
    UBYTE * s ) [extern]
```

Definition at line [1005](#) of file [compcomm.c](#).

4.19.6.397 CoFormat()

```
int CoFormat (
    UBYTE * s ) [extern]
```

Definition at line [154](#) of file [compcomm.c](#).

4.19.6.398 CoGlobal()

```
int CoGlobal (
    UBYTE * inp ) [extern]
```

Definition at line 73 of file [comexpr.c](#).

4.19.6.399 CoGlobalFactorized()

```
int CoGlobalFactorized (
    UBYTE * inp ) [extern]
```

Definition at line 87 of file [comexpr.c](#).

4.19.6.400 CoGoTo()

```
int CoGoTo (
    UBYTE * inp ) [extern]
```

Definition at line 1822 of file [compcomm.c](#).

4.19.6.401 Cold()

```
int CoId (
    UBYTE * inp ) [extern]
```

Definition at line 395 of file [comexpr.c](#).

4.19.6.402 ColdNew()

```
int CoIdNew (
    UBYTE * inp ) [extern]
```

Definition at line 406 of file [comexpr.c](#).

4.19.6.403 ColdOld()

```
int CoIdOld (
    UBYTE * inp ) [extern]
```

Definition at line 384 of file [comexpr.c](#).

4.19.6.404 Colf()

```
int CoIf (
    UBYTE * inp ) [extern]
```

Definition at line 4243 of file [compcomm.c](#).

4.19.6.405 ColfMatch()

```
int ColfMatch (
    UBYTE * inp ) [extern]
```

Definition at line [450](#) of file [comexpr.c](#).

4.19.6.406 ColfNoMatch()

```
int ColfNoMatch (
    UBYTE * inp ) [extern]
```

Definition at line [461](#) of file [comexpr.c](#).

4.19.6.407 ColIndex()

```
int ColIndex (
    UBYTE * s ) [extern]
```

Definition at line [1112](#) of file [names.c](#).

4.19.6.408 ColInsideFirst()

```
int ColInsideFirst (
    UBYTE * s ) [extern]
```

Definition at line [934](#) of file [compcomm.c](#).

4.19.6.409 CoKeep()

```
int CoKeep (
    UBYTE * s ) [extern]
```

Definition at line [993](#) of file [compcomm.c](#).

4.19.6.410 CoLabel()

```
int CoLabel (
    UBYTE * inp ) [extern]
```

Definition at line [1841](#) of file [compcomm.c](#).

4.19.6.411 CoLoad()

```
int CoLoad (
    UBYTE * inp ) [extern]
```

Definition at line [572](#) of file [store.c](#).

4.19.6.412 CoLocal()

```
int CoLocal (
    UBYTE * inp ) [extern]
```

Definition at line 66 of file [comexpr.c](#).

4.19.6.413 CoLocalFactorized()

```
int CoLocalFactorized (
    UBYTE * inp ) [extern]
```

Definition at line 80 of file [comexpr.c](#).

4.19.6.414 CoMany()

```
int CoMany (
    UBYTE * inp ) [extern]
```

Definition at line 428 of file [comexpr.c](#).

4.19.6.415 CoMerge()

```
int CoMerge (
    UBYTE * inp ) [extern]
```

Definition at line 5569 of file [compcomm.c](#).

4.19.6.416 CoStuffle()

```
int CoStuffle (
    UBYTE * inp ) [extern]
```

Definition at line 5625 of file [compcomm.c](#).

4.19.6.417 CoMetric()

```
int CoMetric (
    UBYTE * s ) [extern]
```

Definition at line 1039 of file [compcomm.c](#).

4.19.6.418 CoModOption()

```
int CoModOption (
    UBYTE * s ) [extern]
```

Definition at line 212 of file [module.c](#).

4.19.6.419 CoModuleOption()

```
int CoModuleOption (
    UBYTE * s ) [extern]
```

Definition at line 143 of file [module.c](#).

4.19.6.420 CoModulus()

```
int CoModulus (
    UBYTE * inp ) [extern]
```

Definition at line 3576 of file [compcomm.c](#).

4.19.6.421 CoMulti()

```
int CoMulti (
    UBYTE * inp ) [extern]
```

Definition at line 439 of file [comexpr.c](#).

4.19.6.422 CoMultiply()

```
int CoMultiply (
    UBYTE * inp ) [extern]
```

Definition at line 1031 of file [comexpr.c](#).

4.19.6.423 CoNFunction()

```
int CoNFunction (
    UBYTE * s ) [extern]
```

Definition at line 1626 of file [names.c](#).

4.19.6.424 CoNPrint()

```
int CoNPrint (
    UBYTE * s ) [extern]
```

Definition at line 1277 of file [compcomm.c](#).

4.19.6.425 CoNTensor()

```
int CoNTensor (
    UBYTE * s ) [extern]
```

Definition at line 1628 of file [names.c](#).

4.19.6.426 CoNWrite()

```
int CoNWrite (
    UBYTE * s ) [extern]
```

Definition at line [2413](#) of file [compcomm.c](#).

4.19.6.427 CoNoDrop()

```
int CoNoDrop (
    UBYTE * s ) [extern]
```

Definition at line [1570](#) of file [compcomm.c](#).

4.19.6.428 CoNoSkip()

```
int CoNoSkip (
    UBYTE * s ) [extern]
```

Definition at line [1584](#) of file [compcomm.c](#).

4.19.6.429 CoNormalize()

```
int CoNormalize (
    UBYTE * s ) [extern]
```

Definition at line [2185](#) of file [compcomm.c](#).

4.19.6.430 CoMakeInteger()

```
int CoMakeInteger (
    UBYTE * s ) [extern]
```

Definition at line [2192](#) of file [compcomm.c](#).

4.19.6.431 CoFlags()

```
int CoFlags (
    UBYTE * s,
    int value ) [extern]
```

Definition at line [561](#) of file [compcomm.c](#).

4.19.6.432 CoOff()

```
int CoOff (
    UBYTE * s ) [extern]
```

Definition at line [596](#) of file [compcomm.c](#).

4.19.6.433 CoOn()

```
int CoOn (
    UBYTE * s ) [extern]
```

Definition at line 662 of file [compcomm.c](#).

4.19.6.434 CoOnce()

```
int CoOnce (
    UBYTE * inp ) [extern]
```

Definition at line 472 of file [comexpr.c](#).

4.19.6.435 CoOnly()

```
int CoOnly (
    UBYTE * inp ) [extern]
```

Definition at line 483 of file [comexpr.c](#).

4.19.6.436 CoOptimizeOption()

```
int CoOptimizeOption (
    UBYTE * s ) [extern]
```

Definition at line 6629 of file [compcomm.c](#).

4.19.6.437 CoPolyFun()

```
int CoPolyFun (
    UBYTE * s ) [extern]
```

Definition at line 5352 of file [compcomm.c](#).

4.19.6.438 CoPolyRatFun()

```
int CoPolyRatFun (
    UBYTE * s ) [extern]
```

Definition at line 5399 of file [compcomm.c](#).

4.19.6.439 CoPrint()

```
int CoPrint (
    UBYTE * s ) [extern]
```

Definition at line 1263 of file [compcomm.c](#).

4.19.6.440 CoPrintB()

```
int CoPrintB (
    UBYTE * s ) [extern]
```

Definition at line 1270 of file [compcomm.c](#).

4.19.6.441 CoProperCount()

```
int CoProperCount (
    UBYTE * s ) [extern]
```

Definition at line 941 of file [compcomm.c](#).

4.19.6.442 CoUnitTrace()

```
int CoUnitTrace (
    UBYTE * s ) [extern]
```

Definition at line 5203 of file [compcomm.c](#).

4.19.6.443 CoRCycleSymmetrize()

```
int CoRCycleSymmetrize (
    UBYTE * s ) [extern]
```

Definition at line 2381 of file [compcomm.c](#).

4.19.6.444 CoRatio()

```
int CoRatio (
    UBYTE * s ) [extern]
```

Definition at line 2440 of file [compcomm.c](#).

4.19.6.445 CoRedefine()

```
int CoRedefine (
    UBYTE * s ) [extern]
```

Definition at line 2480 of file [compcomm.c](#).

4.19.6.446 CoRenumber()

```
int CoRenumber (
    UBYTE * s ) [extern]
```

Definition at line 2585 of file [compcomm.c](#).

4.19.6.447 CoRepeat()

```
int CoRepeat (
    UBYTE * inp ) [extern]
```

Definition at line [3714](#) of file [compcomm.c](#).

4.19.6.448 CoSave()

```
int CoSave (
    UBYTE * inp ) [extern]
```

Definition at line [362](#) of file [store.c](#).

4.19.6.449 CoSelect()

```
int CoSelect (
    UBYTE * inp ) [extern]
```

Definition at line [494](#) of file [comexpr.c](#).

4.19.6.450 CoSet()

```
int CoSet (
    UBYTE * s ) [extern]
```

Definition at line [2357](#) of file [names.c](#).

4.19.6.451 CoSetExitFlag()

```
int CoSetExitFlag (
    UBYTE * s ) [extern]
```

Definition at line [3459](#) of file [compcomm.c](#).

4.19.6.452 CoSkip()

```
int CoSkip (
    UBYTE * s ) [extern]
```

Definition at line [1577](#) of file [compcomm.c](#).

4.19.6.453 CoProcessBucket()

```
int CoProcessBucket (
    UBYTE * s ) [extern]
```

Definition at line [5680](#) of file [compcomm.c](#).

4.19.6.454 CoPushHide()

```
int CoPushHide (
    UBYTE * s ) [extern]
```

Definition at line [1284](#) of file [compcomm.c](#).

4.19.6.455 CoPopHide()

```
int CoPopHide (
    UBYTE * s ) [extern]
```

Definition at line [1328](#) of file [compcomm.c](#).

4.19.6.456 CoHide()

```
int CoHide (
    UBYTE * inp ) [extern]
```

Definition at line [1591](#) of file [compcomm.c](#).

4.19.6.457 CoIntoHide()

```
int CoIntoHide (
    UBYTE * inp ) [extern]
```

Definition at line [1609](#) of file [compcomm.c](#).

4.19.6.458 CoNoIntoHide()

```
int CoNoIntoHide (
    UBYTE * inp ) [extern]
```

Definition at line [1627](#) of file [compcomm.c](#).

4.19.6.459 CoNoHide()

```
int CoNoHide (
    UBYTE * inp ) [extern]
```

Definition at line [1634](#) of file [compcomm.c](#).

4.19.6.460 CoUnHide()

```
int CoUnHide (
    UBYTE * inp ) [extern]
```

Definition at line [1641](#) of file [compcomm.c](#).

4.19.6.461 CoNoUnHide()

```
int CoNoUnHide (
    UBYTE * inp ) [extern]
```

Definition at line 1648 of file [compcomm.c](#).

4.19.6.462 CoSort()

```
int CoSort (
    UBYTE * s ) [extern]
```

Definition at line 5313 of file [compcomm.c](#).

4.19.6.463 CoSplitArg()

```
int CoSplitArg (
    UBYTE * s ) [extern]
```

Definition at line 2199 of file [compcomm.c](#).

4.19.6.464 CoSplitFirstArg()

```
int CoSplitFirstArg (
    UBYTE * s ) [extern]
```

Definition at line 2206 of file [compcomm.c](#).

4.19.6.465 CoSplitLastArg()

```
int CoSplitLastArg (
    UBYTE * s ) [extern]
```

Definition at line 2213 of file [compcomm.c](#).

4.19.6.466 CoSum()

```
int CoSum (
    UBYTE * s ) [extern]
```

Definition at line 2606 of file [compcomm.c](#).

4.19.6.467 CoSymbol()

```
int CoSymbol (
    UBYTE * s ) [extern]
```

Definition at line 946 of file [names.c](#).

4.19.6.468 CoSymmetrize()

```
int CoSymmetrize (  
    UBYTE * s ) [extern]
```

Definition at line [2360](#) of file [compcomm.c](#).

4.19.6.469 DoTable()

```
int DoTable (  
    UBYTE * s,  
    int par ) [extern]
```

Definition at line [1668](#) of file [names.c](#).

4.19.6.470 CoTable()

```
int CoTable (  
    UBYTE * s ) [extern]
```

Definition at line [2050](#) of file [names.c](#).

4.19.6.471 CoTerm()

```
int CoTerm (  
    UBYTE * s ) [extern]
```

Definition at line [5238](#) of file [compcomm.c](#).

4.19.6.472 CoNTable()

```
int CoNTable (  
    UBYTE * s ) [extern]
```

Definition at line [2060](#) of file [names.c](#).

4.19.6.473 CoCTable()

```
int CoCTable (  
    UBYTE * s ) [extern]
```

Definition at line [2070](#) of file [names.c](#).

4.19.6.474 EmptyTable()

```
void EmptyTable (  
    TABLES T ) [extern]
```

Definition at line [2080](#) of file [names.c](#).

4.19.6.475 CoToTensor()

```
int CoToTensor (
    UBYTE * s ) [extern]
```

Definition at line [2726](#) of file [compcomm.c](#).

4.19.6.476 CoToVector()

```
int CoToVector (
    UBYTE * s ) [extern]
```

Definition at line [2894](#) of file [compcomm.c](#).

4.19.6.477 CoTrace4()

```
int CoTrace4 (
    UBYTE * s ) [extern]
```

Definition at line [2964](#) of file [compcomm.c](#).

4.19.6.478 CoTraceN()

```
int CoTraceN (
    UBYTE * s ) [extern]
```

Definition at line [3051](#) of file [compcomm.c](#).

4.19.6.479 CoChisholm()

```
int CoChisholm (
    UBYTE * s ) [extern]
```

Definition at line [3111](#) of file [compcomm.c](#).

4.19.6.480 CoTransform()

```
int CoTransform (
    UBYTE * in ) [extern]
```

Definition at line [87](#) of file [transform.c](#).

4.19.6.481 CoClearTable()

```
int CoClearTable (
    UBYTE * s ) [extern]
```

Definition at line [5798](#) of file [compcomm.c](#).

4.19.6.482 DoChain()

```
int DoChain (
    UBYTE * s,
    int option ) [extern]
```

Definition at line 3193 of file [compcomm.c](#).

4.19.6.483 CoChainin()

```
int CoChainin (
    UBYTE * s ) [extern]
```

Definition at line 3237 of file [compcomm.c](#).

4.19.6.484 CoChainout()

```
int CoChainout (
    UBYTE * s ) [extern]
```

Definition at line 3249 of file [compcomm.c](#).

4.19.6.485 CoTryReplace()

```
int CoTryReplace (
    UBYTE * p ) [extern]
```

Definition at line 3473 of file [compcomm.c](#).

4.19.6.486 CoVector()

```
int CoVector (
    UBYTE * s ) [extern]
```

Definition at line 1267 of file [names.c](#).

4.19.6.487 CoWhile()

```
int CoWhile (
    UBYTE * inp ) [extern]
```

Definition at line 4982 of file [compcomm.c](#).

4.19.6.488 CoWrite()

```
int CoWrite (
    UBYTE * s ) [extern]
```

Definition at line 2388 of file [compcomm.c](#).

4.19.6.489 CoAuto()

```
int CoAuto (
    UBYTE * inp ) [extern]
```

Definition at line [2574](#) of file [names.c](#).

4.19.6.490 CoSwitch()

```
int CoSwitch (
    UBYTE * s ) [extern]
```

Definition at line [7188](#) of file [compcomm.c](#).

4.19.6.491 CoCase()

```
int CoCase (
    UBYTE * s ) [extern]
```

Definition at line [7234](#) of file [compcomm.c](#).

4.19.6.492 CoBreak()

```
int CoBreak (
    UBYTE * s ) [extern]
```

Definition at line [7284](#) of file [compcomm.c](#).

4.19.6.493 CoDefault()

```
int CoDefault (
    UBYTE * s ) [extern]
```

Definition at line [7310](#) of file [compcomm.c](#).

4.19.6.494 CoEndSwitch()

```
int CoEndSwitch (
    UBYTE * s ) [extern]
```

Definition at line [7337](#) of file [compcomm.c](#).

4.19.6.495 CoTBaddto()

```
int CoTBaddto (
    UBYTE * s ) [extern]
```

Definition at line [801](#) of file [tables.c](#).

4.19.6.496 CoTBaudit()

```
int CoTBaudit (
    UBYTE * s ) [extern]
```

Definition at line [1474](#) of file [tables.c](#).

4.19.6.497 CoTBcleanup()

```
int CoTBcleanup (
    UBYTE * s ) [extern]
```

Definition at line [1576](#) of file [tables.c](#).

4.19.6.498 CoTBcreate()

```
int CoTBcreate (
    UBYTE * s ) [extern]
```

Definition at line [756](#) of file [tables.c](#).

4.19.6.499 CoTBenter()

```
int CoTBenter (
    UBYTE * s ) [extern]
```

Definition at line [974](#) of file [tables.c](#).

4.19.6.500 CoTBhelp()

```
int CoTBhelp (
    UBYTE * s ) [extern]
```

Definition at line [1884](#) of file [tables.c](#).

4.19.6.501 CoTBload()

```
int CoTBload (
    UBYTE * ss ) [extern]
```

Definition at line [1252](#) of file [tables.c](#).

4.19.6.502 CoTBoff()

```
int CoTBoff (
    UBYTE * s ) [extern]
```

Definition at line [1545](#) of file [tables.c](#).

4.19.6.503 CoTBon()

```
int CoTBon (
    UBYTE * s ) [extern]
```

Definition at line 1514 of file [tables.c](#).

4.19.6.504 CoTBopen()

```
int CoTBopen (
    UBYTE * s ) [extern]
```

Definition at line 772 of file [tables.c](#).

4.19.6.505 CoTBreplace()

```
int CoTBreplace (
    UBYTE * s ) [extern]
```

Definition at line 1588 of file [tables.c](#).

4.19.6.506 CoTBuse()

```
int CoTBuse (
    UBYTE * s ) [extern]
```

Definition at line 1603 of file [tables.c](#).

4.19.6.507 CoTestUse()

```
int CoTestUse (
    UBYTE * s ) [extern]
```

Definition at line 1133 of file [tables.c](#).

4.19.6.508 CoThreadBucket()

```
int CoThreadBucket (
    UBYTE * s ) [extern]
```

Definition at line 5698 of file [compcomm.c](#).

4.19.6.509 AddComString()

```
int AddComString (
    int n,
    WORD * array,
    UBYTE * thestring,
    int par ) [extern]
```

Definition at line 1670 of file [compcomm.c](#).

4.19.6.510 CompileAlgebra()

```
int CompileAlgebra (
    UBYTE * s,
    int leftright,
    WORD * prototype ) [extern]
```

Definition at line 536 of file [compiler.c](#).

4.19.6.511 IsIdStatement()

```
int IsIdStatement (
    UBYTE * s ) [extern]
```

Definition at line 522 of file [compiler.c](#).

4.19.6.512 IsRHS()

```
UBYTE * IsRHS (
    UBYTE * s,
    UBYTE c ) [extern]
```

Definition at line 476 of file [compiler.c](#).

4.19.6.513 ParenthesesTest()

```
int ParenthesesTest (
    UBYTE * sin ) [extern]
```

Definition at line 385 of file [compiler.c](#).

4.19.6.514 tokenize()

```
int tokenize (
    UBYTE * in,
    WORD leftright ) [extern]
```

Definition at line 58 of file [token.c](#).

4.19.6.515 WriteTokens()

```
void WriteTokens (
    SBYTE * in ) [extern]
```

Definition at line 652 of file [token.c](#).

4.19.6.516 simp1token()

```
int simp1token (
    SBYTE * s ) [extern]
```

Definition at line 700 of file [token.c](#).

4.19.6.517 simpwtoken()

```
int simpwtoken (
    SBYTE * s ) [extern]
```

Definition at line 789 of file [token.c](#).

4.19.6.518 simp2token()

```
int simp2token (
    SBYTE * s ) [extern]
```

Definition at line 943 of file [token.c](#).

4.19.6.519 simp3atoken()

```
int simp3atoken (
    SBYTE * s,
    int mode ) [extern]
```

Definition at line 1191 of file [token.c](#).

4.19.6.520 simp3btoken()

```
int simp3btoken (
    SBYTE * s,
    int mode ) [extern]
```

Definition at line 1432 of file [token.c](#).

4.19.6.521 simp4token()

```
int simp4token (
    SBYTE * s ) [extern]
```

Definition at line 1843 of file [token.c](#).

4.19.6.522 simp5token()

```
int simp5token (
    SBYTE * s,
    int mode ) [extern]
```

Definition at line [2003](#) of file [token.c](#).

4.19.6.523 simp6token()

```
int simp6token (
    SBYTE * tokens,
    int mode ) [extern]
```

Definition at line [2049](#) of file [token.c](#).

4.19.6.524 SkipAName()

```
UBYTE * SkipAName (
    UBYTE * s ) [extern]
```

Skips a name and gives a pointer to the character after the name. If there is not a proper name, it emits a compiler message and then returns a null pointer. This function supports formal names (e.g., `[x+a]`) and `$`-variables in addition to ordinary identifiers.

Note

The brackets are assumed to be matched already.

Parameters

<code>in</code>	<code>s</code>	Pointer to a null-terminated input string that starts with a name.

Returns

Pointer to the character after the name, or a null pointer if the name is invalid.

Definition at line [443](#) of file [compiler.c](#).

4.19.6.525 TestTables()

```
int TestTables (
    void ) [extern]
```

Definition at line [706](#) of file [compiler.c](#).

4.19.6.526 GetLabel()

```
int GetLabel (
    UBYTE * name ) [extern]
```

Definition at line [2789](#) of file [names.c](#).

4.19.6.527 ColdExpression()

```
int CoIdExpression (
    UBYTE * inp,
    int type ) [extern]
```

Definition at line [507](#) of file [comexpr.c](#).

4.19.6.528 CoAssign()

```
int CoAssign (
    UBYTE * inp ) [extern]
```

Definition at line [1924](#) of file [comexpr.c](#).

4.19.6.529 DoExpr()

```
int DoExpr (
    UBYTE * inp,
    int type,
    int par ) [extern]
```

Definition at line [96](#) of file [comexpr.c](#).

4.19.6.530 CompileSubExpressions()

```
int CompileSubExpressions (
    SBYTE * tokens ) [extern]
```

Definition at line [750](#) of file [compiler.c](#).

4.19.6.531 CodeGenerator()

```
int CodeGenerator (
    SBYTE * tokens ) [extern]
```

Definition at line [885](#) of file [compiler.c](#).

4.19.6.532 CompleteTerm()

```
int CompleteTerm (
    WORD * term,
    UWORD * numer,
    UWORD * denom,
    WORD nnum,
    WORD nden,
    int sign ) [extern]
```

Definition at line 1948 of file [compiler.c](#).

4.19.6.533 CodeFactors()

```
int CodeFactors (
    SBYTE * s ) [extern]
```

Definition at line 1985 of file [compiler.c](#).

4.19.6.534 GenerateFactors()

```
WORD GenerateFactors (
    WORD n,
    WORD inc ) [extern]
```

Definition at line 2282 of file [compiler.c](#).

4.19.6.535 InsTree()

```
int InsTree (
    int bufnum,
    int h ) [extern]
```

Definition at line 355 of file [comtool.c](#).

4.19.6.536 FindTree()

```
int FindTree (
    int bufnum,
    WORD * subexpr ) [extern]
```

Definition at line 533 of file [comtool.c](#).

4.19.6.537 RedoTree()

```
void RedoTree (
    CBUF * C,
    int size ) [extern]
```

Definition at line 569 of file [comtool.c](#).

4.19.6.538 ClearTree()

```
void ClearTree (
    int i ) [extern]
```

Definition at line 588 of file [comtool.c](#).

4.19.6.539 CatchDollar()

```
int CatchDollar (
    int par ) [extern]
```

Definition at line 54 of file [dollar.c](#).

4.19.6.540 AssignDollar()

```
int AssignDollar (
    PHEAD WORD * term,
    WORD level ) [extern]
```

Definition at line 209 of file [dollar.c](#).

4.19.6.541 WriteDollarToBuffer()

```
UBYTE * WriteDollarToBuffer (
    WORD numdollar,
    WORD par ) [extern]
```

Definition at line 596 of file [dollar.c](#).

4.19.6.542 WriteDollarFactorToBuffer()

```
UBYTE * WriteDollarFactorToBuffer (
    WORD numdollar,
    WORD numfac,
    WORD par ) [extern]
```

Definition at line 687 of file [dollar.c](#).

4.19.6.543 AddToDollarBuffer()

```
void AddToDollarBuffer (
    UBYTE * s ) [extern]
```

Definition at line 762 of file [dollar.c](#).

4.19.6.544 PutTermInDollar()

```
int PutTermInDollar (
    WORD * term,
    WORD numdollar ) [extern]
```

Definition at line 863 of file [dollar.c](#).

4.19.6.545 TermAssign()

```
void TermAssign (
    WORD * term ) [extern]
```

Definition at line 803 of file [dollar.c](#).

4.19.6.546 WildDollars()

```
void WildDollars (
    PHEAD WORD * term ) [extern]
```

Definition at line 891 of file [dollar.c](#).

4.19.6.547 numcommute()

```
LONG numcommute (
    WORD * terms,
    LONG * numterms ) [extern]
```

Definition at line 659 of file [comtool.c](#).

4.19.6.548 FullRenumber()

```
int FullRenumber (
    PHEAD WORD * term,
    WORD par ) [extern]
```

Definition at line 324 of file [reshuf.c](#).

4.19.6.549 Lus()

```
int Lus (
    WORD * term,
    WORD funnum,
    WORD loopsize,
    WORD numargs,
    WORD outfun,
    WORD mode ) [extern]
```

Definition at line 57 of file [lus.c](#).

4.19.6.550 FindLus()

```
int FindLus (
    int from,
    int level,
    int openindex ) [extern]
```

Definition at line 515 of file [lus.c](#).

4.19.6.551 CoReplaceLoop()

```
int CoReplaceLoop (
    UBYTE * inp ) [extern]
```

Definition at line 5156 of file [compcomm.c](#).

4.19.6.552 CoFindLoop()

```
int CoFindLoop (
    UBYTE * inp ) [extern]
```

Definition at line 5148 of file [compcomm.c](#).

4.19.6.553 DoFindLoop()

```
int DoFindLoop (
    UBYTE * inp,
    int mode ) [extern]
```

Definition at line 5031 of file [compcomm.c](#).

4.19.6.554 CoFunPowers()

```
int CoFunPowers (
    UBYTE * inp ) [extern]
```

Definition at line 5176 of file [compcomm.c](#).

4.19.6.555 SortTheList()

```
int SortTheList (
    int * slist,
    int num ) [extern]
```

Definition at line 578 of file [lus.c](#).

4.19.6.556 MatchIsPossible()

```
int MatchIsPossible (
    WORD * pattern,
    WORD * term ) [extern]
```

Definition at line 299 of file [smart.c](#).

4.19.6.557 StudyPattern()

```
int StudyPattern (
    WORD * lhs ) [extern]
```

Definition at line 62 of file [smart.c](#).

4.19.6.558 DolToTensor()

```
WORD DolToTensor (
    PHEAD WORD ) [extern]
```

Definition at line 1082 of file [dollar.c](#).

4.19.6.559 DolToFunction()

```
WORD DolToFunction (
    PHEAD WORD ) [extern]
```

Definition at line 1143 of file [dollar.c](#).

4.19.6.560 DolToVector()

```
WORD DolToVector (
    PHEAD WORD ) [extern]
```

Definition at line 1200 of file [dollar.c](#).

4.19.6.561 DolToNumber()

```
WORD DolToNumber (
    PHEAD WORD ) [extern]
```

Definition at line 1264 of file [dollar.c](#).

4.19.6.562 DolToSymbol()

```
WORD DolToSymbol (
    PHEAD WORD ) [extern]
```

Definition at line 1323 of file [dollar.c](#).

4.19.6.563 DoIToIndex()

```
WORD DoIToIndex (  
    PHEAD WORD ) [extern]
```

Definition at line 1377 of file [dollar.c](#).

4.19.6.564 DoIToLong()

```
LONG DoIToLong (  
    PHEAD WORD ) [extern]
```

Definition at line 1606 of file [dollar.c](#).

4.19.6.565 DollarFactorize()

```
int DollarFactorize (  
    PHEAD WORD ) [extern]
```

Definition at line 2955 of file [dollar.c](#).

4.19.6.566 CoPrintTable()

```
int CoPrintTable (  
    UBYTE * inp ) [extern]
```

Definition at line 1748 of file [comexpr.c](#).

4.19.6.567 CoDeallocateTable()

```
int CoDeallocateTable (  
    UBYTE * inp ) [extern]
```

Definition at line 1982 of file [comexpr.c](#).

4.19.6.568 CleanDollarFactors()

```
void CleanDollarFactors (  
    DOLLARS d ) [extern]
```

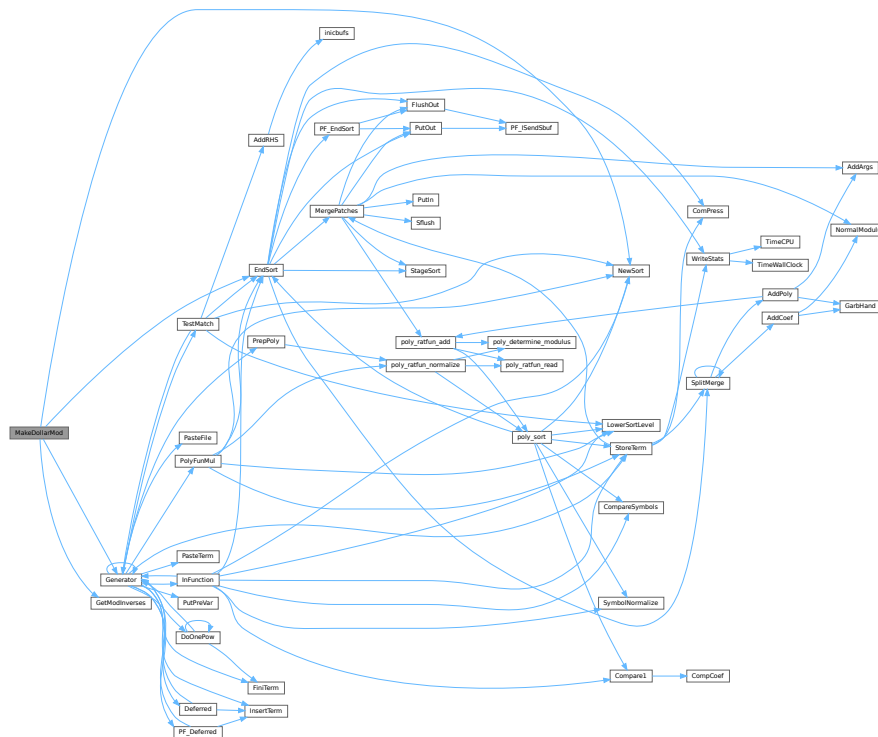
Definition at line 3554 of file [dollar.c](#).

4.19.6.569 TakeDollarContent()

```
WORD * TakeDollarContent (  
    PHEAD WORD * dollarbuffer,  
    WORD ** factor ) [extern]
```

Definition at line 3576 of file [dollar.c](#).

Here is the call graph for this function:



4.19.6.572 GetDolNum()

```
int GetDolNum (
    PHEAD WORD * t,
    WORD * tstop ) [extern]
```

Definition at line 3847 of file [dollar.c](#).

4.19.6.573 AddPotModdollar()

```
void AddPotModdollar (
    WORD numdollar ) [extern]
```

Adds a \$-variable specified by *numdollar* to the list of potentially modified \$-variables unless it has already been included in the list.

Parameters

<i>numdollar</i>	The index of the \$-variable to be added.

Definition at line 3960 of file [dollar.c](#).

4.19.6.574 Optimize()

```
int Optimize (
    WORD exprnr,
    int do_print ) [extern]
```

Optimization of expression

4.19.7 Description

This method takes an input expression and generates optimized code to calculate its value. The following methods are called to do so:

- (1) `get_expression` : to read to expression
- (2) `get_brackets` : find brackets for simultaneous optimization
- (3) `occurrence_order` or `find_Horner_MCTS` : to determine (the) Horner scheme(s) to use; this depends on `AO.optimize.horner`
- (4) `optimize_expression_given_Horner` : to do the optimizations for each Horner scheme; this method does either CSE or greedy optimizations dependings on `AO.optimize.method`
- (5) `generate_output` : to format the output in Form notation and store it in a buffer
- (6a) `optimize_print_code` : to print the expression (for "Print") or (6b) `generate_expression` : to modify the expression (for "#Optimize")

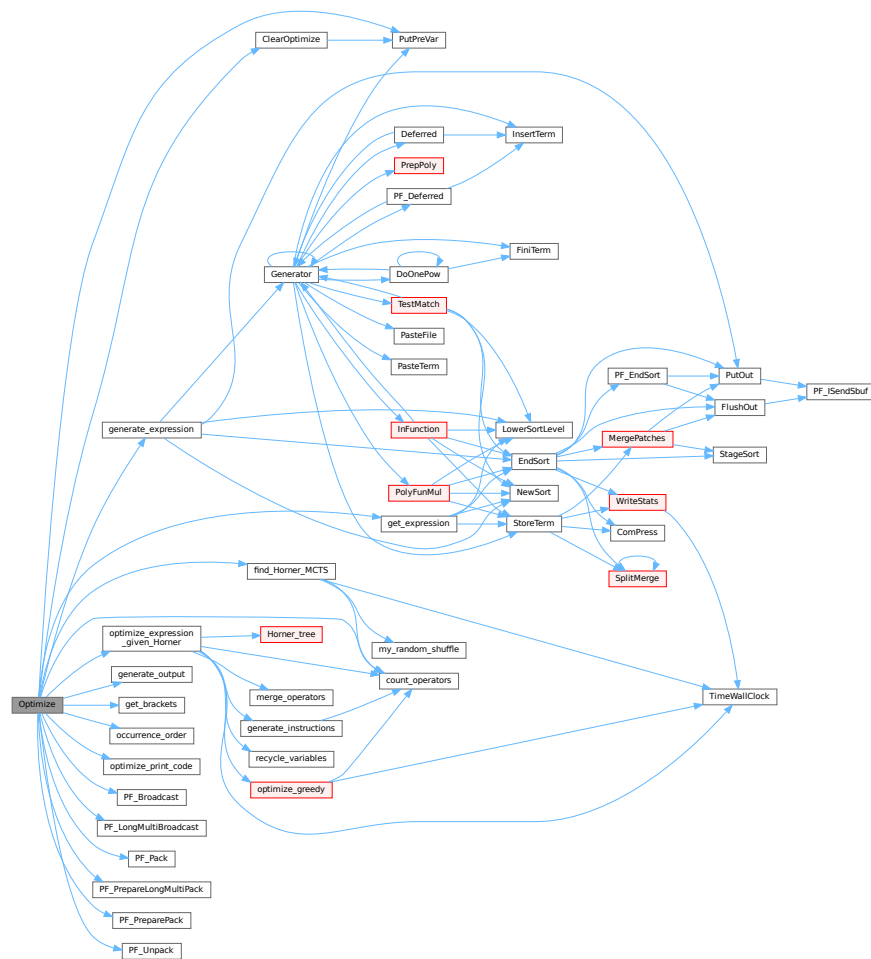
On ParFORM, all the processes must call this function at the same time. Then

- (1) Because only the master can access to the expression to be optimized, the master broadcast the expression to all the slaves after reading the expression (`PF_get_expression`).
- (2) `get_brackets` reads `optimize_expr` as the input and it works also on the slaves. We leave it although the bracket information is not needed on the slaves (used in (5) on the master).
- (3) and (4) `find_Horner_MCTS` and `optimize_expression_given_Horner` are parallelized.
- (5), (6a) and (6b) are needed only on the master.

Definition at line [4637](#) of file [optimize.cc](#).

References [ClearOptimize\(\)](#), [count_operators\(\)](#), [find_Horner_MCTS\(\)](#), [generate_expression\(\)](#), [generate_output\(\)](#), [get_brackets\(\)](#), [get_expression\(\)](#), [occurrence_order\(\)](#), [optimize_expression_given_Horner\(\)](#), [optimize_print_code\(\)](#), [PF_Broadcast\(\)](#), [PF_LongMultiBroadcast\(\)](#), [PF_Pack\(\)](#), [PF_PrepareLongMultiPack\(\)](#), [PF_PreparePack\(\)](#), [PF_Unpack\(\)](#), and [PutPreVar\(\)](#).

Here is the call graph for this function:



4.19.7.1 ClearOptimize()

```
int ClearOptimize (
    void ) [extern]
```

Optimization of expression

4.19.8 Description

Clears the buffers that were used for optimization output. Clears the expression from the buffers (marks it to be dropped). Note: we need to use the expression by its name, because numbers may change if we drop other expressions between when we do the optimizations and clear the results (in [execute.c](#)). Also this is not 100% safe, because we could overwrite the optimized expression. But that can be done only in a Local or Global statement and hence we only have to test there that we might have to call ClearOptimize first. (in file [comexpr.c](#))

Definition at line 4974 of file [optimize.cc](#).

References [PutPreVar\(\)](#).

Referenced by [Optimize\(\)](#).

Here is the call graph for this function:



4.19.8.1 TakeLongRoot()

```
int TakeLongRoot (
    UWORD * a,
    WORD * n,
    WORD power ) [extern]
```

Definition at line [2736](#) of file [reken.c](#).

4.19.8.2 TakeRatRoot()

```
int TakeRatRoot (
    UWORD * a,
    WORD * n,
    WORD power ) [extern]
```

Definition at line [681](#) of file [reken.c](#).

4.19.8.3 MakeRational()

```
int MakeRational (
    WORD a,
    WORD m,
    WORD * b,
    WORD * c ) [extern]
```

Definition at line [2856](#) of file [reken.c](#).

4.19.8.4 MakeLongRational()

```
int MakeLongRational (
    PHEAD UWORD * a,
    WORD na,
    UWORD * m,
    WORD nm,
    UWORD * b,
    WORD * nb ) [extern]
```

Definition at line [2904](#) of file [reken.c](#).

4.19.8.5 ClearTableTree()

```
void ClearTableTree (
    TABLES T ) [extern]
```

Definition at line 72 of file [tables.c](#).

4.19.8.6 InsTableTree()

```
int InsTableTree (
    TABLES T,
    WORD * tp ) [extern]
```

Definition at line 103 of file [tables.c](#).

4.19.8.7 RedoTableTree()

```
void RedoTableTree (
    TABLES T,
    int newsize ) [extern]
```

Definition at line 266 of file [tables.c](#).

4.19.8.8 FindTableTree()

```
int FindTableTree (
    TABLES T,
    WORD * tp,
    int inc ) [extern]
```

Definition at line 291 of file [tables.c](#).

4.19.8.9 finishcbuf()

```
void finishcbuf (
    WORD num ) [extern]
```

Frees a compiler buffer.

Parameters

<i>num</i>	The ID number for the buffer to be freed.

Definition at line 89 of file [comtool.c](#).

References [CbUf::boomlijst](#), [CbUf::Buffer](#), [CbUf::BufferSize](#), [CbUf::CanCommu](#), [CbUf::lhs](#), [CbUf::NumTerms](#), [CbUf::Pointer](#), [CbUf::rhs](#), and [CbUf::Top](#).

Referenced by [PF_BroadcastCBuf\(\)](#).

4.19.8.10 clearcbuf()

```
void clearcbuf (
    WORD num ) [extern]
```

Clears contents in a compiler buffer.

Parameters

<i>num</i>	The ID number for the buffer to be cleared.

Definition at line 116 of file [comtool.c](#).

References [CbUf::boomlijst](#), [CbUf::Buffer](#), [CbUf::Pointer](#), and [CbUf::rhs](#).

4.19.8.11 CleanUpSort()

```
void CleanUpSort (
    int num ) [extern]
```

Partially or completely frees function sort buffers.

Definition at line 4856 of file [sort.c](#).

4.19.8.12 AllocFileHandle()

```
FILEHANDLE * AllocFileHandle (
    WORD par,
    char * name ) [extern]
```

Definition at line 1033 of file [setfile.c](#).

4.19.8.13 DeAllocFileHandle()

```
void DeAllocFileHandle (
    FILEHANDLE * fh ) [extern]
```

Definition at line 1094 of file [setfile.c](#).

4.19.8.14 LowerSortLevel()

```
void LowerSortLevel (
    void ) [extern]
```

Lowers the level in the sort system.

Definition at line 4956 of file [sort.c](#).

Referenced by [generate_expression\(\)](#), [get_expression\(\)](#), [InFunction\(\)](#), [PF_CollectModifiedDollars\(\)](#), [PF_Processor\(\)](#), [poly_factorize_expression\(\)](#), [poly_sort\(\)](#), [poly_unfactorize_expression\(\)](#), [PolyFunMul\(\)](#), [Processor\(\)](#), and [TestMatch\(\)](#).

4.19.8.15 PolyRatFunSpecial()

```
WORD * PolyRatFunSpecial (
    PHEAD WORD * t1,
    WORD * t2 ) [extern]
```

Definition at line [4973](#) of file [sort.c](#).

4.19.8.16 SimpleSplitMergeRec()

```
void SimpleSplitMergeRec (
    WORD * array,
    WORD num,
    WORD * auxarray ) [extern]
```

Definition at line [5075](#) of file [sort.c](#).

4.19.8.17 SimpleSplitMerge()

```
void SimpleSplitMerge (
    WORD * array,
    WORD num ) [extern]
```

Definition at line [5103](#) of file [sort.c](#).

4.19.8.18 BinarySearch()

```
WORD BinarySearch (
    WORD * array,
    WORD num,
    WORD x ) [extern]
```

Definition at line [5121](#) of file [sort.c](#).

4.19.8.19 InsideDollar()

```
int InsideDollar (
    PHEAD WORD * ll,
    WORD level ) [extern]
```

Definition at line [1746](#) of file [dollar.c](#).

4.19.8.20 DolToTerms()

```
DOLLARS DolToTerms (
    PHEAD WORD ) [extern]
```

Definition at line [1458](#) of file [dollar.c](#).

4.19.8.21 EvalDoLoopArg()

```
WORD EvalDoLoopArg (
    PHEAD WORD * arg,
    WORD par ) [extern]
```

Evaluates one argument of a do loop. Such an argument is constructed from SNUMBERS DOLLAREXPRES- SIONS and possibly DOLLAREXPR2s which indicate factors of the preceding dollar. Hence we have SNUM- BER,num DOLLAREXPRESSSION,numdollar DOLLAREXPRESSSION,numdollar,DOLLAREXPR2,numfactor DOL- LAREXPRESSSION,numdollar,DOLLAREXPR2,numfactor,DOLLAREXPR2,numfactor etc. Because we have a do- loop at every stage we should have a number. The notation in DOLLAREXPR2 is that ≥ 0 is number of yet another dollar and < 0 is $-n-1$ with n the array element or zero. The return value is the (short) number. The routine works its way through the list in a recursive manner.

Definition at line 2651 of file [dollar.c](#).

References [EvalDoLoopArg\(\)](#).

Referenced by [EvalDoLoopArg\(\)](#).

Here is the call graph for this function:



4.19.8.22 SetExprCases()

```
int SetExprCases (
    int par,
    int setunset,
    int val ) [extern]
```

Definition at line 1363 of file [compcomm.c](#).

4.19.8.23 TestSelect()

```
int TestSelect (
    WORD * term,
    WORD * setp ) [extern]
```

Definition at line 1748 of file [pattern.c](#).

4.19.8.24 SubsInAll()

```
void SubsInAll (
    PHEAD0 ) [extern]
```

Definition at line 1982 of file [pattern.c](#).

4.19.8.25 TransferBuffer()

```
void TransferBuffer (
    int from,
    int to,
    int spectator ) [extern]
```

Definition at line 2143 of file [pattern.c](#).

4.19.8.26 TakeIDfunction()

```
int TakeIDfunction (
    PHEAD WORD * term ) [extern]
```

Definition at line 2213 of file [pattern.c](#).

4.19.8.27 MakeSetupAllocs()

```
int MakeSetupAllocs (
    void ) [extern]
```

Definition at line 1111 of file [setfile.c](#).

4.19.8.28 TryFileSetups()

```
int TryFileSetups (
    void ) [extern]
```

Definition at line 1131 of file [setfile.c](#).

4.19.8.29 ExchangeExpressions()

```
void ExchangeExpressions (
    int num1,
    int num2 ) [extern]
```

Definition at line 1673 of file [execute.c](#).

4.19.8.30 ExchangeDollars()

```
void ExchangeDollars (
    int num1,
    int num2 ) [extern]
```

Definition at line [1849](#) of file [dollar.c](#).

4.19.8.31 GetFirstBracket()

```
int GetFirstBracket (
    WORD * term,
    int num ) [extern]
```

Definition at line [1747](#) of file [execute.c](#).

4.19.8.32 GetFirstTerm()

```
int GetFirstTerm (
    WORD * term,
    int num,
    int pre ) [extern]
```

Gets the first term of an expression. Routine should be thread safe.

Parameters

out	<i>term</i>	Pointer to the buffer where the term should be written.
in	<i>num</i>	The expression number from which to get the term.
in	<i>pre</i>	Denotes a pre-processor call (1) or not (0). Used to determine the correct source file for the expression.

Returns

First WORD of term, a negative value denotes an error.

Definition at line [1866](#) of file [execute.c](#).

References [FiLe::handle](#), [VaRrEnUm::lo](#), and [ReNuMbEr::syMb](#).

4.19.8.33 GetContent()

```
int GetContent (
    WORD * content,
    int num ) [extern]
```

Definition at line [1974](#) of file [execute.c](#).

4.19.8.34 CleanupTerm()

```
int CleanupTerm (
    WORD * term ) [extern]
```

Definition at line 2091 of file [execute.c](#).

4.19.8.35 ContentMerge()

```
WORD ContentMerge (
    PHEAD WORD * content,
    WORD * term ) [extern]
```

Definition at line 2115 of file [execute.c](#).

4.19.8.36 PreIfDollarEval()

```
UBYTE * PreIfDollarEval (
    UBYTE * s,
    int * value ) [extern]
```

Definition at line 1978 of file [dollar.c](#).

4.19.8.37 TermsInDollar()

```
LONG TermsInDollar (
    WORD num ) [extern]
```

Definition at line 1867 of file [dollar.c](#).

4.19.8.38 SizeOfDollar()

```
LONG SizeOfDollar (
    WORD num ) [extern]
```

Definition at line 1916 of file [dollar.c](#).

4.19.8.39 TermsInExpression()

```
LONG TermsInExpression (
    WORD num ) [extern]
```

Definition at line 2379 of file [execute.c](#).

4.19.8.40 SizeOfExpression()

```
LONG SizeOfExpression (
    WORD num ) [extern]
```

Definition at line 2391 of file [execute.c](#).

4.19.8.41 TranslateExpression()

```
WORD * TranslateExpression (
    UBYTE * s ) [extern]
```

Definition at line [2160](#) of file [dollar.c](#).

4.19.8.42 IsSetMember()

```
int IsSetMember (
    WORD * buffer,
    WORD numset ) [extern]
```

Definition at line [2217](#) of file [dollar.c](#).

4.19.8.43 IsMultipleOf()

```
int IsMultipleOf (
    WORD * buf1,
    WORD * buf2 ) [extern]
```

Definition at line [2398](#) of file [dollar.c](#).

4.19.8.44 TwoExprCompare()

```
int TwoExprCompare (
    WORD * buf1,
    WORD * buf2,
    int oprtr ) [extern]
```

Definition at line [2473](#) of file [dollar.c](#).

4.19.8.45 UpdatePositions()

```
void UpdatePositions (
    void ) [extern]
```

Definition at line [2403](#) of file [execute.c](#).

4.19.8.46 M_print()

```
void M_print (
    void ) [extern]
```

Definition at line [2453](#) of file [tools.c](#).

4.19.8.47 M_check1()

```
void M_check1 (
    void ) [extern]
```

Definition at line 2452 of file [tools.c](#).

4.19.8.48 PrintTime()

```
void PrintTime (
    UBYTE * mess ) [extern]
```

Definition at line 766 of file [sch.c](#).

4.19.8.49 FindBracket()

```
POSITION * FindBracket (
    WORD nexp,
    WORD * bracket ) [extern]
```

Definition at line 65 of file [index.c](#).

4.19.8.50 PutBracketInIndex()

```
void PutBracketInIndex (
    PHEAD WORD * term,
    POSITION * newpos ) [extern]
```

Definition at line 331 of file [index.c](#).

4.19.8.51 ClearBracketIndex()

```
void ClearBracketIndex (
    WORD numexp ) [extern]
```

Definition at line 546 of file [index.c](#).

4.19.8.52 OpenBracketIndex()

```
void OpenBracketIndex (
    WORD nexpr ) [extern]
```

Definition at line 578 of file [index.c](#).

4.19.8.53 DoNoParallel()

```
int DoNoParallel (
    UBYTE * s ) [extern]
```

Definition at line 529 of file [module.c](#).

4.19.8.54 DoParallel()

```
int DoParallel (
    UBYTE * s ) [extern]
```

Definition at line 544 of file [module.c](#).

4.19.8.55 DoModSum()

```
int DoModSum (
    UBYTE * s ) [extern]
```

Definition at line 559 of file [module.c](#).

4.19.8.56 DoModMax()

```
int DoModMax (
    UBYTE * s ) [extern]
```

Definition at line 579 of file [module.c](#).

4.19.8.57 DoModMin()

```
int DoModMin (
    UBYTE * s ) [extern]
```

Definition at line 599 of file [module.c](#).

4.19.8.58 DoModLocal()

```
int DoModLocal (
    UBYTE * s ) [extern]
```

Definition at line 619 of file [module.c](#).

4.19.8.59 DoModDollar()

```
UBYTE * DoModDollar (
    UBYTE * s,
    int type ) [extern]
```

Definition at line 657 of file [module.c](#).

4.19.8.60 DoProcessBucket()

```
int DoProcessBucket (
    UBYTE * s ) [extern]
```

Definition at line 639 of file [module.c](#).

4.19.8.61 DoinParallel()

```
int DoinParallel (
    UBYTE * s ) [extern]
```

Definition at line 758 of file [module.c](#).

4.19.8.62 DonotinParallel()

```
int DonotinParallel (
    UBYTE * s ) [extern]
```

Definition at line 768 of file [module.c](#).

4.19.8.63 FlipTable()

```
int FlipTable (
    FUNCTIONS f,
    int type ) [extern]
```

Definition at line 671 of file [tables.c](#).

4.19.8.64 ChainIn()

```
int ChainIn (
    PHEAD WORD * term,
    WORD funnum ) [extern]
```

Definition at line 691 of file [function.c](#).

4.19.8.65 ChainOut()

```
int ChainOut (
    PHEAD WORD * term,
    WORD funnum ) [extern]
```

Definition at line 748 of file [function.c](#).

4.19.8.66 ArgumentImplode()

```
int ArgumentImplode (
    PHEAD WORD * term,
    WORD * thelist ) [extern]
```

Definition at line 1851 of file [argument.c](#).

4.19.8.67 ArgumentExplode()

```
int ArgumentExplode (
    PHEAD WORD * term,
    WORD * thelist ) [extern]
```

Definition at line 1955 of file [argument.c](#).

4.19.8.68 DenToFunction()

```
int DenToFunction (
    WORD * term,
    WORD numfun ) [extern]
```

Definition at line 2557 of file [wildcard.c](#).

4.19.8.69 HowMany()

```
WORD HowMany (
    PHEAD WORD * ifcode,
    WORD * term ) [extern]
```

Definition at line 840 of file [if.c](#).

4.19.8.70 RemoveDollars()

```
void RemoveDollars (
    void ) [extern]
```

Definition at line 3130 of file [names.c](#).

4.19.8.71 CountTerms1()

```
LONG CountTerms1 (
    PHEAD0 ) [extern]
```

Definition at line 2463 of file [execute.c](#).

4.19.8.72 TermsInBracket()

```
LONG TermsInBracket (
    PHEAD WORD * term,
    WORD level ) [extern]
```

Definition at line 2571 of file [execute.c](#).

4.19.8.73 Crash()

```
int Crash (
    void ) [extern]
```

Definition at line 3770 of file [tools.c](#).

4.19.8.74 str_dup()

```
char * str_dup (
    char * str ) [extern]
```

Definition at line 101 of file [minos.c](#).

4.19.8.75 convertblock()

```
void convertblock (
    INDEXBLOCK * in,
    INDEXBLOCK * out,
    int mode ) [extern]
```

Definition at line 120 of file [minos.c](#).

4.19.8.76 convertnamesblock()

```
void convertnamesblock (
    NAMESBLOCK * in,
    NAMESBLOCK * out,
    int mode ) [extern]
```

Definition at line 171 of file [minos.c](#).

4.19.8.77 convertiniinfo()

```
void convertiniinfo (
    INIINFO * in,
    INIINFO * out,
    int mode ) [extern]
```

Definition at line 197 of file [minos.c](#).

4.19.8.78 ReadIndex()

```
int ReadIndex (
    DBASE * d ) [extern]
```

Definition at line 288 of file [minos.c](#).

4.19.8.79 WriteIndexBlock()

```
int WriteIndexBlock (
    DBASE * d,
    MLONG num ) [extern]
```

Definition at line 399 of file [minos.c](#).

4.19.8.80 WriteNamesBlock()

```
int WriteNamesBlock (
    DBASE * d,
    MLONG num ) [extern]
```

Definition at line 420 of file [minos.c](#).

4.19.8.81 WriteIndex()

```
int WriteIndex (
    DBASE * d ) [extern]
```

Definition at line 443 of file [minos.c](#).

4.19.8.82 WriteIniInfo()

```
int WriteIniInfo (
    DBASE * d ) [extern]
```

Definition at line 506 of file [minos.c](#).

4.19.8.83 ReadIniInfo()

```
int ReadIniInfo (
    DBASE * d ) [extern]
```

Definition at line 523 of file [minos.c](#).

4.19.8.84 AddToIndex()

```
int AddToIndex (
    DBASE * d,
    MLONG number ) [extern]
```

Definition at line 1065 of file [minos.c](#).

4.19.8.85 GetDbase()

```
DBASE * GetDbase (
    char * filename,
    MLONG rwmode ) [extern]
```

Definition at line 546 of file [minos.c](#).

4.19.8.86 OpenDbase()

```
DBASE * OpenDbase (
    char * filename ) [extern]
```

Definition at line 820 of file [minos.c](#).

4.19.8.87 ReadObject()

```
char * ReadObject (
    DBASE * d,
    MLONG tablenumber,
    char * arguments ) [extern]
```

Definition at line 1294 of file [minos.c](#).

4.19.8.88 ReadijObject()

```
char * ReadijObject (
    DBASE * d,
    MLONG i,
    MLONG j,
    char * arguments ) [extern]
```

Definition at line 1375 of file [minos.c](#).

4.19.8.89 ExistsObject()

```
int ExistsObject (
    DBASE * d,
    MLONG tablenumber,
    char * arguments ) [extern]
```

Definition at line 1427 of file [minos.c](#).

4.19.8.90 DeleteObject()

```
int DeleteObject (
    DBASE * d,
    MLONG tablenumber,
    char * arguments ) [extern]
```

Definition at line 1459 of file [minos.c](#).

4.19.8.91 WriteObject()

```
int WriteObject (
    DBASE * d,
    MLONG tablename,
    char * arguments,
    char * rhs,
    MLONG number ) [extern]
```

Definition at line 1194 of file [minos.c](#).

4.19.8.92 AddObject()

```
MLONG AddObject (
    DBASE * d,
    MLONG tablename,
    char * arguments,
    char * rhs ) [extern]
```

Definition at line 1152 of file [minos.c](#).

4.19.8.93 NewDbase()

```
DBASE * NewDbase (
    char * name,
    MLONG number ) [extern]
```

Definition at line 602 of file [minos.c](#).

4.19.8.94 FreeTableBase()

```
void FreeTableBase (
    DBASE * d ) [extern]
```

Definition at line 739 of file [minos.c](#).

4.19.8.95 ComposeTableNames()

```
int ComposeTableNames (
    DBASE * d ) [extern]
```

Definition at line 771 of file [minos.c](#).

4.19.8.96 PutTableNames()

```
int PutTableNames (
    DBASE * d ) [extern]
```

Definition at line 969 of file [minos.c](#).

4.19.8.97 AddTableName()

```
MLONG AddTableName (
    DBASE * d,
    char * name,
    TABLES T ) [extern]
```

Definition at line 854 of file [minos.c](#).

4.19.8.98 GetTableName()

```
MLONG GetTableName (
    DBASE * d,
    char * name ) [extern]
```

Definition at line 937 of file [minos.c](#).

4.19.8.99 FindTableNumber()

```
MLONG FindTableNumber (
    DBASE * d,
    char * name ) [extern]
```

Definition at line 1166 of file [minos.c](#).

4.19.8.100 TryEnvironment()

```
int TryEnvironment (
    void ) [extern]
```

Definition at line 1217 of file [setfile.c](#).

4.19.8.101 CopyExpression()

```
int CopyExpression (
    FILEHANDLE * from,
    FILEHANDLE * to ) [extern]
```

Definition at line 3307 of file [store.c](#).

4.19.8.102 set_in()

```
int set_in (
    UBYTE ch,
    set_of_char set ) [extern]
```

Definition at line 2133 of file [tools.c](#).

4.19.8.103 set_set()

```
one_byte set_set (
    UBYTE ch,
    set_of_char set ) [extern]
```

Definition at line 2153 of file [tools.c](#).

4.19.8.104 set_del()

```
one_byte set_del (
    UBYTE ch,
    set_of_char set ) [extern]
```

Definition at line 2174 of file [tools.c](#).

4.19.8.105 set_sub()

```
one_byte set_sub (
    set_of_char set,
    set_of_char set1,
    set_of_char set2 ) [extern]
```

Definition at line 2196 of file [tools.c](#).

4.19.8.106 DoPreAddSeparator()

```
int DoPreAddSeparator (
    UBYTE * s ) [extern]
```

Definition at line 5460 of file [pre.c](#).

4.19.8.107 DoPreRmSeparator()

```
int DoPreRmSeparator (
    UBYTE * s ) [extern]
```

Definition at line 5485 of file [pre.c](#).

4.19.8.108 openExternalChannel()

```
int openExternalChannel (
    UBYTE * cmd,
    int daemonize,
    UBYTE * shellname,
    UBYTE * stderrname ) [extern]
```

Definition at line 230 of file [extcmd.c](#).

4.19.8.109 initPresetExternalChannels()

```
int initPresetExternalChannels (
    UBYTE * theline,
    int thetimeout ) [extern]
```

Definition at line [232](#) of file [extcmd.c](#).

4.19.8.110 closeExternalChannel()

```
int closeExternalChannel (
    int n ) [extern]
```

Definition at line [233](#) of file [extcmd.c](#).

4.19.8.111 selectExternalChannel()

```
int selectExternalChannel (
    int n ) [extern]
```

Definition at line [234](#) of file [extcmd.c](#).

4.19.8.112 getCurrentExternalChannel()

```
int getCurrentExternalChannel (
    void ) [extern]
```

Definition at line [235](#) of file [extcmd.c](#).

4.19.8.113 closeAllExternalChannels()

```
void closeAllExternalChannels (
    void ) [extern]
```

Definition at line [236](#) of file [extcmd.c](#).

4.19.8.114 defineChannel()

```
UBYTE * defineChannel (
    UBYTE * s,
    HANDLERS * h ) [extern]
```

Definition at line [6111](#) of file [pre.c](#).

4.19.8.115 writeToChannel()

```
int writeToChannel (
    int wtype,
    UBYTE * s,
    HANDLERS * h ) [extern]
```

Definition at line 6146 of file [pre.c](#).

4.19.8.116 ReleaseTB()

```
int ReleaseTB (
    void ) [extern]
```

Definition at line 2317 of file [tables.c](#).

4.19.8.117 SymbolNormalize()

```
int SymbolNormalize (
    WORD * term ) [extern]
```

Routine normalizes terms that contain only symbols. Regular minimum and maximum properties are ignored.

We check whether there are negative powers in the output. This is not allowed.

Definition at line 5164 of file [normal.c](#).

Referenced by [InFunction\(\)](#), and [poly_sort\(\)](#).

4.19.8.118 TestFunFlag()

```
int TestFunFlag (
    PHEAD WORD * tfun ) [extern]
```

Definition at line 5260 of file [normal.c](#).

4.19.8.119 CompareSymbols()

```
WORD CompareSymbols (
    PHEAD WORD * term1,
    WORD * term2,
    WORD par ) [extern]
```

Compares the terms, based on the value of AN.polysortflag. If $term1 < term2$ the return value is -1 If $term1 > term2$ the return value is 1 If $term1 = term2$ the return value is 0 The coefficients may differ. The terms contain only a single subterm of type SYMBOL. If AN.polysortflag = 0 it is a 'regular' compare. If AN.polysortflag = 1 the sum of the powers is more important par is a dummy parameter to make the parameter field identical to that of Compare1 which is the regular compare routine in [sort.c](#)

Definition at line 3110 of file [sort.c](#).

Referenced by [InFunction\(\)](#), and [poly_sort\(\)](#).

4.19.8.120 CompareHSymbols()

```
WORD CompareHSymbols (
    PHEAD WORD * term1,
    WORD * term2,
    WORD par ) [extern]
```

Compares terms that can have only SYMBOL and HAAKJE subterms. If $term1 < term2$ the return value is -1 If $term1 > term2$ the return value is 1 If $term1 = term2$ the return value is 0 par is a dummy parameter to make the parameter field identical to that of Compare1 which is the regular compare routine in [sort.c](#)

Definition at line [3153](#) of file [sort.c](#).

4.19.8.121 NextPrime()

```
WORD NextPrime (
    PHEAD WORD num ) [extern]
```

Gives the next prime number in the list of prime numbers.

If the list isn't long enough we expand it. For ease in ParForm and because these lists shouldn't be very big we let each worker keep its own list.

The list is cut off at MAXPOWER, because we don't want to get into trouble that the power of a variable gets larger than the prime number.

Definition at line [3665](#) of file [reken.c](#).

4.19.8.122 Moebius()

```
WORD Moebius (
    PHEAD WORD ) [extern]
```

Definition at line [3729](#) of file [reken.c](#).

4.19.8.123 wranf()

```
UWORD wranf (
    PHEAD0 ) [extern]
```

Definition at line [3869](#) of file [reken.c](#).

4.19.8.124 iranf()

```
UWORD iranf (
    PHEAD UWORD ) [extern]
```

Definition at line [3888](#) of file [reken.c](#).

4.19.8.125 iniwranf()

```
void iniwranf (
    PHEAD0 ) [extern]
```

Definition at line 3820 of file [reken.c](#).

4.19.8.126 PreRandom()

```
UBYTE * PreRandom (
    UBYTE * s ) [extern]
```

Definition at line 3913 of file [reken.c](#).

4.19.8.127 TreatPolyRatFun()

```
int TreatPolyRatFun (
    PHEAD WORD * prf ) [extern]
```

Definition at line 4980 of file [normal.c](#).

4.19.8.128 ReadSaveHeader()

```
int ReadSaveHeader (
    void ) [extern]
```

Reads the header in the save file and sets function pointers and flags according to the information found there. Must be called before any other ReadSave... function.

Currently works only for the exchange between 32bit and 64bit machines (WORD size must be 2 or 4 bytes)!

It is called by CoLoad().

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 4057 of file [store.c](#).

4.19.8.129 ReadSaveIndex()

```
LONG ReadSaveIndex (
    FILEINDEX * fileind ) [extern]
```

Reads a FILEINDEX from the open save file specified by AO.SaveData.Handle. Translations for adjusting endianness and data sizes are done if necessary.

Depends on the assumption that sizeof(FILEINDEX) is the same everywhere. If FILEINDEX or INDEXENTRY change, then this functions has to be adjusted.

Called by CoLoad() and FindInIndex().

Parameters

<i>fileind</i>	contains the read FILEINDEX after successful return. must point to allocated, big enough memory.

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 4175 of file [store.c](#).

References [INFILEINDEX](#), [FiLeInDeX::next](#), [FiLeInDeX::number](#), and [FuNcTiOn::number](#).

4.19.8.130 ReadSaveExpression()

```
LONG ReadSaveExpression (
    UBYTE * buffer,
    UBYTE * top,
    LONG * size,
    LONG * outsize ) [extern]
```

Reads an expression from the open file specified by AO.SaveData.Handle. The endianness flip and a resizing without renumbering is done in this function. Thereafter the buffer consists of chunks with a uniform maximal word size (32bit at the moment). The actual renumbering is then done by calling the function [ReadSaveTerm32\(\)](#). The result is returned in *buffer*.

If the translation at some point doesn't fit into the buffer anymore, the function returns and must be called again. In any case *size* returns the number of successfully read bytes, *outsize* returns the number of successfully written bytes, and the file will be positioned at the next byte after the successfully read data.

It is called by PutInStore().

Parameters

<i>buffer</i>	output buffer, holds the (translated) expression
<i>top</i>	end of buffer
<i>size</i>	number of read bytes
<i>outsize</i>	number of written bytes

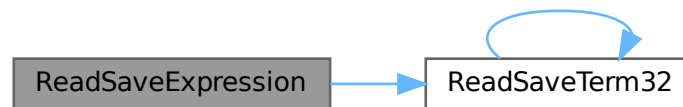
Returns

= 0 everything okay, != 0 an error occurred

Definition at line 5138 of file [store.c](#).

References [ReadSaveTerm32\(\)](#).

Here is the call graph for this function:



4.19.8.131 ReadSaveTerm32()

```

UBYTE * ReadSaveTerm32 (
    UBYTE * bin,
    UBYTE * binend,
    UBYTE ** bout,
    UBYTE * boutend,
    UBYTE * top,
    int terminbuf ) [extern]
  
```

Reads a single term from the given buffer at *bin* and write the translated term back to this buffer at *bout*.

[ReadSaveTerm32\(\)](#) is currently the only instantiation of a `ReadSaveTerm`-function. It only deals with data that already has the correct endianness and that is resized to 32bit words but without being renumbered or translated in any other way. It uses the compress buffer `AR.CompressBuffer`.

The function is reentrant in order to cope with nested function arguments. It is called by [ReadSaveExpression\(\)](#) and itself.

The *return value* indicates the position in the input buffer up to which the data has already been successfully processed. The parameter *bout* returns the corresponding position in the output buffer.

Parameters

<i>bin</i>	start of the input buffer
<i>binend</i>	end of the input buffer
<i>bout</i>	as input points to the beginning of the output buffer, as output points behind the already translated data in the output buffer
<i>boutend</i>	end of already decompressed data in output buffer
<i>top</i>	end of output buffer
<i>terminbuf</i>	flag whether decompressed data is already in the output buffer. used in recursive calls

Returns

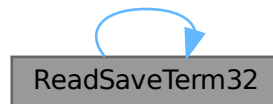
pointer to the next unprocessed data in the input buffer

Definition at line [4728](#) of file [store.c](#).

References [ReadSaveTerm32\(\)](#).

Referenced by [ReadSaveExpression\(\)](#), and [ReadSaveTerm32\(\)](#).

Here is the call graph for this function:



4.19.8.132 ReadSaveVariables()

```

LONG ReadSaveVariables (
    UBYTE * buffer,
    UBYTE * top,
    LONG * size,
    LONG * outsize,
    INDEXENTRY * ind,
    LONG * stage ) [extern]
  
```

Reads the variables from the open file specified by AO.SaveData.Handle. It reads the **size* bytes and writes them to the **buffer*. It is called by PutInStore().

If translation is necessary, the data might shrink or grow in size, then **size* is adjusted so that the reading and writing fits into the memory from the buffer to the top. The actual number of read bytes is returned in **size*, the number of written bytes is returned in **outsize*.

If the **size* is smaller than the actual size of the variables, this function will be called several times and needs to remember the current position in the variable structure. The parameter *stage* does this job. When [ReadSaveVariables\(\)](#) is called for the first time, this parameter should have the value -1.

The parameter *ind* is used to get the number of variables.

Parameters

<i>buffer</i>	read variables are written into this allocated memory
<i>top</i>	upper end of allocated memory
<i>size</i>	number of bytes to read. might return a smaller number of read bytes if translation was necessary
<i>outsize</i>	if translation has be done, outsize contains the number of written bytes
<i>ind</i>	pointer of INDEXENTRY for the current expression. read-only
<i>stage</i>	should be -1 for the first call, will be increased by ReadSaveVariables to memorize the position in the variable structure

Returns

= 0 everything okay, != 0 an error occurred

Definition at line [4361](#) of file [store.c](#).

References [InDeXeNtRy::nfunctions](#), [InDeXeNtRy::nindices](#), [InDeXeNtRy::nsymbols](#), [InDeXeNtRy::nvectors](#), and [FuNcTiOn::spec](#).

4.19.8.133 WriteStoreHeader()

```
LONG WriteStoreHeader (
    WORD handle ) [extern]
```

Writes header with information about system architecture and FORM revision to an open store file.

Called by [SetFileIndex\(\)](#).

Parameters

<i>handle</i>	specifies open file to which header will be written

Returns

= 0 everything okay, != 0 an error occurred

Definition at line [3964](#) of file [store.c](#).

References [STOREHEADER::endianness](#), [STOREHEADER::lenLONG](#), [STOREHEADER::lenPOINTER](#), [STOREHEADER::lenPOS](#), [STOREHEADER::lenWORD](#), [STOREHEADER::maxpower](#), [STOREHEADER::sFun](#), [STOREHEADER::sInd](#), [STOREHEADER::sSym](#), [STOREHEADER::sVec](#), and [STOREHEADER::wildoffset](#).

Referenced by [SetFileIndex\(\)](#).

4.19.8.134 InitRecovery()

```
void InitRecovery (
    void ) [extern]
```

Sets up the strings for the filenames of the recovery files. This functions should only be called once to avoid memory leaks and after AM.TempDir has been initialized.

Definition at line [399](#) of file [checkpoint.c](#).

4.19.8.135 CheckRecoveryFile()

```
int CheckRecoveryFile (
    void ) [extern]
```

Checks whether a snapshot/recovery file exists. Returns 1 if it exists, 0 otherwise.

Definition at line [278](#) of file [checkpoint.c](#).

References [RecoveryFilename\(\)](#).

Here is the call graph for this function:



4.19.8.136 DeleteRecoveryFile()

```
void DeleteRecoveryFile (  
    void ) [extern]
```

Deletes the recovery files. It is called by `CleanUp()` in the case of a successful completion.

Definition at line 333 of file [checkpoint.c](#).

4.19.8.137 RecoveryFilename()

```
char * RecoveryFilename (  
    void ) [extern]
```

Returns pointer to recovery filename.

Definition at line 364 of file [checkpoint.c](#).

Referenced by [CheckRecoveryFile\(\)](#), and [DoRecovery\(\)](#).

4.19.8.138 DoRecovery()

```
int DoRecovery (  
    int * moduletype ) [extern]
```

Reads from the recovery file and restores all necessary variables and states in FORM, so that the execution can recommence in preprocessor() as if no restart of FORM had occurred.

The recovery file is read into memory as a whole. The pointer `p` then points into this memory at the next non-processed data. The macros by which variables are restored, like `R_SET`, automatically increase `p` appropriately.

If something goes wrong, the function returns with a non-zero value.

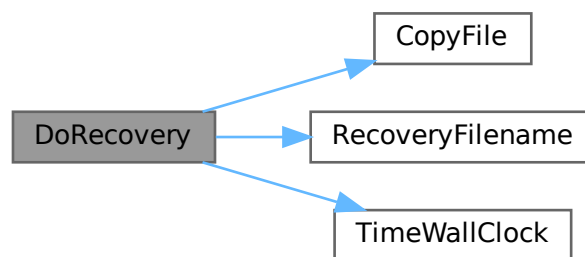
Allocated memory that would be lost when overwriting the global structs with data from the file is freed first. A major part of the code deals with the restoration of pointers. The idiom we use is to memorize the original pointer value (`org`), allocate new memory and copy the data from the file into this memory, calculate the offset between the old pointer value and the new allocated memory position (`ofs`), and then correct all affected pointers (`+=ofs`).

We rely on the fact that several variables (especially in AM) are already assigned the correct values by the startup functions. That means, in principle, that a change in the setup files between snapshot creation and recovery will be noticed.

Definition at line 1402 of file [checkpoint.c](#).

References [TaBIEs::argtail](#), [TaBIEs::boomlijst](#), [BrAcKeTiNfO::bracketbuffer](#), [TaBIEs::buffers](#), [TaBIEs::buffersize](#), [CopyFile\(\)](#), [TaBIEs::flags](#), [ReNuMbEr::func](#), [ReNuMbEr::funnum](#), [VaRrEnUm::hi](#), [BrAcKeTiNfO::indexbuffer](#), [ReNuMbEr::indi](#), [ReNuMbEr::indnum](#), [VaRrEnUm::lo](#), [TaBIEs::MaxTreeSize](#), [TaBIEs::mm](#), [TaBIEs::numind](#), [TaBIEs::pattern](#), [TaBIEs::prototype](#), [TaBIEs::prototypeSize](#), [RecoveryFilename\(\)](#), [ExPrEsSiOn::renumlists](#), [TaBIEs::reserved](#), [TaBIEs::spare](#), [TaBIEs::sparse](#), [VaRrEnUm::start](#), [ReNuMbEr::symb](#), [ReNuMbEr::symnum](#), [TaBIEs::tablepointers](#), [TimeWallClock\(\)](#), [TaBIEs::totind](#), [ReNuMbEr::vecnum](#), and [ReNuMbEr::vect](#).

Here is the call graph for this function:



4.19.8.139 DoCheckpoint()

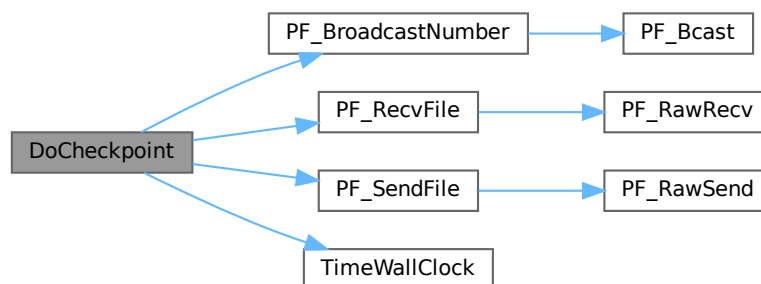
```
void DoCheckpoint (
    int moduletype ) [extern]
```

Checks whether a snapshot should be done. Calls `DoSnapshot()` to create the snapshot.

Definition at line 3111 of file [checkpoint.c](#).

References [PF_BroadcastNumber\(\)](#), [PF_RecvFile\(\)](#), [PF_SendFile\(\)](#), and [TimeWallClock\(\)](#).

Here is the call graph for this function:



4.19.8.140 NumberMallocAddMemory()

```
void NumberMallocAddMemory (
    PHEAD0 ) [extern]
```

Definition at line 2590 of file [tools.c](#).

4.19.8.141 CacheNumberMallocAddMemory()

```
void CacheNumberMallocAddMemory (
    PHEAD0 ) [extern]
```

Definition at line 2664 of file [tools.c](#).

4.19.8.142 TermMallocAddMemory()

```
void TermMallocAddMemory (
    PHEAD0 ) [extern]
```

Definition at line 2486 of file [tools.c](#).

4.19.8.143 ExprStatus()

```
void ExprStatus (
    EXPRESSIONS e ) [extern]
```

Definition at line 66 of file [bugtool.c](#).

4.19.8.144 iniTools()

```
void iniTools (
    void ) [extern]
```

Definition at line 2222 of file [tools.c](#).

4.19.8.145 TestTerm()

```
int TestTerm (
    WORD * term ) [extern]
```

Tests the consistency of the term. Returns 0 when the term is OK. Any nonzero value is trouble. In the current version the testing isn't 100% complete. For instance, we don't check the validity of the symbols nor do we check the range of their powers. Etc. This should be extended when the need is there.

Parameters

<i>term</i>	the term to be tested

Definition at line [3798](#) of file [tools.c](#).

References [TestTerm\(\)](#).

Referenced by [TestTerm\(\)](#).

Here is the call graph for this function:



4.19.8.146 RunTransform()

```
int RunTransform (
    PHEAD WORD * term,
    WORD * params ) [extern]
```

Definition at line [750](#) of file [transform.c](#).

4.19.8.147 RunEncode()

```
int RunEncode (
    PHEAD WORD * fun,
    WORD * args,
    WORD * info ) [extern]
```

Definition at line [1006](#) of file [transform.c](#).

4.19.8.148 RunDecode()

```
int RunDecode (
    PHEAD WORD * fun,
    WORD * args,
    WORD * info ) [extern]
```

Definition at line [1193](#) of file [transform.c](#).

4.19.8.149 RunReplace()

```
int RunReplace (
    PHEAD WORD * fun,
    WORD * args,
    WORD * info ) [extern]
```

Definition at line [1363](#) of file [transform.c](#).

4.19.8.150 RunImplode()

```
int RunImplode (
    WORD * fun,
    WORD * args ) [extern]
```

Definition at line 1841 of file [transform.c](#).

4.19.8.151 RunExplode()

```
int RunExplode (
    PHEAD WORD * fun,
    WORD * args ) [extern]
```

Definition at line 2042 of file [transform.c](#).

4.19.8.152 TestArgNum()

```
int TestArgNum (
    int n,
    int totarg,
    WORD * args ) [extern]
```

Definition at line 3307 of file [transform.c](#).

4.19.8.153 PutArgInScratch()

```
WORD PutArgInScratch (
    WORD * arg,
    UWORD * scrat ) [extern]
```

Definition at line 3376 of file [transform.c](#).

4.19.8.154 ReadRange()

```
UBYTE * ReadRange (
    UBYTE * s,
    WORD * out,
    int par ) [extern]
```

Definition at line 3412 of file [transform.c](#).

4.19.8.155 FindRange()

```
int FindRange (
    PHEAD WORD * args,
    WORD * arg1,
    WORD * arg2,
    WORD totarg ) [extern]
```

Definition at line 3572 of file [transform.c](#).

4.19.8.156 RunPermute()

```
int RunPermute (
    PHEAD WORD * fun,
    WORD * args,
    WORD * info ) [extern]
```

Definition at line 2156 of file [transform.c](#).

4.19.8.157 RunReverse()

```
int RunReverse (
    PHEAD WORD * fun,
    WORD * args ) [extern]
```

Definition at line 2383 of file [transform.c](#).

4.19.8.158 RunCycle()

```
int RunCycle (
    PHEAD WORD * fun,
    WORD * args,
    WORD * info ) [extern]
```

Definition at line 2559 of file [transform.c](#).

4.19.8.159 RunAddArg()

```
int RunAddArg (
    PHEAD WORD * fun,
    WORD * args ) [extern]
```

Definition at line 2711 of file [transform.c](#).

4.19.8.160 RunMulArg()

```
int RunMulArg (
    PHEAD WORD * fun,
    WORD * args ) [extern]
```

Definition at line 2800 of file [transform.c](#).

4.19.8.161 RunIsLyndon()

```
int RunIsLyndon (
    PHEAD WORD * fun,
    WORD * args,
    int par ) [extern]
```

Definition at line 2941 of file [transform.c](#).

4.19.8.162 RunToLyndon()

```
WORD RunToLyndon (
    PHEAD WORD * fun,
    WORD * args,
    int par ) [extern]
```

Definition at line [3019](#) of file [transform.c](#).

4.19.8.163 RunDropArg()

```
int RunDropArg (
    PHEAD WORD * fun,
    WORD * args ) [extern]
```

Definition at line [3131](#) of file [transform.c](#).

4.19.8.164 RunSelectArg()

```
int RunSelectArg (
    PHEAD WORD * fun,
    WORD * args ) [extern]
```

Definition at line [3157](#) of file [transform.c](#).

4.19.8.165 RunDedup()

```
int RunDedup (
    PHEAD WORD * fun,
    WORD * args ) [extern]
```

Definition at line [2470](#) of file [transform.c](#).

4.19.8.166 RunZtoHArg()

```
int RunZtoHArg (
    PHEAD WORD * fun,
    WORD * args ) [extern]
```

Definition at line [3185](#) of file [transform.c](#).

4.19.8.167 RunHtoZArg()

```
int RunHtoZArg (
    PHEAD WORD * fun,
    WORD * args ) [extern]
```

Definition at line [3241](#) of file [transform.c](#).

4.19.8.168 NormPolyTerm()

```
int NormPolyTerm (
    PHEAD WORD * term ) [extern]
```

Definition at line 53 of file [notation.c](#).

4.19.8.169 ConvertToPoly()

```
int ConvertToPoly (
    PHEAD WORD * term,
    WORD * outterm,
    WORD * comlist,
    WORD par ) [extern]
```

Definition at line 307 of file [notation.c](#).

4.19.8.170 LocalConvertToPoly()

```
int LocalConvertToPoly (
    PHEAD WORD * term,
    WORD * outterm,
    WORD startebuf,
    WORD par ) [extern]
```

Converts a generic term to polynomial notation in which there are only symbols and brackets. During conversion there will be only symbols. Brackets are stripped. Objects that need 'translation' are put inside a special compiler buffer and represented by a symbol. The numbering of the extra symbols is down from the maximum. In principle there can be a problem when running into the already assigned ones. This uses the FindTree for searching in the global tree and then looks further in the AT.ebufnum. This allows fully parallel processing. Hence we need no locks. Cannot be used in the same module as ConvertToPoly.

Definition at line 510 of file [notation.c](#).

Referenced by [poly_factorize_expression\(\)](#).

4.19.8.171 ConvertFromPoly()

```
int ConvertFromPoly (
    PHEAD WORD * term,
    WORD * outterm,
    WORD from,
    WORD to,
    WORD offset,
    WORD par ) [extern]
```

Definition at line 660 of file [notation.c](#).

4.19.8.172 FindSubterm()

```
int FindSubterm (
    WORD * subterm ) [extern]
```

Definition at line 773 of file [notation.c](#).

4.19.8.173 FindLocalSubterm()

```
int FindLocalSubterm (
    PHEAD WORD * subterm,
    WORD startebuf ) [extern]
```

Definition at line [843](#) of file [notation.c](#).

4.19.8.174 PrintSubtermList()

```
void PrintSubtermList (
    int from,
    int to ) [extern]
```

Definition at line [897](#) of file [notation.c](#).

4.19.8.175 PrintExtraSymbol()

```
void PrintExtraSymbol (
    int num,
    WORD * terms,
    int par ) [extern]
```

Definition at line [1009](#) of file [notation.c](#).

4.19.8.176 FindSubexpression()

```
int FindSubexpression (
    WORD * subexpr ) [extern]
```

Definition at line [1096](#) of file [notation.c](#).

4.19.8.177 UpdateMaxSize()

```
void UpdateMaxSize (
    void ) [extern]
```

Definition at line [1612](#) of file [tools.c](#).

4.19.8.178 CoToPolynomial()

```
int CoToPolynomial (
    UBYTE * inp ) [extern]
```

Definition at line [5946](#) of file [compcomm.c](#).

4.19.8.179 CoFromPolynomial()

```
int CoFromPolynomial (
    UBYTE * inp ) [extern]
```

Definition at line [6021](#) of file [compcomm.c](#).

4.19.8.180 CoArgToExtraSymbol()

```
int CoArgToExtraSymbol (
    UBYTE * s ) [extern]
```

Definition at line [6047](#) of file [compcomm.c](#).

4.19.8.181 CoExtraSymbols()

```
int CoExtraSymbols (
    UBYTE * inp ) [extern]
```

Definition at line [6097](#) of file [compcomm.c](#).

4.19.8.182 GetDoParam()

```
UBYTE * GetDoParam (
    UBYTE * inp,
    WORD ** wp,
    int par ) [extern]
```

Definition at line [6228](#) of file [compcomm.c](#).

4.19.8.183 GetIfDollarFactor()

```
WORD * GetIfDollarFactor (
    UBYTE ** inp,
    WORD * w ) [extern]
```

Definition at line [6170](#) of file [compcomm.c](#).

4.19.8.184 CoDo()

```
int CoDo (
    UBYTE * inp ) [extern]
```

Definition at line [6305](#) of file [compcomm.c](#).

4.19.8.185 CoEndDo()

```
int CoEndDo (
    UBYTE * inp ) [extern]
```

Definition at line 6408 of file [compcomm.c](#).

4.19.8.186 ExtraSymFun()

```
int ExtraSymFun (
    PHEAD WORD * term,
    WORD level ) [extern]
```

Definition at line 1149 of file [notation.c](#).

4.19.8.187 PruneExtraSymbols()

```
int PruneExtraSymbols (
    WORD downto ) [extern]
```

Definition at line 1217 of file [notation.c](#).

4.19.8.188 IniFbuffer()

```
int IniFbuffer (
    WORD bufnum ) [extern]
```

Initialize a factorization cache buffer. We set the size of the rhs and boomlijst buffers immediately to their final values.

Definition at line 614 of file [comtool.c](#).

References [tree::blnce](#), [CbUf::boomlijst](#), [CbUf::CanCommu](#), [CbUf::dimension](#), [tree::left](#), [CbUf::numdum](#), [CbUf::NumTerms](#), [tree::parent](#), [CbUf::rhs](#), [tree::right](#), [tree::usage](#), and [tree::value](#).

4.19.8.189 GCDfunction()

```
int GCDfunction (
    PHEAD WORD * term,
    WORD level ) [extern]
```

Definition at line 609 of file [ratio.c](#).

4.19.8.190 GCDfunction3()

```
WORD * GCDfunction3 (
    PHEAD WORD * in1,
    WORD * in2 ) [extern]
```

Definition at line 1141 of file [ratio.c](#).

4.19.8.191 ReadPolyRatFun()

```
int ReadPolyRatFun (
    PHEAD WORD * term ) [extern]
```

Definition at line 2264 of file [ratio.c](#).

4.19.8.192 FromPolyRatFun()

```
int FromPolyRatFun (
    PHEAD WORD * fun,
    WORD ** numout,
    WORD ** denout ) [extern]
```

Definition at line 2383 of file [ratio.c](#).

4.19.8.193 PolyDiv()

```
WORD * PolyDiv (
    PHEAD WORD * a,
    WORD * b,
    char * text ) [extern]
```

Definition at line 2741 of file [ratio.c](#).

4.19.8.194 GCDclean()

```
void GCDclean (
    PHEAD WORD * num,
    WORD * den ) [extern]
```

Definition at line 2644 of file [ratio.c](#).

4.19.8.195 TakeSymbolContent()

```
WORD * TakeSymbolContent (
    PHEAD WORD * in,
    WORD * term ) [extern]
```

Implements part of the old ExecArg in which we take common factors from arguments with more than one term. We allow only symbols as this code is used for the polyratfun only. We have a special routine, because the generic TakeContent does too much work and speed is at a premium here. Input: in is the input expression as a sequence of terms. **Output:** term: the content return value: the contentfree expression. it is in new allocation, made by TermMalloc. (should be in a TermMalloc space?)

Definition at line 2434 of file [ratio.c](#).

References [GetModInverses\(\)](#).

Here is the call graph for this function:



4.19.8.196 GCDterms()

```
int GCDterms (
    PHEAD WORD * term1,
    WORD * term2,
    WORD * termout ) [extern]
```

Definition at line 2076 of file [ratio.c](#).

4.19.8.197 PutExtraSymbols()

```
WORD * PutExtraSymbols (
    PHEAD WORD * in,
    WORD startebuf,
    int * actionflag ) [extern]
```

Definition at line 1244 of file [ratio.c](#).

4.19.8.198 TakeExtraSymbols()

```
WORD * TakeExtraSymbols (
    PHEAD WORD * in,
    WORD startebuf ) [extern]
```

Definition at line 1273 of file [ratio.c](#).

4.19.8.199 MultiplyWithTerm()

```
WORD * MultiplyWithTerm (
    PHEAD WORD * in,
    WORD * term,
    WORD par ) [extern]
```

Definition at line 1312 of file [ratio.c](#).

4.19.8.200 TakeContent()

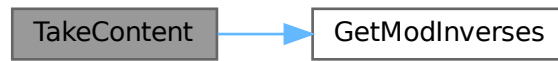
```
WORD * TakeContent (
    PHEAD WORD * in,
    WORD * term ) [extern]
```

Implements part of the old ExecArg in which we take common factors from arguments with more than one term. Here the input is a sequence of terms in 'in' and the answer is a content-free sequence of terms. This sequence has been allocated by the Malloc1 routine in a call to EndSort, unless the expression was already content-free. In that case the input pointer is returned. The content is returned in term. This is supposed to be a separate allocation, made by TermMalloc in the calling routine.

Definition at line 1376 of file [ratio.c](#).

References [GetModInverses\(\)](#).

Here is the call graph for this function:



4.19.8.201 MergeSymbolLists()

```
int MergeSymbolLists (  
    PHEAD WORD * old,  
    WORD * extra,  
    int par ) [extern]
```

Definition at line 1808 of file [ratio.c](#).

4.19.8.202 MergeDotproductLists()

```
int MergeDotproductLists (  
    PHEAD WORD * old,  
    WORD * extra,  
    int par ) [extern]
```

Definition at line 1927 of file [ratio.c](#).

4.19.8.203 CreateExpression()

```
WORD * CreateExpression (  
    PHEAD WORD ) [extern]
```

Definition at line 2020 of file [ratio.c](#).

4.19.8.204 DIVfunction()

```
int DIVfunction (  
    PHEAD WORD * term,  
    WORD level,  
    int par ) [extern]
```

Definition at line 2788 of file [ratio.c](#).

4.19.8.205 MULfunc()

```
WORD * MULfunc (
    PHEAD WORD * p1,
    WORD * p2 ) [extern]
```

Definition at line [2957](#) of file [ratio.c](#).

4.19.8.206 ConvertArgument()

```
WORD * ConvertArgument (
    PHEAD WORD * arg,
    int * type ) [extern]
```

Definition at line [3018](#) of file [ratio.c](#).

4.19.8.207 ExpandRat()

```
int ExpandRat (
    PHEAD WORD * fun ) [extern]
```

Definition at line [3113](#) of file [ratio.c](#).

4.19.8.208 InvPoly()

```
int InvPoly (
    PHEAD WORD * inpoly,
    WORD maxpow,
    WORD sym ) [extern]
```

Definition at line [3592](#) of file [ratio.c](#).

4.19.8.209 TestDoLoop()

```
WORD TestDoLoop (
    PHEAD WORD * lhsbuf,
    WORD level ) [extern]
```

Definition at line [2762](#) of file [dollar.c](#).

4.19.8.210 TestEndDoLoop()

```
WORD TestEndDoLoop (
    PHEAD WORD * lhsbuf,
    WORD level ) [extern]
```

Definition at line [2840](#) of file [dollar.c](#).

4.19.8.211 poly_gcd()

```
WORD * poly_gcd (  
    PHEAD WORD * a,  
    WORD * b,  
    WORD fit ) [extern]
```

Polynomial gcd

4.19.9 Description

This method calculates the greatest common divisor of two polynomials, given by two zero-terminated Form-style term lists.

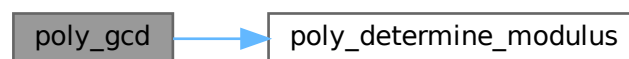
4.19.10 Notes

- The result is written at newly allocated memory
- Called from [ratio.c](#)
- Calls `polygcd::gcd`

Definition at line [124](#) of file [polywrap.cc](#).

References [poly_determine_modulus\(\)](#).

Here is the call graph for this function:



4.19.10.1 poly_div()

```
WORD * poly_div (  
    PHEAD WORD * a,  
    WORD * b,  
    WORD fit ) [extern]
```

Definition at line [435](#) of file [polywrap.cc](#).

4.19.10.2 poly_rem()

```
WORD * poly_rem (
    PHEAD WORD * a,
    WORD * b,
    WORD fit ) [extern]
```

Definition at line 463 of file [polywrap.cc](#).

4.19.10.3 poly_inverse()

```
WORD * poly_inverse (
    PHEAD WORD * arga,
    WORD * argb ) [extern]
```

Definition at line 1743 of file [polywrap.cc](#).

4.19.10.4 poly_mul()

```
WORD * poly_mul (
    PHEAD WORD * a,
    WORD * b ) [extern]
```

Definition at line 1948 of file [polywrap.cc](#).

4.19.10.5 poly_ratfun_add()

```
WORD * poly_ratfun_add (
    PHEAD WORD * t1,
    WORD * t2 ) [extern]
```

Addition of PolyRatFuns

4.19.11 Description

This method gets two pointers to polyratfuns with up to two arguments each and calculates the sum.

4.19.14 Notes

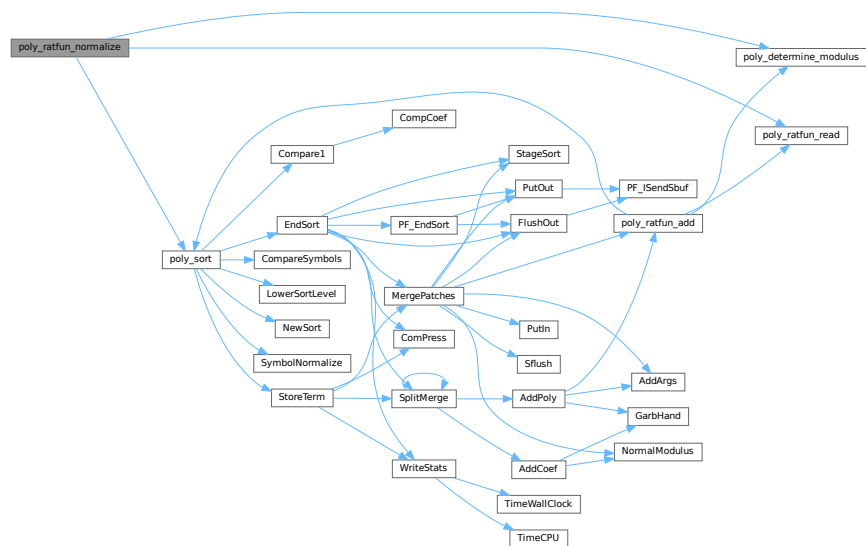
- The result overwrites the original term
- Called from [proces.c](#)
- Calls `poly::operators` and `polygcd::gcd`

Definition at line 769 of file [polywrap.cc](#).

References [poly_determine_modulus\(\)](#), [poly_ratfun_read\(\)](#), and [poly_sort\(\)](#).

Referenced by [PolyFunMul\(\)](#), and [PrepPoly\(\)](#).

Here is the call graph for this function:



4.19.14.1 poly_factorize_argument()

```

int poly_factorize_argument (
    PHEAD WORD * argin,
    WORD * argout ) [extern]
  
```

Factorization of function arguments

4.19.15 Description

This method factorizes the Form-style argument `argin`.

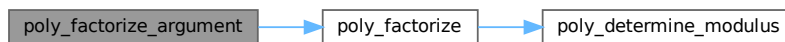
4.19.16 Notes

- The result is written at argout
- Called from [argument.c](#)
- Calls `poly_factorize`

Definition at line 1112 of file [polywrap.cc](#).

References [poly_factorize\(\)](#).

Here is the call graph for this function:



4.19.16.1 `poly_factorize_dollar()`

```
WORD * poly_factorize_dollar (
    PHEAD WORD * argin ) [extern]
```

Factorization of dollar variables

4.19.17 Description

This method factorizes a dollar variable.

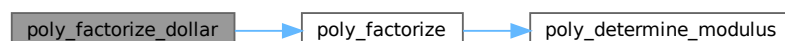
4.19.18 Notes

- The result is written at newly allocated memory.
- Called from [dollar.c](#)
- Calls `poly_factorize`

Definition at line 1146 of file [polywrap.cc](#).

References [poly_factorize\(\)](#).

Here is the call graph for this function:



4.19.18.1 poly_factorize_expression()

```
int poly_factorize_expression (
    EXPRESSIONS expr ) [extern]
```

Factorization of expressions

4.19.19 Description

This method factorizes an expression.

4.19.20 Notes

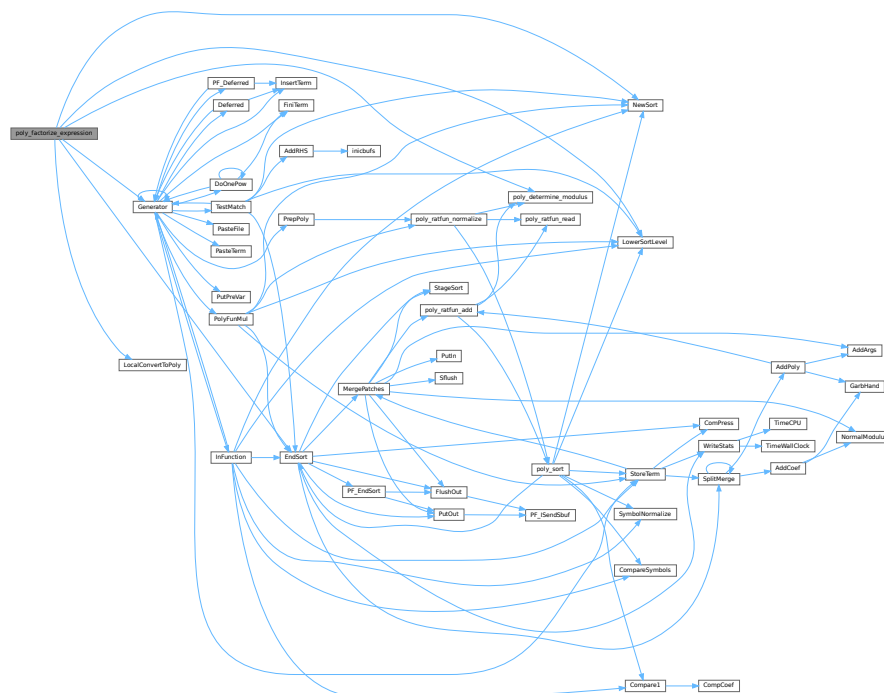
- The result overwrites the input expression
- Called from [proces.c](#)
- Calls `polyfact::factorize`

Definition at line 1178 of file [polywrap.cc](#).

References [EndSort\(\)](#), [Generator\(\)](#), [File::handle](#), [LocalConvertToPoly\(\)](#), [LowerSortLevel\(\)](#), [NewSort\(\)](#), and [poly_determine_modulus\(\)](#).

Referenced by [PF_Processor\(\)](#), and [Processor\(\)](#).

Here is the call graph for this function:



4.19.20.1 poly_unfactorize_expression()

```
int poly_unfactorize_expression (
    EXPRESSIONS expr ) [extern]
```

Unfactorization of expressions

4.19.21 Description

This method expands a factorized expression.

4.19.22 Notes

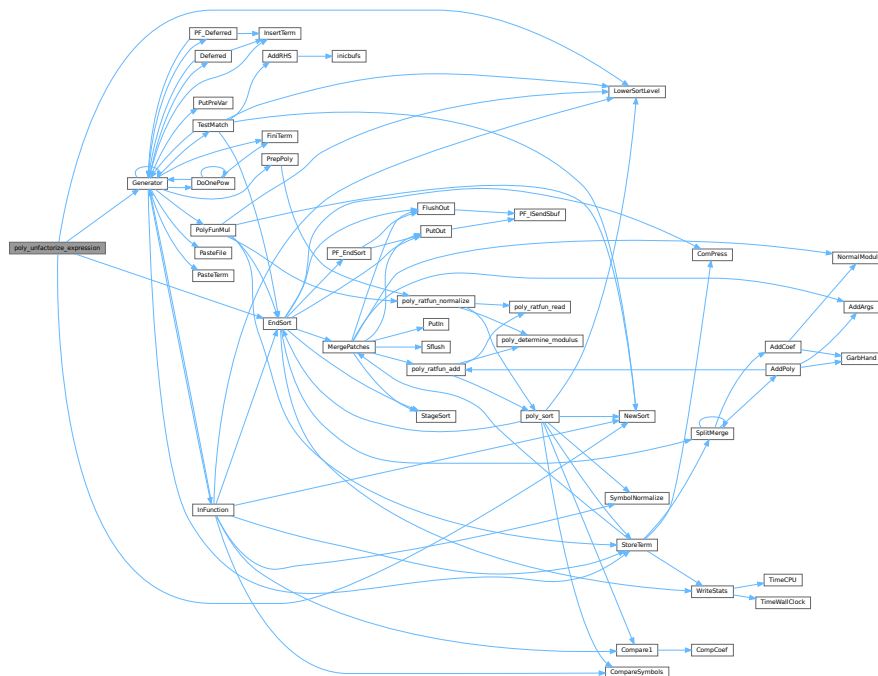
- The result overwrites the input expression
- Called from [proces.c](#)

Definition at line 1535 of file [polywrap.cc](#).

References [EndSort\(\)](#), [Generator\(\)](#), [FiLe::handle](#), [LowerSortLevel\(\)](#), and [NewSort\(\)](#).

Referenced by [PF_Processor\(\)](#), and [Processor\(\)](#).

Here is the call graph for this function:



4.19.22.1 poly_free_poly_vars()

```
void poly_free_poly_vars (
    PHEAD const char * text ) [extern]
```

Definition at line 2014 of file [polywrap.cc](#).

4.19.22.2 optimize_print_code()

```
void optimize_print_code (
    int print_expr ) [extern]
```

Print optimized code

4.19.23 Description

This method prints the optimized code via "PrintExtraSymbol". Depending on the flag, the original expression is printed (for "Print") or not (for "#Optimize / #write "O").

Definition at line 4524 of file [optimize.cc](#).

Referenced by [Optimize\(\)](#).

4.19.23.1 DoPreAdd()

```
int DoPreAdd (
    UBYTE * s ) [extern]
```

Definition at line 1067 of file [dict.c](#).

4.19.23.2 DoPreUseDictionary()

```
int DoPreUseDictionary (
    UBYTE * s ) [extern]
```

Definition at line 1022 of file [dict.c](#).

4.19.23.3 DoPreCloseDictionary()

```
int DoPreCloseDictionary (
    UBYTE * s ) [extern]
```

Definition at line 998 of file [dict.c](#).

4.19.23.4 DoPreOpenDictionary()

```
int DoPreOpenDictionary (
    UBYTE * s ) [extern]
```

Definition at line 959 of file [dict.c](#).

4.19.23.5 RemoveDictionary()

```
void RemoveDictionary (
    DICTIONARY * dict ) [extern]
```

Definition at line 902 of file [dict.c](#).

4.19.23.6 UnSetDictionary()

```
void UnSetDictionary (
    void ) [extern]
```

Definition at line 883 of file [dict.c](#).

4.19.23.7 SetDictionaryOptions()

```
int SetDictionaryOptions (
    UBYTE * options ) [extern]
```

Definition at line 790 of file [dict.c](#).

4.19.23.8 AddToDictionary()

```
int AddToDictionary (
    DICTIONARY * dict,
    UBYTE * left,
    UBYTE * right ) [extern]
```

Definition at line 491 of file [dict.c](#).

4.19.23.9 AddDictionary()

```
int AddDictionary (
    UBYTE * name ) [extern]
```

Definition at line 449 of file [dict.c](#).

4.19.23.10 FindDictionary()

```
int FindDictionary (
    UBYTE * name ) [extern]
```

Definition at line 434 of file [dict.c](#).

4.19.23.11 IsExponentSign()

```
UBYTE * IsExponentSign (
    void ) [extern]
```

Definition at line 238 of file [dict.c](#).

4.19.23.12 IsMultiplySign()

```
UBYTE * IsMultiplySign (
    void ) [extern]
```

Definition at line 218 of file [dict.c](#).

4.19.23.13 TransformRational()

```
void TransformRational (
    UWORD * a,
    WORD na ) [extern]
```

Definition at line 71 of file [dict.c](#).

4.19.23.14 WriteDictionary()

```
void WriteDictionary (
    DICTIONARY * dict ) [extern]
```

Definition at line 1289 of file [sch.c](#).

4.19.23.15 ShrinkDictionary()

```
void ShrinkDictionary (
    DICTIONARY * dict ) [extern]
```

Definition at line 945 of file [dict.c](#).

4.19.23.16 MultiplyToLine()

```
void MultiplyToLine (
    void ) [extern]
```

Definition at line 653 of file [sch.c](#).

4.19.23.17 FindSymbol()

```
UBYTE * FindSymbol (
    WORD num ) [extern]
```

Definition at line 258 of file [dict.c](#).

4.19.23.18 FindVector()

```
UBYTE * FindVector (
    WORD num ) [extern]
```

Definition at line 279 of file [dict.c](#).

4.19.23.19 FindIndex()

```
UBYTE * FindIndex (
    WORD num ) [extern]
```

Definition at line 301 of file [dict.c](#).

4.19.23.20 FindFunction()

```
UBYTE * FindFunction (
    WORD num ) [extern]
```

Definition at line 323 of file [dict.c](#).

4.19.23.21 FindFunWithArgs()

```
UBYTE * FindFunWithArgs (
    WORD * t ) [extern]
```

Definition at line 345 of file [dict.c](#).

4.19.23.22 FindExtraSymbol()

```
UBYTE * FindExtraSymbol (
    WORD num ) [extern]
```

Definition at line 377 of file [dict.c](#).

4.19.23.23 DictToBytes()

```
LONG DictToBytes (
    DICTIONARY * dict,
    UBYTE * buf ) [extern]
```

Definition at line 1119 of file [dict.c](#).

4.19.23.24 DictFromBytes()

```
DICTIONARY * DictFromBytes (
    UBYTE * buf ) [extern]
```

Definition at line 1152 of file [dict.c](#).

4.19.23.25 CoCreateSpectator()

```
int CoCreateSpectator (
    UBYTE * inp ) [extern]
```

Definition at line 89 of file [spectator.c](#).

4.19.23.26 CoToSpectator()

```
int CoToSpectator (
    UBYTE * inp ) [extern]
```

Definition at line 181 of file [spectator.c](#).

4.19.23.27 CoRemoveSpectator()

```
int CoRemoveSpectator (
    UBYTE * inp ) [extern]
```

Definition at line 233 of file [spectator.c](#).

4.19.23.28 CoEmptySpectator()

```
int CoEmptySpectator (
    UBYTE * inp ) [extern]
```

Definition at line 293 of file [spectator.c](#).

4.19.23.29 CoCopySpectator()

```
int CoCopySpectator (
    UBYTE * inp ) [extern]
```

Definition at line 459 of file [spectator.c](#).

4.19.23.30 PutInSpectator()

```
int PutInSpectator (
    WORD * term,
    WORD specnum ) [extern]
```

Definition at line 345 of file [spectator.c](#).

4.19.23.31 ClearSpectators()

```
void ClearSpectators (
    WORD par ) [extern]
```

Definition at line 675 of file [spectator.c](#).

4.19.23.32 GetFromSpectator()

```
WORD GetFromSpectator (
    WORD * term,
    WORD specnum ) [extern]
```

Definition at line 601 of file [spectator.c](#).

4.19.23.33 FlushSpectators()

```
void FlushSpectators (
    void ) [extern]
```

Definition at line 402 of file [spectator.c](#).

4.19.23.34 ReadFromScratch()

```
int ReadFromScratch (
    FILEHANDLE * fi,
    POSITION * pos,
    UBYTE * buffer,
    POSITION * length ) [extern]
```

Definition at line 272 of file [store.c](#).

4.19.23.35 AddToScratch()

```
int AddToScratch (
    FILEHANDLE * fi,
    POSITION * pos,
    UBYTE * buffer,
    POSITION * length,
    int withflush ) [extern]
```

Definition at line 299 of file [store.c](#).

4.19.23.36 DoPreAppendPath()

```
int DoPreAppendPath (
    UBYTE * s ) [extern]
```

Appends the given path (absolute or relative to the current file directory) to the FORM path.

Syntax: #appendpath <path>

Definition at line 7231 of file [pre.c](#).

4.19.23.37 DoPrePrependPath()

```
int DoPrePrependPath (
    UBYTE * s ) [extern]
```

Prepends the given path (absolute or relative to the current file directory) to the FORM path.

Syntax: #prependpath <path>

Definition at line 7248 of file [pre.c](#).

4.19.23.38 DoSwitch()

```
int DoSwitch (
    PHEAD WORD * term,
    WORD * lhs ) [extern]
```

Definition at line 1070 of file [if.c](#).

4.19.23.39 DoEndSwitch()

```
int DoEndSwitch (
    PHEAD WORD * term,
    WORD * lhs ) [extern]
```

Definition at line 1086 of file [if.c](#).

4.19.23.40 FindCase()

```
SWITCHTABLE * FindCase (
    WORD nswitch,
    WORD ncase ) [extern]
```

Definition at line 1097 of file [if.c](#).

4.19.23.41 SwitchSplitMergeRec()

```
void SwitchSplitMergeRec (
    SWITCHTABLE * array,
    WORD num,
    SWITCHTABLE * auxarray ) [extern]
```

Definition at line 1184 of file [if.c](#).

4.19.23.42 SwitchSplitMerge()

```
void SwitchSplitMerge (
    SWITCHTABLE * array,
    WORD num ) [extern]
```

Definition at line 1213 of file [if.c](#).

4.19.23.43 DoubleSwitchBuffers()

```
int DoubleSwitchBuffers (
    void ) [extern]
```

Definition at line [1135](#) of file [if.c](#).

4.19.23.44 DistrN()

```
int DistrN (
    int n,
    int * cpl,
    int ncpl,
    int * scratch ) [extern]
```

Definition at line [3984](#) of file [tools.c](#).

4.19.23.45 DoNamespace()

```
int DoNamespace (
    UBYTE * s ) [extern]
```

Definition at line [7312](#) of file [pre.c](#).

4.19.23.46 DoEndNamespace()

```
int DoEndNamespace (
    UBYTE * s ) [extern]
```

Definition at line [7360](#) of file [pre.c](#).

4.19.23.47 DoUse()

```
int DoUse (
    UBYTE * s ) [extern]
```

Definition at line [7575](#) of file [pre.c](#).

4.19.23.48 SkipName()

```
UBYTE * SkipName (
    UBYTE * s ) [extern]
```

Definition at line [7383](#) of file [pre.c](#).

4.19.23.49 ConstructName()

```
UBYTE * ConstructName (
    UBYTE * s,
    UBYTE type ) [extern]
```

Definition at line 7483 of file [pre.c](#).

4.19.23.50 DoSetUserFlag()

```
int DoSetUserFlag (
    UBYTE * s ) [extern]
```

Definition at line 7740 of file [pre.c](#).

4.19.23.51 DoClearUserFlag()

```
int DoClearUserFlag (
    UBYTE * s ) [extern]
```

Definition at line 7730 of file [pre.c](#).

4.19.23.52 CreateVertex()

```
VERTEX * CreateVertex (
    MODEL * m )
```

Definition at line 76 of file [model.c](#).

4.19.23.53 ReadParticle()

```
UBYTE * ReadParticle (
    UBYTE * s,
    VERTEX * v,
    MODEL * m,
    int par )
```

Definition at line 106 of file [model.c](#).

4.19.23.54 CoModel()

```
int CoModel (
    UBYTE * s )
```

Definition at line 149 of file [model.c](#).

4.19.23.55 CoParticle()

```
int CoParticle (
    UBYTE * s )
```

Definition at line 224 of file [model.c](#).

4.19.23.56 CoVertex()

```
int CoVertex (
    UBYTE * s )
```

Definition at line 306 of file [model.c](#).

4.19.23.57 CoEndModel()

```
int CoEndModel (
    UBYTE * s )
```

Definition at line 418 of file [model.c](#).

4.19.23.58 LoadModel()

```
int LoadModel (
    MODEL * m )
```

Definition at line 57 of file [diawrap.cc](#).

4.19.23.59 CoSetUserFlag()

```
int CoSetUserFlag (
    UBYTE * s )
```

Definition at line 7405 of file [compcomm.c](#).

4.19.23.60 CoClearUserFlag()

```
int CoClearUserFlag (
    UBYTE * s )
```

Definition at line 7433 of file [compcomm.c](#).

4.19.23.61 CoCreateAllLoops()

```
int CoCreateAllLoops (
    UBYTE * s )
```

Definition at line 7470 of file [compcomm.c](#).

4.19.23.62 CoCreateAllPaths()

```
int CoCreateAllPaths (
    UBYTE * s )
```

Definition at line [7590](#) of file [compcomm.c](#).

4.19.23.63 CoCreateAll()

```
int CoCreateAll (
    UBYTE * s )
```

Definition at line [7718](#) of file [compcomm.c](#).

4.19.23.64 AllLoops()

```
int AllLoops (
    PHEAD WORD * term,
    WORD level )
```

Definition at line [650](#) of file [lus.c](#).

4.19.23.65 StartLoops()

```
LONG StartLoops (
    PHEAD WORD * term,
    WORD level,
    LONG vert,
    WORD nvert,
    WORD * arglist,
    WORD nargs,
    WORD * loop,
    WORD nloop )
```

Definition at line [894](#) of file [lus.c](#).

4.19.23.66 GenLoops()

```
LONG GenLoops (
    PHEAD WORD * term,
    WORD level,
    LONG vert,
    WORD nvert,
    WORD * arglist,
    WORD nargs,
    WORD * loop,
    WORD nloop )
```

Definition at line [980](#) of file [lus.c](#).

4.19.23.67 LoopOutput()

```
void LoopOutput (
    PHEAD WORD * term,
    WORD level,
    WORD * loop,
    WORD nloop )
```

Definition at line [1059](#) of file [lus.c](#).

4.19.23.68 AllPaths()

```
int AllPaths (
    PHEAD WORD * term,
    WORD level )
```

Definition at line [1123](#) of file [lus.c](#).

4.19.23.69 GenPaths()

```
LONG GenPaths (
    PHEAD WORD * term,
    WORD level,
    LONG vert,
    WORD nvert,
    WORD * arglist,
    WORD nargs,
    WORD * path,
    WORD npath )
```

Definition at line [1413](#) of file [lus.c](#).

4.19.23.70 PathOutput()

```
void PathOutput (
    PHEAD WORD * term,
    WORD level,
    WORD * path,
    WORD npath )
```

Definition at line [1486](#) of file [lus.c](#).

4.20 declare.h

[Go to the documentation of this file.](#)

```

00001 #ifndef __FDECLARE__
00002 #define __FDECLARE__
00003
00009 /* #[ License : */
00010 /*
00011 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00012 * When using this file you are requested to refer to the publication
00013 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014 * This is considered a matter of courtesy as the development was paid
00015 * for by FOM the Dutch physics granting agency and we would like to
00016 * be able to track its scientific use to convince FOM of its value
00017 * for the community.
00018 *
00019 * This file is part of FORM.
00020 *
00021 * FORM is free software: you can redistribute it and/or modify it under the
00022 * terms of the GNU General Public License as published by the Free Software
00023 * Foundation, either version 3 of the License, or (at your option) any later
00024 * version.
00025 *
00026 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00029 * details.
00030 *
00031 * You should have received a copy of the GNU General Public License along
00032 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00033 */
00034 /* #] License : */
00035
00036 /*
00037 #[ Macro's :
00038 */
00039
00040 #define MaX(x,y) ((x) > (y) ? (x) : (y))
00041 #define MiN(x,y) ((x) < (y) ? (x) : (y))
00042 #define ABS(x) ( (x) < 0 ? -(x) : (x) )
00043 #define SGN(x) ( (x) > 0 ? 1 : (x) < 0 ? -1 : 0 )
00044 #define REDLENG(x) (((x)<0)?((x)+1):((x)-1))/2)
00045 #define INCLENG(x) (((x)<0)?((x)*2)-1:((x)*2)+1))
00046 #define GETCOEF(x,y) x += *x;y = x[-1];x -= ABS(y);y=REDLENG(y)
00047 #define GETSTOP(x,y) y=x+(*x)-1;y -= ABS(*y)-1
00048 #define StuffAdd(x,y) (((x)<0?-1:1)*(y)+((y)<0?-1:1)*(x))
00049
00050 #define EXCHN(t1,t2,n) { WORD a,i; for(i=0;i<n;i++){a=t1[i];t1[i]=t2[i];t2[i]=a;} }
00051 #define EXCH(x,y) { WORD a = (x); (x) = (y); (y) = a; }
00052
00053 #define TOKENTOline(x,y) if ( AC.OutputSpaces == NOSPACEFORMAT ) { \
00054 TokenToLine((UBYTE *) (y)); } else { TokenToLine((UBYTE *) (x)); }
00055
00056 #define UngetFromStream(stream,c) ((stream)->nextchar[(stream)->isnextchar++]=c)
00057 #ifdef WITHRETURN
00058 #define AddLineFeed(s,n) { (s)[(n)++] = CARRIAGERETURN; (s)[(n)++] = LINEFEED; }
00059 #else
00060 #define AddLineFeed(s,n) { (s)[(n)++] = LINEFEED; }
00061 #endif
00062 #define TryRecover(x) Terminate(-1)
00063 #define UngetChar(c) { pushbackchar = c; }
00064 #define ParseNumber(x,s) { (x)=0;while(*(s)>='0'&&*(s)<='9')(x)=10*(x)+*(s)++ -'0';}
00065 #define ParseSign(sgn,s) { (sgn)=0;while(*(s)=='-'||*(s)=='+'){ \
00066 if ( *(s)++ == '-' ) sgn ^= 1;}}
00067 #define ParseSignedNumber(x,s) { int sgn; ParseSign(sgn,s)\
00068 ParseNumber(x,s) if ( sgn ) x = -x; }
00069
00070 /* (n) is necessary here, since the macro is sometimes passed dereferenced pointers for n */
00071 #define NCOPY(s,t,n) { while ( (n)-- > 0 ) { *s++ = *t++; } }
00072 /*#define NCOPY(s,t,n) { memcpy(s,t,n*sizeof(WORD)); s+=n; t+=n; n = -1; }*/
00073 #define NCOPYI(s,t,n) { while ( (n)-- > 0 ) { *s++ = *t++; } }
00074 #define NCOPYB(s,t,n) { while ( (n)-- > 0 ) { *s++ = *t++; } }
00075 #define NCOPYI32(s,t,n) { while ( (n)-- > 0 ) { *s++ = *t++; } }
00076 #define WCOPY(s,t,n) { int nn=n; WORD *ss=(WORD *)s, *tt=(WORD *)t; while ( (nn)-- > 0 ) { *ss++ =
*tt++; } }
00077
00078 #define NeedNumber(x,s,err) { int sgn = 1;
00079 while ( *s == ' ' || *s == '\t' || *s == '-' || *s == '+' ) {
00080 if ( *s == '-' ) {sgn = -sgn;} s++; }
00081 if ( chartype[*s] != 1 ) goto err;
00082 ParseNumber(x,s)
00083 if ( sgn < 0 ) {(x) = -(x);} while ( *s == ' ' || *s == '\t' ) s++; \
00084 }
00085 #define SKIPBLANKS(s) { while ( *(s) == ' ' || *(s) == '\t' ) (s)++; }
00086 #define FLUSHCONSOLE if ( AP.InOutBuf > 0 ) CharOut(LINEFEED)

```

```

00087
00088 #define SKIPBRA1(s) { int lev1=0; s++; while(*s) { if(*s=='{') {lev1++;} \
00089 else {if(*s=='}')&&--lev1<0}{break;}} s++;} }
00090 #define SKIPBRA2(s) { int lev2=0; s++; while(*s) { if(*s=='{') {lev2++;} \
00091 else {if(*s=='}')&&--lev2<0}{break;}} \
00092 else {if(*s=='{') {SKIPBRA1(s)}} s++;} }
00093 #define SKIPBRA3(s) { int lev3=0; s++; while(*s) { if(*s=='{') {lev3++;} \
00094 else {if(*s=='}')&&--lev3<0}{break;}} \
00095 else {if(*s=='{') {SKIPBRA2(s)}} \
00096 else {if(*s=='{') {SKIPBRA1(s)}}} s++;} }
00097 #define SKIPBRA4(s) { int lev4=0; s++; while(*s) { if(*s=='{') {lev4++;} \
00098 else {if(*s=='}')&&--lev4<0}{break;}} \
00099 else {if(*s=='{') {SKIPBRA3(s)}} s++;} }
00100 #define SKIPBRA5(s) { int lev5=0; s++; while(*s) { if(*s=='{') {lev5++;} \
00101 else {if(*s=='}')&&--lev5<0}{break;}} \
00102 else {if(*s=='{') {SKIPBRA4(s)}} \
00103 else {if(*s=='{') {SKIPBRA1(s)}}} s++;} }
00104
00105 /*
00106 #define CYCLE1(a,i) {WORD iX,jX; iX=*a; for(jX=1;jX<i;jX++)a[jX-1]=a[jX]; a[i-1]=iX;}
00107 */
00108 #define CYCLE1(t,a,i) {t iX=*a; WORD jX; for(jX=1;jX<i;jX++)a[jX-1]=a[jX]; a[i-1]=iX;}
00109
00110 #define AddToCB(c,wx) if(c->Pointer>=c->Top) \
00111 {DoubleCbuffer(c-cbuf,c->Pointer,21);} \
00112 *(c->Pointer)++ = wx;
00113
00114 #define EXCHINOUT { FILEHANDLE *ffFi = AR.outfile; \
00115 AR.outfile = AR.infile; AR.infile = ffFi; }
00116 #define BACKINOUT { FILEHANDLE *ffFi = AR.outfile; POSITION posi; \
00117 AR.outfile = AR.infile; AR.infile = ffFi; \
00118 SetEndScratch(AR.infile,&posi); }
00119
00120 #define CopyArg(to,from) { if ( *from > 0 ) { int ica = *from; NCOPY(to,from,ica) } \
00121 else if ( *from <= -FUNCTION ) *to++ = *from++; \
00122 else { *to++ = *from++; *to++ = *from++; } }
00123
00124 #if ARGHEAD > 2
00125 #define FILLARG(w) { int i = ARGHEAD-2; while ( --i >= 0 ) *w++ = 0; }
00126 #define COPYARG(w,t) { int i = ARGHEAD-2; while ( --i >= 0 ) *w++ = *t++; }
00127 #define ZEROARG(w) { int i; for ( i = 2; i < ARGHEAD; i++ ) w[i] = 0; }
00128 #else
00129 #define FILLARG(w)
00130 #define COPYARG(w,t)
00131 #define ZEROARG(w)
00132 #endif
00133
00134 #if FUNHEAD > 2
00135 #define FILLFUN(w) { *w++ = 0; FILLFUN3(w) }
00136 #define COPYFUN(w,t) { *w++ = *t++; COPYFUN3(w,t) }
00137 #else
00138 #define FILLFUN(w)
00139 #define COPYFUN(w,t)
00140 #endif
00141
00142 #if FUNHEAD > 3
00143 #define FILLFUN3(w) { int ie = FUNHEAD-3; while ( --ie >= 0 ) *w++ = 0; }
00144 #define COPYFUN3(w,t) { int ie = FUNHEAD-3; while ( --ie >= 0 ) *w++ = *t++; }
00145 #else
00146 #define COPYFUN3(w,t)
00147 #define FILLFUN3(w)
00148 #endif
00149
00150 #if SUBEXPSIZE > 5
00151 #define FILLSUB(w) { int ie = SUBEXPSIZE-5; while ( --ie >= 0 ) *w++ = 0; }
00152 #define COPYSUB(w,ww) { int ie = SUBEXPSIZE-5; while ( --ie >= 0 ) *w++ = *ww++; }
00153 #else
00154 #define FILLSUB(w)
00155 #define COPYSUB(w,ww)
00156 #endif
00157
00158 #if EXPRHEAD > 4
00159 #define FILLEXPR(w) { int ie = EXPRHEAD-4; while ( --ie >= 0 ) *w++ = 0; }
00160 #else
00161 #define FILLEXPR(w)
00162 #endif
00163
00164 #define NEXTARG(x) if(*x>0) x += *x; else if(*x <= -FUNCTION)x++; else x += 2;
00165 #define COPY1ARG(sl,t1) { int ica; if ( (ica==t1) > 0 ) { NCOPY(sl,t1,ica) } \
00166 else if(*t1<=-FUNCTION){*sl++=*t1++;} else{*sl++=*t1++;*sl++=*t1++;} }
00167
00175 #define ZeroFillRange(w,begin,end) do { \
00176 int tmp_i; \
00177 for ( tmp_i = begin; tmp_i < end; tmp_i++ ) { (w)[tmp_i] = 0; } \
00178 } while (0)
00179
00180 #define TABLESIZE(a,b) (((WORD)sizeof(a))/((WORD)sizeof(b)))

```

```

00181 #define WORDDIFF(x,y) (WORD)(x-y)
00182 #define wsizeof(a) ((WORD)sizeof(a))
00183 #define VARNAME(type,num) (AC.varnames->namebuffer+type[num].name)
00184 #define DOLLARNAME(type,num) (AC.dollarnames->namebuffer+type[num].name)
00185 #define EXPRNAME(num) (AC.exprnames->namebuffer+Expressions[num].name)
00186
00187 #define PREV(x) prevorder?prevorder:x
00188
00189 #define Terminate(x) do { TerminateImpl(x, __FILE__, __LINE__, __FUNCTION__); } while(0)
00190 #define SETERROR(x) { Terminate(-1); return(-1); }
00191
00192 /* use this macro to avoid the unused parameter warning */
00193 #define DUMMYUSE(x) (void)(x);
00194
00195 #ifdef _FILE_OFFSET_BITS
00196 #if _FILE_OFFSET_BITS==64
00197 /*:[19mar2004 mt]*/
00198
00199 #define ADDPOS(pp,x) (pp).p1 = ((pp).p1+(off_t)(x))
00200 #define SETBASELENGTH(ss,x) (ss).p1 = (off_t)(x)
00201 #define SETBASEPOSITION(pp,x) (pp).p1 = (off_t)(x)
00202 #define ISEQUALPOSINC(pp1,pp2,x) ( (pp1).p1 == ((pp2).p1+(off_t)(x)) )
00203 #define ISGEPOSINC(pp1,pp2,x) ( (pp1).p1 >= ((pp2).p1+(off_t)(x)) )
00204 #define DIVPOS(pp,n) ( (pp).p1/(off_t)(n) )
00205 #define MULPOS(pp,n) (pp).p1 *= (off_t)(n)
00206
00207 #else
00208
00209 #define ADDPOS(pp,x) (pp).p1 = ((pp).p1+(x))
00210 #define SETBASELENGTH(ss,x) (ss).p1 = (x)
00211 #define SETBASEPOSITION(pp,x) (pp).p1 = (x)
00212 #define ISEQUALPOSINC(pp1,pp2,x) ( (pp1).p1 == ((pp2).p1+(LONG)(x)) )
00213 #define ISGEPOSINC(pp1,pp2,x) ( (pp1).p1 >= ((pp2).p1+(LONG)(x)) )
00214 #define DIVPOS(pp,n) ( (pp).p1/(n) )
00215 #define MULPOS(pp,n) (pp).p1 *= (n)
00216 #endif
00217 #else
00218
00219 #define ADDPOS(pp,x) (pp).p1 = ((pp).p1+(LONG)(x))
00220 #define SETBASELENGTH(ss,x) (ss).p1 = (LONG)(x)
00221 #define SETBASEPOSITION(pp,x) (pp).p1 = (LONG)(x)
00222 #define ISEQUALPOSINC(pp1,pp2,x) ( (pp1).p1 == ((pp2).p1+(LONG)(x)) )
00223 #define ISGEPOSINC(pp1,pp2,x) ( (pp1).p1 >= ((pp2).p1+(LONG)(x)) )
00224 #define DIVPOS(pp,n) ( (pp).p1/(LONG)(n) )
00225 #define MULPOS(pp,n) (pp).p1 *= (LONG)(n)
00226
00227 #endif
00228 #define DIFPOS(ss,pp1,pp2) (ss).p1 = ((pp1).p1-(pp2).p1)
00229 #define DIFBASE(pp1,pp2) ((pp1).p1-(pp2).p1)
00230 #define ADD2POS(pp1,pp2) (pp1).p1 += (pp2).p1
00231 #define PUTZERO(pp) (pp).p1 = 0
00232 #define BASEPOSITION(pp) ((pp).p1)
00233 #define SETSTARTPOS(pp) (pp).p1 = -2
00234 #define NOTSTARTPOS(pp) ( (pp).p1 > -2 )
00235 #define ISMINPOS(pp) ( (pp).p1 == -1 )
00236 #define ISEQUALPOS(pp1,pp2) ( (pp1).p1 == (pp2).p1 )
00237 #define ISNOTEQUALPOS(pp1,pp2) ( (pp1).p1 != (pp2).p1 )
00238 #define ISLESSPOS(pp1,pp2) ( (pp1).p1 < (pp2).p1 )
00239 #define ISGEPOS(pp1,pp2) ( (pp1).p1 >= (pp2).p1 )
00240 #define ISNOTZEROPOS(pp) ( (pp).p1 != 0 )
00241 #define ISZEROPOS(pp) ( (pp).p1 == 0 )
00242 #define ISPOSPOS(pp) ( (pp).p1 > 0 )
00243 #define ISNEGPOS(pp) ( (pp).p1 < 0 )
00244 extern void TELLFILE(int,POSITION *);
00245
00246 #define TOLONG(x) ((LONG)(x))
00247
00248 #define Add2Com(x) { WORD cod[2]; cod[0] = x; cod[1] = 2; AddNtoL(2,cod); }
00249 #define Add3Com(x1,x2) { WORD cod[3]; cod[0] = x1; cod[1] = 3; cod[2] = x2; AddNtoL(3,cod); }
00250 #define Add4Com(x1,x2,x3) { WORD cod[4]; cod[0] = x1; cod[1] = 4; \
00251     cod[2] = x2; cod[3] = x3; AddNtoL(4,cod); }
00252 #define Add5Com(x1,x2,x3,x4) { WORD cod[5]; cod[0] = x1; cod[1] = 5; \
00253     cod[2] = x2; cod[3] = x3; cod[4] = x4; AddNtoL(5,cod); }
00254
00255 /*
00256     The temporary variable ppp is to avoid a compiler warning about strict aliasing
00257 */
00258 #define WantAddPointers(x) while((AT.pWorkPointer+(x))>AR.pWorkSize){WORD ***ppp=&AT.pWorkSpace;\
00259     ExpandBuffer((void **)ppp,&AR.pWorkSize,(int)(sizeof(WORD *)));}
00260 #define WantAddLongs(x) while((AT.lWorkPointer+(x))>AR.lWorkSize){LONG **ppp=&AT.lWorkSpace;\
00261     ExpandBuffer((void **)ppp,&AR.lWorkSize,sizeof(LONG));}
00262 #define WantAddPositions(x) while((AT.posWorkPointer+(x))>AR.posWorkSize){POSITION
00263     **ppp=&AT.posWorkSpace;\
00264     ExpandBuffer((void **)ppp,&AR.posWorkSize,sizeof(POSITION));}
00265
00266 /* inline in form3.h (or config.h). */
00267 #define FORM_INLINE inline

```

```

00267
00268 /*
00269     Macro's for memory management. This can be done by routines, but that
00270     would be slower. Inline routines could do this, but we don't want to
00271     leave this to the friendliness of the compiler(s).
00272     The routines can be found in the file tools.c
00273 */
00274 #define MEMORYMACROS
00275
00276 #ifndef MEMORYMACROS
00277
00278 #define TermMalloc(x) ( (AT.TermMemTop <= 0) ? TermMallocAddMemory(BHEAD0),
AT.TermMemHeap[--AT.TermMemTop]: AT.TermMemHeap[--AT.TermMemTop] )
00279 #define NumberMalloc(x) ( (AT.NumberMemTop <= 0) ? NumberMallocAddMemory(BHEAD0),
AT.NumberMemHeap[--AT.NumberMemTop]: AT.NumberMemHeap[--AT.NumberMemTop] )
00280 #define CacheNumberMalloc(x) ( (AT.CacheNumberMemTop <= 0) ? CacheNumberMallocAddMemory(BHEAD0),
AT.CacheNumberMemHeap[--AT.CacheNumberMemTop]: AT.CacheNumberMemHeap[--AT.CacheNumberMemTop] )
00281 #define TermFree(TermMem,x) AT.TermMemHeap[AT.TermMemTop++] = (WORD *) (TermMem)
00282 #define NumberFree(NumberMem,x) AT.NumberMemHeap[AT.NumberMemTop++] = (UWORD *) (NumberMem)
00283 #define CacheNumberFree(NumberMem,x) AT.CacheNumberMemHeap[AT.CacheNumberMemTop++] = (UWORD
*) (NumberMem)
00284
00285 #else
00286
00287 #define TermMalloc(x) TermMalloc2(BHEAD (char *) (x))
00288 #define NumberMalloc(x) NumberMalloc2(BHEAD (char *) (x))
00289 #define CacheNumberMalloc(x) CacheNumberMalloc2(BHEAD (char *) (x))
00290 #define TermFree(x,y) TermFree2(BHEAD (WORD *) (x), (char *) (y))
00291 #define NumberFree(x,y) NumberFree2(BHEAD (UWORD *) (x), (char *) (y))
00292 #define CacheNumberFree(x,y) CacheNumberFree2(BHEAD (UWORD *) (x), (char *) (y))
00293
00294 #endif
00295
00296 /*
00297 * Macros for checking nesting levels in the compiler, used as follows:
00298 *
00299 *   AC.IfSumCheck[AC.IfLevel] = NestingChecksum();
00300 *   AC.IfLevel++;
00301 *
00302 *   AC.IfLevel--;
00303 *   if ( AC.IfSumCheck[AC.IfLevel] != NestingChecksum() ) {
00304 *       MesNesting();
00305 *   }
00306 *
00307 * Note that NestingChecksum() also contains AC.IfLevel and so in this case
00308 * using increment/decrement operators on it in the left-hand side may be
00309 * confusing.
00310 */
00311 #define NestingChecksum() (AC.IfLevel + AC.RepLevel + AC.arglevel + AC.insidelevel + AC.termlevel +
AC.inexprlevel + AC.dolooplevel + AC.SwitchLevel)
00312 #define MesNesting() MesPrint("&Illegal nesting of if, repeat, argument, inside, term, inexpression
and do")
00313
00314 #define MarkPolyRatFunDirty(T) {if(*T&&AR.PolyFunType==2){WORD *TP,*TT;TT=T+*T;TT-=ABS(TT[-1]);\
TP=T+1;while(TP<TT){if(*TP==AR.PolyFun){TP[2]|=(DIRTYFLAG|MUSTCLEANPRF);}TP+=TP[1];}}\
00315
00316
00317 /*
00318     Macros for nesting input levels for #name = ...; assign instructions.
00319     Note that the level should never go below zero.
00320 */
00321 #define PUSHPREASSIGNLEVEL AP.PreAssignLevel++; { GETIDENTITY \
00322     if ( AP.PreAssignLevel >= AP.MaxPreAssignLevel ) { int i; \
00323         LONG *ap = (LONG *) Malloc1(2*AP.MaxPreAssignLevel*sizeof(LONG *),"PreAssignStack"); \
00324         for ( i = 0; i < AP.MaxPreAssignLevel; i++ ) ap[i] = AP.PreAssignStack[i]; \
00325         M_free(AP.PreAssignStack,"PreAssignStack"); \
00326         AP.MaxPreAssignLevel *= 2; AP.PreAssignStack = ap; \
00327     } \
00328     *AT.WorkPointer++ = AP.PreContinuation; AP.PreContinuation = 0; \
00329     AP.PreAssignStack[AP.PreAssignLevel] = AC.iPointer - AC.iBuffer; }
00330
00331 #define POPPREASSIGNLEVEL if ( AP.PreAssignLevel > 0 ) { GETIDENTITY \
00332     AC.iPointer = AC.iBuffer + AP.PreAssignStack[AP.PreAssignLevel--]; \
00333     AP.PreContinuation = *--AT.WorkPointer; \
00334     *AC.iPointer = 0; }
00335
00336 #ifndef WITHFLOAT
00337 /*
00338     The following macro's are needed to avoid problems with the compilers
00339     and gmp.h. For the C++ files we need the Form include files to be inside
00340     and extern C {} environment, but then the structs.h needs gmp.h to
00341     recognise the mpf_t datatype. This causes no end of problems.
00342     Hence we collect the sensitive objects as (void *) and cast them to
00343     something usable with the macro's below. This way the gmp.h file
00344     needs to be included only in a very limited number of .c files.
00345 */
00346 #define mpftab1 ((mpf_t *) (AT.mpf_tab1))
00347 #define mpftab2 ((mpf_t *) (AT.mpf_tab2))

```



```

00348 #define mpfaux_ ((mpf_t *) (AT.aux_))
00349 #define aux1 ((mpf_t *) (AT.aux_)) [0]
00350 #define aux2 ((mpf_t *) (AT.aux_)) [1]
00351 #define aux3 ((mpf_t *) (AT.aux_)) [2]
00352 #define aux4 ((mpf_t *) (AT.aux_)) [3]
00353 #define aux5 ((mpf_t *) (AT.aux_)) [4]
00354 #define auxjm ((mpf_t *) (AT.aux_)) [5]
00355 #define auxjjm ((mpf_t *) (AT.aux_)) [6]
00356 #define auxsum ((mpf_t *) (AT.aux_)) [7]
00357
00358 #define mpfdelta1 ((mpf_t *) (AS.delta_1)) [0]
00359
00360 #endif
00361
00362
00363 /*
00364     MesPrint("P-level popped to %d with %d", AP.PreAssignLevel, (WORD) (AC.iPointer - AC.iBuffer));
00365
00366     #] Macro's :
00367     #[ Inline functions :
00368 */
00369
00370 /*
00371 * The following three functions give the unsigned absolute value of a signed
00372 * integer even for the most negative integer. This is beyond the scope of
00373 * the standard abs() function and its family, whose return-values are signed.
00374 * In short, we should not use the unary minus operator with signed numbers
00375 * unless we are sure that there are no integer overflows. Instead, we rely on
00376 * two well-defined operations: (i) signed-to-unsigned conversion and
00377 * (ii) unary minus of unsigned operands.
00378 *
00379 * See also:
00380 *   https://stackoverflow.com/a/4536188 (Unary minus and signed-to-unsigned conversion)
00381 *   https://stackoverflow.com/q/8026694 (C: unary minus operator behavior with unsigned operands)
00382 *   https://stackoverflow.com/q/1610947 (Why does stdlib.h's abs() family of functions return a
signed value?)
00383 *   https://blog.regehr.org/archives/226 (A Guide to Undefined Behavior in C and C++, Part 2)
00384 */
00385 static inline unsigned int IntAbs(int x)
00386 {
00387     if ( x >= 0 ) return (unsigned int)x;
00388     return(-(unsigned int)x);
00389 }
00390
00391 static inline UWORD WordAbs(WORD x)
00392 {
00393     if ( x >= 0 ) return (UWORD)x;
00394     return(-(UWORD)x);
00395 }
00396
00397 static inline ULONG LongAbs(LONG x)
00398 {
00399     if ( x >= 0 ) return (ULONG)x;
00400     return(-(ULONG)x);
00401 }
00402
00403 /*
00404 * The following functions provide portable unsigned-to-signed conversions
00405 * (to avoid the implementation-defined behaviour), which is expected to be
00406 * optimized to a no-op.
00407 *
00408 * See also:
00409 *   https://stackoverflow.com/a/13208789 (Efficient unsigned-to-signed cast avoiding
implementation-defined behavior)
00410 */
00411 static inline int UnsignedToInt(unsigned int x)
00412 {
00413     extern void TerminateImpl(int, const char*, int, const char*) NORETURN;
00414     if ( x <= INT_MAX ) return (int)x;
00415     if ( x >= (unsigned int)INT_MIN )
00416         return((int)(x - (unsigned int)INT_MIN) + INT_MIN);
00417     Terminate(1);
00418     return(0);
00419 }
00420
00421 static inline WORD UWordToWord(UWORD x)
00422 {
00423     extern void TerminateImpl(int, const char*, int, const char*) NORETURN;
00424     if ( x <= WORD_MAX_VALUE ) return (WORD)x;
00425     if ( x >= (UWORD)WORD_MIN_VALUE )
00426         return((WORD)(x - (UWORD)WORD_MIN_VALUE) + WORD_MIN_VALUE);
00427     Terminate(1);
00428     return(0);
00429 }
00430
00431 static inline LONG ULongToLong(ULONG x)
00432 {

```

```

00433     extern void TerminateImpl(int, const char*, int, const char*) NORETURN;
00434     if ( x <= LONG_MAX_VALUE ) return((LONG)x);
00435     if ( x >= (ULONG)LONG_MIN_VALUE )
00436         return((LONG)(x - (ULONG)LONG_MIN_VALUE) + LONG_MIN_VALUE);
00437     Terminate(1);
00438     return(0);
00439 }
00440
00441 /*
00442     #] Inline functions :
00443     #[ Thread objects :
00444 */
00445
00452 #ifdef WITHPTHREADS
00453
00454 #define EXTERNLOCK(x) extern pthread_mutex_t x;
00455 #define INILOCK(x) pthread_mutex_t x = PTHREAD_MUTEX_INITIALIZER;
00456 #define EXTERNRWLOCK(x) extern pthread_rwlock_t x;
00457 #define INIRWLOCK(x) pthread_rwlock_t x = PTHREAD_RWLOCK_INITIALIZER;
00458 #ifdef DEBUGGINGLOCKS
00459 #include <asm/errno.h>
00460 #define LOCK(x) while ( pthread_mutex_trylock(&(x)) == EBUSY ) {}
00461 #define RWLOCKR(x) while ( pthread_rwlock_tryrdlock(&(x)) == EBUSY ) {}
00462 #define RWLOCKW(x) while ( pthread_rwlock_trywrlock(&(x)) == EBUSY ) {}
00463 #else
00464 #define LOCK(x) pthread_mutex_lock(&(x))
00465 #define RWLOCKR(x) pthread_rwlock_rdlock(&(x))
00466 #define RWLOCKW(x) pthread_rwlock_wrlock(&(x))
00467 #endif
00468 #define UNLOCK(x) pthread_mutex_unlock(&(x))
00469 #define UNRWLOCK(x) pthread_rwlock_unlock(&(x))
00470 #define MLOCK(x) LOCK(x)
00471 #define MUNLOCK(x) UNLOCK(x)
00472
00473 #define GETBIDENTITY
00474 #define GETIDENTITY int identity = WhoAmI(); ALLPRIVATES *B = AB[identity];
00475 #else
00476
00477 #define EXTERNLOCK(x)
00478 #define INILOCK(x)
00479 #define LOCK(x)
00480 #define UNLOCK(x)
00481 #define EXTERNRWLOCK(x)
00482 #define INIRWLOCK(x)
00483 #define RWLOCKR(x)
00484 #define RWLOCKW(x)
00485 #define UNRWLOCK(x)
00486 #ifdef WITHMPI
00487 #define MLOCK(x) do { if ( PF.me != MASTER ) PF_MLock(); } while (0)
00488 #define MUNLOCK(x) do { if ( PF.me != MASTER ) PF_MUnlock(); } while (0)
00489 #else
00490 #define MLOCK(x)
00491 #define MUNLOCK(x)
00492 #endif
00493 #define GETIDENTITY
00494 #define GETBIDENTITY
00495
00496 #endif
00497
00498 /*
00499     #] Thread objects :
00500     #[ Declarations :
00501 */
00502
00503 #ifdef TERMMALLOCDEBUG
00504 extern WORD **DebugHeap1, **DebugHeap2;
00505 #endif
00506
00511 extern void StartVariables(void);
00512 extern void setSignalHandlers(void);
00513 extern UBYTE *CodeToLine(WORD, UBYTE *);
00514 extern UBYTE *AddArrayIndex(WORD, UBYTE *);
00515 extern INDEXENTRY *FindInIndex(WORD, FILEDATA *, WORD, WORD);
00516 extern INDEXENTRY *NextFileIndex(POSITION *);
00517 extern WORD *PasteTerm(PHEAD WORD, WORD *, WORD *, WORD, WORD);
00518 extern UBYTE *StrCopy(UBYTE *, UBYTE *);
00519 extern UBYTE *WrtPower(UBYTE *, WORD);
00520 extern int AccumGCD(PHEAD UWORD *, WORD *, UWORD *, WORD);
00521 extern void AddArgs(PHEAD WORD *, WORD *, WORD *);
00522 extern int AddCoef(PHEAD WORD **, WORD **);
00523 extern int AddLong(UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *);
00524 extern int AddPLon(UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *);
00525 extern int AddPoly(PHEAD WORD **, WORD **);
00526 extern int AddRat(PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *);
00527 extern void AddToLine(UBYTE *);
00528 extern int AddWild(PHEAD WORD, WORD, WORD);
00529 extern int BigLong(UWORD *, WORD, UWORD *, WORD);

```

```

00530 extern int      BinomGen(PHEAD WORD *,WORD,WORD **,WORD,WORD,WORD,WORD,WORD,UWORD *,WORD);
00531 extern int      CheckWild(PHEAD WORD,WORD,WORD,WORD *);
00532 extern int      Chisholm(PHEAD WORD *,WORD);
00533 extern int      CleanExpr(WORD);
00534 extern void     CleanUp(WORD);
00535 extern void     ClearWild(PHEAD0);
00536 extern int      CompareFunctions(WORD *,WORD *);
00537 extern int      Commute(WORD *,WORD *);
00538 extern WORD     DetCommu(WORD *);
00539 extern WORD     DoesCommu(WORD *);
00540 extern int      CompArg(WORD *,WORD *);
00541 extern WORD     CompCoef(WORD *, WORD *);
00542 extern int      CompGroup(PHEAD WORD,WORD **,WORD *,WORD *,WORD);
00543 extern WORD     Compare1(PHEAD WORD *,WORD *,WORD);
00544 extern WORD     CountDo(WORD *,WORD *);
00545 extern WORD     CountFun(WORD *,WORD *);
00546 extern WORD     DimensionSubterm(WORD *);
00547 extern WORD     DimensionTerm(WORD *);
00548 extern WORD     DimensionExpression(PHEAD WORD *);
00549 extern int      Deferred(PHEAD WORD *,WORD);
00550 extern int      DeleteStore(WORD);
00551 extern WORD     DetCurDum(PHEAD WORD *);
00552 extern void     DetVars(WORD *,WORD);
00553 extern int      Distribute(DISTRIBUTE *,WORD);
00554 extern int      DivLong(UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *,UWORD *,WORD *);
00555 extern int      DivRat(PHEAD UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
00556 extern int      Divvy(PHEAD UWORD *,WORD *,UWORD *,WORD);
00557 extern int      DoDelta(WORD *);
00558 extern int      DoDelta3(PHEAD WORD *,WORD);
00559 extern int      TestPartitions(WORD *, PARTI *);
00560 extern int      DoPartitions(PHEAD WORD *,WORD);
00561 extern int      CoCanonicalize(UBYTE *);
00562 extern int      DoCanonicalize(PHEAD WORD *, WORD *);
00563 extern int      GenDiagrams(PHEAD WORD *,WORD);
00564 extern int      DoTopologyCanonicalize(PHEAD WORD *,WORD,WORD,WORD *);
00565 extern int      DoShattering(PHEAD WORD *,WORD *,WORD *,WORD *);
00566 extern int      DoTableExpansion(WORD *,WORD);
00567 extern int      DoDistrib(PHEAD WORD *,WORD);
00568 extern int      DoShuffle(WORD *,WORD,WORD,WORD);
00569 extern int      DoPermutations(PHEAD WORD *,WORD);
00570 extern int      Shuffle(WORD *, WORD *, WORD *);
00571 extern int      FinishShuffle(WORD *);
00572 extern int      DoStuffle(WORD *,WORD,WORD,WORD);
00573 extern int      Stuffle(WORD *, WORD *, WORD *);
00574 extern int      FinishStuffle(WORD *);
00575 extern WORD     *StuffRootAdd(WORD *, WORD *, WORD *);
00576 extern int      TestUse(WORD *,WORD);
00577 extern DBASE    *FindTB(UBYTE *);
00578 extern int      CheckTableDeclarations(DBASE *);
00579 extern void     Apply(WORD *,WORD);
00580 extern int      ApplyExec(WORD *,int,WORD);
00581 extern void     ApplyReset(WORD);
00582 extern void     TableReset(void);
00583 extern void     ReWorkT(WORD *,WORD *,WORD);
00584 extern WORD     GetIfDollarNum(WORD *, WORD *);
00585 extern int      FindVar(WORD *,WORD *);
00586 extern int      DoIfStatement(PHEAD WORD *,WORD *);
00587 extern int      DoOnePow(PHEAD WORD *,WORD,WORD,WORD *,WORD *,WORD,WORD *);
00588 extern void     DoRevert(WORD *,WORD *);
00589 extern int      DoSumF1(PHEAD WORD *,WORD *,WORD,WORD);
00590 extern int      DoSumF2(PHEAD WORD *,WORD *,WORD,WORD);
00591 extern int      DoTheta(PHEAD WORD *);
00592 extern LONG     EndSort(PHEAD WORD *,int);
00593 extern int      EntVar(WORD,UBYTE *,WORD,WORD,WORD,WORD);
00594 extern int      EpfCon(PHEAD WORD *,WORD *,WORD,WORD);
00595 extern int      EpfFind(PHEAD WORD *,WORD *);
00596 extern WORD     EpfGen(WORD,WORD *,WORD *,WORD *,WORD);
00597 extern int      EqualArg(WORD *,WORD,WORD);
00598 extern int      Factorial(PHEAD WORD,UWORD *,WORD *);
00599 extern int      Bernoulli(WORD,UWORD *,WORD *);
00600 extern int      FactorIn(PHEAD WORD *,WORD);
00601 extern int      FactorInExpr(PHEAD WORD *,WORD);
00602 extern int      FindAll(PHEAD WORD *,WORD *,WORD,WORD *);
00603 extern WORD     FindMulti(PHEAD WORD *,WORD *);
00604 extern int      FindOnce(PHEAD WORD *,WORD *);
00605 extern int      FindOnly(PHEAD WORD *,WORD *);
00606 extern int      FindRest(PHEAD WORD *,WORD *);
00607 extern void     FindSpecial(WORD *);
00608 extern WORD     FindrNumber(WORD,VARRENUM *);
00609 extern void     FiniLine(void);
00610 extern int      FiniTerm(PHEAD WORD *,WORD *,WORD *,WORD,WORD);
00611 extern int      FlushOut(POSITION *,FILEHANDLE *,int);
00612 extern void     FunLevel(PHEAD WORD *);
00613 extern void     AdjustRenumScratch(PHEAD0);
00614 extern void     GarbHand(void);
00615 extern int      GcdLong(PHEAD UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
00616 extern int      LcmLong(PHEAD UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);

```

```

00617 extern void GCD(UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
00618 extern ULONG GCD2(ULONG,ULONG);
00619 extern int Generator(PHEAD WORD *,WORD);
00620 extern int GetBinom(UWORD *,WORD *,WORD,WORD);
00621 extern WORD GetFromStore(WORD *,POSITION *,RENUMBER,WORD *,WORD);
00622 extern int GetLong(UBYTE *,UWORD *,WORD *);
00623 extern WORD GetMoreTerms(WORD *);
00624 extern int GetMoreFromMem(WORD *,WORD **);
00625 extern WORD GetOneTerm(PHEAD WORD *,FILEHANDLE *,POSITION *,int);
00626 extern RENUMBER GetTable(WORD,POSITION *,WORD);
00627 extern WORD GetTerm(PHEAD WORD *);
00628 extern int Glue(PHEAD WORD *,WORD *,WORD *,WORD);
00629 extern int InFunction(PHEAD WORD *,WORD *);
00630 extern void IniLine(WORD);
00631 extern void IniVars(void);
00632 extern int InsertTerm(PHEAD WORD *,WORD,WORD,WORD *,WORD *,WORD);
00633 extern void LongToLine(UWORD *,WORD);
00634 extern int MakeDirty(WORD *,WORD *,WORD);
00635 extern void MarkDirty(WORD *,WORD);
00636 extern void PolyFunDirty(PHEAD WORD *);
00637 extern void PolyFunClean(PHEAD WORD *);
00638 extern int MakeModTable(void);
00639 extern int MatchE(PHEAD WORD *,WORD *,WORD *,WORD);
00640 extern int MatchCy(PHEAD WORD *,WORD *,WORD *,WORD);
00641 extern int FunMatchCy(PHEAD WORD *,WORD *,WORD *,WORD);
00642 extern int FunMatchSy(PHEAD WORD *,WORD *,WORD *,WORD);
00643 extern int MatchArgument(PHEAD WORD *,WORD *);
00644 extern int MatchFunction(PHEAD WORD *,WORD *,WORD *);
00645 extern int MergePatches(WORD);
00646 extern int MesCerr(char *, UBYTE *);
00647 extern int MesComp(char *, UBYTE *, UBYTE *);
00648 extern int Modulus(WORD *);
00649 extern void MoveDummies(PHEAD WORD *,WORD);
00650 extern int MulLong(UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
00651 extern int MulRat(PHEAD UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
00652 extern int Mully(PHEAD UWORD *,WORD *,UWORD *,WORD);
00653 extern int MultDo(PHEAD WORD *,WORD *);
00654 extern int NewSort(PHEAD0);
00655 extern int ExtraSymbol(WORD,WORD,WORD,WORD *,WORD *);
00656 extern int Normalize(PHEAD WORD *);
00657 extern int BracketNormalize(PHEAD WORD *);
00658 extern void DropCoefficient(PHEAD WORD *);
00659 extern void DropSymbols(PHEAD WORD *);
00660 extern int PutInside(PHEAD WORD *, WORD *);
00661 extern void OpenTemp(void);
00662 extern void Pack(UWORD *,WORD *,UWORD *,WORD );
00663 extern LONG PasteFile(PHEAD WORD,WORD *,POSITION *,WORD **,RENUMBER,WORD *,WORD);
00664 extern int Permute(PERM *,WORD);
00665 extern int PermuteP(PERMP *,WORD);
00666 extern int PolyFunMul(PHEAD WORD *);
00667 extern int PopVariables(void);
00668 extern int PrepPoly(PHEAD WORD *,WORD);
00669 extern int Processor(void);
00670 extern int Product(UWORD *,WORD *,WORD);
00671 extern void PrtLong(UWORD *,WORD,UBYTE *);
00672 extern void PrtTerms(void);
00673 extern void PrintDeprecation(const char *,const char *);
00674 extern void PrintRunningTime(void);
00675 extern LONG GetRunningTime(void);
00676 extern int PutBracket(PHEAD WORD *);
00677 extern LONG PutIn(FILEHANDLE *,POSITION *,WORD *,WORD **,int);
00678 extern int PutInStore(INDEXENTRY *,WORD);
00679 extern WORD PutOut(PHEAD WORD *,POSITION *,FILEHANDLE *,WORD);
00680 extern UWORD Quotient(UWORD *,WORD *,WORD);
00681 extern int RaisPow(PHEAD UWORD *,WORD *,UWORD);
00682 extern void RaisPowCached (PHEAD WORD, WORD, UWORD **, WORD *);
00683 extern WORD RaisPowMod (WORD, WORD, WORD);
00684 extern int NormalModulus(UWORD *,WORD *);
00685 extern int MakeInverses(void);
00686 extern int GetModInverses(WORD,WORD,WORD *,WORD *);
00687 extern int GetLongModInverses(PHEAD UWORD *, WORD, UWORD *, WORD, UWORD *, WORD *, UWORD *, WORD
*);
00688 extern void RatToLine(UWORD *,WORD);
00689 extern int RatioFind(PHEAD WORD *,WORD *);
00690 extern int RatioGen(PHEAD WORD *,WORD *,WORD,WORD);
00691 extern WORD ReNumber(PHEAD WORD *);
00692 extern WORD ReadSnum(UBYTE **);
00693 extern WORD Remain10(UWORD *,WORD *);
00694 extern WORD Remain4(UWORD *,WORD *);
00695 extern int ResetScratch(void);
00696 extern int ResolveSet(PHEAD WORD *,WORD *,WORD *);
00697 extern int RevertScratch(void);
00698 extern int ScanFunctions(PHEAD WORD *,WORD *,WORD);
00699 extern void SeekScratch(FILEHANDLE *,POSITION *);
00700 extern void SetEndScratch(FILEHANDLE *,POSITION *);
00701 extern void SetEndHScratch(FILEHANDLE *,POSITION *);
00702 extern int SetFileIndex(void);

```

```

00703 extern int      Sflush(FILEHANDLE *);
00704 extern int      Simplify(PHEAD UWORD *,WORD *,UWORD *,WORD *);
00705 extern int      SortWild(WORD *,WORD);
00706 extern FILE *LocateBase(char **,char **,char *);
00707 extern LONG      SplitMerge(PHEAD WORD **,LONG);
00708 extern int      StoreTerm(PHEAD WORD *);
00709 extern void      SubPLon(UWORD *,WORD,UWORD *,WORD,UWORD *,WORD *);
00710 extern void      Substitute(PHEAD WORD *,WORD *,WORD);
00711 extern int      SymFind(PHEAD WORD *,WORD *);
00712 extern int      SymGen(PHEAD WORD *,WORD *,WORD,WORD);
00713 extern WORD      Symmetrize(PHEAD WORD *,WORD *,WORD,WORD,WORD);
00714 extern int      FullSymmetrize(PHEAD WORD *,int);
00715 extern int      TakeModulus(UWORD *,WORD *,UWORD *,WORD,WORD);
00716 extern int      TakeNormalModulus(UWORD *,WORD *,UWORD *,WORD,WORD);
00717 extern void      TalToLine(UWORD);
00718 extern int      TenVec(PHEAD WORD *,WORD *,WORD,WORD);
00719 extern int      TenVecFind(PHEAD WORD *,WORD *);
00720 extern int      TermRenumbr(WORD *,RENUMBER,WORD);
00721 extern void      TestDrop(void);
00722 extern void      PutInVflags(WORD);
00723 extern int      TestMatch(PHEAD WORD *,WORD *);
00724 extern WORD      TestSub(PHEAD WORD *,WORD);
00725 extern LONG      TimeCPU(WORD);
00726 extern LONG      TimeChildren(WORD);
00727 extern LONG      TimeWallClock(WORD);
00728 extern LONG      Timer(int);
00729 extern int      GetTimerInfo(LONG **,LONG **);
00730 extern void      WriteTimerInfo(LONG *,LONG *);
00731 extern LONG      GetWorkerTimes(void);
00732 extern int      ToStorage(EXPRESSIONS,POSITION *);
00733 extern void      TokenToLine(UBYTE *);
00734 extern int      Trace4(PHEAD WORD *,WORD *,WORD,WORD);
00735 extern int      Trace4Gen(PHEAD TRACES *,WORD);
00736 extern int      Trace4no(WORD,WORD *,TRACES *);
00737 extern int      TraceFind(PHEAD WORD *,WORD *);
00738 extern int      TraceN(PHEAD WORD *,WORD *,WORD,WORD);
00739 extern int      TraceNgen(PHEAD TRACES *,WORD);
00740 extern WORD      TraceNno(WORD,WORD *,TRACES *);
00741 extern int      Traces(PHEAD WORD *,WORD *,WORD,WORD);
00742 extern WORD      Trick(WORD *,TRACES *);
00743 extern int      TryDo(PHEAD WORD *,WORD *,WORD);
00744 extern void      UnPack(UWORD *,WORD,WORD *,WORD *);
00745 extern int      VarStore(UBYTE *,WORD,WORD,WORD);
00746 extern WORD      WildFill(PHEAD WORD *,WORD *,WORD *);
00747 extern int      WriteAll(void);
00748 extern int      WriteOne(UBYTE *,int,int,WORD);
00749 extern void      WriteArgument(WORD *);
00750 extern int      WriteExpression(WORD *,LONG);
00751 extern int      WriteInnerTerm(WORD *,WORD);
00752 extern void      WriteLists(void);
00753 extern void      WriteSetup(void);
00754 extern void      WriteStats(POSITION *,WORD,WORD);
00755 extern int      WriteSubTerm(WORD *,WORD);
00756 extern int      WriteTerm(WORD *,WORD *,WORD,WORD,WORD);
00757 extern int      execarg(PHEAD WORD *,WORD);
00758 extern int      execterm(PHEAD WORD *,WORD);
00759 extern void      SpecialCleanup(PHEAD0);
00760 extern void      SetMods(void);
00761 extern void      UnSetMods(void);
00762
00763 /*-----*/
00764
00765 extern int      DoExecute(WORD,WORD);
00766 extern void      SetScratch(FILEHANDLE *,POSITION *);
00767 extern void      Warning(char *);
00768 extern void      HighWarning(char *);
00769 extern int      SpareTable(TABLES);
00770
00771 extern UBYTE *strDup1(UBYTE *,char *);
00772 extern void *Malloc1(LONG,const char *);
00773 extern int      DoTail(int,UBYTE **);
00774 extern int      OpenInput(void);
00775 extern int      PutPreVar(UBYTE *,UBYTE *,UBYTE *,int);
00776 extern void      Error0(char *);
00777 extern void      Error1(char *,UBYTE *);
00778 extern void      Error2(char *,char *,UBYTE *);
00779 extern UBYTE ReadFromStream(STREAM *);
00780 extern UBYTE GetFromStream(STREAM *);
00781 extern UBYTE LookInStream(STREAM *);
00782 extern STREAM *OpenStream(UBYTE *,int,int,int);
00783 extern int      LocateFile(UBYTE **,int);
00784 extern STREAM *CloseStream(STREAM *);
00785 extern void      PositionStream(STREAM *,LONG);
00786 extern int      ReverseStatements(STREAM *);
00787 extern int      ProcessOption(UBYTE *,UBYTE *,int);
00788 extern int      DoSetups(void);
00789 extern void      TerminateImpl(int, const char *,int, const char *) NORETURN;

```

```

00790 extern NAMENODE *GetNode(NAMETREE *, UBYTE *);
00791 extern int AddName(NAMETREE *, UBYTE *, WORD, WORD, int *);
00792 extern int GetName(NAMETREE *, UBYTE *, WORD *, int);
00793 extern UBYTE *GetFunction(UBYTE *, WORD *);
00794 extern UBYTE *GetNumber(UBYTE *, WORD *);
00795 extern int GetLastExprName(UBYTE *, WORD *);
00796 extern int GetAutoName(UBYTE *, WORD *);
00797 extern int GetVar(UBYTE *, WORD *, WORD *, int, int);
00798 extern int MakeDubious(NAMETREE *, UBYTE *, WORD *);
00799 extern int GetOName(NAMETREE *, UBYTE *, WORD *, int);
00800 extern void DumpTree(NAMETREE *);
00801 extern void DumpNode(NAMETREE *, WORD, WORD);
00802 extern void LinkTree(NAMETREE *, WORD, WORD);
00803 extern void CopyTree(NAMETREE *, NAMETREE *, WORD, WORD);
00804 extern int CompactifyTree(NAMETREE *, WORD);
00805 extern NAMETREE *MakeNameTree(void);
00806 extern void FreeNameTree(NAMETREE *);
00807 extern int AddExpression(UBYTE *, int, int);
00808 extern int AddSymbol(UBYTE *, int, int, int, int);
00809 extern int AddDollar(UBYTE *, WORD, WORD *, LONG);
00810 extern int ReplaceDollar(WORD, WORD, WORD *, LONG);
00811 extern int DollarRaiseLow(UBYTE *, LONG);
00812 extern int AddVector(UBYTE *, int, int);
00813 extern int AddDubious(UBYTE *);
00814 extern int AddIndex(UBYTE *, int, int);
00815 extern UBYTE *DoDimension(UBYTE *, int *, int *);
00816 extern int AddFunction(UBYTE *, int, int, int, int, int, int);
00817 extern int CoCommuteInSet(UBYTE *);
00818 extern int CoFunction(UBYTE *, int, int);
00819 extern int TestName(UBYTE *);
00820 extern int AddSet(UBYTE *, WORD);
00821 extern int DoElements(UBYTE *, SETS, UBYTE *);
00822 extern int DoTempSet(UBYTE *, UBYTE *);
00823 extern int NameConflict(int, UBYTE *);
00824 extern int OpenFile(char *);
00825 extern int OpenAddFile(char *);
00826 extern int ReOpenFile(char *);
00827 extern int CreateFile(char *);
00828 extern int CreateLogFile(char *);
00829 extern void CloseFile(int);
00830 extern int CopyFile(char *, char *);
00831 extern int CreateHandle(void);
00832 extern LONG ReadFile(int, UBYTE *, LONG);
00833 extern LONG ReadPosFile(PHEAD FILEHANDLE *, UBYTE *, LONG, POSITION *);
00834 extern LONG WriteFileToFile(int, UBYTE *, LONG);
00835 extern void SeekFile(int, POSITION *, int);
00836 extern LONG TellFile(int);
00837 extern void FlushFile(int);
00838 extern int GetPosFile(int, fpos_t *);
00839 extern int SetPosFile(int, fpos_t *);
00840 extern void SynchFile(int);
00841 extern void TruncateFile(int);
00842 extern int GetChannel(char *, int);
00843 extern int GetAppendChannel(char *);
00844 extern int CloseChannel(char *);
00845 extern void inictable(void);
00846 extern KEYWORD *findcommand(UBYTE *);
00847 extern int inicbufs(void);
00848 extern void StartFiles(void);
00849 extern UBYTE *MakeDate(void);
00850 extern void PreProcessor(void);
00851 extern void *FromList(LIST *);
00852 extern void *FromUList(LIST *);
00853 extern void *FromVarList(LIST *);
00854 extern int DoubleList(void ***, int *, int, char *);
00855 extern int DoubleLList(void ***, LONG *, int, char *);
00856 extern void DoubleBuffer(void **, void **, int, char *);
00857 extern void ExpandBuffer(void **, LONG *, int);
00858 extern LONG iexp(LONG, int);
00859 extern int IsLikeVector(WORD *);
00860 extern int AreArgsEqual(WORD *, WORD *);
00861 extern int CompareArgs(WORD *, WORD *);
00862 extern UBYTE *SkipField(UBYTE *, int);
00863 extern int StrCmp(UBYTE *, UBYTE *);
00864 extern int StrICmp(UBYTE *, UBYTE *);
00865 extern int StrHICmp(UBYTE *, UBYTE *);
00866 extern int StrICont(UBYTE *, UBYTE *);
00867 extern int CmpArray(WORD *, WORD *, WORD);
00868 extern int ConWord(UBYTE *, UBYTE *);
00869 extern int StrLen(UBYTE *);
00870 extern UBYTE *GetPreVar(UBYTE *, int);
00871 extern void ToGeneral(WORD *, WORD *, WORD);
00872 extern WORD ToPolyFunGeneral(PHEAD WORD *);
00873 extern int ToFast(WORD *, WORD *);
00874 extern SETUPPARAMETERS *GetSetupPar(UBYTE *);
00875 extern int RecalcSetups(void);
00876 extern int AllocSetups(void);

```

```

00877 extern SORTING *AllocSort (LONG, LONG, LONG, LONG, int, int, LONG, int);
00878 extern void AllocSortFileName (SORTING *);
00879 extern UBYTE *LoadInputFile (UBYTE *, int);
00880 extern UBYTE GetInput (void);
00881 extern void ClearPushback (void);
00882 extern UBYTE GetChar (int);
00883 extern void CharOut (UBYTE);
00884 extern void UnsetAllowDelay (void);
00885 extern void PopPreVars (int);
00886 extern void IniModule (int);
00887 extern void IniSpecialModule (int);
00888 extern int ModuleInstruction (int *, int *);
00889 extern int PreProInstruction (void);
00890 extern int LoadInstruction (int);
00891 extern int LoadStatement (int);
00892 extern KEYWORD *FindKeyWord (UBYTE *, KEYWORD *, int);
00893 extern KEYWORD *FindInKeyWord (UBYTE *, KEYWORD *, int);
00894 extern int DoDefine (UBYTE *);
00895 extern int DoRedefine (UBYTE *);
00896 extern int TheDefine (UBYTE *, int);
00897 extern int TheUndefine (UBYTE *);
00898 extern int ClearMacro (UBYTE *);
00899 extern int DoUndefine (UBYTE *);
00900 extern int DoInclude (UBYTE *);
00901 extern int DoReverseInclude (UBYTE *);
00902 extern int Include (UBYTE *, int);
00903 /*[14apr2004 mt]:*/
00904 extern int DoExternal (UBYTE *);
00905 extern int DoToExternal (UBYTE *);
00906 extern int DoFromExternal (UBYTE *);
00907 extern int DoPrompt (UBYTE *);
00908 extern int DoSetExternal (UBYTE *);
00909 /*[10may2006 mt]:*/
00910 extern int DoSetExternalAttr (UBYTE *);
00911 /*:[10may2006 mt]*/
00912 extern int DoRmExternal (UBYTE *);
00913 /*:[14apr2004 mt]*/
00914 extern int DoFactDollar (UBYTE *);
00915 extern WORD GetDollarNumber (UBYTE **, DOLLARS);
00916 extern int DoSetRandom (UBYTE *);
00917 extern int DoOptimize (UBYTE *);
00918 extern int DoClearOptimize (UBYTE *);
00919 extern int DoSkipExtraSymbols (UBYTE *);
00920 extern int DoTimeOutAfter (UBYTE *);
00921 extern int DoMessage (UBYTE *);
00922 extern int DoPreOut (UBYTE *);
00923 extern int DoPreAppend (UBYTE *);
00924 extern int DoPreCreate (UBYTE *);
00925 extern int DoPreAssign (UBYTE *);
00926 extern int DoPreBreak (UBYTE *);
00927 extern int DoPreDefault (UBYTE *);
00928 extern int DoPreSwitch (UBYTE *);
00929 extern int DoPreEndSwitch (UBYTE *);
00930 extern int DoPreCase (UBYTE *);
00931 extern int DoPreShow (UBYTE *);
00932 extern int DoPreExchange (UBYTE *);
00933 extern int DoSystem (UBYTE *);
00934 extern int DoPipe (UBYTE *);
00935 extern void StartPrepro (void);
00936 extern int DoIfdef (UBYTE *, int);
00937 extern int DoIfydef (UBYTE *);
00938 extern int DoIfndef (UBYTE *);
00939 extern int DoElse (UBYTE *);
00940 extern int DoElseif (UBYTE *);
00941 extern int DoEndif (UBYTE *);
00942 extern int DoTerminate (UBYTE *);
00943 extern int DoIf (UBYTE *);
00944 extern int DoCall (UBYTE *);
00945 extern int DoDebug (UBYTE *);
00946 extern int DoContinueDo (UBYTE *);
00947 extern int DoDo (UBYTE *);
00948 extern int DoBreakDo (UBYTE *);
00949 extern int DoEnddo (UBYTE *);
00950 extern int DoEndprocedure (UBYTE *);
00951 extern int DoInside (UBYTE *);
00952 extern int DoEndInside (UBYTE *);
00953 extern int DoProcedure (UBYTE *);
00954 extern int DoPrePrintTimes (UBYTE *);
00955 extern int DoPreWrite (UBYTE *);
00956 extern int DoPreClose (UBYTE *);
00957 extern int DoPreRemove (UBYTE *);
00958 extern int DoPreSortReallocate (UBYTE *);
00959 extern int DoCommentChar (UBYTE *);
00960 extern int DoPrcExtension (UBYTE *);
00961 extern int DoPreReset (UBYTE *);
00962 extern void WriteString (int, UBYTE *, int);
00963 extern void WriteUnfinString (int, UBYTE *, int);

```



```

00964 extern UBYTE *AddToString(UBYTE *,UBYTE *,int);
00965 extern UBYTE *PreCalc(void);
00966 extern UBYTE *PreEval(UBYTE *,LONG *);
00967 extern void NumToStr(UBYTE *,LONG);
00968 extern int PreCmp(int,int,UBYTE *,int,int,UBYTE *,int);
00969 extern int PreEq(int,int,UBYTE *,int,int,UBYTE *,int);
00970 extern UBYTE *pParseObject(UBYTE *,int *,LONG *);
00971 extern UBYTE *PreIfEval(UBYTE *,int *);
00972 extern int EvalPreIf(UBYTE *);
00973 extern int PreLoad(PRELOAD *,UBYTE *,UBYTE *,int,char *);
00974 extern int PreSkip(UBYTE *,UBYTE *,int);
00975 extern UBYTE *EndOfToken(UBYTE *);
00976 extern void SetSpecialMode(int,int);
00977 extern void MakeGlobal(void);
00978 extern int ExecModule(int);
00979 extern int ExecStore(void);
00980 extern void FullCleanup(void);
00981 extern int DoExecStatement(void);
00982 extern int DoPipeStatement(void);
00983 extern int DoPolyfun(UBYTE *);
00984 extern int DoPolyratfun(UBYTE *);
00985 extern int CompileStatement(UBYTE *);
00986 extern UBYTE *ToToken(UBYTE *);
00987 extern int GetDollar(UBYTE *);
00988 extern void MesWork(void) NORETURN;
00989 extern int MesPrint(const char *,...);
00990 extern int MesCall(char *);
00991 extern UBYTE *NumCopy(WORD,UBYTE *);
00992 extern char *LongCopy(LONG,char *);
00993 extern char *LongLongCopy(off_t *,char *);
00994 extern void ReserveTempFiles(int);
00995 extern void PrintTerm(WORD *,char *);
00996 extern void PrintTermC(WORD *,char *);
00997 extern void PrintSubTerm(WORD *,char *);
00998 extern void PrintWords(WORD *,LONG);
00999 extern void PrintSeq(WORD *,char *);
01000 extern int ExpandTripleDots(int);
01001 extern LONG ComPress(WORD **,LONG *);
01002 extern void StageSort(FILEHANDLE *);
01003
01004 #define M_alloc(x)      malloc((size_t)(x))
01005
01006 extern void M_free(void *,const char *);
01007 extern void ClearWildcardNames(void);
01008 extern int AddWildcardName(UBYTE *);
01009 extern int GetWildcardName(UBYTE *);
01010 extern void Globalize(int);
01011 extern void ResetVariables(int);
01012 extern void AddToPreTypes(int);
01013 extern void MessPreNesting(int);
01014 extern LONG GetStreamPosition(STREAM *);
01015 extern WORD *DoubleCbuffer(int,WORD *,int);
01016 extern WORD *AddLHS(int);
01017 extern WORD *AddRHS(int,int);
01018 extern int AddNtoL(int,WORD *);
01019 extern int AddNtoC(int,int,WORD *,int);
01020 extern void DoubleIfBuffers(void);
01021 extern STREAM *CreateStream(UBYTE *);
01022
01023 extern int setonoff(UBYTE *,int *,int,int);
01024 extern int DoPrint(UBYTE *,int);
01025 extern int SetExpr(UBYTE *,int,int);
01026 extern void AddToCom(int,WORD *);
01027 extern int Add2ComStrings(int,WORD *,UBYTE *,UBYTE *);
01028 extern int DoSymmetrize(UBYTE *,int);
01029 extern int DoArgument(UBYTE *,int);
01030 extern int ArgFactorize(PHEAD WORD *,WORD *);
01031 extern WORD *TakeArgContent(PHEAD WORD *,WORD *);
01032 extern WORD *MakeInteger(PHEAD WORD *,WORD *,WORD *);
01033 extern WORD *MakeMod(PHEAD WORD *,WORD *,WORD *);
01034 extern WORD *FindArg(PHEAD WORD *);
01035 extern int InsertArg(PHEAD WORD *,WORD *,int);
01036 extern int CleanupArgCache(PHEAD WORD);
01037 extern int ArgSymbolMerge(WORD *,WORD *);
01038 extern int ArgDotproductMerge(WORD *,WORD *);
01039 extern void SortWeights(LONG *,LONG *,WORD);
01040 extern int DoBrackets(UBYTE *,int);
01041 extern int DoPutInside(UBYTE *,int);
01042 extern WORD *CountComp(UBYTE *,WORD *);
01043 extern int CoAntiBracket(UBYTE *);
01044 extern int CoAntiSymmetrize(UBYTE *);
01045 extern int DoArgPlode(UBYTE *,int);
01046 extern int CoArgExplode(UBYTE *);
01047 extern int CoArgImplode(UBYTE *);
01048 extern int CoArgument(UBYTE *);
01049 extern int CoInside(UBYTE *);
01050 extern int ExecInside(UBYTE *);

```



```
01051 extern int      CoInExpression(UBYTE *);
01052 extern int      CoInParallel(UBYTE *);
01053 extern int      CoNotInParallel(UBYTE *);
01054 extern int      DoInParallel(UBYTE *,int);
01055 extern int      CoEndInExpression(UBYTE *);
01056 extern int      CoBracket(UBYTE *);
01057 extern int      CoPutInside(UBYTE *);
01058 extern int      CoAntiPutInside(UBYTE *);
01059 extern int      CoMultiBracket(UBYTE *);
01060 extern int      CoCFunction(UBYTE *);
01061 extern int      CoCTensor(UBYTE *);
01062 extern int      CoCollect(UBYTE *);
01063 extern int      CoCompress(UBYTE *);
01064 extern int      CoContract(UBYTE *);
01065 extern int      CoCycleSymmetrize(UBYTE *);
01066 extern int      CoDelete(UBYTE *);
01067 extern int      CoTableBase(UBYTE *);
01068 extern int      CoApply(UBYTE *);
01069 extern int      CoDenominators(UBYTE *);
01070 extern int      CoDimension(UBYTE *);
01071 extern int      CoDiscard(UBYTE *);
01072 extern int      CoDisorder(UBYTE *);
01073 extern int      CoDrop(UBYTE *);
01074 extern int      CoDropCoefficient(UBYTE *);
01075 extern int      CoDropSymbols(UBYTE *);
01076 extern int      CoElse(UBYTE *);
01077 extern int      CoElseIf(UBYTE *);
01078 extern int      CoEndArgument(UBYTE *);
01079 extern int      CoEndInside(UBYTE *);
01080 extern int      CoEndIf(UBYTE *);
01081 extern int      CoEndRepeat(UBYTE *);
01082 extern int      CoEndTerm(UBYTE *);
01083 extern int      CoEndWhile(UBYTE *);
01084 extern int      CoExit(UBYTE *);
01085 extern int      CoFactArg(UBYTE *);
01086 extern int      CoFactDollar(UBYTE *);
01087 extern int      CoFactorize(UBYTE *);
01088 extern int      CoNFactorize(UBYTE *);
01089 extern int      CoUnFactorize(UBYTE *);
01090 extern int      CoNUnFactorize(UBYTE *);
01091 extern int      DoFactorize(UBYTE *,int);
01092 extern int      CoFill(UBYTE *);
01093 extern int      CoFillExpression(UBYTE *);
01094 extern int      CoFixIndex(UBYTE *);
01095 extern int      CoFormat(UBYTE *);
01096 extern int      CoGlobal(UBYTE *);
01097 extern int      CoGlobalFactorized(UBYTE *);
01098 extern int      CoGoTo(UBYTE *);
01099 extern int      CoId(UBYTE *);
01100 extern int      CoIdNew(UBYTE *);
01101 extern int      CoIdOld(UBYTE *);
01102 extern int      CoIf(UBYTE *);
01103 extern int      CoIfMatch(UBYTE *);
01104 extern int      CoIfNoMatch(UBYTE *);
01105 extern int      CoIndex(UBYTE *);
01106 extern int      CoInsideFirst(UBYTE *);
01107 extern int      CoKeep(UBYTE *);
01108 extern int      CoLabel(UBYTE *);
01109 extern int      CoLoad(UBYTE *);
01110 extern int      CoLocal(UBYTE *);
01111 extern int      CoLocalFactorized(UBYTE *);
01112 extern int      CoMany(UBYTE *);
01113 extern int      CoMerge(UBYTE *);
01114 extern int      CoShuffle(UBYTE *);
01115 extern int      CoMetric(UBYTE *);
01116 extern int      CoModOption(UBYTE *);
01117 extern int      CoModuleOption(UBYTE *);
01118 extern int      CoModulus(UBYTE *);
01119 extern int      CoMulti(UBYTE *);
01120 extern int      CoMultiply(UBYTE *);
01121 extern int      CoNFunction(UBYTE *);
01122 extern int      CoNPrint(UBYTE *);
01123 extern int      CoNTensor(UBYTE *);
01124 extern int      CoNWrite(UBYTE *);
01125 extern int      CoNoDrop(UBYTE *);
01126 extern int      CoNoSkip(UBYTE *);
01127 extern int      CoNormalize(UBYTE *);
01128 extern int      CoMakeInteger(UBYTE *);
01129 extern int      CoFlags(UBYTE *,int);
01130 extern int      CoOff(UBYTE *);
01131 extern int      CoOn(UBYTE *);
01132 extern int      CoOnce(UBYTE *);
01133 extern int      CoOnly(UBYTE *);
01134 extern int      CoOptimizeOption(UBYTE *);
01135 extern int      CoPolyFun(UBYTE *);
01136 extern int      CoPolyRatFun(UBYTE *);
01137 extern int      CoPrint(UBYTE *);
```

```

01138 extern int      CoPrintB(UBYTE *);
01139 extern int      CoProperCount(UBYTE *);
01140 extern int      CoUnitTrace(UBYTE *);
01141 extern int      CoRCycleSymmetrize(UBYTE *);
01142 extern int      CoRatio(UBYTE *);
01143 extern int      CoRedefine(UBYTE *);
01144 extern int      CoRenumber(UBYTE *);
01145 extern int      CoRepeat(UBYTE *);
01146 extern int      CoSave(UBYTE *);
01147 extern int      CoSelect(UBYTE *);
01148 extern int      CoSet(UBYTE *);
01149 extern int      CoSetExitFlag(UBYTE *);
01150 extern int      CoSkip(UBYTE *);
01151 extern int      CoProcessBucket(UBYTE *);
01152 extern int      CoPushHide(UBYTE *);
01153 extern int      CoPopHide(UBYTE *);
01154 extern int      CoHide(UBYTE *);
01155 extern int      CoIntoHide(UBYTE *);
01156 extern int      CoNoIntoHide(UBYTE *);
01157 extern int      CoNoHide(UBYTE *);
01158 extern int      CoUnHide(UBYTE *);
01159 extern int      CoNoUnHide(UBYTE *);
01160 extern int      CoSort(UBYTE *);
01161 extern int      CoSplitArg(UBYTE *);
01162 extern int      CoSplitFirstArg(UBYTE *);
01163 extern int      CoSplitLastArg(UBYTE *);
01164 extern int      CoSum(UBYTE *);
01165 extern int      CoSymbol(UBYTE *);
01166 extern int      CoSymmetrize(UBYTE *);
01167 extern int      DoTable(UBYTE *,int);
01168 extern int      CoTable(UBYTE *);
01169 extern int      CoTerm(UBYTE *);
01170 extern int      CoNTable(UBYTE *);
01171 extern int      CoCTable(UBYTE *);
01172 extern void      EmptyTable(TABLES);
01173 extern int      CoToTensor(UBYTE *);
01174 extern int      CoToVector(UBYTE *);
01175 extern int      CoTrace4(UBYTE *);
01176 extern int      CoTraceN(UBYTE *);
01177 extern int      CoChisholm(UBYTE *);
01178 extern int      CoTransform(UBYTE *);
01179 extern int      CoClearTable(UBYTE *);
01180 extern int      DoChain(UBYTE *,int);
01181 extern int      CoChainin(UBYTE *);
01182 extern int      CoChainout(UBYTE *);
01183 extern int      CoTryReplace(UBYTE *);
01184 extern int      CoVector(UBYTE *);
01185 extern int      CoWhile(UBYTE *);
01186 extern int      CoWrite(UBYTE *);
01187 extern int      CoAuto(UBYTE *);
01188 extern int      CoSwitch(UBYTE *);
01189 extern int      CoCase(UBYTE *);
01190 extern int      CoBreak(UBYTE *);
01191 extern int      CoDefault(UBYTE *);
01192 extern int      CoEndSwitch(UBYTE *);
01193 extern int      CoTBaddto(UBYTE *);
01194 extern int      CoTBaudit(UBYTE *);
01195 extern int      CoTbcleanup(UBYTE *);
01196 extern int      CoTbcreate(UBYTE *);
01197 extern int      CoTBenter(UBYTE *);
01198 extern int      CoTBhelp(UBYTE *);
01199 extern int      CoTBload(UBYTE *);
01200 extern int      CoTBoff(UBYTE *);
01201 extern int      CoTBon(UBYTE *);
01202 extern int      CoTBopen(UBYTE *);
01203 extern int      CoTBreplace(UBYTE *);
01204 extern int      CoTBuse(UBYTE *);
01205 extern int      CoTestUse(UBYTE *);
01206 extern int      CoThreadBucket(UBYTE *);
01207 extern int      AddComString(int,WORD *,UBYTE *,int);
01208 extern int      CompileAlgebra(UBYTE *,int,WORD *);
01209 extern int      IsIdStatement(UBYTE *);
01210 extern UBYTE *  IsRHS(UBYTE *,UBYTE);
01211 extern int      ParenthesesTest(UBYTE *);
01212 extern int      tokenize(UBYTE *,WORD);
01213 extern void      WriteTokens(SBYTE *);
01214 extern int      simpltoken(SBYTE *);
01215 extern int      simpwtoken(SBYTE *);
01216 extern int      simp2token(SBYTE *);
01217 extern int      simp3atoken(SBYTE *,int);
01218 extern int      simp3btoken(SBYTE *,int);
01219 extern int      simp4token(SBYTE *);
01220 extern int      simp5token(SBYTE *,int);
01221 extern int      simp6token(SBYTE *,int);
01222 extern UBYTE *  SkipAName(UBYTE *);
01223 extern int      TestTables(void);
01224 extern int      GetLabel(UBYTE *);

```

```

01225 extern int      CoIdExpression(UBYTE *,int);
01226 extern int      CoAssign(UBYTE *);
01227 extern int      DoExpr(UBYTE *,int,int);
01228 extern int      CompileSubExpressions(SBYTE *);
01229 extern int      CodeGenerator(SBYTE *);
01230 extern int      CompleteTerm(WORD *,UWORD *,WORD,WORD,int);
01231 extern int      CodeFactors(SBYTE *s);
01232 extern WORD     GenerateFactors(WORD,WORD);
01233 extern int      InsTree(int,int);
01234 extern int      FindTree(int,WORD *);
01235 extern void     RedoTree(CBUF *,int);
01236 extern void     ClearTree(int);
01237 extern int      CatchDollar(int);
01238 extern int      AssignDollar(PHEAD WORD *,WORD);
01239 extern UBYTE    *WriteDollarToBuffer(WORD,WORD);
01240 extern UBYTE    *WriteDollarFactorToBuffer(WORD,WORD,WORD);
01241 extern void     AddToDollarBuffer(UBYTE *);
01242 extern int      PutTermInDollar(WORD *,WORD);
01243 extern void     TermAssign(WORD *);
01244 extern void     WildDollars(PHEAD WORD *);
01245 extern LONG     numcommute(WORD *,LONG *);
01246 extern int      FullRenumber(PHEAD WORD *,WORD);
01247 extern int      Lus(WORD *,WORD,WORD,WORD,WORD,WORD);
01248 extern int      FindLus(int,int,int);
01249 extern int      CoReplaceLoop(UBYTE *);
01250 extern int      CoFindLoop(UBYTE *);
01251 extern int      DoFindLoop(UBYTE *,int);
01252 extern int      CoFunPowers(UBYTE *);
01253 extern int      SortTheList(int *,int);
01254 extern int      MatchIsPossible(WORD *,WORD *);
01255 extern int      StudyPattern(WORD *);
01256 extern WORD     DolToTensor(PHEAD WORD);
01257 extern WORD     DolToFunction(PHEAD WORD);
01258 extern WORD     DolToVector(PHEAD WORD);
01259 extern WORD     DolToNumber(PHEAD WORD);
01260 extern WORD     DolToSymbol(PHEAD WORD);
01261 extern WORD     DolToIndex(PHEAD WORD);
01262 extern LONG     DolToLong(PHEAD WORD);
01263 extern int      DollarFactorize(PHEAD WORD);
01264 extern int      CoPrintTable(UBYTE *);
01265 extern int      CoDeallocateTable(UBYTE *);
01266 extern void     CleanDollarFactors(DOLLARS);
01267 extern WORD     *TakeDollarContent(PHEAD WORD *,WORD **);
01268 extern WORD     *MakeDollarInteger(PHEAD WORD *,WORD **);
01269 extern WORD     *MakeDollarMod(PHEAD WORD *,WORD **);
01270 extern int      GetDolNum(PHEAD WORD *,WORD *);
01271 extern void     AddPotModdollar(WORD);
01272
01273 extern int      Optimize(WORD, int);
01274 extern int      ClearOptimize(void);
01275 extern int      TakeLongRoot(UWORD *,WORD *,WORD);
01276 extern int      TakeRatRoot(UWORD *,WORD *,WORD);
01277 extern int      MakeRational(WORD ,WORD ,WORD *,WORD *);
01278 extern int      MakeLongRational(PHEAD UWORD *,WORD ,UWORD *,WORD ,UWORD *,WORD *);
01279 extern void     ClearTableTree(TABLES);
01280 extern int      InsTableTree(TABLES,WORD *);
01281 extern void     RedoTableTree(TABLES,int);
01282 extern int      FindTableTree(TABLES,WORD *,int);
01283 extern void     finishcbuf(WORD);
01284 extern void     clearcbuf(WORD);
01285 extern void     CleanupSort(int);
01286 extern FILEHANDLE *AllocFileHandle(WORD,char *);
01287 extern void     DeAllocFileHandle(FILEHANDLE *);
01288 extern void     LowerSortLevel(void);
01289 extern WORD     *PolyRatFunSpecial(PHEAD WORD *,WORD *);
01290 extern void     SimpleSplitMergeRec(WORD *,WORD,WORD *);
01291 extern void     SimpleSplitMerge(WORD *,WORD);
01292 extern WORD     BinarySearch(WORD *,WORD,WORD);
01293 extern int      InsideDollar(PHEAD WORD *,WORD);
01294 extern DOLLARS  DolToTerms(PHEAD WORD);
01295 extern WORD     EvalDoLoopArg(PHEAD WORD *,WORD);
01296 extern int      SetExprCases(int,int,int);
01297 extern int      TestSelect(WORD *,WORD *);
01298 extern void     SubsInAll(PHEAD0);
01299 extern void     TransferBuffer(int,int,int);
01300 extern int      TakeIDfunction(PHEAD WORD *);
01301 extern int      MakeSetupAllocs(void);
01302 extern int      TryFileSetups(void);
01303 extern void     ExchangeExpressions(int,int);
01304 extern void     ExchangeDollars(int,int);
01305 extern int      GetFirstBracket(WORD *,int);
01306 extern int      GetFirstTerm(WORD *,int,int);
01307 extern int      GetContent(WORD *,int);
01308 extern int      CleanupTerm(WORD *);
01309 extern WORD     ContentMerge(PHEAD WORD *,WORD *);
01310 extern UBYTE    *PreIfDollarEval(UBYTE *,int *);
01311 extern LONG     TermsInDollar(WORD);

```

```

01312 extern LONG      SizeOfDollar(WORD);
01313 extern LONG      TermsInExpression(WORD);
01314 extern LONG      SizeOfExpression(WORD);
01315 extern WORD      *TranslateExpression(UBYTE *);
01316 extern int       IsSetMember(WORD *,WORD);
01317 extern int       IsMultipleOf(WORD *,WORD *);
01318 extern int       TwoExprCompare(WORD *,WORD *,int);
01319 extern void      UpdatePositions(void);
01320 extern void      M_check(void);
01321 extern void      M_print(void);
01322 extern void      M_check1(void);
01323 extern void      PrintTime(UBYTE *);
01324
01325 extern POSITION   *FindBracket(WORD,WORD *);
01326 extern void      PutBracketInIndex(PHEAD WORD *,POSITION *);
01327 extern void      ClearBracketIndex(WORD);
01328 extern void      OpenBracketIndex(WORD);
01329 extern int       DoNoParallel(UBYTE *);
01330 extern int       DoParallel(UBYTE *);
01331 extern int       DoModSum(UBYTE *);
01332 extern int       DoModMax(UBYTE *);
01333 extern int       DoModMin(UBYTE *);
01334 extern int       DoModLocal(UBYTE *);
01335 extern UBYTE     *DoModDollar(UBYTE *,int);
01336 extern int       DoProcessBucket(UBYTE *);
01337 extern int       DoinParallel(UBYTE *);
01338 extern int       DonotinParallel(UBYTE *);
01339
01340 extern int       FlipTable(FUNCTIONS,int);
01341 extern int       ChainIn(PHEAD WORD *,WORD);
01342 extern int       ChainOut(PHEAD WORD *,WORD);
01343 extern int       ArgumentImplode(PHEAD WORD *,WORD *);
01344 extern int       ArgumentExplode(PHEAD WORD *,WORD *);
01345 extern int       DenToFunction(WORD *,WORD);
01346
01347 extern WORD      HowMany(PHEAD WORD *,WORD *);
01348 extern void      RemoveDollars(void);
01349 extern LONG      CountTerms1(PHEAD0);
01350 extern LONG      TermsInBracket(PHEAD WORD *,WORD);
01351 extern int       Crash(void);
01352
01353 extern char      *str_dup(char *);
01354 extern void      convertblock(INDEXBLOCK *,INDEXBLOCK *,int);
01355 extern void      convertnamesblock(NAMESBLOCK *,NAMESBLOCK *,int);
01356 extern void      convertiniinfo(INIINFO *,INIINFO *,int);
01357 extern int       ReadIndex(DBASE *);
01358 extern int       WriteIndexBlock(DBASE *,MLONG);
01359 extern int       WriteNamesBlock(DBASE *,MLONG);
01360 extern int       WriteIndex(DBASE *);
01361 extern int       WriteIniInfo(DBASE *);
01362 extern int       ReadIniInfo(DBASE *);
01363 extern int       AddToIndex(DBASE *,MLONG);
01364 extern DBASE     *GetDbase(char *,MLONG);
01365 extern DBASE     *OpenDbase(char *);
01366 extern char      *ReadObject(DBASE *,MLONG,char *);
01367 extern char      *ReadiObject(DBASE *,MLONG,MLONG,char *);
01368 extern int       ExistsObject(DBASE *,MLONG,char *);
01369 extern int       DeleteObject(DBASE *,MLONG,char *);
01370 extern int       WriteObject(DBASE *,MLONG,char *,char *,MLONG);
01371 extern MLONG     AddObject(DBASE *,MLONG,char *,char *);
01372 extern DBASE     *NewDbase(char *,MLONG);
01373 extern void      FreeTableBase(DBASE *);
01374 extern int       ComposeTableNames(DBASE *);
01375 extern int       PutTableNames(DBASE *);
01376 extern MLONG     AddTableName(DBASE *,char *,TABLES);
01377 extern MLONG     GetTableName(DBASE *,char *);
01378 extern MLONG     FindTableNumber(DBASE *,char *);
01379 extern int       TryEnvironment(void);
01380
01381 #ifdef WITHZLIB
01382 extern int       SetupOutputGZIP(FILEHANDLE *);
01383 extern int       PutOutputGZIP(FILEHANDLE *);
01384 extern int       FlushOutputGZIP(FILEHANDLE *);
01385 extern int       SetupAllInputGZIP(SORTING *);
01386 extern LONG      FillInputGZIP(FILEHANDLE *,POSITION *,UBYTE *,LONG,int);
01387 extern void      ClearSortGZIP(FILEHANDLE *f);
01388 #endif
01389
01390 #ifdef WITHPTHREADS
01391 extern void      BeginIdentities(void);
01392 extern int       WhoAmI(void);
01393 extern int       StartAllThreads(int);
01394 extern void      StartHandleLock(void);
01395 extern void      TerminateAllThreads(void);
01396 extern int       GetAvailableThread(void);
01397 extern int       ConditionalGetAvailableThread(void);
01398 extern int       BalanceRunThread(PHEAD int,WORD *,WORD);

```

```

01399 extern void    WakeupThread(int,int);
01400 extern int      MasterWait(void);
01401 extern int      InParallelProcessor(void);
01402 extern int      ThreadsProcessor(EXPRESSIONS,WORD,WORD);
01403 extern int      MasterMerge(void);
01404 extern int      PutToMaster(PHEAD WORD *);
01405 extern void     SetWorkerFiles(void);
01406 extern int      MakeThreadBuckets(int,int);
01407 extern int      SendOneBucket(int);
01408 extern int      LoadOneThread(int,int,THREADBUCKET *,int);
01409 extern void     *RunSortBot(void *);
01410 extern void     MasterWaitAllSortBots(void);
01411 extern int      SortBotMerge(PHEAD0);
01412 extern int      SortBotOut(PHEAD WORD *);
01413 extern void     DefineSortBotTree(void);
01414 extern int      SortBotMasterMerge(void);
01415 extern int      SortBotWait(int);
01416 extern void     StartIdentity(void);
01417 extern void     FinishIdentity(void *);
01418 extern int      SetIdentity(int *);
01419 extern ALLPRIVATES *InitializeOneThread(int);
01420 extern void     FinalizeOneThread(int);
01421 extern void     ClearAllThreads(void);
01422 extern void     *RunThread(void *);
01423 extern void     IAmAvailable(int);
01424 extern int      ThreadWait(int);
01425 extern int      ThreadClaimedBlock(int);
01426 extern int      GetThread(int);
01427 extern int      UpdateOneThread(int);
01428 extern void     MasterWaitAll(void);
01429 extern void     MasterWaitAllBlocks(void);
01430 extern int      MasterWaitThread(int);
01431 extern void     WakeupMasterFromThread(int,int);
01432 extern int      LoadReadjusted(void);
01433 extern int      IniSortBlocks(int);
01434 extern int      UpdateSortBlocks(int);
01435 extern int      TreatIndexEntry(PHEAD LONG);
01436 extern WORD     GetTerm2(PHEAD WORD *);
01437 extern void     SetHideFiles(void);
01438
01439 #endif
01440
01441 extern int      CopyExpression(FILEHANDLE *,FILEHANDLE *);
01442
01443 extern int      set_in(UBYTE, set_of_char);
01444 extern one_byte set_set(UBYTE, set_of_char);
01445 extern one_byte set_del(UBYTE, set_of_char);
01446 extern one_byte set_sub(set_of_char, set_of_char);
01447 extern int      DoPreAddSeparator(UBYTE *);
01448 extern int      DoPreRmSeparator(UBYTE *);
01449
01450 /*See the file extcmd.c*/
01451 extern int      openExternalChannel(UBYTE *,int,UBYTE *,UBYTE *);
01452 extern int      initPresetExternalChannels(UBYTE *, int);
01453 extern int      closeExternalChannel(int);
01454 extern int      selectExternalChannel(int);
01455 extern int      getCurrentExternalChannel(void);
01456 extern void     closeAllExternalChannels(void);
01457
01458 typedef int      (*WRITEBUFTOEXTCHANNEL)(char *,size_t);
01459 typedef int      (*GETCFROMEXTCHANNEL)(void);
01460 typedef int      (*SETTERMINATORFOREXTCHANNEL)(char *);
01461 typedef int      (*SETKILLMODEFOREXTCHANNEL)(int,int);
01462 typedef LONG     (*WRITEFILE)(int,UBYTE *,LONG);
01463 typedef WORD     (*GETTERM)(PHEAD WORD *);
01464
01465 #define CompareTerms ((COMPARE)AR.CompareRoutine)
01466 #define FiniShuffle AN.SHvar.finishuf
01467 #define DoShtuffle ((DO_UFFLE)AN.SHvar.do_uffle)
01468
01469 extern UBYTE *defineChannel(UBYTE*, HANDLERS*);
01470 extern int    writeToChannel(int,UBYTE *,HANDLERS*);
01471 #ifdef WITHEXTCHANNEL
01472 extern LONG   WriteToExternalChannel(int,UBYTE *,LONG);
01473 #endif
01474 extern int    writeBufToExtChannelOk(char *,size_t);
01475 extern int    getcFromExtChannelOk(void);
01476 extern int    setKillModeForExternalChannelOk(int,int);
01477 extern int    setTerminatorForExternalChannelOk(char *);
01478 extern int    getcFromExtChannelFailure(void);
01479 extern int    setKillModeForExternalChannelFailure(int,int);
01480 extern int    setTerminatorForExternalChannelFailure(char *);
01481 extern int    writeBufToExtChannelFailure(char *,size_t);
01482
01483 extern int    ReleaseTB(void);
01484
01485 extern int    SymbolNormalize(WORD *);

```

```

01486 extern int      TestFunFlag(PHEAD WORD *);
01487 extern WORD      CompareSymbols(PHEAD WORD *,WORD *,WORD);
01488 extern WORD      CompareHSymbols(PHEAD WORD *,WORD *,WORD);
01489 extern WORD      NextPrime(PHEAD WORD);
01490 extern WORD      Moebius(PHEAD WORD);
01491 extern UWORD     wranf(PHEAD0);
01492 extern UWORD     iranf(PHEAD UWORD);
01493 extern void      iniwranf(PHEAD0);
01494 extern UBYTE     *PreRandom(UBYTE *);
01495
01496 extern WORD      *PolyNormPoly(PHEAD WORD);
01497 extern WORD      *EvaluateGcd(PHEAD WORD *);
01498 extern int       TreatPolyRatFun(PHEAD WORD *);
01499
01500 extern int       ReadSaveHeader(void);
01501 extern LONG      ReadSaveIndex(FILEINDEX *);
01502 extern LONG      ReadSaveExpression(UBYTE *,UBYTE *,LONG *,LONG *);
01503 extern UBYTE     *ReadSaveTerm32(UBYTE *,UBYTE **,UBYTE *,UBYTE *,int);
01504 extern LONG      ReadSaveVariables(UBYTE *,UBYTE *,LONG *,LONG *,INDEXENTRY *,LONG *);
01505 extern LONG      WriteStoreHeader(WORD);
01506
01507 extern void      InitRecovery(void);
01508 extern int       CheckRecoveryFile(void);
01509 extern void      DeleteRecoveryFile(void);
01510 extern char      *RecoveryFilename(void);
01511 extern int       DoRecovery(int *);
01512 extern void      DoCheckpoint(int);
01513
01514 extern void      NumberMallocAddMemory(PHEAD0);
01515 extern void      CacheNumberMallocAddMemory(PHEAD0);
01516 extern void      TermMallocAddMemory(PHEAD0);
01517 #ifndef MEMORYMACROS
01518 extern WORD      *TermMalloc2(PHEAD char *text);
01519 extern void      TermFree2(PHEAD WORD *term,char *text);
01520 extern UWORD     *NumberMalloc2(PHEAD char *text);
01521 extern UWORD     *CacheNumberMalloc2(PHEAD char *text);
01522 extern void      NumberFree2(PHEAD UWORD *NumberMem,char *text);
01523 extern void      CacheNumberFree2(PHEAD UWORD *NumberMem,char *text);
01524 #endif
01525
01526 extern void      ExprStatus(EXPRESSIONS);
01527 extern void      iniTools(void);
01528 extern int       TestTerm(WORD *);
01529
01530 extern int       RunTransform(PHEAD WORD *term, WORD *params);
01531 extern int       RunEncode(PHEAD WORD *fun, WORD *args, WORD *info);
01532 extern int       RunDecode(PHEAD WORD *fun, WORD *args, WORD *info);
01533 extern int       RunReplace(PHEAD WORD *fun, WORD *args, WORD *info);
01534 extern int       RunImplode(WORD *fun, WORD *args);
01535 extern int       RunExplode(PHEAD WORD *fun, WORD *args);
01536 extern int       TestArgNum(int n, int totarg, WORD *args);
01537 extern WORD      PutArgInScratch(WORD *arg,UWORD *scrat);
01538 extern UBYTE     *ReadRange(UBYTE *s, WORD *out, int par);
01539 extern int       FindRange(PHEAD WORD *,WORD *,WORD *,WORD);
01540 extern int       RunPermute(PHEAD WORD *fun, WORD *args, WORD *info);
01541 extern int       RunReverse(PHEAD WORD *fun, WORD *args);
01542 extern int       RunCycle(PHEAD WORD *fun, WORD *args, WORD *info);
01543 extern int       RunAddArg(PHEAD WORD *fun, WORD *args);
01544 extern int       RunMulArg(PHEAD WORD *fun, WORD *args);
01545 extern int       RunIsLyndon(PHEAD WORD *fun, WORD *args, int par);
01546 extern WORD      RunToLyndon(PHEAD WORD *fun, WORD *args, int par);
01547 extern int       RunDropArg(PHEAD WORD *fun, WORD *args);
01548 extern int       RunSelectArg(PHEAD WORD *fun, WORD *args);
01549 extern int       RunDedup(PHEAD WORD *fun, WORD *args);
01550 extern int       RunZtoHArg(PHEAD WORD *fun, WORD *args);
01551 extern int       RunHtoZArg(PHEAD WORD *fun, WORD *args);
01552
01553 extern int       NormPolyTerm(PHEAD WORD *);
01554 extern WORD      ComparePoly(WORD *, WORD *, WORD);
01555 extern int       ConvertToPoly(PHEAD WORD *, WORD *,WORD *,WORD);
01556 extern int       LocalConvertToPoly(PHEAD WORD *, WORD *, WORD,WORD);
01557 extern int       ConvertFromPoly(PHEAD WORD *, WORD *, WORD, WORD, WORD, WORD);
01558 extern int       FindSubterm(WORD *);
01559 extern int       FindLocalSubterm(PHEAD WORD *, WORD);
01560 extern void      PrintSubtermList(int,int);
01561 extern void      PrintExtraSymbol(int,WORD *,int);
01562 extern int       FindSubexpression(WORD *);
01563
01564 extern void      UpdateMaxSize(void);
01565
01566 extern int       CoToPolynomial(UBYTE *);
01567 extern int       CoFromPolynomial(UBYTE *);
01568 extern int       CoArgToExtraSymbol(UBYTE *);
01569 extern int       CoExtraSymbols(UBYTE *);
01570 extern UBYTE     *GetDoParam(UBYTE *, WORD **, int);
01571 extern WORD      *GetIfDollarFactor(UBYTE **, WORD *);
01572 extern int       CoDo(UBYTE *);

```



```

01573 extern int CoEndDo(UBYTE *);
01574 extern int ExtraSymFun(PHEAD WORD *,WORD);
01575 extern int PruneExtraSymbols(WORD);
01576 extern int IniFbuffer(WORD);
01577 extern void IniFbufs(void);
01578 extern int GCDfunction(PHEAD WORD *,WORD);
01579 extern WORD *GCDfunction3(PHEAD WORD *,WORD *);
01580 extern int ReadPolyRatFun(PHEAD WORD *);
01581 extern int FromPolyRatFun(PHEAD WORD *, WORD **, WORD **);
01582 extern WORD *PolyDiv(PHEAD WORD *,WORD *,char *);
01583 extern void GCDclean(PHEAD WORD *, WORD *);
01584 extern WORD *TakeSymbolContent(PHEAD WORD *,WORD *);
01585 extern int GCDterms(PHEAD WORD *,WORD *,WORD *);
01586 extern WORD *PutExtraSymbols(PHEAD WORD *,WORD,int *);
01587 extern WORD *TakeExtraSymbols(PHEAD WORD *,WORD);
01588 extern WORD *MultiplyWithTerm(PHEAD WORD *, WORD *,WORD);
01589 extern WORD *TakeContent(PHEAD WORD *, WORD *);
01590 extern int MergeSymbolLists(PHEAD WORD *, WORD *, int);
01591 extern int MergeDotproductLists(PHEAD WORD *, WORD *, int);
01592 extern WORD *CreateExpression(PHEAD WORD);
01593 extern int DIVfunction(PHEAD WORD *,WORD,int);
01594 extern WORD *MULfunc(PHEAD WORD *, WORD *);
01595 extern WORD *ConvertArgument(PHEAD WORD *,int *);
01596 extern int ExpandRat(PHEAD WORD *);
01597 extern int InvPoly(PHEAD WORD *,WORD,WORD);
01598 extern WORD TestDoLoop(PHEAD WORD *,WORD);
01599 extern WORD TestEndDoLoop(PHEAD WORD *,WORD);
01600
01601 extern WORD *poly_gcd(PHEAD WORD *, WORD *, WORD);
01602 extern WORD *poly_div(PHEAD WORD *, WORD *, WORD);
01603 extern WORD *poly_rem(PHEAD WORD *, WORD *, WORD);
01604 extern WORD *poly_inverse(PHEAD WORD *, WORD *);
01605 extern WORD *poly_mul(PHEAD WORD *, WORD *);
01606 extern WORD *poly_ratfun_add(PHEAD WORD *, WORD *);
01607 extern int poly_ratfun_normalize(PHEAD WORD *);
01608 extern int poly_factorize_argument(PHEAD WORD *, WORD *);
01609 extern WORD *poly_factorize_dollar(PHEAD WORD *);
01610 extern int poly_factorize_expression(EXPRESSIONS);
01611 extern int poly_unfactorize_expression(EXPRESSIONS);
01612 extern void poly_free_poly_vars(PHEAD const char *);
01613
01614 #ifdef WITHFLINT
01615 extern void flint_final_cleanup_thread(void);
01616 extern void flint_final_cleanup_master(void);
01617 extern WORD* flint_div(PHEAD WORD *, WORD *, const WORD);
01618 extern int flint_factorize_argument(PHEAD WORD *, WORD *);
01619 extern WORD* flint_factorize_dollar(PHEAD WORD *);
01620 extern WORD* flint_gcd(PHEAD WORD *, WORD *, const WORD);
01621 extern WORD* flint_inverse(PHEAD WORD *, WORD *);
01622 extern WORD* flint_mul(PHEAD WORD *, WORD *);
01623 extern WORD* flint_ratfun_add(PHEAD WORD *, WORD *);
01624 extern int flint_ratfun_normalize(PHEAD WORD *);
01625 extern WORD* flint_rem(PHEAD WORD *, WORD *, const WORD);
01626 extern void flint_check_version(void);
01627 #endif
01628
01629 extern void optimize_print_code (int);
01630
01631 #ifdef WITHPTHREADS
01632 extern void find_Horner_MCTS_expand_tree(void);
01633 extern void find_Horner_MCTS_expand_tree_threaded(void);
01634 extern void optimize_expression_given_Horner(void);
01635 extern void optimize_expression_given_Horner_threaded(void);
01636 #endif
01637
01638 extern int DoPreAdd(UBYTE *s);
01639 extern int DoPreUseDictionary(UBYTE *s);
01640 extern int DoPreCloseDictionary(UBYTE *s);
01641 extern int DoPreOpenDictionary(UBYTE *s);
01642 extern void RemoveDictionary(DICTIONARY *dict);
01643 extern void UnSetDictionary(void);
01644 extern int SetDictionaryOptions(UBYTE *options);
01645 extern int AddToDictionary(DICTIONARY *dict,UBYTE *left,UBYTE *right);
01646 extern int AddDictionary(UBYTE *name);
01647 extern int FindDictionary(UBYTE *name);
01648 extern UBYTE *IsExponentSign(void);
01649 extern UBYTE *IsMultiplySign(void);
01650 extern void TransformRational(UWORD *a, WORD na);
01651 extern void WriteDictionary(DICTIONARY *);
01652 extern void ShrinkDictionary(DICTIONARY *);
01653 extern void MultiplyToLine(void);
01654 extern UBYTE *FindSymbol(WORD num);
01655 extern UBYTE *FindVector(WORD num);
01656 extern UBYTE *FindIndex(WORD num);
01657 extern UBYTE *FindFunction(WORD num);
01658 extern UBYTE *FindFunWithArgs(WORD *t);
01659 extern UBYTE *FindExtraSymbol(WORD num);

```

```

01660 extern LONG DictToBytes(DICTIONARY *dict, UBYTE *buf);
01661 extern DICTIONARY *DictFromBytes(UBYTE *buf);
01662 extern int CoCreateSpectator(UBYTE *inp);
01663 extern int CoToSpectator(UBYTE *inp);
01664 extern int CoRemoveSpectator(UBYTE *inp);
01665 extern int CoEmptySpectator(UBYTE *inp);
01666 extern int CoCopySpectator(UBYTE *inp);
01667 extern int PutInSpectator(WORD *, WORD);
01668 extern void ClearSpectators(WORD);
01669 extern WORD GetFromSpectator(WORD *, WORD);
01670 extern void FlushSpectators(void);
01671
01672 extern WORD *PreGCD(PHEAD WORD *, WORD *, int);
01673 extern int InvPoly(PHEAD WORD *, WORD, WORD);
01674
01675 extern int ReadFromScratch(FILEHANDLE *, POSITION *, UBYTE *, POSITION *);
01676 extern int AddToScratch(FILEHANDLE *, POSITION *, UBYTE *, POSITION *, int);
01677
01678 extern int DoPreAppendPath(UBYTE *);
01679 extern int DoPrePrependPath(UBYTE *);
01680
01681 extern int DoSwitch(PHEAD WORD *, WORD *);
01682 extern int DoEndSwitch(PHEAD WORD *, WORD *);
01683 extern SWITCHTABLE *FindCase(WORD, WORD);
01684 extern void SwitchSplitMergeRec(SWITCHTABLE *, WORD, SWITCHTABLE *);
01685 extern void SwitchSplitMerge(SWITCHTABLE *, WORD);
01686 extern int DoubleSwitchBuffers(void);
01687
01688 extern int DistrN(int, int *, int, int *);
01689
01690 extern int DoNamespace(UBYTE *);
01691 extern int DoEndNamespace(UBYTE *);
01692 extern int DoUse(UBYTE *);
01693 extern UBYTE *SkipName(UBYTE *);
01694 extern UBYTE *ConstructName(UBYTE *, UBYTE);
01695 extern int DoSetUserFlag(UBYTE *);
01696 extern int DoClearUserFlag(UBYTE *);
01697
01698 VERTEX *CreateVertex(MODEL *);
01699 UBYTE *ReadParticle(UBYTE *, VERTEX *, MODEL *, int);
01700 int CoModel(UBYTE *);
01701 int CoParticle(UBYTE *);
01702 int CoVertex(UBYTE *);
01703 int CoEndModel(UBYTE *);
01704 int LoadModel(MODEL *);
01705
01706 int CoSetUserFlag(UBYTE *);
01707 int CoClearUserFlag(UBYTE *);
01708 int CoCreateAllLoops(UBYTE *);
01709 int CoCreateAllPaths(UBYTE *);
01710 int CoCreateAll(UBYTE *);
01711
01712 int AllLoops(PHEAD WORD *, WORD);
01713 LONG StartLoops(PHEAD WORD *, WORD, LONG, WORD, WORD *, WORD, WORD *, WORD);
01714 LONG GenLoops(PHEAD WORD *, WORD, LONG, WORD, WORD *, WORD, WORD *, WORD);
01715 void LoopOutput(PHEAD WORD *, WORD, WORD *, WORD);
01716 int AllPaths(PHEAD WORD *, WORD);
01717 LONG GenPaths(PHEAD WORD *, WORD, LONG, WORD, WORD *, WORD, WORD *, WORD);
01718 void PathOutput(PHEAD WORD *, WORD, WORD *, WORD);
01719
01720 #ifdef WITHFLOAT
01721 int DoStartFloat(UBYTE *);
01722 int DoEndFloat(UBYTE *);
01723 int SetFloatPrecision(WORD);
01724 void SetupMZVTables(void);
01725 void SetupMPFTTables(void);
01726 void ClearMZVTables(void);
01727 int EvaluateEuler(PHEAD WORD *, WORD, WORD);
01728 int EvaluateSqrt(PHEAD WORD *, WORD, WORD);
01729 int CoEvaluate(UBYTE *);
01730 int PrintFloat(WORD *fun, int numdigits);
01731 int CoToRat(UBYTE *);
01732 int CoToFloat(UBYTE *);
01733 int CoChop(UBYTE *);
01734 int ToRat(PHEAD WORD *, WORD);
01735 int ToFloat(PHEAD WORD *, WORD);
01736 int Chop(PHEAD WORD *, WORD);
01737 WORD FloatFunToRat(PHEAD UWORD *, WORD *);
01738 int AddWithFloat(PHEAD WORD **, WORD **);
01739 int MergeWithFloat(PHEAD WORD **, WORD **);
01740 void ClearfFloat(void);
01741 int TestFloat(WORD *);
01742 SBYTE *ReadFloat(SBYTE *);
01743 UBYTE *CheckFloat(UBYTE *, int *);
01744 void SetfFloatPrecision(LONG);
01745 int EvaluateFun(PHEAD WORD *, WORD, WORD *);
01746 int CoStrictRounding(UBYTE *);

```



```

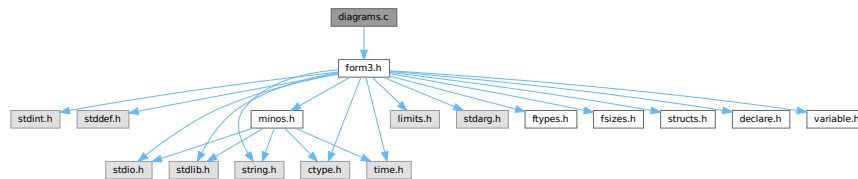
01747 int StrictRounding(PHEAD WORD *, WORD, WORD, WORD);
01748 #endif
01749
01750 /*
01751  #] Declarations :
01752  */
01753 #endif

```

4.21 diagrams.c File Reference

```
#include "form3.h"
```

Include dependency graph for diagrams.c:



Functions

- int [CoCanonicalize](#) (UBYTE *s)
- int [DoCanonicalize](#) (PHEAD WORD *term, WORD *params)
- int [DoTopologyCanonicalize](#) (PHEAD WORD *term, WORD vert, WORD edge, WORD *args)
- int [DoShattering](#) (PHEAD WORD *connect, WORD *environ, WORD *partitions, WORD nvert)

4.21.1 Detailed Description

Contains the wrapper routines for diagram manipulations.

Definition in file [diagrams.c](#).

4.21.2 Function Documentation

4.21.2.1 CoCanonicalize()

```

int CoCanonicalize (
    UBYTE * s )

```

Definition at line 64 of file [diagrams.c](#).

4.21.2.2 DoCanonicalize()

```

int DoCanonicalize (
    PHEAD WORD * term,
    WORD * params )

```

Definition at line 185 of file [diagrams.c](#).

4.21.2.3 DoTopologyCanonicalize()

```
int DoTopologyCanonicalize (
    PHEAD WORD * term,
    WORD vert,
    WORD edge,
    WORD * args )
```

Definition at line 346 of file [diagrams.c](#).

4.21.2.4 DoShattering()

```
int DoShattering (
    PHEAD WORD * connect,
    WORD * environ,
    WORD * partitions,
    WORD nvert )
```

Definition at line 555 of file [diagrams.c](#).

4.22 diagrams.c

[Go to the documentation of this file.](#)

```
00001
00005 /* #[] License : */
00006 /*
00007 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00008 * When using this file you are requested to refer to the publication
00009 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010 * This is considered a matter of courtesy as the development was paid
00011 * for by FOM the Dutch physics granting agency and we would like to
00012 * be able to track its scientific use to convince FOM of its value
00013 * for the community.
00014 *
00015 * This file is part of FORM.
00016 *
00017 * FORM is free software: you can redistribute it and/or modify it under the
00018 * terms of the GNU General Public License as published by the Free Software
00019 * Foundation, either version 3 of the License, or (at your option) any later
00020 * version.
00021 *
00022 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00025 * details.
00026 *
00027 * You should have received a copy of the GNU General Public License along
00028 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00029 */
00030 /* #[] License : */
00031 /*
00032 * #[] Includes : diagrams.c
00033 */
00034 #include "form3.h"
00035
00036 static WORD one = 1;
00037
00038 /*
00039 * #[] Includes :
00040 * #[] CoCanonicalize :
00041
00042 Syntax:
00043 Canonicalize,mainoption,...
00044 With the main options currently:
00045 Canonicalize,topology,vertexfunction,edgefunction,OutDollar,extraoptions;
00046 Canonicalize,polynomial,InDollar,set_or_setname_or_dollar,OutDollar,extraoptions;
00047 The vertex function needs to have the format (assume it is called vx):
00048 vx(p1,p2,-p3) or vx(-p1,p2,p3,-p4) etc.
```

```

00050     The external lines have a vertex with only one line.
00051     All momenta that form connections should be unique.
00052     In principle the - signs are not relevant for the topology, but they
00053     may exist already in the remaining part of the diagram. They are also
00054     part of the canonical form.
00055     The edge function can be used in different ways, depending on the options.
00056     The extraoption(s) should be nonnegative integers or $variables that evaluate
00057     into nonnegative integers (integers less than 2^31).
00058     The Indollar variable contains the polynomial to be canonicalized.
00059     The OutDollar variable should be the name of a $-variable (as in $out) which
00060     will be filled with a replace_ function. The canonicalization can then
00061     be executed in the whole term with the    Multiply $out;    command.
00062  */
00063
00064 int CoCanonicalize(UBYTE *s)
00065 {
00066     WORD args[10], *a, num;
00067     UBYTE *t, c;
00068     args[0] = TYPECANONICALIZE;
00069     a = args+2;
00070     while ( *s == ',' || *s == '\t' || *s == ' ' ) s++;
00071     t = s; while ( FG.cTable[*s] == 0 ) s++;
00072     c = *s; *s = 0;
00073     if ( StrICmp(t, (UBYTE *)("topology")) == 0 ) {
00074         *s = c; *a++ = 0;
00075         while ( *s == ',' || *s == '\t' || *s == ' ' ) s++;
00076         s = GetFunction(s,a);
00077         while ( *s == ',' || *s == '\t' || *s == ' ' ) s++;
00078         s = GetFunction(s,a+1);
00079         if ( *a == 0 || a[1] == 0 ) return(1);
00080         a += 2;
00081     }
00082     else if ( StrICmp(t, (UBYTE *)("polynomial")) == 0 ) {
00083         *s = c; *a++ = 1;
00084         while ( *s == ',' || *s == '\t' || *s == ' ' ) s++;
00085     /*
00086         Now get the name of the input $-variable
00087     */
00088         if ( *s != '$' ) {
00089             MesPrint("&Canonicalize statement needs a $-variable for its input.");
00090             return(1);
00091         }
00092         s++; t = s; while ( FG.cTable[*s] < 2 ) s++;
00093         c = *s; *s = 0;
00094         if ( GetName(AC.dollarnames,t,&num,NOAUTO) == CDOLLAR ) *a++ = num;
00095         else { *a++ = AddDollar(t,DOLINDEX,&one,1); }
00096         *s = c;
00097     /*
00098         Now the set
00099     */
00100         if ( *s == '{' ) {
00101             t = s+1; SKIPBRA2(s)
00102             c = *s; *s = 0;
00103             *a++ = DoTempSet(t,s);
00104             *s++ = c;
00105         }
00106         else if ( FG.cTable[*s] == 0 || *s == '[' ) {
00107             t = s;
00108             if ( ( s = SkipAName(s) ) == 0 ) {
00109                 MesPrint("&Illegal name for set in Canonicalize statement: %s",t);
00110                 return(1);
00111             }
00112             c = *s; *s = 0;
00113             if ( GetName(AC.varnames,t,a,WITHAUTO) == CSET ) {
00114                 if ( Sets[*a].type != CSYMBOL ) {
00115                     MesPrint("&In Canonicalize: %s is not a set of symbols.",t);
00116                     return(1);
00117                 }
00118             }
00119             else {
00120                 MesPrint("&In Canonicalize: %s is not a set.",t);
00121                 return(1);
00122             }
00123             *s = c; a++;
00124         }
00125         else if ( *s == '$' ) {
00126             s++; t = s; while ( FG.cTable[*s] < 2 ) s++;
00127             c = *s; *s = 0;
00128             if ( GetName(AC.dollarnames,t,&num,NOAUTO) == CDOLLAR ) *a++ = -num-2;
00129             else {
00130                 MesPrint("&In Canonicalize: %s is undefined.",t-1);
00131                 return(1);
00132             }
00133             *s = c;
00134         }
00135         else {
00136             MesPrint("&In Canonicalize: Illegal third(=set) argument.");

```

```

00137         return(1);
00138     }
00139 }
00140 else {
00141     MesPrint("&Unrecognized option in Canonicalize statement: %s",t);
00142     return(1);
00143 }
00144 /*
00145 Now get the name of the output $-variable
00146 */
00147 while ( *s == ',' || *s == '\t' || *s == ' ' ) s++;
00148 if ( *s != '$' ) {
00149     MesPrint("&Canonicalize statement needs a $-variable for its output.");
00150     return(1);
00151 }
00152 s++; t = s; while ( FG.cTable[*s] < 2 ) s++;
00153 c = *s; *s = 0;
00154 if ( GetName(AC.dollarnames,t,&num,NOAUTO) == CDOLLAR ) *a++ = num;
00155 else { *a++ = AddDollar(t,DOLINDEX,&one,1); }
00156 *s = c;
00157 /*
00158 Now the options. At the moment we just do one of them.
00159 (the first extra option is relevant to determine the use of the edge function)
00160 In the future we may have to be more flexible.
00161 */
00162 a[0] = 0; /* default value */
00163 while ( *s == ',' || *s == '\t' || *s == ' ' ) s++;
00164 if ( *s != 0 ) {
00165     s = GetNumber(s,a);
00166     if ( *a == -1 ) return(1);
00167     while ( *s == ',' || *s == '\t' || *s == ' ' ) s++;
00168     a++;
00169 }
00170 /*
00171 Now complete the args string and put it in the compiler buffer
00172 */
00173 args[1] = a-args;
00174 AddNtoL(args[1],args);
00175 return(0);
00176 }
00177 /*
00178 #] CoCanonicalize :
00179 #[ DoCanonicalize :
00180
00181 Does the canonicalization. The output term overwrites the input term.
00182 */
00183 int DoCanonicalize(PHEAD WORD *term, WORD *params)
00184 {
00185     WORD args[10];
00186     int i;
00187     /*
00188     First check whether we need to expand dollars;
00189     */
00190     for ( i = 0; i < params[1]; i++ ) args[i] = params[i];
00191     if ( args[2] == 0 ) { /* topology */
00192         for ( i = 3; i < 5; i++ ) {
00193             if ( args[i] < 0 ) { /* This is a dollar */
00194                 args[i] = DolToFunction(BHEAD -args[i]-2);
00195                 if ( args[i] == 0 ) {
00196                     MLOCK(ErrorMessageLock);
00197                     MesPrint("Value of $-variable in Canonicalize statement should be a function.");
00198                     MUNLOCK(ErrorMessageLock);
00199                     Terminate(-1);
00200                 }
00201             }
00202         }
00203     }
00204     for ( i = 6; i < args[1]; i++ ) { /* Extra options */
00205         if ( args[i] < 0 ) { /* This is a dollar */
00206             args[i] = DolToNumber(BHEAD -args[i]-2);
00207             if ( args[i] < 0 ) {
00208                 MLOCK(ErrorMessageLock);
00209                 MesPrint("Value of $-variable in Canonicalize statement should be a nonnegative
00210 number < %l.",(LONG)MAXPOSITIVE);
00211                 MUNLOCK(ErrorMessageLock);
00212                 Terminate(-1);
00213             }
00214         }
00215     }
00216     switch ( args[6] ) {
00217     case 1: { /* pass the vertex and edge functions. */
00218         WORD *tstop, *t, *tedge, *te;
00219         tstop = term + *term; tstop -= ABS(tstop[-1]);
00220         t = term+1;
00221         tedge = AT.WorkPointer; te = tedge+1;
00222         while ( t < tstop ) {

```

```

00223         if ( *t != args[3] && *t != args[4] ) { t += t[1]; continue; }
00224         for ( i = 0; i < t[1]; i++ ) te[i] = t[i];
00225         te += t[1]; t += t[1];
00226     }
00227     *te++ = 1; *te++ = 1; *te++ = 3;
00228     tedge[0] = te - tedge;
00229     AT.WorkPointer = te;
00230 /*
00231     DoVertexCanonicalize(BHEAD term, tedge, args[3], args[4], args[5], args[6]);
00232 */
00233     AT.WorkPointer = tedge;
00234 } break;
00235 case 2: { /* pass the edge functions only */
00236     WORD *tstop, *t, *tedge, *te;
00237     tstop = term + *term; tstop -= ABS(tstop[-1]);
00238     t = term + 1;
00239     tedge = AT.WorkPointer; te = tedge + 1;
00240     while ( t < tstop ) {
00241         if ( *t != args[4] ) { t += t[1]; continue; }
00242         for ( i = 0; i < t[1]; i++ ) te[i] = t[i];
00243         te += t[1]; t += t[1];
00244     }
00245     *te++ = 1; *te++ = 1; *te++ = 3;
00246     tedge[0] = te - tedge;
00247     AT.WorkPointer = te;
00248 /*
00249     DoEdgeCanonicalize(BHEAD term, tedge, args[5]);
00250 */
00251     AT.WorkPointer = tedge;
00252 } break;
00253 default: {
00254     DoTopologyCanonicalize(BHEAD term, args[3], args[4], args+5);
00255 } break;
00256 }
00257 /*
00258     Call here the topology canonicalization
00259     We will have the arguments:
00260     args[3]: The function used as vertex function
00261     args[4]: The function used as edge function
00262     args[5]: The number of the dollar to be used for the output
00263     args[6]: Potentially other options (like saying how to use args[4]).
00264     term: The term in which the topology resides.
00265 */
00266 }
00267 else if ( args[2] == 1 ) { /* polynomial */
00268     WORD *symlist, *nsymlist;
00269     for ( i = 6; i < args[1]; i++ ) { /* Extra options */
00270         if ( args[i] < 0 ) { /* This is a dollar */
00271             args[i] = DolToNumber(BHEAD -args[i]-2);
00272             if ( args[i] < 0 ) {
00273                 MLOCK(ErrorMessageLock);
00274                 MesPrint("Value of $-variable in Canonicalize statement should be a nonnegative
00275 number < %1.", (LONG)MAXPOSITIVE);
00276                 MUNLOCK(ErrorMessageLock);
00277                 Terminate(-1);
00278             }
00279         }
00280     }
00281 /*
00282     Now we sort out the set. We create a pointer to the list of set
00283     elements, and we determine the number of elements in the set.
00284 */
00285     symlist = AT.WorkPointer;
00286     if ( args[4] < -1 ) { /* Dollar that should expand into a list of symbols */
00287         DOLLARS d = Dollars - args[4] - 2;
00288         WORD *ds, *insym;
00289         if ( d->type != DOLWILDARGS ) {
00290 NoWildArg:
00291             MLOCK(ErrorMessageLock);
00292             MesPrint("Value of $-variable in Canonicalize statement should be a argument wildcard
00293 of symbol arguments.");
00294             MUNLOCK(ErrorMessageLock);
00295             Terminate(-1);
00296         }
00297         insym = symlist; ds = d->where+1;
00298         while ( *ds ) {
00299             if ( *ds != -SYMBOL ) goto NoWildArg;
00300             *insym++ = ds[1];
00301             ds += 2;
00302         }
00303         nsymlist = insym - symlist;
00304     } else { /*if ( args[4] >= 0 ) */
00305         WORD *ss, *sy, n;
00306         ss = (WORD *) (AC.SetElementList.lijst) + Sets[args[4]].first;
00307         nsymlist = n = Sets[args[4]].last - Sets[args[4]].first;

```

```

00308         sy = symlist = AT.WorkPointer;
00309         NCOPY(sy,ss,n);
00310     }
00311     AT.WorkPointer = symlist+nsymlist;
00312 /*
00313     Call here the polynomial canonicalization
00314     We will have the arguments:
00315     args[3]: The number of the dollar to be used for the input
00316     symlist: an array of symbols
00317     nsymlist: the number of symbols in symlist
00318     args[5]: The number of the dollar to be used for the output
00319     args[6]: Potentially other options.
00320 */
00321 /*
00322     DoPolynomialCanonicalize(BHEAD args[3],symlist,nsymlist,args[5],args[6]);
00323 */
00324     AT.WorkPointer = symlist;
00325 }
00326 return(0);
00327 }
00328
00329 /*
00330 #] DoCanonicalize :
00331 #[ DoTopologyCanonicalize :
00332
00333     term: The term
00334     vert: the vertex function
00335     edge: the edge function
00336     args[0]: the number of the output dollar
00337     args[1]: options
00338     return value should be zero if all is correct.
00339
00340     The external lines connect to an 'external vertex' which has only one line.
00341     The vertices are of a type: vertex_(p1,p2,-p3)*vertex_(p3,p4,p5) etc.
00342     The edges indicate noninteger powers of the lines:
00343     edge_(p1,2) means 1/p1.pl^(2*ep)
00344 */
00345
00346 int DoTopologyCanonicalize(PHEAD WORD *term,WORD vert,WORD edge,WORD *args)
00347 {
00348     int nvert = 0, nvert2, i, ii, jj, flipnames = 0, nparts/*, level, num */;
00349     WORD *tstop, *t, *tt, *tend, *td;
00350     WORD *oldworkpointer = AT.WorkPointer;
00351     WORD *termcopy = TermMalloc("TopologyCanonize1");
00352     WORD *vet= TermMalloc("TopologyCanonize2");
00353     WORD *partition, *environ, *connect, *pparts, *p;
00354 /*
00355     WORD *pparts;
00356 */
00357     WORD momenta[150],flips[50],nmomenta = 0, nflips = 0;
00358 /*
00359     Step one: the vertices should get a number. We copy the term for this.
00360     We need a high number for the vertex function to make sure
00361     that it comes after the edge function in the sorting.
00362 */
00363     if ( args[0] < args[1] ) { flipnames = 1; }
00364     tend = term + *term; tend -= ABS(tend[-1]); t = term+1; tt = termcopy+1;
00365     while ( t < tend ) {
00366         if ( *t == vert ) {
00367             for ( i = FUNHEAD; i < t[1]; i += 2 ) {
00368                 if ( t[i] == -VECTOR || ( t[i] == -INDEX && t[i+1] < 0 ) ) {
00369                     momenta[nmomenta++] = -VECTOR;
00370                     momenta[nmomenta++] = t[i+1];
00371                 }
00372                 else if ( t[i] == -MINVECTOR ) {
00373                     momenta[nmomenta++] = -MINVECTOR;
00374                     momenta[nmomenta++] = t[i+1];
00375                 }
00376                 else goto notgoodvertex;
00377                 momenta[nmomenta++] = nvert;
00378             }
00379             ii = FUNHEAD; i = t[1]-FUNHEAD;
00380             NCOPY(tt,t,ii)
00381             if ( flipnames ) tt[-FUNHEAD] = edge;
00382             tt[-FUNHEAD+1] += 2;
00383             *tt++ = -CNUMBER; *tt++ = nvert++;
00384         }
00385         else if ( *t == edge && flipnames ) {
00386             i = t[1] - 1; *tt++ = vert; t++;
00387         }
00388         else {
00389             notgoodvertex:
00390                 i = t[1];
00391             }
00392             NCOPY(tt,t,i)
00393         }
00394         while ( t < tend ) *tt++ = *t++;

```

```

00395     termcopy[0] = tt - termcopy;
00396     if ( flipnames ) EXCH(edge,vert)
00397     nvert2 = nvert*nvert;
00398 /*
00399     Sort the momenta. Keep the sign order.
00400 */
00401     for ( i = 0; i < nmomenta-3; i+=3 ) {
00402         jj = i;
00403         while ( jj >= 0 && momenta[jj+4] > momenta[jj+1] ) {
00404             EXCH(momenta[jj+5],momenta[jj+2])
00405             EXCH(momenta[jj+4],momenta[jj+1])
00406             EXCH(momenta[jj+3],momenta[jj])
00407             jj -= 3;
00408         }
00409     }
00410 /*
00411     Step two: make now the edge functions in the proper notation.
00412 */
00413     t = vet+1;
00414     for ( i = 0; i < nmomenta; i += 6 ) {
00415         if ( momenta[i] == -VECTOR && momenta[i+3] == -MINVECTOR
00416             && momenta[i+1] == momenta[i+4] ) {
00417             }
00418         else if ( momenta[i] == -MINVECTOR && momenta[i+3] == -VECTOR
00419             && momenta[i+1] == momenta[i+4] ) {
00420             flips[nflips++] = momenta[i+1];
00421             DUMMYUSE(flips[nflips-1]);
00422         }
00423         else { /* something wrong with the momenta */
00424             MLOCK(ErrorMessageLock);
00425             MesPrint("No momentum conservation or wrong momenta in Canonicalize statement");
00426             MUNLOCK(ErrorMessageLock);
00427             Terminate(-1);
00428         }
00429         *t++ = EDGE; *t++ = FUNHEAD+10; FILLFUN(t)
00430         *t++ = -SNUMBER; *t++ = momenta[i+2];
00431         *t++ = -SNUMBER; *t++ = momenta[i+5];
00432         *t++ = -VECTOR; *t++ = momenta[i+1];
00433         *t++ = -SNUMBER; *t++ = 0; /* provisional power/color, multiple of ep */
00434         *t++ = -SNUMBER; *t++ = 0; /* provisional power/color, integer */
00435     }
00436     tend = t;
00437     *t++ = 1; *t++ = 1; *t++ = 3; vet[0] = t-vet; *t = 0;
00438 /*
00439     Now the powers of the denominators
00440 */
00441     tstop = termcopy+termcopy; tstop -= ABS(tstop[-1]); td = termcopy+1;
00442     while ( td < tstop ) {
00443         if ( *td == edge && td[1] == FUNHEAD+4 ) {
00444             if ( td[FUNHEAD+2] == -SNUMBER && ( td[FUNHEAD] == -VECTOR || td[FUNHEAD] == -INDEX
00445                 || td[FUNHEAD] == -MINVECTOR ) ) {}
00446             else {
00447                 MLOCK(ErrorMessageLock);
00448                 MesPrint("Illegal argument in edge function in Canonicalize statement");
00449                 MUNLOCK(ErrorMessageLock);
00450                 Terminate(-1);
00451             }
00452             tt = vet+1;
00453             while ( tt < tend ) {
00454                 if ( tt[FUNHEAD+5] == td[FUNHEAD+1] ) { tt[FUNHEAD+7] = td[FUNHEAD+3]; break; }
00455                 tt += tt[1];
00456             }
00457         }
00458         else if ( *td == DOTPRODUCT ) break;
00459         td += td[1];
00460     }
00461     if ( td < tstop ) {
00462         tt = vet+1;
00463         while ( tt < tend ) {
00464             /*
00465                 tt[FUNHEAD+5] is a vector. We look for dotproducts with twice tt[FUNHEAD+5]
00466             */
00467             for ( i = 2; i < td[1]; i += 3 ) {
00468                 if ( td[i] == tt[FUNHEAD+5] && td[i+1] == tt[FUNHEAD+5] ) {
00469                     tt[FUNHEAD+9] = td[i+2];
00470                     break;
00471                 }
00472             }
00473             tt += tt[1];
00474         }
00475     }
00476     Normalize(BHEAD vet);
00477 /*
00478     Now we have a term `vet` in the proper notation and we can start.
00479     To keep track of the shattering we use an array of 2*nvert.
00480     each entry is Number,marker
00481     When the marker is zero, the vertices are in the same partition.

```

```

00482     For the environment we need a matrix that is nvert x nvert
00483     At the same time we keep the connectivity matrix, because that will
00484     save much time later.
00485     The partitions are stored in a matrix as well. This allows them to be
00486     treated as a stack. The entries are separated by 0 if they belong to
00487     the same part, and by a 1 when they belong to different parts.
00488 */
00489     partition = AT.WorkPointer; AT.WorkPointer += 2*nvert2;
00490     for ( i = 0; i < nvert; i++ ) { partition[2*i] = i; partition[2*i+1] = 0; }
00491     partition[2*i-1] = -1; /* end of the first part which is currently all vertices */
00492     nparts = 1;
00493
00494     connect = AT.WorkPointer; AT.WorkPointer += nvert2;
00495     for ( i = 0; i < nvert2; i++ ) connect[i] = 0;
00496     tstop = vet+vet; tstop -= ABS(tstop[-1]); t = vet+1;
00497     while ( t < tstop ) {
00498         if ( *t == EDGE ) {
00499             connect[t[FUNHEAD+1]*nvert+t[FUNHEAD+3]]++;
00500             connect[t[FUNHEAD+3]*nvert+t[FUNHEAD+1]]++;
00501         }
00502         t += t[1];
00503     }
00504     for ( i = 0; i < nvert; i++ ) {
00505         MesPrint("connectivity: %d -- %a",i,nvert,connect+i*nvert);
00506     }
00507     /*
00508     Create the environment matrix and sort it.
00509 */
00510     environ = AT.WorkPointer; AT.WorkPointer += nvert2;
00511     /*
00512     And now the refinement process starts.
00513 */
00514     WantAddPointers(nvert+1);
00515     for ( i = 0; i < nvert2; i++ ) environ[i] = 0;
00516     /* level = 0; */
00517     pparts = partition;
00518     while ( nparts < nvert ) {
00519         nparts = DoShattering(BHEAD connect,environ,pparts,nvert);
00520         if ( nparts < nvert ) { /* raise level and make a copy and split a part */
00521             p = pparts + 2*nvert;
00522             /* level++; */
00523             for ( i = 0; i < 2*nvert; i++ ) p[i] = pparts[i];
00524             for ( ii = 0; ii < 2*nvert; ii += 2 ) {
00525                 if ( p[ii+1] == 0 ) { /* found a part with more than one */
00526                     /* num = 2; */ i = ii+2;
00527                     while ( p[i+1] == 0 ) { /* num++; */ i += 2; }
00528                     p[ii+1] = -1; pparts = p;
00529                     break;
00530                 }
00531             }
00532         }
00533     }
00534     MesPrint("partition: %d -- %a",nparts,2*nvert,pparts);
00535 }
00536 /*
00537 Just for now
00538 */
00539 PutTermInDollar(vet,args[0]);
00540
00541 TermFree(vet,"TopologyCanonize2");
00542 TermFree(termcopy,"TopologyCanonize1");
00543 AT.WorkPointer = oldworkpointer;
00544 return(0);
00545 }
00546
00547 /*
00548 #] DoTopologyCanonicalize :
00549 #[ DoShattering :
00550 */
00551 int DoShattering(PHEAD WORD *connect, WORD *environ, WORD *partitions, WORD nvert)
00552 {
00553     int nparts, i, j, ii, jj, iii, jjj, newmarker;
00554     WORD *p = AT.pWorkSpace + AT.pWorkPointer, *part, *endpart;
00555     WORD *poin1, *poin2;
00556     #ifdef SHATBUG
00557     MesPrint("Entering DoShattering. partitions = %a",2*nvert,partitions);
00558     #endif
00559     restart:
00560     /*
00561     Determine the number of parts
00562     p will be an array with pointers to the parts.
00563     We made space for this array in the calling routine and because this
00564     routine is not calling any other routines we do not need to raise

```



```

00569     the pointer in this stack (AT.pWorkPointer).
00570 */
00571     nparts = 0; newmarker = 0;
00572     part = partitions; endpart = part + 2*nvert;
00573     p[0] = part;
00574     while ( part < endpart ) {
00575         if ( part[1] != 0 ) { p[++nparts] = part+2; }
00576         part += 2;
00577     }
00578     for ( i = 0; i < nparts; i++ )
00579         AT.WorkPointer[i] = (p[i+1]-p[i])/2;
00580 #ifdef SHATBUG
00581 MesPrint("DoShattering: calculated the pointers");
00582 MesPrint("DoShattering: sizes: %a", nparts, AT.WorkPointer);
00583 MesPrint("DoShattering: p[0]: %a, p[1]: %a", 6, p[0], 6, p[1]);
00584 #endif
00585     for ( i = 0; i < nparts; i++ ) {
00586         if ( AT.WorkPointer[i] > 1 ) {
00587             for ( j = 0; j < nparts; j++ ) {
00588 /*
00589             Shatter part i wrt part j.
00590             if there is action, go to restart.
00591 */
00592                 for ( ii = 0; ii < AT.WorkPointer[i]; ii++ ) {
00593                     for ( jj = 0; jj < AT.WorkPointer[j]; jj++ ) {
00594                         environ[ii*AT.WorkPointer[j]+jj] += connect[p[i][2*ii]*nvert+p[j][2*jj]];
00595                     }
00596                 }
00597 #ifdef SHATBUG
00598                 for ( ii = 0; ii < AT.WorkPointer[i]; ii++ ) {
00599                     MesPrint("Environ(%d,%d): %a", i, j, AT.WorkPointer[j], environ+ii*AT.WorkPointer[j]);
00600                 }
00601 #endif
00602 /*
00603                 Sort the rows internally, then sort the rows wrt each other
00604                 and finally place new markers. If a new marker, we restart.
00605                 Don't forget to clean up the environ array.
00606 */
00607                 for ( ii = 0; ii < nvert; ii++ ) {
00608                     poin1 = environ+ii*AT.WorkPointer[j];
00609                     for ( jj = 0; jj < AT.WorkPointer[j]-1; jj++ ) {
00610                         jjj = jj;
00611                         while ( jjj >= 0 && poin1[jjj+1] > poin1[jjj] ) {
00612                             EXCH(poin1[jjj+1], poin1[jjj]);
00613                             jjj--;
00614                         }
00615                     }
00616                 }
00617 #ifdef SHATBUG
00618                 for ( ii = 0; ii < AT.WorkPointer[i]; ii++ ) {
00619                     MesPrint("environ(%d,%d): %a", i, j, AT.WorkPointer[j], environ+ii*AT.WorkPointer[j]);
00620                 }
00621 #endif
00622                 for ( ii = 0; ii < AT.WorkPointer[i]-1; ii++ ) {
00623                     poin2 = environ+ii*AT.WorkPointer[j]; poin1 = poin2+AT.WorkPointer[j];
00624                     iii = ii;
00625                     while ( iii >= 0 && ( CmpArray(poin1, poin2, AT.WorkPointer[j]) < 0 ) ) {
00626                         EXCHN(poin2, poin1, AT.WorkPointer[j]);
00627                         EXCH(p[i][2*iii+2], p[i][2*iii]);
00628                         iii--; poin1 = poin2; poin2 = poin1-AT.WorkPointer[j];
00629                     }
00630                 }
00631 #ifdef SHATBUG
00632                 for ( ii = 0; ii < AT.WorkPointer[i]; ii++ ) {
00633                     MesPrint("environ(%d,%d): %a", i, j, AT.WorkPointer[j], environ+ii*AT.WorkPointer[j]);
00634                 }
00635                 MesPrint("partitions = %a", 2*nvert, partitions);
00636 #endif
00637                 for ( ii = 0; ii < AT.WorkPointer[i]-1; ii++ ) {
00638                     poin2 = environ+ii*AT.WorkPointer[j]; poin1 = poin2+AT.WorkPointer[j];
00639                     if ( CmpArray(poin1, poin2, AT.WorkPointer[j]) == 0 ) continue;
00640                     p[i][2*ii+1] = -1; nparts++; newmarker++;
00641                 }
00642 #ifdef SHATBUG
00643                 MesPrint("partitions = %a", 2*nvert, partitions);
00644 #endif
00645 /*
00646                 Clear environ. This is probably faster than just clearing the whole array.
00647                 Maybe in the future a test could be done on nvert to decide how to clear.
00648 */
00649                 for ( ii = 0; ii < AT.WorkPointer[i]; ii++ ) {
00650                     for ( jj = 0; jj < AT.WorkPointer[j]; jj++ ) {
00651                         environ[ii*AT.WorkPointer[j]+jj] = 0;
00652                     }
00653                 }
00654                 if ( newmarker ) { goto restart; }
00655             }

```

```

00656     }
00657     }
00658     return(nparts);
00659 }
00660
00661 /*
00662  #] DoShattering :
00663  */

```

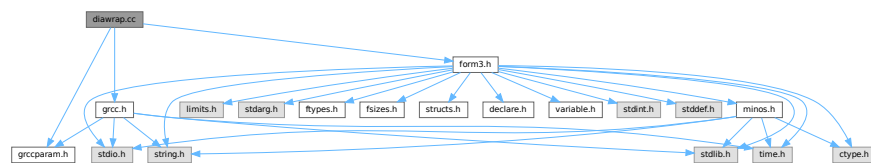
4.23 diawrap.cc File Reference

```

#include "form3.h"
#include "grccparam.h"
#include "grcc.h"

```

Include dependency graph for diawrap.cc:



Data Structures

- struct [ToPoTyPe](#)

Macros

- #define [MAXPOINTS](#) 120
- #define [WITHEARLYVETO](#)

Typedefs

- typedef struct [ToPoTyPe](#) [TOPOTYPE](#)

Functions

- int [LoadModel](#) ([MODEL](#) *m)
- int [ConvertParticle](#) ([Model](#) *model, int formnum)
- int [ReConvertParticle](#) ([Model](#) *model, int grccnum)
- int [numParticle](#) ([MODEL](#) *m, [WORD](#) n)
- Bool [fendMG](#) ([EGraph](#) *eg, void *ti)
- Bool [ProcessTopology](#) ([EGraph](#) *eg, void *ti)
- void [SetDualOpts](#) (int *opt, const [WORD](#) num, const int key, const char *key_name, const int dual, const char *dual_name, const int val, const int dval)
- int [GenDiagrams](#) ([PHEAD](#) [WORD](#) *term, [WORD](#) level)

4.23.1 Detailed Description

Functions with interface FORM with gcc, to implement the `diagrams_` function.

Definition in file [diawrap.cc](#).

4.23.2 Macro Definition Documentation

4.23.2.1 MAXPOINTS

```
#define MAXPOINTS 120
```

Definition at line 40 of file [diawrap.cc](#).

4.23.3 Function Documentation

4.23.3.1 LoadModel()

```
int LoadModel (  
    MODEL * m )
```

Definition at line 57 of file [diawrap.cc](#).

4.23.3.2 ConvertParticle()

```
int ConvertParticle (  
    Model * model,  
    int formnum )
```

Definition at line 204 of file [diawrap.cc](#).

4.23.3.3 ReConvertParticle()

```
int ReConvertParticle (  
    Model * model,  
    int gccnum )
```

Definition at line 222 of file [diawrap.cc](#).

4.23.3.4 numParticle()

```
int numParticle (  
    MODEL * m,  
    WORD n )
```

Definition at line 234 of file [diawrap.cc](#).

4.23.3.5 fendMG()

```
Bool fendMG (
    EGraph * eg,
    void * ti )
```

Definition at line 505 of file [diawrap.cc](#).

4.23.3.6 ProcessTopology()

```
Bool ProcessTopology (
    EGraph * eg,
    void * ti )
```

Definition at line 515 of file [diawrap.cc](#).

4.23.3.7 SetDualOpts()

```
void SetDualOpts (
    int * opt,
    const WORD num,
    const int key,
    const char * key_name,
    const int dual,
    const char * dual_name,
    const int val,
    const int dval )
```

Definition at line 778 of file [diawrap.cc](#).

4.23.3.8 GenDiagrams()

```
int GenDiagrams (
    PHEAD WORD * term,
    WORD level )
```

Definition at line 805 of file [diawrap.cc](#).

4.24 diawrap.cc

[Go to the documentation of this file.](#)

```
00001
00005 /* # [ License : */
00006 /*
00007 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00008 * When using this file you are requested to refer to the publication
00009 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010 * This is considered a matter of courtesy as the development was paid
00011 * for by FOM the Dutch physics granting agency and we would like to
00012 * be able to track its scientific use to convince FOM of its value
00013 * for the community.
00014 *
00015 * This file is part of FORM.
00016 *
00017 * FORM is free software: you can redistribute it and/or modify it under the
```

```

00018 *   terms of the GNU General Public License as published by the Free Software
00019 *   Foundation, either version 3 of the License, or (at your option) any later
00020 *   version.
00021 *
00022 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00025 *   details.
00026 *
00027 *   You should have received a copy of the GNU General Public License along
00028 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00029 */
00030 /* #] License : */
00031 // #[ Includes : diawrap.cc
00032
00033 extern "C" {
00034 #include "form3.h"
00035 }
00036
00037 #include "grccparam.h"
00038 #include "grcc.h"
00039
00040 #define MAXPOINTS 120
00041
00042 typedef struct ToPoTyPe {
00043     WORD *vert;
00044     WORD *vertmax;
00045     Options *opt;
00046     int cldeg[MAXPOINTS], clnum[MAXPOINTS], cleft[MAXPOINTS];
00047     int cmind[MAXLEGS+1], cmaxd[MAXLEGS+1];
00048     int ncl, nvert;
00049 } TOPOTYPE;
00050
00051 static void ProcessDiagram(EGraph *eg, void *ti);
00052 static int processVertex(TOPOTYPE *TopoInf, int pointsremaining, int level);
00053
00054 // #] Includes :
00055 // #[ LoadModel :
00056
00057 int LoadModel(MODEL *m)
00058 {
00059 //
00060 // First check whether there was a model already.
00061 // In the Kaneko setup there can be only one at the same time.
00062 // Hence if there was one we need to remove it first.
00063 // Unless of course, it was already the model we want.
00064 //
00065     if ( m->grccmodel != NULL ) return(0);
00066
00067     int i,j,k;
00068 //
00069 // First the information that goes into the Model struct.
00070 // Note that new Model takes over (does not copy) minp.cnamlist.
00071 //
00072     MInput minp;
00073     PInput pinp;
00074     IInput iinp;
00075     if ( m->ncouplings > GRCC_MAXNCPLG ) {
00076         MesPrint("Too many coupling constants in model. Current limit is %d.", (WORD)GRCC_MAXNCPLG);
00077         MesPrint("Suggestion: recompile Form with a larger value for GRCC_MAXNCPLG");
00078         return(-1);
00079     }
00080     minp.defpart = GRCC_DEFBYCODE;
00081     minp.name = (char *) (m->name);
00082     minp.ncouple = m->ncouplings;
00083     for ( i = 0; i < GRCC_MAXNCPLG; i++ ) minp.cnamlist[i] = NULL;
00084     for ( i = 0; i < minp.ncouple; i++ )
00085         minp.cnamlist[i] = (char *) (VARNAME(symbols,m->couplings[i]));
00086     Model *mdl = new Model(&minp);
00087 //
00088 // Now the particles
00089 //
00090     for ( i = 0; i < m->nparticles; i++ ) {
00091         if ( minp.defpart == GRCC_DEFBYCODE ) {
00092             pinp.name = NULL;
00093             pinp.aname = NULL;
00094             pinp.pcode = m->vertices[i]->particles[0].number;
00095             pinp.acode = m->vertices[i]->particles[1].number;
00096             switch ( m->vertices[i]->particles[0].spin ) {
00097                 case 1:
00098                     pinp.ptypec = GRCC_PT_Scalar;
00099                     break;
00100                 case -1:
00101                     pinp.ptypec = GRCC_PT_Ghost;
00102                     break;
00103                 case 2:
00104                     if ( m->vertices[i]->particles[0].type == 0 )

```

```

00105         pinp.ptypec = GRCC_PT_Majorana;
00106     else
00107         pinp.ptypec = GRCC_PT_Undef;
00108     break;
00109     case -2:
00110         if ( m->vertices[i]->particles[0].type == 0 )
00111             pinp.ptypec = GRCC_PT_Majorana;
00112         else
00113             pinp.ptypec = GRCC_PT_Dirac;
00114     break;
00115     case 3:
00116         pinp.ptypec = GRCC_PT_Vector;
00117     break;
00118     default:
00119         pinp.ptypec = GRCC_PT_Undef;
00120     break;
00121 }
00122 }
00123 else {
00124     pinp.name = (char *) (VARNAME(functions,m->vertices[i]->particles[0].number));
00125     pinp.aname = (char *) (VARNAME(functions,m->vertices[i]->particles[1].number));
00126     switch ( m->vertices[i]->particles[0].spin ) {
00127         case 1:
00128             pinp.ptypen = "scalar";
00129             break;
00130         case -1:
00131             pinp.ptypen = "ghost";
00132             break;
00133         case 2:
00134             if ( m->vertices[i]->particles[0].type == 0 )
00135                 pinp.ptypen = "majorana";
00136             else
00137                 pinp.ptypen = "undef";
00138             break;
00139         case -2:
00140             if ( m->vertices[i]->particles[0].type == 0 )
00141                 pinp.ptypen = "majorana";
00142             else
00143                 pinp.ptypen = "dirac";
00144             break;
00145         case 3:
00146             pinp.ptypen = "vector";
00147             break;
00148         default:
00149             pinp.ptypen = "undef";
00150             break;
00151     }
00152 }
00153 pinp.extonly = m->vertices[i]->externonly;
00154 mdl->addParticle(&pinp);
00155 }
00156 mdl->addParticleEnd();
00157 //
00158 // Now the vertices
00159 //
00160 for ( i = m->nparticles; i < m->invertices; i++ ) {
00161     VERTEX *v = m->vertices[i];
00162     if ( minp.defpart == GRCC_DEFBYCODE ) {
00163         iinp.icode = NODEFUNCTION+i;
00164         iinp.name = NULL;
00165     }
00166     else {
00167         iinp.name = "node_";
00168     }
00169     iinp.nplistn = v->nparticles;
00170     for ( j = 0; j < iinp.nplistn; j++ ) {
00171         if ( minp.defpart == GRCC_DEFBYCODE ) {
00172             iinp.plistc[j] = v->particles[j].number;
00173         }
00174         else {
00175             iinp.plistn[j] = (char *) (VARNAME(functions,v->particles[j].number));
00176         }
00177     }
00178 //
00179 // We need a properly ordered list of coupling constants
00180 // The ordered list is in m->couplings.
00181 // For each vertex they are in v->couplings.
00182 //
00183 // iinp.cvallist = (int *)Mallocl(m->ncouplings*sizeof(int),"couplings");
00184
00185     for ( j = 0; j < m->ncouplings; j++ ) {
00186         iinp.cvallist[j] = 0;
00187         for ( k = 0; k < v->ncouplings; k += 2 ) {
00188             if ( v->couplings[k] == m->couplings[j] ) {
00189                 iinp.cvallist[j] = v->couplings[k+1];
00190                 break;
00191             }

```

```

00192     }
00193     }
00194     mdl->addInteraction(&iinp);
00195 }
00196 mdl->addInteractionEnd();
00197 m->grccmodel = (void *)mdl;
00198 return(0);
00199 }
00200
00201 // #] LoadModel :
00202 // #[ ConvertParticle :
00203
00204 int ConvertParticle(Model *model,int formnum)
00205 {
00206 //
00207 // Returns the grcc number of the particle, because grcc does not convert
00208 //
00209     int i;
00210     for ( i = 0; i < model->nParticles; i++ ) {
00211         if ( model->particles[i]->pcode == formnum ) { return(i); }
00212         else if ( model->particles[i]->acode == formnum ) { return(-i); }
00213     }
00214     MesPrint("Particle %d not found in model %s",formnum,model->name);
00215     Terminate(-1);
00216     return(0);
00217 }
00218
00219 // #] ConvertParticle :
00220 // #[ ReConvertParticle :
00221
00222 int ReConvertParticle(Model *model,int grccnum)
00223 {
00224 //
00225 // Returns the grcc number of the particle, because grcc does not convert
00226 //
00227     if ( grccnum < 0 ) { return(model->particles[-grccnum]->acode); }
00228     else { return(model->particles[grccnum]->pcode); }
00229 }
00230
00231 // #] ReConvertParticle :
00232 // #[ numParticle :
00233
00234 int numParticle(MODEL *m,WORD n)
00235 {
00236     int i;
00237     for ( i = 0; i < m->nparticles; i++ ) {
00238         if ( m->vertices[i]->particles[0].number == n ) return(i);
00239         if ( m->vertices[i]->particles[1].number == n ) return(i);
00240     }
00241     MesPrint("numParticle: particle %d not found in model",n);
00242     Terminate(-1);
00243     return(-1);
00244 }
00245
00246 // #] numParticle :
00247 // #[ ProcessDiagram :
00248
00249 void ProcessDiagram(EGraph *eg, void *ti)
00250 {
00251 //
00252 // This is the routine that gets a complete diagram and passes it on
00253 // to Form (Generator) for further algebraic manipulations.
00254 // The term is picked up from AT.diaterm and a new term is constructed
00255 // in the workspace.
00256 //
00257     GETIDENTITY
00258     TERMINFO *info = (TERMINFO *)ti;
00259
00260     if ( ( info->flags & TOPOLOGIESONLY ) == TOPOLOGIESONLY ) return;
00261
00262     WORD *term = info->term, *newterm, *oldworkpointer = AT.WorkPointer;
00263     WORD *tdia = term + info->diaoffset;
00264     WORD *tail = tdia + tdia[1];
00265     WORD *tend = term + *term;
00266     WORD *fill, *startfill, *cfill, *afill;
00267     int i, j, intr;
00268     Model *model = (Model *)info->currentModel;
00269     MODEL *m = (MODEL *)info->currentMODEL;
00270     int numlegs, vect, edge, maxmom = 0;
00271
00272     newterm = term + *term;
00273     for ( i = 1; i < info->diaoffset; i++ ) newterm[i] = term[i];
00274     fill = newterm + info->diaoffset;
00275 //
00276 // Now get the nodes
00277 //
00278     if ( ( info->flags & WITHOUTNODES ) == 0 ) {

```

```

00279     for ( i = 0; i < eg->nNodes; i++ ) {
00280 //
00281 //         node_(number,coupling,particle_1(momentum_1),...,particle_n(momentum_n))
00282 //
00283         numlegs = eg->nNodes[i]->deg;
00284         startfill = fill;
00285         *fill++ = NODEFUNCTION;
00286         *fill++ = 0;
00287         FILLFUN(fill)
00288         *fill++ = -SNUMBER; *fill++ = i+1;
00289 //
00290 //     Now we put the coupling constants. This is done inside the
00291 //     function for when we want to work with counterterms.
00292 //
00293     if ( !eg->isExternal(i) ) {
00294         afill = fill; *fill++ = 0; *fill++ = 1; FILLARG(fill)
00295         cfill = fill; *fill++ = 0;
00296         intr = eg->nNodes[i]->intrct;
00297         for ( j = 0; j < model->interacts[intr]->nclist; j++ ) {
00298             if ( model->interacts[intr]->clist[j] != 0 ) {
00299                 *fill++ = SYMBOL; *fill++ = 4;
00300                 *fill++ = m->couplings[j];
00301                 *fill++ = model->interacts[intr]->clist[j];
00302             }
00303         }
00304         *fill++ = 1; *fill++ = 1; *fill++ = 3;
00305         *cfill = fill - cfill;
00306         *afill = fill - afill;
00307     }
00308     else {
00309         *fill++ = -SNUMBER;
00310         *fill++ = 1;
00311     }
00312 //
00313 //     Now the particles and their momenta.
00314 //
00315     for ( j = 0; j < numlegs; j++ ) {
00316         *fill++ = ARGHEAD+FUNHEAD+6;
00317         *fill++ = 0;
00318         FILLARG(fill)
00319         edge = eg->nNodes[i]->edges[j];
00320         vect = ABS(edge)-1;
00321         *fill++ = 6+FUNHEAD;
00322         int a;
00323         if ( edge < 0 ) { a = ReConvertParticle(model,-eg->edges[vect]->ptcl); }
00324         else { a = ReConvertParticle(model,eg->edges[vect]->ptcl); }
00325         *fill++ = a;
00326         *fill++ = FUNHEAD+2;
00327         FILLFUN(fill)
00328         *fill++ = edge < 0 ? -MINVECTOR: -VECTOR;
00329         if ( numlegs == 1 || vect < info->numextern ) { // Look up in set of external momenta
00330             *fill++ = SetElements[Sets[info->externalset].first+vect];
00331         }
00332         else { // Look up in set of internal momenta set
00333             *fill++ = SetElements[Sets[info->internalset].first+(vect-eg->nExtern)];
00334             // determine the number of momenta required from internalset:
00335             maxmom = MaX(maxmom, vect-eg->nExtern);
00336         }
00337         *fill++ = 1; *fill++ = 1; *fill++ = 3;
00338     }
00339     startfill[1] = fill-startfill;
00340 }
00341 }
00342 if ( ( info->flags & WITHEDGES ) == WITHEDGES ) {
00343     for ( i = 0; i < eg->nEdges; i++ ) {
00344         int n1 = eg->edges[i]->nNodes[0];
00345         int n2 = eg->edges[i]->nNodes[1];
00346 //         int l1 = eg->edges[i]->nlegs[0];
00347 //         int l2 = eg->edges[i]->nlegs[1];
00348
00349         startfill = fill;
00350         *fill++ = EDGE;
00351         *fill++ = 0;
00352         FILLFUN(fill)
00353         *fill++ = -SNUMBER; *fill++ = i+1; // number of the edge
00354 //
00355         *fill++ = ARGHEAD+FUNHEAD+6;
00356         *fill++ = 0;
00357         FILLARG(fill)
00358         *fill++ = 6+FUNHEAD;
00359         int a = ReConvertParticle(model,eg->edges[i]->ptcl);
00360         *fill++ = a;
00361         *fill++ = FUNHEAD+2;
00362         FILLFUN(fill)
00363         *fill++ = -VECTOR;
00364         // Look up in set of internal momenta set
00365         if ( i < info->numextern ) { // Look up in set of external momenta

```



```

00366         *fill++ = SetElements[Sets[info->externalset].first+i];
00367     }
00368     else { // Look up in set of internal momenta set
00369         *fill++ = SetElements[Sets[info->internalset].first+(i-eg->nExtern)];
00370         maxmom = MaX(maxmom, i-eg->nExtern);
00371     }
00372     *fill++ = 1; *fill++ = 1; *fill++ = 3;
00373 //
00374     *fill++ = -SNUMBER; *fill++ = n1+1; // number of the node from
00375     *fill++ = -SNUMBER; *fill++ = n2+1; // number of the node to
00376     startfill[1] = fill - startfill;
00377 }
00378 }
00379 if ( ( info->flags & WITHBLOCKS ) == WITHBLOCKS ) {
00380     for ( i = 0; i < eg->econn->nblocks; i++ ) {
00381         startfill = fill;
00382         *fill++ = BLOCK;
00383         *fill++ = 0;
00384         FILLFUN(fill);
00385         *fill++ = -SNUMBER;
00386         *fill++ = i+1;
00387         *fill++ = -SNUMBER;
00388         *fill++ = eg->econn->blocks[i].loop;
00389 //
00390 // Now we have to make a list of all nodes inside this block
00391 //
00392         int bnodes[GRCC_MAXNODES], k;
00393         WORD *argfill = fill, *funfill;
00394         *fill++ = 0; *fill++ = 0; FILLARG(fill)
00395         *fill++ = 0;
00396         for ( k = 0; k < GRCC_MAXNODES; k++ ) bnodes[k] = 0;
00397         for ( k = 0; k < eg->econn->blocks[i].nmedges; k++ ) {
00398             bnodes[eg->econn->blocks[i].edges[k][0]] = 1;
00399             bnodes[eg->econn->blocks[i].edges[k][1]] = 1;
00400         }
00401         for ( k = 0; k < GRCC_MAXNODES; k++ ) {
00402             if ( bnodes[k] == 0 ) continue;
00403 //
00404 // Now we put the node inside this argument.
00405 //
00406             funfill = fill;
00407             *fill++ = NODEFUNCTION;
00408             *fill++ = 0;
00409             FILLFUN(fill)
00410             *fill++ = -SNUMBER; *fill++ = k+1;
00411             numlegs = eg->nodes[k]->deg;
00412             for ( j = 0; j < numlegs; j++ ) {
00413                 edge = eg->nodes[k]->edges[j];
00414                 vect = ABS(edge)-1;
00415                 *fill++ = -VECTOR;
00416                 if ( numlegs == 1 || vect < info->numextern ) { // Look up in set of external
momenta
00417                     *fill++ = SetElements[Sets[info->externalset].first+vect];
00418                 }
00419                 else { // Look up in set of internal momenta set
00420                     *fill++ = SetElements[Sets[info->internalset].first+(vect-info->numextern)];
00421                     maxmom = MaX(maxmom, vect-info->numextern);
00422                 }
00423             }
00424             funfill[1] = fill-funfill;
00425         }
00426         *fill++ = 1; *fill++ = 1; *fill++ = 3;
00427         *argfill = fill - argfill;
00428         argfill[ARGHEAD] = argfill[0] - ARGHEAD;
00429         startfill[1] = fill-startfill;
00430     }
00431 }
00432 if ( ( info->flags & WITHONEPISETS ) == WITHONEPISETS ) {
00433     for ( i = 0; i < eg->econn->nopic; i++ ) {
00434         startfill = fill;
00435         *fill++ = ONEPI;
00436         *fill++ = 0;
00437         FILLFUN(fill);
00438         *fill++ = -SNUMBER;
00439         *fill++ = i+1;
00440         *fill++ = -ONEPI;
00441         for ( j = 0; j < eg->econn->opics[i].nnodes; j++ ) {
00442             *fill++ = -SNUMBER;
00443             *fill++ = eg->econn->opics[i].nodes[j]+1;
00444         }
00445         startfill[1] = fill-startfill;
00446     }
00447 }
00448 //
00449 // Topology counter. We have exaggerated a bit with the eye on the far future.
00450 //
00451 if ( info->numtopo-1 < MAXPOSITIVE ) {

```

```

00452         *fill++ = TOPO; *fill++ = FUNHEAD+2; FILLFUN(fill)
00453         *fill++ = -SNUMBER; *fill++ = (WORD)(info->numtopo-1);
00454     }
00455     else if ( info->numtopo-1 < FULLMAX-1 ) {
00456         *fill++ = TOPO; *fill++ = FUNHEAD+ARGHEAD+4; FILLFUN(fill)
00457         *fill++ = ARGHEAD+4; *fill++ = 0; FILLARG(fill)
00458         *fill++ = 4;
00459         *fill++ = (WORD)((info->numtopo-1) & WORDMASK);
00460         *fill++ = 1; *fill++ = 3;
00461     }
00462     else { // for now: science fiction
00463         *fill++ = TOPO; *fill++ = FUNHEAD+ARGHEAD+6; FILLFUN(fill)
00464         *fill++ = ARGHEAD+6; *fill++ = 0; FILLARG(fill)
00465         *fill++ = 6; *fill++ = (WORD)((info->numtopo-1) » BITSINWORD);
00466         *fill++ = (WORD)((info->numtopo-1) & WORDMASK);
00467         *fill++ = 0; *fill++ = 1; *fill++ = 5;
00468     }
00469 //
00470 // Symmetry factors. We let Normalize do the multiplication.
00471 //
00472     if ( eg->nsym != 1 ) {
00473         *fill++ = SNUMBER; *fill++ = 4; *fill++ = (WORD)eg->nsym; *fill++ = -1;
00474     }
00475     if ( eg->esym != 1 ) {
00476         *fill++ = SNUMBER; *fill++ = 4; *fill++ = (WORD)eg->esym; *fill++ = -1;
00477     }
00478     if ( eg->extperm != 1 ) {
00479         *fill++ = SNUMBER; *fill++ = 4; *fill++ = (WORD)eg->extperm; *fill++ = 1;
00480     }
00481 //
00482 // verify internalset has sufficient momenta:
00483 //
00484     if ( maxmom >= Sets[info->internalset].last - Sets[info->internalset].first ) {
00485         MLOCK(ErrorMessageLock);
00486         MesPrint("&Insufficient internal momenta in diagrams_");
00487         MUNLOCK(ErrorMessageLock);
00488         Terminate(-1);
00489     }
00490 //
00491 // finish it off
00492 //
00493     while ( tail < tend ) *fill++ = *tail++;
00494     if ( eg->fsign < 0 ) fill[-1] = -fill[-1];
00495     *newterm = fill - newterm;
00496     AT.WorkPointer = fill;
00497
00498     Generator(BHEAD newterm,info->level);
00499     AT.WorkPointer = oldworkpointer;
00500 }
00501
00502 // #] ProcessDiagram :
00503 // #[ fendMG :
00504
00505 Bool fendMG(EGraph *eg, void *ti)
00506 {
00507     DUMMYUSE(eg);
00508     DUMMYUSE(ti);
00509     return True;
00510 }
00511
00512 // #] fendMG :
00513 // #[ ProcessTopology :
00514
00515 Bool ProcessTopology(EGraph *eg, void *ti)
00516 {
00517 //
00518 // This routine is called for each new topology.
00519 // New convention: return True; generate the corresponding diagrams if needed
00520 // return False; skip diagram generation (when asked for).
00521 //
00522     TERMINFO *info = (TERMINFO *)ti;
00523
00524 // This seems to work properly. It was disabled before.
00525 #define WITHEARLYVETO
00526 #ifdef WITHEARLYVETO
00527     if ( ( ( info->flags & CHECKEXTERN ) == CHECKEXTERN ) && info->currentMODEL != NULL ) {
00528         int i, j;
00529         int numlegs, vect, edge;
00530         for ( i = 0; i < eg->nNodes; i++ ) {
00531             if ( eg->isExternal(i) ) continue;
00532             numlegs = eg->nodes[i]->deg;
00533             for ( j = 0; j < numlegs; j++ ) {
00534                 edge = eg->nodes[i]->edges[j];
00535                 vect = ABS(edge)-1;
00536                 if ( vect < info->numextern && info->legcouple[vect][numlegs] == 0 ) {
00537
00538 // This cannot be.

```

```

00539
00540             info->numtopo++;
00541             return False;
00542         }
00543     }
00544 }
00545 }
00546 #endif
00547
00548 if ( ( info->flags & TOPOLOGIESONLY ) == 0 ) {
00549     info->numtopo++;
00550     return True;
00551 }
00552 //
00553 // Now we are just generating topologies.
00554 //
00555 GETIDENTITY
00556 WORD *term = info->term, *newterm, *oldworkpointer = AT.WorkPointer;
00557 WORD *tdia = term + info->diaoffset;
00558 WORD *tail = tdia + tdia[1];
00559 WORD *tend = term + *term;
00560 WORD *fill, *startfill;
00561 Model *model = (Model *)info->currentModel;
00562 MODEL *m = (MODEL *)info->currentMODEL;
00563 int i, j;
00564 int numlegs, vect, edge, maxmom = 0;
00565
00566 newterm = term + *term;
00567 for ( i = 1; i < info->diaoffset; i++ ) newterm[i] = term[i];
00568 fill = newterm + info->diaoffset;
00569 //
00570 // Now get the nodes
00571 //
00572 for ( i = 0; i < eg->nNodes; i++ ) {
00573 //
00574 //     node_(number,momentum_1,...,momentum_n)
00575 //
00576     numlegs = eg->nodes[i]->deg;
00577     startfill = fill;
00578     *fill++ = NODEFUNCTION;
00579     *fill++ = 0;
00580     FILLFUN(fill)
00581     *fill++ = -SNUMBER; *fill++ = i+1;
00582     if ( model != NULL && m != NULL ) {
00583         if ( !eg->isExternal(i) ) {
00584             WORD *afill = fill; *fill++ = 0; *fill++ = 1; FILLARG(fill)
00585             WORD *cfill = fill; *fill++ = 0;
00586             int cpl = 2*eg->nodes[i]->extloop+eg->nodes[i]->deg-2;
00587             *fill++ = SYMBOL; *fill++ = 4;
00588             *fill++ = m->couplings[0];
00589             *fill++ = cpl;
00590             *fill++ = 1; *fill++ = 1; *fill++ = 3;
00591             *cfill = fill - cfill;
00592             *afill = fill - afill;
00593             if ( *afill == ARGHEAD+8 && afill[ARGHEAD+4] == 1 ) {
00594                 fill = afill; *fill++ = -SYMBOL; *fill++ = afill[ARGHEAD+3];
00595             }
00596         }
00597         else {
00598             *fill++ = -SNUMBER;
00599             *fill++ = 1;
00600         }
00601     }
00602 //
00603 // Now the momenta.
00604 //
00605     for ( j = 0; j < numlegs; j++ ) {
00606         edge = eg->nodes[i]->edges[j];
00607         vect = ABS(edge)-1;
00608         *fill++ = edge < 0 ? -MINVECTOR: -VECTOR;
00609         if ( numlegs == 1 || vect < info->numextern ) { // Look up in set of external momenta
00610             *fill++ = SetElements[Sets[info->externalset].first+vect];
00611         }
00612         else { // Look up in set of internal momenta set
00613             *fill++ = SetElements[Sets[info->internalset].first+(vect-info->numextern)];
00614             // determine the number of momenta required from internalset:
00615             maxmom = MaX(maxmom, vect-info->numextern);
00616         }
00617     }
00618     startfill[1] = fill-startfill;
00619 }
00620 if ( ( info->flags & WITHEDGES ) == WITHEDGES ) {
00621     for ( i = 0; i < eg->nEdges; i++ ) {
00622         int n1 = eg->edges[i]->nodes[0];
00623         int n2 = eg->edges[i]->nodes[1];
00624         int l1 = eg->edges[i]->nlegs[0];
00625         int l2 = eg->edges[i]->nlegs[1];

```

```

00626         startfill = fill;
00627         *fill++ = EDGE;
00628         *fill++ = 0;
00629         FILLFUN(fill)
00630         *fill++ = -SNUMBER; *fill++ = i+1; // number of the edge
00631 //
00632         *fill++ = -VECTOR;
00633         if ( i < info->numextern ) { // Look up in set of external momenta
00634             *fill++ = SetElements[Sets[info->externalset].first+i];
00635         }
00636         else { // Look up in set of internal momenta set
00637             *fill++ = SetElements[Sets[info->internalset].first+(i-eg->nExtern)];
00638             maxmom = MaX(maxmom, i-eg->nExtern);
00639         }
00640 //
00641         *fill++ = -SNUMBER; *fill++ = n1+1; // number of the node from
00642         *fill++ = -SNUMBER; *fill++ = n2+1; // number of the node to
00643         startfill[1] = fill - startfill;
00644     }
00645 }
00646 //
00647 // Block information
00648 //
00649 if ( ( info->flags & WITHBLOCKS ) == WITHBLOCKS ) {
00650     for ( i = 0; i < eg->econn->nblocks; i++ ) {
00651         startfill = fill;
00652         *fill++ = BLOCK;
00653         *fill++ = 0;
00654         FILLFUN(fill);
00655         *fill++ = -SNUMBER;
00656         *fill++ = i+1;
00657         *fill++ = -SNUMBER;
00658         *fill++ = eg->econn->blocks[i].loop;
00659 //
00660 // Now we have to make a list of all nodes inside this block
00661 //
00662         int bnodes[GRCC_MAXNODES], k;
00663         WORD *argfill = fill, *funfill;
00664         *fill++ = 0; *fill++ = 0; FILLARG(fill)
00665         *fill++ = 0;
00666         for ( k = 0; k < GRCC_MAXNODES; k++ ) bnodes[k] = 0;
00667         for ( k = 0; k < eg->econn->blocks[i].nmedges; k++ ) {
00668             bnodes[eg->econn->blocks[i].edges[k][0]] = 1;
00669             bnodes[eg->econn->blocks[i].edges[k][1]] = 1;
00670         }
00671         for ( k = 0; k < GRCC_MAXNODES; k++ ) {
00672             if ( bnodes[k] == 0 ) continue;
00673 //
00674 // Now we put the node inside this argument.
00675 //
00676             funfill = fill;
00677             *fill++ = NODEFUNCTION;
00678             *fill++ = 0;
00679             FILLFUN(fill)
00680             *fill++ = -SNUMBER; *fill++ = k+1;
00681             numlegs = eg->nodes[k]->deg;
00682             for ( j = 0; j < numlegs; j++ ) {
00683                 edge = eg->nodes[k]->edges[j];
00684                 vect = ABS(edge)-1;
00685                 *fill++ = -VECTOR;
00686                 if ( numlegs == 1 || vect < info->numextern ) { // Look up in set of external
momenta
00687                     *fill++ = SetElements[Sets[info->externalset].first+vect];
00688                 }
00689                 else { // Look up in set of internal momenta set
00690                     *fill++ = SetElements[Sets[info->internalset].first+(vect-info->numextern)];
00691                     maxmom = MaX(maxmom, vect-info->numextern);
00692                 }
00693             }
00694             funfill[1] = fill-funfill;
00695         }
00696         *fill++ = 1; *fill++ = 1; *fill++ = 3;
00697         *argfill = fill - argfill;
00698         argfill[ARGHEAD] = argfill[0] - ARGHEAD;
00699         startfill[1] = fill-startfill;
00700     }
00701 }
00702 //
00703 // if ( eg->econn->narticuls > 0 ) {
00704 //     startfill = fill;
00705 //     *fill++ = BLOCK;
00706 //     *fill++ = 0;
00707 //     FILLFUN(fill);
00708 //     for ( i = 0; i < eg->econn->snodes; i++ ) {
00709 //         if ( eg->econn->articuls[i] != 0 ) {
00710 //             *fill++ = -SNUMBER;
00711 //             *fill++ = i+1;

```

```

00712 //      }
00713 //      startfill[1] = fill-startfill;
00714 //      }
00715 }
00716 if ( ( info->flags & WITHONEPISETS ) == WITHONEPISETS ) {
00717     for ( i = 0; i < eg->econn->nopic; i++ ) {
00718         startfill = fill;
00719         *fill++ = ONEPI;
00720         *fill++ = 0;
00721         FILLFUN(fill);
00722         *fill++ = -SNUMBER;
00723         *fill++ = i+1;
00724         *fill++ = -ONEPI;
00725         for ( j = 0; j < eg->econn->opics[i].nnodes; j++ ) {
00726             *fill++ = -SNUMBER;
00727             *fill++ = eg->econn->opics[i].nodes[j]+1;
00728         }
00729         startfill[1] = fill-startfill;
00730     }
00731 }
00732 //
00733 // Topology counter. We have exaggerated a bit with the eye on the far future.
00734 //
00735 if ( info->numtopo < MAXPOSITIVE ) {
00736     *fill++ = TOPO; *fill++ = FUNHEAD+2; FILLFUN(fill)
00737     *fill++ = -SNUMBER; *fill++ = (WORD)(info->numtopo);
00738 }
00739 else if ( info->numtopo < FULLMAX-1 ) {
00740     *fill++ = TOPO; *fill++ = FUNHEAD+ARGHEAD+4; FILLFUN(fill)
00741     *fill++ = ARGHEAD+4; *fill++ = 0; FILLARG(fill)
00742     *fill++ = 4;
00743     *fill++ = (WORD)(info->numtopo & WORDMASK);
00744     *fill++ = 1; *fill++ = 3;
00745 }
00746 else { // for now: science fiction
00747     *fill++ = TOPO; *fill++ = FUNHEAD+ARGHEAD+6; FILLFUN(fill)
00748     *fill++ = ARGHEAD+6; *fill++ = 0; FILLARG(fill)
00749     *fill++ = 6; *fill++ = (WORD)(info->numtopo » BITSINWORD);
00750     *fill++ = (WORD)(info->numtopo & WORDMASK);
00751     *fill++ = 0; *fill++ = 1; *fill++ = 5;
00752 }
00753 //
00754 // verify internalset has sufficient momenta:
00755 //
00756 if ( maxmom >= Sets[info->internalset].last - Sets[info->internalset].first ) {
00757     MLOCK(ErrorMessageLock);
00758     MesPrint("&Insufficient internal momenta in diagrams_");
00759     MUNLOCK(ErrorMessageLock);
00760     Terminate(-1);
00761 }
00762 //
00763 // finish it off
00764 //
00765 while ( tail < tend ) *fill++ = *tail++;
00766 if ( eg->fsign < 0 ) fill[-1] = -fill[-1];
00767 *newterm = fill - newterm;
00768 AT.WorkPointer = fill;
00769
00770 Generator(BHEAD newterm,info->level);
00771 AT.WorkPointer = oldworkpointer;
00772 info->numtopo++;
00773 return False;
00774 }
00775
00776 // #] ProcessTopology :
00777 // #[ SetDualOpts :
00778 void SetDualOpts(int *opt, const WORD num, const int key, const char* key_name,
00779     const int dual, const char* dual_name, const int val, const int dval) {
00780
00781     if ( ( num & key ) == key ) {
00782         if ( ( num & dual ) == dual ) {
00783             MLOCK(ErrorMessageLock);
00784             MesPrint("&Conflicting diagram filters: %s and %s.", key_name, dual_name);
00785             MUNLOCK(ErrorMessageLock);
00786             Terminate(-1);
00787         }
00788         else {
00789             *opt = val;
00790         }
00791     }
00792     else {
00793         if ( ( num & dual ) == dual ) {
00794             *opt = dval;
00795         }
00796         else {
00797             // The default value is always 0.
00798             *opt = 0;

```

```

00799     }
00800   }
00801 }
00802 // #] SetDualOpts :
00803 // #[ GenDiagrams :
00804
00805 int GenDiagrams(PHEAD WORD *term, WORD level)
00806 {
00807     Model *model;
00808     MODEL *m;
00809     Options *opt;
00810     Process *proc;
00811     int pid = 1, x;
00812     int babble = 0;    // Later we may set this at the FORM code level
00813     TERMINFO info;
00814     WORD inset,outset,*coupl,setnum,optionnumber = 0;
00815     int i, j, cpl[GRCC_MAXNCPLG];
00816     int ninitl, initlPart[GRCC_MAXLEGS], nfinal, finalPart[GRCC_MAXLEGS];
00817     for ( i = 0; i < GRCC_MAXNCPLG; i++ ) cpl[i] = 0;
00818 //
00819 // Here we create an object of type Option and load it up.
00820 // Next we run the diagram generation on it.
00821 //
00822     info.term = term;
00823     info.level = level;
00824     info.diaoffset = AR.funoffset;
00825     info.externalset = term[info.diaoffset+FUNHEAD+7];
00826     info.internalset = term[info.diaoffset+FUNHEAD+9];
00827     info.flags = 0;
00828     inset = term[info.diaoffset+FUNHEAD+3];
00829     outset = term[info.diaoffset+FUNHEAD+5];
00830     coupl = term + info.diaoffset + FUNHEAD + 10;
00831     if ( *coupl < 0 ) {
00832         if ( term[info.diaoffset+1] > FUNHEAD + 12 ) {
00833             optionnumber = term[info.diaoffset+FUNHEAD+13];
00834         }
00835     }
00836     else {
00837         if ( term[info.diaoffset+1] > *coupl+FUNHEAD+10 )
00838             optionnumber = term[info.diaoffset+*coupl+FUNHEAD+11];
00839     }
00840     setnum = term[info.diaoffset+FUNHEAD+1];
00841
00842     m = AC.models[SetElements[Sets[setnum].first]];
00843     LoadModel(m);
00844     model = (Model *)m->grccmodel;
00845
00846     info.currentModel = (void *)model;
00847     info.currentMODEL = (void *)m;
00848     info.numdia = 0;
00849     info.numtopo = 1;
00850     info.flags = optionnumber;
00851
00852     opt = new Options();
00853
00854     opt->setOutAG(ProcessDiagram, &info);
00855     opt->setOutMG(ProcessTopology, &info);
00856
00857     opt->values[GRCC_OPT_SymmInitial] = ( optionnumber & WITHSYMMETRIZEI ) == WITHSYMMETRIZEI;
00858     opt->values[GRCC_OPT_SymmFinal]   = ( optionnumber & WITHSYMMETRIZEF ) == WITHSYMMETRIZEF;
00859
00860     // WITHBLOCKS controls output formatting. We could introduce an extra filtering option
00861     // corresponding to GRCC_OPT_Block, which is somewhat like Qgraf "onevi" but not quite
00862     // the same currently.
00863     //opt->values[GRCC_OPT_Block] = ;
00864
00865     // Now the "qgraf-compatible filtering options":
00866     int qgopt[GRCC_QGRAF_OPT_Size];
00867     SetDualOpts(&qgopt[GRCC_QGRAF_OPT_ONEPI],    optionnumber,ONEPARTI, "ONEPI_",    ONEPARTR,
00868 "ONEPR_",    1,-1);
00869     SetDualOpts(&qgopt[GRCC_QGRAF_OPT_ONSHELL],  optionnumber,ONSHELL,  "ONSHELL_", OFFSHELL,
00870 "OFFSHELL_", 1,-1);
00871     SetDualOpts(&qgopt[GRCC_QGRAF_OPT_NOSIGMA],   optionnumber,NOSIGMA, "NOSIGMA_", SIGMA,
00872 "SIGMA_",    1,-1);
00873     SetDualOpts(&qgopt[GRCC_QGRAF_OPT_NOSNAIL],   optionnumber,NOSNAIL, "NOSNAIL_", SNAIL,
00874 "SNAIL_",    1,-1);
00875     SetDualOpts(&qgopt[GRCC_QGRAF_OPT_NOTADPOLE], optionnumber,NOTADPOLE,"NOTADPOLE_",TADPOLE ,
00876 "TADPOLE_",  1,-1);
00877     SetDualOpts(&qgopt[GRCC_QGRAF_OPT_SIMPLE],    optionnumber,SIMPLE,  "SIMPLE_",
00878 NOTSIMPLE,"NOTSIMPLE_",1,-1);
00879     SetDualOpts(&qgopt[GRCC_QGRAF_OPT_BIPART],     optionnumber,BIPART,  "BIPART_",
00880 NONBIPART,"NONBIPART_",1,-1);
00881     SetDualOpts(&qgopt[GRCC_QGRAF_OPT_CYCLI],      optionnumber,CYCLI,   "CYCLI_",   CYCLR,
00882 "CYCLR_",    1,-1);
00883     SetDualOpts(&qgopt[GRCC_QGRAF_OPT_FLOOP],      optionnumber,FLOOP,   "FLOOP_",   NOTFLOOP,
00884 "NOTFLOOP_", 1,-1);
00885 // Now set the options internally:

```

```

00877     opt->setQGrafOpt(qgopt);
00878
00879     opt->setOutputF(False,"");
00880     opt->setOutputP(False,"");
00881     opt->printLevel(babble);
00882
00883 // Load the various arrays.
00884 ninitl = Sets[inset].last - Sets[inset].first;
00885 for ( i = 0; i < ninitl; i++ ) {
00886     x = SetElements[Sets[inset].first+i];
00887     initlPart[i] = ConvertParticle(model,x);
00888     info.legcouple[i] = m->vertices[numParticle(m,x)]->couplings;
00889 }
00890 nfinal = Sets[outset].last - Sets[outset].first;
00891 for ( i = 0; i < nfinal; i++ ) {
00892     x = SetElements[Sets[outset].first+i];
00893     finalPart[i] = ConvertParticle(model,x);
00894     info.legcouple[i+ninitl] = m->vertices[numParticle(m,x)]->couplings;
00895 }
00896 info.numextern = ninitl + nfinal;
00897 // Check that we have sufficient external momenta in the set:
00898 if ( info.numextern > Sets[info.externalset].last - Sets[info.externalset].first ) {
00899     MLOCK(ErrorMessageLock);
00900     MesPrint("&Insufficient external momenta in diagrams_");
00901     MUNLOCK(ErrorMessageLock);
00902     Terminate(-1);
00903 }
00904 for ( i = 2; i <= MAXLEGS; i++ ) {
00905     if ( m->legcouple[i] == 1 ) {
00906         for ( j = 0; j < info.numextern; j++ ) {
00907             if ( info.legcouple[j][i] == 0 ) { info.flags |= CHECKEXTERN; goto Go_on; }
00908         }
00909     }
00910 }
00911 Go_on;;
00912 //
00913 // Now we have to sort out the coupling constants.
00914 // The argument at coupl can be of type -SNUMBER, -SYMBOL or generic
00915 // It has however already be tested for syntax.
00916 // Note that one cannot have 1 for the coupling constants.
00917 // In that case one should select 0 loops or something equivalent.
00918 //
00919 if ( *coupl == -SNUMBER ) { // Number of loops
00920 //
00921 // This is the complicated case.
00922 // We have to compute the number of coupling constants and then
00923 // generate diagrams for all combinations with the proper power.
00924 //
00925 int nc = coupl[1]*2 + ninitl + nfinal - 2;
00926 int *scratch = (int *)Malloca(nc*sizeof(int),"DistrN");
00927 scratch[0] = -2; // indicating startup cq first call.
00928
00929 if ( ( info.flags & TOPOLOGIESONLY ) == 0 ) {
00930     while ( DistrN(nc,cpl,m->ncouplings,scratch) ) {
00931         proc = new Process(pid, model, opt, ninitl, initlPart, nfinal, finalPart, cpl);
00932         delete proc;
00933         info.numtopo = 1;
00934     }
00935 }
00936 else {
00937     cpl[0] = nc;
00938     proc = new Process(pid, model, opt, ninitl, initlPart, nfinal, finalPart, cpl);
00939     delete proc;
00940 }
00941 M_free(scratch,"DistrN");
00942 opt->end();
00943 delete opt;
00944 return(0);
00945 }
00946 else if ( *coupl == -SYMBOL ) { // Just a single power of one constant
00947     for ( i = 0; i < m->ncouplings; i++ ) {
00948         if ( m->couplings[i] == coupl[1] ) {
00949             cpl[i] = 1;
00950             break;
00951         }
00952     }
00953 }
00954 else { // One term with powers of coupling constants
00955     WORD *t, *tstop;
00956     t = coupl + ARGHEAD+3;
00957     tstop = coupl+*coupl; tstop -= ABS(tstop[-1]);
00958     while ( t < tstop ) {
00959         for ( i = 0; i < m->ncouplings; i++ ) {
00960             if ( m->couplings[i] == *t ) {
00961                 cpl[i] = t[1];
00962                 break;
00963             }

```

```

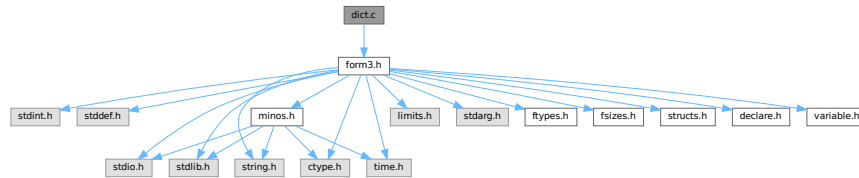
00964         }
00965         t += 2;
00966     }
00967 }
00968 /*
00969 And now the generation:
00970 */
00971 proc = new Process(pid, model, opt, ninitl, initlPart, nfinal, finalPart, cpl);
00972 opt->end();
00973 delete proc;
00974 delete opt;
00975 return(0);
00976 }
00977
00978 // #] GenDiagrams :
00979 // #[ processVertex :
00980
00981 // Routine is to be used recursively to work its way through a list
00982 // of possible vertices. The array of vertices is in TopoInf->vert
00983 // with TopoInf->nvert the number of possible vertices.
00984 // Currently we allow in TopoInf->vert only vertices with 3 or more edges.
00985 //
00986 // We work with a point system. Each n-point vertex contributes n-2 points.
00987 // When all points are assigned, we can call mgraph->generate().
00988 //
00989 // The number of vertices of a given number of edges is stored in
00990 // TopoInf->cnum[...] but the loop that determines how many there are
00991 // may be limited by the corresponding element in TopoInf->vertmax[level]
00992
00993 int processVertex(TOPOTYPE *TopoInf, int pointsremaining, int level)
00994 {
00995     int i, j;
00996
00997     for ( i = pointsremaining, j = 0; i >= 0; i -= TopoInf->vert[level]-2, j++ ) {
00998         if ( TopoInf->vertmax && TopoInf->vertmax[level] >= 0
00999             && j > TopoInf->vertmax[level] ) break;
01000         if ( i == 0 ) { // We got one!
01001             TopoInf->cldeg[TopoInf->ncl] = TopoInf->vert[level];
01002             TopoInf->cnum[TopoInf->ncl] = j;
01003             TopoInf->clxt[TopoInf->ncl] = 0;
01004             TopoInf->ncl++;
01005
01006             MGraph *mgraph = new MGraph(1, TopoInf->ncl, TopoInf->cldeg,
01007                                         TopoInf->cnum, TopoInf->clxt,
01008                                         TopoInf->cmind, TopoInf->cmxd, TopoInf->opt);
01009
01010             mgraph->generate();
01011
01012             delete mgraph;
01013
01014             TopoInf->ncl--;
01015
01016             break;
01017         }
01018         if ( level < TopoInf->nvert-1 ) {
01019             if ( j > 0 ) {
01020                 TopoInf->cldeg[TopoInf->ncl] = TopoInf->vert[level];
01021                 TopoInf->cnum[TopoInf->ncl] = j;
01022                 TopoInf->clxt[TopoInf->ncl] = 0;
01023                 TopoInf->ncl++;
01024             }
01025             if ( processVertex(TopoInf,i,level+1) < 0 ) return(-1);
01026             if ( j > 0 ) { TopoInf->ncl--; }
01027         }
01028     }
01029     return(0);
01030 }
01031
01032 // #] processVertex :
01033

```


4.25 dict.c File Reference

```
#include "form3.h"
```

Include dependency graph for dict.c:



Functions

- void [TransformRational](#) (UWORD *a, WORD na)
- UBYTE * [IsMultiplySign](#) (void)
- UBYTE * [IsExponentSign](#) (void)
- UBYTE * [FindSymbol](#) (WORD num)
- UBYTE * [FindVector](#) (WORD num)
- UBYTE * [FindIndex](#) (WORD num)
- UBYTE * [FindFunction](#) (WORD num)
- UBYTE * [FindFunWithArgs](#) (WORD *t)
- UBYTE * [FindExtraSymbol](#) (WORD num)
- int [FindDictionary](#) (UBYTE *name)
- int [AddDictionary](#) (UBYTE *name)
- int [AddToDictionary](#) (DICTIONARY *dict, UBYTE *left, UBYTE *right)
- int [UseDictionary](#) (UBYTE *name, UBYTE *options)
- int [SetDictionaryOptions](#) (UBYTE *options)
- void [UnSetDictionary](#) (void)
- void [RemoveDictionary](#) (DICTIONARY *dict)
- void [ShrinkDictionary](#) (DICTIONARY *dict)
- int [DoPreOpenDictionary](#) (UBYTE *s)
- int [DoPreCloseDictionary](#) (UBYTE *s)
- int [DoPreUseDictionary](#) (UBYTE *s)
- int [DoPreAdd](#) (UBYTE *s)
- LONG [DictToBytes](#) (DICTIONARY *dict, UBYTE *buf)
- DICTIONARY * [DictFromBytes](#) (UBYTE *buf)

4.25.1 Detailed Description

Contains the code pertaining to dictionaries. Commands are: #opendictionary name #closedictionary #selectdictionary name <options>. There can be several dictionaries, but only one can be active. Defining elements is done with #add object: "replacement". Replacements are strings when a dictionary is for output translation. Objects can be 1: a number (rational) 2: a variable 3: * ^ 4: a function with arguments

Definition in file [dict.c](#).

4.25.2 Function Documentation

4.25.2.1 TransformRational()

```
void TransformRational (
    UWORD * a,
    WORD na )
```

Definition at line 71 of file [dict.c](#).

4.25.2.2 IsMultiplySign()

```
UBYTE * IsMultiplySign (
    void )
```

Definition at line 218 of file [dict.c](#).

4.25.2.3 IsExponentSign()

```
UBYTE * IsExponentSign (
    void )
```

Definition at line 238 of file [dict.c](#).

4.25.2.4 FindSymbol()

```
UBYTE * FindSymbol (
    WORD num )
```

Definition at line 258 of file [dict.c](#).

4.25.2.5 FindVector()

```
UBYTE * FindVector (
    WORD num )
```

Definition at line 279 of file [dict.c](#).

4.25.2.6 FindIndex()

```
UBYTE * FindIndex (
    WORD num )
```

Definition at line 301 of file [dict.c](#).

4.25.2.7 FindFunction()

```
UBYTE * FindFunction (
    WORD num )
```

Definition at line 323 of file [dict.c](#).

4.25.2.8 FindFunWithArgs()

```
UBYTE * FindFunWithArgs (
    WORD * t )
```

Definition at line 345 of file [dict.c](#).

4.25.2.9 FindExtraSymbol()

```
UBYTE * FindExtraSymbol (
    WORD num )
```

Definition at line 377 of file [dict.c](#).

4.25.2.10 FindDictionary()

```
int FindDictionary (
    UBYTE * name )
```

Definition at line 434 of file [dict.c](#).

4.25.2.11 AddDictionary()

```
int AddDictionary (
    UBYTE * name )
```

Definition at line 449 of file [dict.c](#).

4.25.2.12 AddToDictionary()

```
int AddToDictionary (
    DICTIONARY * dict,
    UBYTE * left,
    UBYTE * right )
```

Definition at line 491 of file [dict.c](#).

4.25.2.13 UseDictionary()

```
int UseDictionary (
    UBYTE * name,
    UBYTE * options )
```

Definition at line 766 of file [dict.c](#).

4.25.2.14 SetDictionaryOptions()

```
int SetDictionaryOptions (
    UBYTE * options )
```

Definition at line 790 of file [dict.c](#).

4.25.2.15 UnSetDictionary()

```
void UnSetDictionary (
    void )
```

Definition at line 883 of file [dict.c](#).

4.25.2.16 RemoveDictionary()

```
void RemoveDictionary (
    DICTIONARY * dict )
```

Definition at line 902 of file [dict.c](#).

4.25.2.17 ShrinkDictionary()

```
void ShrinkDictionary (
    DICTIONARY * dict )
```

Definition at line 945 of file [dict.c](#).

4.25.2.18 DoPreOpenDictionary()

```
int DoPreOpenDictionary (
    UBYTE * s )
```

Definition at line 959 of file [dict.c](#).

4.25.2.19 DoPreCloseDictionary()

```
int DoPreCloseDictionary (
    UBYTE * s )
```

Definition at line 998 of file [dict.c](#).

4.25.2.20 DoPreUseDictionary()

```
int DoPreUseDictionary (
    UBYTE * s )
```

Definition at line 1022 of file [dict.c](#).

4.25.2.21 DoPreAdd()

```
int DoPreAdd (
    UBYTE * s )
```

Definition at line 1067 of file [dict.c](#).

4.25.2.22 DictToBytes()

```
LONG DictToBytes (
    DICTIONARY * dict,
    UBYTE * buf )
```

Definition at line 1119 of file [dict.c](#).

4.25.2.23 DictFromBytes()

```
DICTIONARY * DictFromBytes (
    UBYTE * buf )
```

Definition at line 1152 of file [dict.c](#).

4.26 dict.c

[Go to the documentation of this file.](#)

```
00001
00018 /* #[] License : */
00019 /*
00020  * Copyright (C) 1984-2026 J.A.M. Vermaseren
00021  * When using this file you are requested to refer to the publication
00022  * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00023  * This is considered a matter of courtesy as the development was paid
00024  * for by FOM the Dutch physics granting agency and we would like to
00025  * be able to track its scientific use to convince FOM of its value
00026  * for the community.
00027  *
00028  * This file is part of FORM.
00029  *
00030  * FORM is free software: you can redistribute it and/or modify it under the
00031  * terms of the GNU General Public License as published by the Free Software
00032  * Foundation, either version 3 of the License, or (at your option) any later
00033  * version.
00034  *
00035  * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00036  * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00037  * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00038  * details.
00039  *
00040  * You should have received a copy of the GNU General Public License along
00041  * with FORM. If not, see <http://www.gnu.org/licenses/>.
00042  */
00043 /* #[] License : */
```

```

00044  /*
00045     #[ Includes : ratio.c
00046
00047     Data setup:
00048         AO.Dictionaries          Array of pointers to DICTIONARY
00049         AO.NumDictionaries
00050         AO.SizeDictionaries
00051         AO.CurrentDictionary
00052         AO.CurDictNumbers
00053         AO.CurDictVariables
00054         AO.CurDictSpecials
00055         AP.OpenDictionary
00056  */
00057
00058  #include "form3.h"
00059
00060  /*
00061     #] Includes :
00062     #[ TransformRational:
00063
00064     Tries to transform the rational number a according to the rules of
00065     the current dictionary. Whatever cannot be translated goes to the
00066     regular output.
00067     Options for AO.CurDictNumbers are:
00068         DICT_ALLNUMBERS, DICT_RATIONALONLY, DICT_INTEGERONLY, DICT_NONNUMBERS
00069  */
00070
00071  void TransformRational(UWORD *a, WORD na)
00072  {
00073      DICTIONARY *dict;
00074      WORD i, j, nb, i1, i2; UWORD *b;
00075      if ( AO.CurrentDictionary <= 0 ) goto NoAction;
00076      dict = AO.Dictionaries[AO.CurrentDictionary-1];
00077      if ( na < 0 ) na = -na;
00078      switch ( AO.CurDictNumbers ) {
00079          case DICT_NONNUMBERS:
00080              goto NoAction;
00081          case DICT_INTEGERONLY:
00082              if ( a[na] != 1 ) goto NoAction;
00083              if ( na > 1 ) {
00084                  for ( i = 1; i < na; i++ ) {
00085                      if ( a[na+i] != 0 ) goto NoAction;
00086                  }
00087              }
00088              Numeratoronly;;
00089              for ( i = dict->numelements-1; i >= 0; i-- ) {
00090                  if ( dict->elements[i]->type == DICT_INTEGERNUMBER ) {
00091                      if ( dict->elements[i]->size == na ) {
00092                          for ( j = 0; j < na; j++ ) {
00093                              if ( (UWORD)(dict->elements[i]->lhs[j]) != a[j] ) break;
00094                          }
00095                          if ( j == na ) { /* Got it */
00096                              TokenToLine((UBYTE *) (dict->elements[i]->rhs));
00097                              return;
00098                          }
00099                      }
00100                  }
00101              }
00102              goto NotFound;
00103          case DICT_RATIONALONLY:
00104              nb = 2*na;
00105              for ( i = dict->numelements-1; i >= 0; i-- ) {
00106                  if ( dict->elements[i]->type == DICT_RATIONALNUMBER ) {
00107                      if ( dict->elements[i]->size == nb+2 ) {
00108                          for ( j = 0; j < nb; j++ ) {
00109                              if ( (UWORD)(dict->elements[i]->lhs[j+1]) != a[j] ) break;
00110                          }
00111                          if ( j == nb ) { /* Got it */
00112                              TokenToLine((UBYTE *) (dict->elements[i]->rhs));
00113                              return;
00114                          }
00115                      }
00116                  }
00117              }
00118              goto NotFound;
00119          case DICT_ALLNUMBERS:
00120              /*
00121               First fish for rationals
00122              */
00123              nb = 2*na;
00124              for ( i = dict->numelements-1; i >= 0; i-- ) {
00125                  if ( dict->elements[i]->type == DICT_RATIONALNUMBER ) {
00126                      if ( dict->elements[i]->size == nb+2 ) {
00127                          for ( j = 0; j < nb; j++ ) {
00128                              if ( (UWORD)(dict->elements[i]->lhs[j+1]) != a[j] ) break;
00129                          }
00130                          if ( j == nb ) { /* Got it */

```

```

00131             TokenToLine((UBYTE *) (dict->elements[i]->rhs));
00132             return;
00133         }
00134     }
00135 }
00136 }
00137 /*
00138     Now look for element[j1]/element[j2]
00139 */
00140 nb = na; b = a+na;
00141 while ( b[nb-1] == 0 ) nb--;
00142 if ( nb == 1 && b[0] == 1 ) goto Numeratoronly;
00143 while ( a[na-1] == 0 ) na--;
00144 for ( i1 = dict->numelements-1; i1 >= 0; i1-- ) {
00145     if ( dict->elements[i1]->type == DICT_INTEGERNUMBER ) {
00146         if ( dict->elements[i1]->size == na ) {
00147             for ( j = 0; j < na; j++ ) {
00148                 if ( (UWORD) (dict->elements[i1]->lhs[j]) != a[j] ) break;
00149             }
00150             if ( j == na ) break;
00151         }
00152     }
00153 }
00154 for ( i2 = dict->numelements-1; i2 >= 0; i2-- ) {
00155     if ( dict->elements[i2]->type == DICT_INTEGERNUMBER ) {
00156         if ( dict->elements[i2]->size == nb ) {
00157             for ( j = 0; j < nb; j++ ) {
00158                 if ( (UWORD) (dict->elements[i2]->lhs[j]) != b[j] ) break;
00159             }
00160             if ( j == nb ) break;
00161         }
00162     }
00163 }
00164 if ( i1 < 0 ) {
00165     if ( i2 < 0 ) goto NotFound;
00166     else { /* number/replacement[i2] */
00167         LongToLine(a,na);
00168         if ( na > 1 || ( AO.DoubleFlag & 4 ) == 4 ) {
00169             if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
00170                 || AC.OutputMode == CMODE ) {
00171                 if ( ( AO.DoubleFlag & 2 ) == 2 ) { AddToLine((UBYTE *) ".Q0/"); }
00172                 else if ( ( AO.DoubleFlag & 1 ) == 1 ) { AddToLine((UBYTE *) ".D0/"); }
00173                 else { AddToLine((UBYTE *) "/"); }
00174             }
00175         }
00176         else AddToLine((UBYTE *) ("/"));
00177         TokenToLine((UBYTE *) (dict->elements[i2]->rhs));
00178     }
00179 }
00180 else if ( i2 < 0 ) { /* replacement[i1]/number */
00181     TokenToLine((UBYTE *) (dict->elements[i1]->rhs));
00182     AddToLine((UBYTE *) ("/"));
00183     LongToLine((UWORD *) (b),nb);
00184     if ( nb > 1 || ( AO.DoubleFlag & 4 ) == 4 ) {
00185         if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
00186             || AC.OutputMode == CMODE ) {
00187             if ( ( AO.DoubleFlag & 2 ) == 2 ) { AddToLine((UBYTE *) ".Q0"); }
00188             else if ( ( AO.DoubleFlag & 1 ) == 1 ) { AddToLine((UBYTE *) ".D0"); }
00189         }
00190     }
00191 }
00192 else { /* replacement[i1]/replacement[i2] */
00193     TokenToLine((UBYTE *) (dict->elements[i1]->rhs));
00194     AddToLine((UBYTE *) ("/"));
00195     TokenToLine((UBYTE *) (dict->elements[i2]->rhs));
00196 }
00197 break;
00198 default:
00199     MesPrint("Illegal code in TransformRational: %d",AO.CurDictNumbers);
00200     Terminate(-1);
00201 }
00202 return;
00203 NotFound:
00204 if ( na != 1 || a[1] != 1 ) {
00205     if ( AO.CurDictNumberWarning ) {
00206         MesPrint("»»»»»Could not translate coefficient with dictionary %s«««««",dict->name);
00207     }
00208 NoAction:
00209     RatToLine(a,na);
00210     return;
00211 }
00212
00213 /*
00214     #] TransformRational:
00215     #[ IsMultiplySign:
00216 */
00217

```

```

00218 UBYTE *IsMultiplySign(void)
00219 {
00220     DICTIONARY *dict;
00221     int i;
00222     if ( AO.CurrentDictionary <= 0 ) return(0);
00223     dict = AO.Dictionaries[AO.CurrentDictionary-1];
00224     if ( dict->characters == 0 ) return(0);
00225     for ( i = dict->numelements-1; i >= 0; i-- ) {
00226         if ( ( dict->elements[i]->type == DICT_SPECIALCHARACTER )
00227             && ( dict->elements[i]->lhs[0] == (WORD)('*') ) )
00228             return((UBYTE *) (dict->elements[i]->rhs));
00229     }
00230     return(0);
00231 }
00232
00233 /*
00234 #] IsMultiplySign:
00235 #[ IsExponentSign:
00236 */
00237
00238 UBYTE *IsExponentSign(void)
00239 {
00240     DICTIONARY *dict;
00241     int i;
00242     if ( AO.CurrentDictionary <= 0 ) return(0);
00243     dict = AO.Dictionaries[AO.CurrentDictionary-1];
00244     if ( dict->characters == 0 ) return(0);
00245     for ( i = dict->numelements-1; i >= 0; i-- ) {
00246         if ( ( dict->elements[i]->type == DICT_SPECIALCHARACTER )
00247             && ( dict->elements[i]->lhs[0] == (WORD)('^') ) )
00248             return((UBYTE *) (dict->elements[i]->rhs));
00249     }
00250     return(0);
00251 }
00252
00253 /*
00254 #] IsExponentSign:
00255 #[ FindSymbol :
00256 */
00257
00258 UBYTE *FindSymbol(WORD num)
00259 {
00260     if ( AO.CurrentDictionary > 0 ) {
00261         DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
00262         int i;
00263         if ( dict->variables > 0 && AO.CurDictVariables == DICT_DOVARIABLES ) {
00264             for ( i = dict->numelements-1; i >= 0; i-- ) {
00265                 if ( dict->elements[i]->type == DICT_SYMBOL &&
00266                     dict->elements[i]->lhs[0] == num )
00267                     return((UBYTE *) (dict->elements[i]->rhs));
00268             }
00269         }
00270     }
00271     return(VARNAME(symbols,num));
00272 }
00273
00274 /*
00275 #] FindSymbol :
00276 #[ FindVector :
00277 */
00278
00279 UBYTE *FindVector(WORD num)
00280 {
00281     if ( AO.CurrentDictionary > 0 ) {
00282         DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
00283         int i;
00284         if ( dict->variables > 0 && AO.CurDictVariables == DICT_DOVARIABLES ) {
00285             for ( i = dict->numelements-1; i >= 0; i-- ) {
00286                 if ( dict->elements[i]->type == DICT_VECTOR &&
00287                     dict->elements[i]->lhs[0] == num )
00288                     return((UBYTE *) (dict->elements[i]->rhs));
00289             }
00290         }
00291     }
00292     num -= AM.OffsetVector;
00293     return(VARNAME(vectors,num));
00294 }
00295
00296 /*
00297 #] FindVector :
00298 #[ FindIndex :
00299 */
00300
00301 UBYTE *FindIndex(WORD num)
00302 {
00303     if ( AO.CurrentDictionary > 0 ) {
00304         DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];

```



```

00305     int i;
00306     if ( dict->variables > 0 && AO.CurDictVariables == DICT_DOVARIABLES ) {
00307         for ( i = dict->numelements-1; i >= 0; i-- ) {
00308             if ( dict->elements[i]->type == DICT_INDEX &&
00309                 dict->elements[i]->lhs[0] == num )
00310                 return((UBYTE *) (dict->elements[i]->rhs));
00311         }
00312     }
00313 }
00314 num -= AM.OffsetIndex;
00315 return(VARNAME(indices,num));
00316 }
00317
00318 /*
00319 #] FindIndex :
00320 #[ FindFunction :
00321 */
00322
00323 UBYTE *FindFunction(WORD num)
00324 {
00325     if ( AO.CurrentDictionary > 0 ) {
00326         DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
00327         int i;
00328         if ( dict->variables > 0 && AO.CurDictVariables == DICT_DOVARIABLES ) {
00329             for ( i = dict->numelements-1; i >= 0; i-- ) {
00330                 if ( dict->elements[i]->type == DICT_FUNCTION &&
00331                     dict->elements[i]->lhs[0] == num )
00332                     return((UBYTE *) (dict->elements[i]->rhs));
00333             }
00334         }
00335     }
00336     num -= FUNCTION;
00337     return(VARNAME(functions,num));
00338 }
00339
00340 /*
00341 #] FindFunction :
00342 #[ FindFunWithArgs :
00343 */
00344
00345 UBYTE *FindFunWithArgs(WORD *t)
00346 {
00347     if ( AO.CurrentDictionary > 0 ) {
00348         DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
00349         int i, j;
00350         if ( dict->funwith > 0
00351             && AO.CurDictFunWithArgs == DICT_DOFUNWITHARGS ) {
00352             for ( i = dict->numelements-1; i >= 0; i-- ) {
00353                 if ( dict->elements[i]->type == DICT_FUNCTION_WITH_ARGUMENTS &&
00354                     (WORD)(dict->elements[i]->lhs[0]) == t[0] &&
00355                     (WORD)(dict->elements[i]->lhs[1]) == t[1] ) {
00356                     for ( j = 2; j < t[1]; j++ ) {
00357                         if ( (WORD)(dict->elements[i]->lhs[j]) != t[j] ) break;
00358                     }
00359                     if ( j >= t[1] ) return((UBYTE *) (dict->elements[i]->rhs));
00360                 }
00361             }
00362         }
00363     }
00364     return(0);
00365 }
00366
00367 /*
00368 #] FindFunWithArgs :
00369 #[ FindExtraSymbol :
00370
00371 The extra symbol is constructed in the Workspace. This way we do not
00372 have to worry about Malloc and freeing the object later.
00373 The input value num is already the number of the extra symbol.
00374 We do NOT need num = MAXVARIABLES-num;
00375 */
00376
00377 UBYTE *FindExtraSymbol(WORD num)
00378 {
00379     GETIDENTITY;
00380     UBYTE *out = (UBYTE *) (AT.WorkPointer);
00381     *out = 0;
00382     if ( AO.CurrentDictionary > 0 ) {
00383         DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
00384         int i;
00385         if ( dict->ranges > 0 && AO.CurDictVariables == DICT_DOVARIABLES ) {
00386             for ( i = dict->numelements-1; i >= 0; i-- ) {
00387                 if ( dict->elements[i]->type == DICT_RANGE
00388                     && num >= dict->elements[i]->lhs[0]
00389                     && num <= dict->elements[i]->lhs[1] ) {
00390
00391                     Now we have to translate the rhs

```

```

00392             %# gives the number
00393             %@ gives the number as its position in the range
00394         */
00395         UBYTE *r = (UBYTE *) (dict->elements[i]->rhs);
00396         while ( *r ) {
00397             if ( *r == (UBYTE) '%' && ( r[1] == (UBYTE) '#'
00398             || r[1] == (UBYTE) '@' ) ) {
00399                 if ( r[1] == (UBYTE) '#' ) {
00400                     out = NumCopy(num,out);
00401                 }
00402                 else {
00403                     out = NumCopy(num-dict->elements[i]->lhs[0]+1,out);
00404                 }
00405                 r += 2;
00406             }
00407             else {
00408                 *out++ = *r++;
00409             }
00410         }
00411         *out = 0;
00412         return((UBYTE *) (AT.WorkPointer));
00413     }
00414 }
00415 }
00416 }
00417
00418 out = StrCopy((UBYTE *)AC.extrasym,out);
00419 if ( AC.extrasymbols == 0 ) {
00420     out = NumCopy(num,out);
00421     out = StrCopy((UBYTE *) "_",out);
00422 }
00423 else if ( AC.extrasymbols == 1 ) {
00424     out = AddArrayIndex(num,out);
00425 }
00426 return((UBYTE *) (AT.WorkPointer));
00427 }
00428
00429 /*
00430 #] FindExtraSymbol :
00431 #[ FindDictionary :
00432 */
00433
00434 int FindDictionary(UBYTE *name)
00435 {
00436     int i;
00437     for ( i = 0; i < AO.NumDictionaries; i++ ) {
00438         if ( StrCmp(AO.Dictionaries[i]->name,name) == 0 )
00439             return(i+1);
00440     }
00441     return(0);
00442 }
00443
00444 /*
00445 #] FindDictionary :
00446 #[ AddDictionary :
00447 */
00448
00449 int AddDictionary(UBYTE *name)
00450 {
00451     DICTIONARY *dict;
00452     /*
00453     First make space for the pointer in the list.
00454     */
00455     if ( AO.NumDictionaries >= AO.SizeDictionaries-1 ) {
00456         DICTIONARY **d;
00457         int i;
00458         if ( AO.SizeDictionaries <= 0 ) AO.SizeDictionaries = 10;
00459         else AO.SizeDictionaries = 2*AO.SizeDictionaries;
00460         d = (DICTIONARY **)Malloc1(AO.SizeDictionaries*sizeof(DICTIONARY *),"Dictionaries");
00461         for ( i = 0; i < AO.NumDictionaries; i++ ) d[i] = AO.Dictionaries[i];
00462         if ( AO.Dictionaries != 0 ) M_free(AO.Dictionaries,"Dictionaries");
00463         AO.Dictionaries = d;
00464     }
00465     /*
00466     Now create an empty dictionary.
00467     */
00468     dict = (DICTIONARY *)Malloc1(sizeof(DICTIONARY),"Dictionary");
00469     AO.Dictionaries[AO.NumDictionaries++] = dict;
00470     dict->elements = 0;
00471     dict->name = strDup1(name,"DictionaryName");
00472     dict->sizeelements = 0;
00473     dict->numelements = 0;
00474     dict->numbers = 0;
00475     dict->variables = 0;
00476     dict->characters = 0;
00477     dict->funwith = 0;
00478     dict->gnumelements = 0;

```

```

00479     dict->ranges = 0;
00480
00481     return(AO.NumDictionaries);
00482 }
00483
00484 /*
00485  #] AddDictionary :
00486  #[ AddToDictionary :
00487
00488      To be called from #add left:right
00489 */
00490
00491 int AddToDictionary(DICTIONARY *dict, UBYTE *left, UBYTE *right)
00492 {
00493     GETIDENTITY
00494     CBUF *C = cbuf+AC.cbufnum;
00495     WORD *w = AT.WorkPointer;
00496     WORD *OldWork = AT.WorkPointer;
00497     WORD *s, oldnumrhs = C->numrhs, oldnumlhs = C->numlhs;
00498     WORD *ow, *ww, *mm, oldEside, *where = 0, type, number, range[3];
00499     LONG oldcpointer;
00500     int error = 0, sizelhs, sizerhs, i, retcode;
00501     UBYTE *r;
00502     DICTIONARY_ELEMENT *new;
00503     WORD power = (WORD)('^'), times = (WORD)('*');
00504     if ( ( left[0] == '^' && left[1] == 0 )
00505         || ( left[0] == '*' && left[1] == '*' && left[2] == 0 ) ) {
00506         type = DICT_SPECIALCHARACTER;
00507         number = 1;
00508         where = &power;
00509         goto TestDouble;
00510     }
00511     else if ( left[0] == '*' && left[1] == 0 ) {
00512         type = DICT_SPECIALCHARACTER;
00513         number = 1;
00514         where = &times;
00515         goto TestDouble;
00516     }
00517     else if ( left[0] == '(' ) { /* range of extra symbols */
00518         WORD x1 = 0, x2 = 0;
00519         r = left+1;
00520         while ( FG.cTable[*r] == 1 ) x1 = 10*x1 + *r++ - '0';
00521         if ( *r == ',' ) {
00522             r++;
00523             while ( FG.cTable[*r] == 1 ) x2 = 10*x2 + *r++ - '0';
00524         }
00525         else x2 = x1;
00526         number = 2;
00527         if ( *r != ')' ) {
00528             MesPrint("&Illegal range specification in LHS of %#add instruction.");
00529             return(1);
00530         }
00531         type = DICT_RANGE;
00532         if ( x1 <= 0 || x2 <= 0 || x1 > x2 ) {
00533             MesPrint("&Illegal range in LHS of %#add instruction.");
00534             return(1);
00535         }
00536         range[0] = x1;
00537         range[1] = x2;
00538         range[2] = 0;
00539         where = range;
00540         goto TestDouble;
00541     }
00542 /*
00543     Translate the left part. Determine type.
00544     We follow the code in ColdExpression and then veto what we do not like.
00545     Just make sure to pop what needs to be popped in the compiler buffer.
00546 */
00547     AC.ProtoType = w;
00548     *w++ = SUBEXPRESSION;
00549     *w++ = SUBEXPSIZE;
00550     *w++ = C->numrhs+1;
00551     *w++ = 1;
00552     *w++ = AC.cbufnum;
00553     FILLSUB(w)
00554     AC.WildC = w;
00555     AC.NwildC = 0;
00556     AT.WorkPointer = s = w + 4*AM.MaxWildcards + 8;
00557 /*
00558     Now read the LHS
00559 */
00560     oldcpointer = AddLHS(AC.cbufnum) - C->Buffer;
00561
00562     if ( ( retcode = CompileAlgebra(left, LHSIDE, AC.ProtoType) ) < 0 ) { error = 1; }
00563     else AC.ProtoType[2] = retcode;
00564     AT.WorkPointer = s;
00565     if ( AC.NwildC && SortWild(w, AC.NwildC) ) error = 1;

```

```

00566
00567     OldWork[1] = AC.WildC-OldWork;
00568     w = AC.WildC;
00569     AT.WorkPointer = w;
00570     s = C->rhs[C->numrhs];
00571 /*
00572     We have the expression in the compiler buffers.
00573     The main level is at lhs[numlhs]
00574     The partial lhs (including ProtoType) is in OldWork (in Workspace)
00575     We need to load the result at w after the prototype
00576     Because these sort routines don't use the Workspace
00577     there should not be a conflict
00578 */
00579     if ( !error && *s == 0 ) {
00580 IllLeft:MesPrint("&Illegal LHS in dictionary");
00581         AC.lhdollarflag = 0;
00582         return(1);
00583     }
00584     if ( !error && *(s+s) != 0 ) {
00585         MesPrint("&LHS in dictionary should be one term only");
00586         return(1);
00587     }
00588     if ( error == 0 ) {
00589         if ( NewSort(BHEAD0) || NewSort(BHEAD0) ) {
00590             if ( !error ) error = 1;
00591             return(error);
00592         }
00593         AN.RepPoint = AT.RepCount + 1;
00594         ow = (WORD *) ((UBYTE *) (AT.WorkPointer)) + AM.MaxTer;
00595         mm = s; ww = ow; i = *mm;
00596         while ( --i >= 0 ) { *ww++ = *mm++; } AT.WorkPointer = ww;
00597         AC.lhdollarflag = 0; oldEside = AR.Eside; AR.Eside = LHSIDE;
00598         AR.Cnumlhs = C->numlhs;
00599         if ( Generator(BHEAD ow,C->numlhs) ) {
00600             AR.Eside = oldEside;
00601             LowerSortLevel(); goto IllLeft;
00602         }
00603         AR.Eside = oldEside;
00604         AT.WorkPointer = w;
00605         if ( EndSort(BHEAD w,0) < 0 ) { LowerSortLevel(); goto IllLeft; }
00606         if ( *w == 0 || *(w+w) != 0 ) {
00607             MesPrint("&LHS must be one term");
00608             AC.lhdollarflag = 0;
00609             return(1);
00610         }
00611         LowerSortLevel();
00612     }
00613     AT.WorkPointer = w + *w;
00614     AC.DumNum = 0;
00615 /*
00616     Everything is now after OldWork. We can pop the compilerbuffer.
00617     Next test for illegal things like a coefficient
00618     At this point we have:
00619     w = the term of the LHS
00620 */
00621     C->Pointer = C->Buffer + oldcpointer;
00622     C->numrhs = oldnumrhs;
00623     C->numlhs = oldnumlhs;
00624     AC.lhdollarflag = 0;
00625 /*
00626     Test for undesirables.
00627         1: wildcards
00628         2: sign
00629         3: more than one term
00630         4: composite terms
00631 */
00632     if ( AC.ProtoType[1] != SUBEXPSIZE ) {
00633         MesPrint("& Currently no wildcards allowed in dictionaries.");
00634         return(1);
00635     }
00636     if ( w[w[0]-1] < 0 ) {
00637         MesPrint("& Currently no sign allowed in dictionaries.");
00638         return(1);
00639     }
00640     if ( w[w[0]] != 0 ) {
00641         MesPrint("& More than one term in dictionary element.");
00642         return(1);
00643     }
00644     if ( w[0] == w[w[0]-1]+1 ) { /* Only coefficient */
00645         WORD *numer, *denom;
00646         WORD nsize, dsize;
00647         nsize = dsize = (w[w[0]-1]-1)/2;
00648         numer = w+1;
00649         denom = numer+nsize;
00650         while ( numer[nsize-1] == 0 ) nsize--;
00651         while ( denom[dsize-1] == 0 ) dsize--;
00652         if ( dsize == 1 && denom[0] == 1 ) {

```

```

00653         type = DICT_INTEGERNUMBER;
00654         number = nsize;
00655         where = number;
00656     }
00657     else {
00658         type = DICT_RATIONALNUMBER;
00659         number = w[0];
00660         where = w;
00661     }
00662 }
00663 else {
00664     s = w + w[0]-1;
00665     if ( s[0] != 3 || s[-1] != 1 || s[-2] != 1 ) {
00666 Compositeness:;
00667         MesPrint("& Currently no composite objects allowed in dictionaries.");
00668         return(1);
00669     }
00670     if ( w[0] != w[2]+4 ) goto Compositeness;
00671     s = w+1;
00672     switch ( *s ) {
00673     case SYMBOL:
00674         if ( s[1] != 4 || s[3] != 1 ) goto Compositeness;
00675         type = DICT_SYMBOL;
00676         number = 1;
00677         where = s+2;
00678         break;
00679     case INDEX:
00680         if ( s[1] != 3 ) goto Compositeness;
00681         if ( s[2] < 0 ) type = DICT_VECTOR;
00682         else type = DICT_INDEX;
00683         number = 1;
00684         where = s+2;
00685         break;
00686     default:
00687         if ( *s < FUNCTION ) {
00688             MesPrint("& Illegal object in dictionary.");
00689             return(1);
00690         }
00691         if ( s[1] == FUNHEAD ) {
00692             type = DICT_FUNCTION;
00693             number = 1;
00694             where = s;
00695             break;
00696         }
00697         else {
00698             type = DICT_FUNCTION_WITH_ARGUMENTS;
00699             number = s[1];
00700             where = s;
00701         }
00702         break;
00703     }
00704 }
00705 TestDouble:;
00706 /*
00707  Create a new element
00708 */
00709 if ( dict->numelements >= dict->sizeelements ) {
00710     DICTIONARY_ELEMENT **d;
00711     if ( dict->sizeelements <= 0 ) dict->sizeelements = 10;
00712     else dict->sizeelements *= 2;
00713     d = (DICTIONARY_ELEMENT **)Malloc1(
00714         sizeof(DICTIONARY_ELEMENT *)*dict->sizeelements,"Dictionary elements");
00715     for ( i = 0; i < dict->numelements; i++ )
00716         d[i] = dict->elements[i];
00717     if ( dict->elements ) M_free(dict->elements,"Dictionary elements");
00718     dict->elements = d;
00719 }
00720 sizelhs = number+1;
00721 sizerhs = 1; r = right; while ( *r++ ) sizerhs++;
00722 sizerhs = (sizerhs+sizeof(WORD)-1)/sizeof(WORD)+1;
00723 new = (DICTIONARY_ELEMENT *)Malloc1(sizeof(DICTIONARY_ELEMENT)
00724     +sizeof(WORD)*(sizelhs+sizerhs),"Dictionary element");
00725 new->lhs = (WORD *) (new+1);
00726 new->rhs = new->lhs+sizelhs;
00727 new->type = type;
00728 new->size = number;
00729 for ( i = 0; i < number; i++ ) new->lhs[i] = where[i];
00730 new->lhs[i] = 0;
00731 r = (UBYTE *) (new->rhs);
00732 while ( *right ) {
00733     if ( *right == '\\' && ( right[1] == '\'' || right[1] == '\"' ) ) right++;
00734     *r++ = *right++;
00735 }
00736 *r = 0;
00737
00738 dict->elements[dict->numelements++] = new;
00739

```

```

00740     switch ( type ) {
00741         case DICT_INTEGERNUMBER:
00742         case DICT_RATIONALNUMBER:
00743             dict->numbers++; break;
00744         case DICT_SYMBOL:
00745         case DICT_VECTOR:
00746         case DICT_INDEX:
00747         case DICT_FUNCTION:
00748             dict->variables++; break;
00749         case DICT_FUNCTION_WITH_ARGUMENTS:
00750             dict->funwith++; break;
00751         case DICT_SPECIALCHARACTER:
00752             dict->characters++; break;
00753         case DICT_RANGE:
00754             dict->ranges++; break;
00755     }
00756
00757     AT.WorkPointer = OldWork;
00758     return(0);
00759 }
00760
00761 /*
00762  #] AddToDictionary :
00763  #[ UseDictionary :
00764  */
00765
00766 int UseDictionary(UBYTE *name,UBYTE *options)
00767 {
00768     int i;
00769     for ( i = 0; i < AO.NumDictionaries; i++ ) {
00770         if ( StrCmp(AO.Dictionaries[i]->name,name) == 0 ) {
00771             AO.CurrentDictionary = i+1;
00772             if ( SetDictionaryOptions(options) < 0 ) {
00773                 AO.CurrentDictionary = 0;
00774                 return(-1);
00775             }
00776             else { /* Now test whether what is requested is really there? */
00777                 return(0);
00778             }
00779         }
00780     }
00781     MesPrint("@There is no dictionary with the name %s",name);
00782     exit(-1);
00783 }
00784
00785 /*
00786  #] UseDictionary :
00787  #[ SetDictionaryOptions :
00788  */
00789
00790 int SetDictionaryOptions(UBYTE *options)
00791 {
00792     UBYTE *opt, *s, c;
00793     int retval = 0;
00794     s = options;
00795     AO.CurDictNumbers = DICT_ALLNUMBERS;
00796     AO.CurDictVariables = DICT_DOVARIABLES;
00797     AO.CurDictSpecials = DICT_DOSPECIALS;
00798     AO.CurDictFunWithArgs = DICT_DOFUNWITHARGS;
00799     AO.CurDictNumberWarning = 0;
00800     AO.CurDictNotInFunctions= 0;
00801     AO.CurDictInDollars = DICT_NOTINDOLLARS;
00802     while ( *s ) {
00803         opt = s;
00804         while ( *s && *s != ',' ) s++;
00805         c = *s; *s = 0;
00806         if ( opt[0] == '$' && opt[1] == 0 ) {
00807             AO.CurDictInDollars = DICT_INDOLLARS;
00808         }
00809         else if ( StrICmp(opt,(UBYTE *)"nonumbers") == 0 ) {
00810             AO.CurDictNumbers = DICT_NONUMBERS;
00811         }
00812         else if ( StrICmp(opt,(UBYTE *)"integersonly") == 0 ) {
00813             AO.CurDictNumbers = DICT_INTEGERONLY;
00814         }
00815         else if ( StrICmp(opt,(UBYTE *)"rationalsoonly") == 0 ) {
00816             AO.CurDictNumbers = DICT_RATIONALONLY;
00817         }
00818         else if ( StrICmp(opt,(UBYTE *)"allnumbers") == 0 ) {
00819             AO.CurDictNumbers = DICT_ALLNUMBERS;
00820         }
00821         else if ( StrICmp(opt,(UBYTE *)"novariables") == 0 ) {
00822             AO.CurDictVariables = DICT_NOVARIABLES;
00823         }
00824         else if ( StrICmp(opt,(UBYTE *)"numbersonly") == 0 ) {
00825             AO.CurDictNumbers = DICT_ALLNUMBERS;
00826             AO.CurDictVariables = DICT_NOVARIABLES;

```

```

00827         AO.CurDictSpecials = DICT_NOSPECIALS;
00828         AO.CurDictFunWithArgs = DICT_NOFUNWITHARGS;
00829     }
00830     else if ( StrICmp(opt, (UBYTE *)"variablesonly") == 0 ) {
00831         AO.CurDictNumbers = DICT_NONUMBERS;
00832         AO.CurDictVariables = DICT_DOVARIABLES;
00833         AO.CurDictSpecials = DICT_NOSPECIALS;
00834         AO.CurDictFunWithArgs = DICT_NOFUNWITHARGS;
00835     }
00836     else if ( StrICmp(opt, (UBYTE *)"nospecials") == 0 ) {
00837         AO.CurDictSpecials = DICT_NOSPECIALS;
00838     }
00839     else if ( StrICmp(opt, (UBYTE *)"specialsonly") == 0 ) {
00840         AO.CurDictNumbers = DICT_NONUMBERS;
00841         AO.CurDictVariables = DICT_NOVARIABLES;
00842         AO.CurDictSpecials = DICT_DOSPECIALS;
00843         AO.CurDictFunWithArgs = DICT_NOFUNWITHARGS;
00844     }
00845     else if ( StrICmp(opt, (UBYTE *)"nofunwithargs") == 0 ) {
00846         AO.CurDictFunWithArgs = DICT_NOFUNWITHARGS;
00847     }
00848     else if ( StrICmp(opt, (UBYTE *)"funwithargsonly") == 0 ) {
00849         AO.CurDictNumbers = DICT_NONUMBERS;
00850         AO.CurDictVariables = DICT_NOVARIABLES;
00851         AO.CurDictSpecials = DICT_NOSPECIALS;
00852         AO.CurDictFunWithArgs = DICT_DOFUNWITHARGS;
00853     }
00854     else if ( StrICmp(opt, (UBYTE *)"warnings") == 0
00855              || StrICmp(opt, (UBYTE *)"warning") == 0 ) {
00856         AO.CurDictNumberWarning = 1;
00857     }
00858     else if ( StrICmp(opt, (UBYTE *)"nowarnings") == 0
00859              || StrICmp(opt, (UBYTE *)"nowarning") == 0 ) {
00860         AO.CurDictNumberWarning = 0;
00861     }
00862     else if ( StrICmp(opt, (UBYTE *)"infunctions") == 0 ) {
00863         AO.CurDictNotInFunctions= 0;
00864     }
00865     else if ( StrICmp(opt, (UBYTE *)"notinfunctions") == 0 ) {
00866         AO.CurDictNotInFunctions= 1;
00867     }
00868     else {
00869         MesPrint("@ Unrecognized option in %#SetDictionary: %s",opt);
00870         retval = -1;
00871     }
00872     *s = c;
00873     if ( c == ',' ) s++;
00874 }
00875 return(retval);
00876 }
00877
00878 /*
00879  #] SetDictionaryOptions :
00880  #[ UnSetDictionary :
00881  */
00882
00883 void UnSetDictionary(void)
00884 {
00885     AO.CurrentDictionary = 0;
00886     AO.CurDictNumbers = -1;
00887     AO.CurDictVariables = -1;
00888     AO.CurDictSpecials = -1;
00889     AO.CurDictFunWithArgs = -1;
00890     AO.CurDictFunWithArgs = -1;
00891     AO.CurDictNumberWarning = -1;
00892     AO.CurDictNotInFunctions= -1;
00893 }
00894
00895 /*
00896  #] UnSetDictionary :
00897  #[ RemoveDictionary :
00898
00899     Mostly needed for .clear
00900  */
00901
00902 void RemoveDictionary(DICTIONARY *dict)
00903 {
00904     int i;
00905     if ( dict == 0 ) return;
00906     for ( i = 0; i < AO.NumDictionaries; i++ ) {
00907         if ( AO.Dictionaries[i] == dict ) {
00908             for ( i++; i < AO.NumDictionaries; i++ ) {
00909                 AO.Dictionaries[i-1] = AO.Dictionaries[i];
00910             }
00911             AO.NumDictionaries--;
00912             goto removeit;
00913         }

```

```

00914     }
00915     MesPrint("@ Dictionary not found in RemoveDictionary");
00916     exit(-1);
00917 removeit:;
00918     for ( i = 0; i < dict->numelements; i++ )
00919         M_free(dict->elements[i],"Dictionary element");
00920     for ( i = 0; i < dict->numelements; i++ ) dict->elements[i] = 0;
00921     if ( dict->elements ) M_free(dict->elements,"Dictionary elements");
00922     if ( dict->name ) {
00923         M_free(dict->name,"DictionaryName");
00924         dict->name = 0;
00925     }
00926     dict->sizeelements = 0;
00927     dict->numelements = 0;
00928     dict->numbers = 0;
00929     dict->variables = 0;
00930     dict->characters = 0;
00931     dict->funwith = 0;
00932     dict->gnumelements = 0;
00933     dict->ranges = 0;
00934 }
00935
00936 /*
00937 #] RemoveDictionary :
00938 #[ ShrinkDictionary :
00939
00940     To be called after a .store to restore the dictionary to the state
00941     it had at the last .global
00942     We do not make the elements array shorter.
00943 */
00944
00945 void ShrinkDictionary(DICTIONARY *dict)
00946 {
00947     while ( dict->numelements > dict->gnumelements ) {
00948         dict->numelements--;
00949         M_free(dict->elements[dict->numelements],"Dictionary element");
00950         dict->elements[dict->numelements] = 0;
00951     }
00952 }
00953
00954 /*
00955 #] ShrinkDictionary :
00956 #[ DoPreOpenDictionary :
00957 */
00958
00959 int DoPreOpenDictionary(UBYTE *s)
00960 {
00961     UBYTE *name;
00962     int dict;
00963     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
00964     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
00965     while ( *s == ' ' ) s++;
00966
00967     name = s; s = SkipAName(s);
00968     if ( *s != 0 && *s != ';' ) {
00969         MesPrint("@proper syntax is #opendictionary name");
00970         return(-1);
00971     }
00972     *s = 0;
00973
00974     if ( AP.OpenDictionary > 0 ) {
00975         MesPrint("@you cannot nest #opendictionary instructions");
00976         MesPrint("@dictionary %s is open already",
00977             AO.Dictionaries[AP.OpenDictionary-1]->name);
00978         return(-1);
00979     }
00980     if ( AO.CurrentDictionary > 0 ) {
00981         MesPrint("@before opening a dictionary you have to first close the selected dictionary");
00982         return(-1);
00983     }
00984 /*
00985     Do we have this dictionary already?
00986 */
00987     dict = FindDictionary(name);
00988     if ( dict == 0 ) dict = AddDictionary(name);
00989     AP.OpenDictionary = dict;
00990     return(0);
00991 }
00992
00993 /*
00994 #] DoPreOpenDictionary :
00995 #[ DoPreCloseDictionary :
00996 */
00997
00998 int DoPreCloseDictionary(UBYTE *s)
00999 {
01000     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);

```



```

01001     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
01002     while ( *s == ' ' ) s++;
01003
01004     if ( AP.OpenDictionary == 0 && AO.CurrentDictionary == 0 ) {
01005         MesPrint("@you have neither an open, nor a selected dictionary");
01006         return(-1);
01007     }
01008
01009     AP.OpenDictionary = 0;
01010     AO.CurrentDictionary = 0;
01011
01012     AO.CurDictNotInFunctions = 0;
01013
01014     return(0);
01015 }
01016
01017 /*
01018 #] DoPreCloseDictionary :
01019 #[ DoPreUseDictionary :
01020 */
01021
01022 int DoPreUseDictionary(UBYTE *s)
01023 {
01024     UBYTE *options, c, *ss, *sss, *name;
01025     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
01026     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
01027     while ( *s == ' ' ) s++;
01028
01029     if ( AP.OpenDictionary > 0 ) {
01030         MesPrint("@before selecting a dictionary you have to first close the open dictionary");
01031         return(-1);
01032     }
01033
01034     name = s; s = SkipAName(s);
01035     ss = s; while ( *s && *s != ' ' ) s++;
01036     c = *ss; *ss = 0;
01037     if ( c == 0 ) {
01038         options = ss;
01039     }
01040     else {
01041         options = s+1; SKIPBRA3(s)
01042         if ( *s != ')' ) {
01043             MesPrint("@Irregular end of %#UseDictionary instruction");
01044             return(-1);
01045         }
01046         sss = s;
01047         s++; while ( *s == ' ' || *s == '\t' || *s == ';' ) s++;
01048         *sss = 0;
01049         if ( *s ) {
01050             MesPrint("@Irregular end of %#UseDictionary instruction");
01051             return(-1);
01052         }
01053     }
01054     return(UseDictionary(name,options));
01055 }
01056
01057 /*
01058 #] DoPreUseDictionary :
01059 #[ DoPreAdd :
01060
01061 Syntax:
01062     #add left :right
01063     #add left : "right"
01064 Adds to the currently open dictionary
01065 */
01066
01067 int DoPreAdd(UBYTE *s)
01068 {
01069     UBYTE *left, *right;
01070
01071     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
01072     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
01073     while ( *s == ' ' ) s++;
01074
01075     if ( AP.OpenDictionary == 0 ) {
01076         MesPrint("@there is no open dictionary to add to");
01077         return(-1);
01078     }
01079     /*
01080 Scan to the : and mark the left and right parts.
01081 */
01082     left = s;
01083     while ( *s && *s != ':' ) {
01084         if ( *s == '[' ) { SKIPBRA1(s) s++; }
01085         else if ( *s == '{' ) { SKIPBRA2(s) s++; }
01086         else if ( *s == '(' ) { SKIPBRA3(s) s++; }
01087         else if ( *s == ']' || *s == '}' || *s == ')' ) {

```

```

01088         MesPrint("@unmatched brackets in #add instruction");
01089         return(-1);
01090     }
01091     else s++;
01092 }
01093 if ( *s == 0 ) {
01094     MesPrint("@Missing : in #add instruction");
01095     return(-1);
01096 }
01097 *s++ = 0;
01098 right = s;
01099 while ( *s == ' ' || *s == '\t' ) s++;
01100 if ( *s == '"' && s[1] ) {
01101     right = s+1;
01102     s = s+2;
01103     while ( *s ) s++;
01104     while ( s[-1] != '"' ) s--;
01105     if ( s <= right ) {
01106         MesPrint("@Irregular use of double quotes in #add instruction");
01107         return(-1);
01108     }
01109     s[-1] = 0;
01110 }
01111 return(AddToDictionary(AO.Dictionaries[AP.OpenDictionary-1],left,right));
01112 }
01113
01114 /*
01115 #] DoPreAdd :
01116 #[ DictToBytes :
01117 */
01118
01119 LONG DictToBytes(DICTIONARY *dict,UBYTE *buf)
01120 {
01121     int numelements = dict->numelements, sizeelement, i, j, x;
01122     UBYTE *s1, *s2 = buf;
01123     DICTIONARY_ELEMENT *e;
01124     /*
01125     First copy the struct
01126     */
01127     s1 = (UBYTE *)dict; j = sizeof(DICTIONARY);
01128     NCOPY(s2,s1,j)
01129     /*
01130     Now the elements. Put a size indicator in front of each of them.
01131     */
01132     for ( i = 0; i < numelements; i++ ) {
01133         e = dict->elements[i];
01134         sizeelement = sizeof(DICTIONARY_ELEMENT)+(e->size+1)*sizeof(WORD);
01135         s1 = (UBYTE *)e->rhs; x = 0;
01136         while ( *s1 ) { s1++; x++; }
01137         x /= sizeof(WORD);
01138         sizeelement += (x+1) * sizeof(WORD);
01139         s1 = (UBYTE *)&sizeelement; j = sizeof(WORD); NCOPY(s2,s1,j)
01140         s1 = (UBYTE *)e; j = sizeof(DICTIONARY_ELEMENT); NCOPY(s2,s1,j)
01141         s1 = (UBYTE *)e->lhs; j = (e->size+1)*(sizeof(WORD)); NCOPY(s2,s1,j)
01142         s1 = (UBYTE *)e->rhs; j = (x+1)*(sizeof(WORD)); NCOPY(s2,s1,j)
01143     }
01144     return(s2-buf);
01145 }
01146
01147 /*
01148 #] DictToBytes :
01149 #[ DictFromBytes :
01150 */
01151
01152 DICTIONARY *DictFromBytes(UBYTE *buf)
01153 {
01154     DICTIONARY *dict = Malloc1(sizeof(DICTIONARY),"Dictionary");
01155     UBYTE *s1, *s2;
01156     int i, j, sizeelement;
01157     DICTIONARY_ELEMENT *e;
01158     /*
01159     First read the dictionary itself
01160     */
01161     s1 = buf;
01162     s2 = (UBYTE *)dict; j = sizeof(DICTIONARY); NCOPY(s2,s1,j)
01163     /*
01164     Allocate the elements array:
01165     */
01166     dict->elements = (DICTIONARY_ELEMENT **)Malloc1(
01167         sizeof(DICTIONARY_ELEMENT *)*dict->sizeelements,"dictionary elements");
01168     for ( i = 0; i < dict->numelements; i++ ) {
01169         s2 = (UBYTE *)&sizeelement; j = sizeof(WORD); NCOPY(s2,s1,j)
01170         e = (DICTIONARY_ELEMENT *)Malloc1(sizeelement*sizeof(UBYTE),"dictionary element");
01171         dict->elements[i] = e;
01172         j = sizeelement; s2 = (UBYTE *)e; NCOPY(s2,s1,j)
01173         e->lhs = (WORD *) (e+1);
01174         e->rhs = e->lhs + e->size+1;

```

```

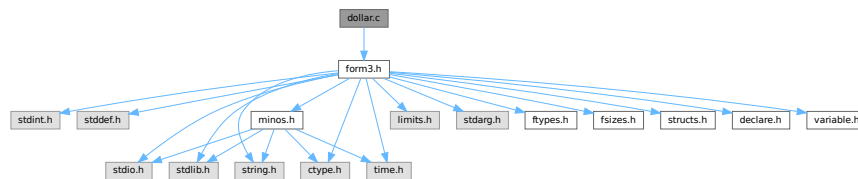
01175     }
01176     return(dict);
01177 }
01178
01179 /*
01180  #] DictFromBytes :
01181  */

```

4.27 dollar.c File Reference

```
#include "form3.h"
```

Include dependency graph for dollar.c:



Macros

- `#define` [STEP2](#)

Functions

- `int` [CatchDollar](#) (`int` par)
- `int` [AssignDollar](#) (`PHEAD WORD *term`, `WORD level`)
- `UBYTE *` [WriteDollarToBuffer](#) (`WORD numdollar`, `WORD par`)
- `UBYTE *` [WriteDollarFactorToBuffer](#) (`WORD numdollar`, `WORD numfac`, `WORD par`)
- `void` [AddToDollarBuffer](#) (`UBYTE *s`)
- `void` [TermAssign](#) (`WORD *term`)
- `int` [PutTermInDollar](#) (`WORD *term`, `WORD numdollar`)
- `void` [WildDollars](#) (`PHEAD WORD *term`)
- `WORD` [DolToTensor](#) (`PHEAD WORD numdollar`)
- `WORD` [DolToFunction](#) (`PHEAD WORD numdollar`)
- `WORD` [DolToVector](#) (`PHEAD WORD numdollar`)
- `WORD` [DolToNumber](#) (`PHEAD WORD numdollar`)
- `WORD` [DolToSymbol](#) (`PHEAD WORD numdollar`)
- `WORD` [DolToIndex](#) (`PHEAD WORD numdollar`)
- `DOLLARS` [DolToTerms](#) (`PHEAD WORD numdollar`)
- `LONG` [DolToLong](#) (`PHEAD WORD numdollar`)
- `int` [ExecInside](#) (`UBYTE *s`)
- `int` [InsideDollar](#) (`PHEAD WORD *ll`, `WORD level`)
- `void` [ExchangeDollars](#) (`int num1`, `int num2`)
- `LONG` [TermsInDollar](#) (`WORD num`)
- `LONG` [SizeOfDollar](#) (`WORD num`)
- `UBYTE *` [PrelfDollarEval](#) (`UBYTE *s`, `int *value`)
- `WORD *` [TranslateExpression](#) (`UBYTE *s`)
- `int` [IsSetMember](#) (`WORD *buffer`, `WORD numset`)
- `int` [IsMultipleOf](#) (`WORD *buf1`, `WORD *buf2`)

- int [TwoExprCompare](#) (WORD *buf1, WORD *buf2, int oprtr)
- int [DollarRaiseLow](#) (UBYTE *name, LONG value)
- WORD [EvalDoLoopArg](#) (PHEAD WORD *arg, WORD par)
- WORD [TestDoLoop](#) (PHEAD WORD *lhsbuf, WORD level)
- WORD [TestEndDoLoop](#) (PHEAD WORD *lhsbuf, WORD level)
- int [DollarFactorize](#) (PHEAD WORD numdollar)
- void [CleanDollarFactors](#) (DOLLARS d)
- WORD * [TakeDollarContent](#) (PHEAD WORD *dollarbuffer, WORD **factor)
- WORD * [MakeDollarInteger](#) (PHEAD WORD *bufin, WORD **bufout)
- WORD * [MakeDollarMod](#) (PHEAD WORD *buffer, WORD **bufout)
- int [GetDolNum](#) (PHEAD WORD *t, WORD *tstop)
- void [AddPotModdollar](#) (WORD numdollar)

4.27.1 Detailed Description

The routines that deal with the dollar variables. The name administration is to be found in the file [names.c](#)

Definition in file [dollar.c](#).

4.27.2 Macro Definition Documentation

4.27.2.1 STEP2

```
#define STEP2
```

Factors a dollar expression. Notation: d->nfactors becomes nonzero. if the number of factors is one, we leave d->factors zero. Otherwise factors is an array of pointers to the factors. These are pointers of the type FAC-DOLLAR. fd->where pointer to contents in term notation fd->size size of the buffer fd->where points to fd->type DOLNUMBER or DOLTERMS fd->value value if type is DOLNUMBER and it fits in a WORD.

Definition at line [2953](#) of file [dollar.c](#).

4.27.3 Function Documentation

4.27.3.1 CatchDollar()

```
int CatchDollar (
    int par )
```

Definition at line [54](#) of file [dollar.c](#).

4.27.3.2 AssignDollar()

```
int AssignDollar (
    PHEAD WORD * term,
    WORD level )
```

Definition at line [209](#) of file [dollar.c](#).

4.27.3.3 WriteDollarToBuffer()

```
UBYTE * WriteDollarToBuffer (
    WORD numdollar,
    WORD par )
```

Definition at line [596](#) of file [dollar.c](#).

4.27.3.4 WriteDollarFactorToBuffer()

```
UBYTE * WriteDollarFactorToBuffer (
    WORD numdollar,
    WORD numfac,
    WORD par )
```

Definition at line [687](#) of file [dollar.c](#).

4.27.3.5 AddToDollarBuffer()

```
void AddToDollarBuffer (
    UBYTE * s )
```

Definition at line [762](#) of file [dollar.c](#).

4.27.3.6 TermAssign()

```
void TermAssign (
    WORD * term )
```

Definition at line [803](#) of file [dollar.c](#).

4.27.3.7 PutTermInDollar()

```
int PutTermInDollar (
    WORD * term,
    WORD numdollar )
```

Definition at line [863](#) of file [dollar.c](#).

4.27.3.8 WildDollars()

```
void WildDollars (
    PHEAD WORD * term )
```

Definition at line [891](#) of file [dollar.c](#).

4.27.3.9 DolToTensor()

```
WORD DolToTensor (
    PHEAD WORD numdollar )
```

Definition at line [1082](#) of file [dollar.c](#).

4.27.3.10 DolToFunction()

```
WORD DolToFunction (
    PHEAD WORD numdollar )
```

Definition at line [1143](#) of file [dollar.c](#).

4.27.3.11 DolToVector()

```
WORD DolToVector (
    PHEAD WORD numdollar )
```

Definition at line [1200](#) of file [dollar.c](#).

4.27.3.12 DolToNumber()

```
WORD DolToNumber (
    PHEAD WORD numdollar )
```

Definition at line [1264](#) of file [dollar.c](#).

4.27.3.13 DolToSymbol()

```
WORD DolToSymbol (
    PHEAD WORD numdollar )
```

Definition at line [1323](#) of file [dollar.c](#).

4.27.3.14 DolToIndex()

```
WORD DolToIndex (
    PHEAD WORD numdollar )
```

Definition at line [1377](#) of file [dollar.c](#).

4.27.3.15 DolToTerms()

```
DOLLARS DolToTerms (
    PHEAD WORD numdollar )
```

Definition at line [1458](#) of file [dollar.c](#).

4.27.3.16 DolToLong()

```
LONG DolToLong (
    PHEAD WORD numdollar )
```

Definition at line [1606](#) of file [dollar.c](#).

4.27.3.17 ExecInside()

```
int ExecInside (
    UBYTE * s )
```

Definition at line [1681](#) of file [dollar.c](#).

4.27.3.18 InsideDollar()

```
int InsideDollar (
    PHEAD WORD * ll,
    WORD level )
```

Definition at line [1746](#) of file [dollar.c](#).

4.27.3.19 ExchangeDollars()

```
void ExchangeDollars (
    int num1,
    int num2 )
```

Definition at line [1849](#) of file [dollar.c](#).

4.27.3.20 TermsInDollar()

```
LONG TermsInDollar (
    WORD num )
```

Definition at line [1867](#) of file [dollar.c](#).

4.27.3.21 SizeOfDollar()

```
LONG SizeOfDollar (
    WORD num )
```

Definition at line [1916](#) of file [dollar.c](#).

4.27.3.22 PreIfDollarEval()

```
UBYTE * PreIfDollarEval (
    UBYTE * s,
    int * value )
```

Definition at line [1978](#) of file [dollar.c](#).

4.27.3.23 TranslateExpression()

```
WORD * TranslateExpression (
    UBYTE * s )
```

Definition at line [2160](#) of file [dollar.c](#).

4.27.3.24 IsSetMember()

```
int IsSetMember (
    WORD * buffer,
    WORD numset )
```

Definition at line [2217](#) of file [dollar.c](#).

4.27.3.25 IsMultipleOf()

```
int IsMultipleOf (
    WORD * buf1,
    WORD * buf2 )
```

Definition at line [2398](#) of file [dollar.c](#).

4.27.3.26 TwoExprCompare()

```
int TwoExprCompare (
    WORD * buf1,
    WORD * buf2,
    int oprtr )
```

Definition at line [2473](#) of file [dollar.c](#).

4.27.3.27 DollarRaiseLow()

```
int DollarRaiseLow (
    UBYTE * name,
    LONG value )
```

Definition at line [2552](#) of file [dollar.c](#).

4.27.3.28 EvalDoLoopArg()

```
WORD EvalDoLoopArg (  
    PHEAD WORD * arg,  
    WORD par )
```

Evaluates one argument of a do loop. Such an argument is constructed from SNUMBERS DOLLAREXPRES-
SIONs and possibly DOLLAREXPR2s which indicate factors of the preceding dollar. Hence we have SNUM-
BER,num DOLLAREXPRESSSION,numdollar DOLLAREXPRESSSION,numdollar,DOLLAREXPR2,numfactor DOL-
LAREXPRESSSION,numdollar,DOLLAREXPR2,numfactor,DOLLAREXPR2,numfactor etc. Because we have a do-
loop at every stage we should have a number. The notation in DOLLAREXPR2 is that ≥ 0 is number of yet another
dollar and < 0 is $-n-1$ with n the array element or zero. The return value is the (short) number. The routine works its
way through the list in a recursive manner.

Definition at line [2651](#) of file [dollar.c](#).

References [EvalDoLoopArg\(\)](#).

Referenced by [EvalDoLoopArg\(\)](#).

Here is the call graph for this function:



4.27.3.29 TestDoLoop()

```
WORD TestDoLoop (  
    PHEAD WORD * lhsbuf,  
    WORD level )
```

Definition at line [2762](#) of file [dollar.c](#).

4.27.3.30 TestEndDoLoop()

```
WORD TestEndDoLoop (  
    PHEAD WORD * lhsbuf,  
    WORD level )
```

Definition at line [2840](#) of file [dollar.c](#).

4.27.3.31 DollarFactorize()

```
int DollarFactorize (
    PHEAD WORD numdollar )
```

Definition at line [2955](#) of file [dollar.c](#).

4.27.3.32 CleanDollarFactors()

```
void CleanDollarFactors (
    DOLLARS d )
```

Definition at line [3554](#) of file [dollar.c](#).

4.27.3.33 TakeDollarContent()

```
WORD * TakeDollarContent (
    PHEAD WORD * dollarbuffer,
    WORD ** factor )
```

Definition at line [3576](#) of file [dollar.c](#).

4.27.3.34 MakeDollarInteger()

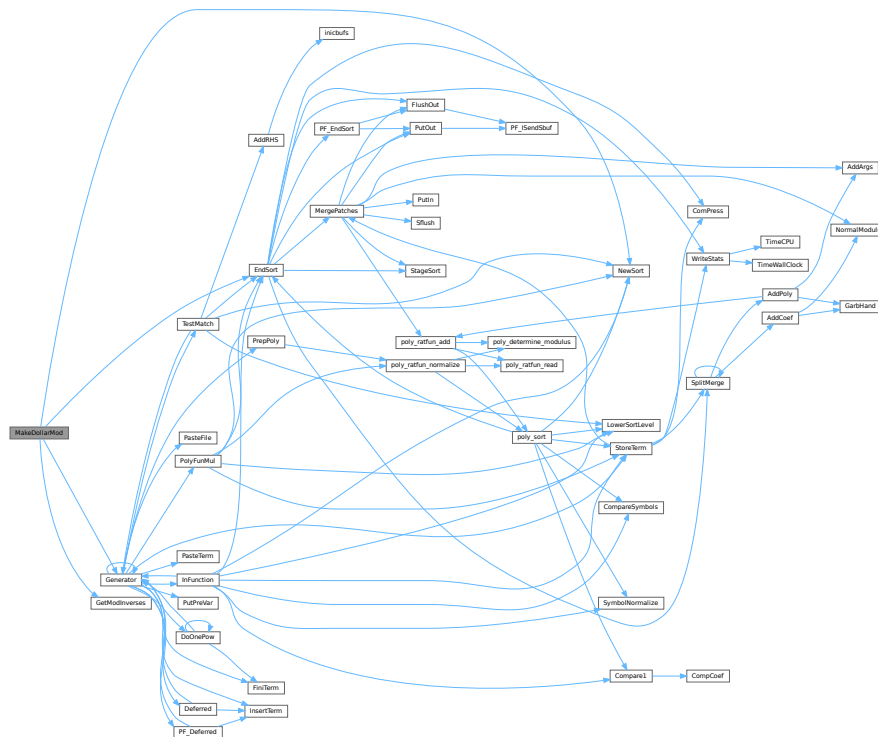
```
WORD * MakeDollarInteger (
    PHEAD WORD * bufin,
    WORD ** bufout )
```

For normalizing everything to integers we have to determine for all elements of this argument the LCM of the denominators and the GCD of the numerators. The input argument is in *bufin*. The number that comes out is the return value. The normalized argument is in *bufout*.

Definition at line [3628](#) of file [dollar.c](#).

References [EndSort\(\)](#), [Generator\(\)](#), and [NewSort\(\)](#).

Here is the call graph for this function:



4.27.3.36 GetDolNum()

```
int GetDolNum (
    PHEAD WORD * t,
    WORD * tstop )
```

Definition at line 3847 of file [dollar.c](#).

4.27.3.37 AddPotModdollar()

```
void AddPotModdollar (
    WORD numdollar )
```

Adds a \$-variable specified by *numdollar* to the list of potentially modified \$-variables unless it has already been included in the list.

Parameters

<i>numdollar</i>	The index of the \$-variable to be added.

Definition at line 3960 of file [dollar.c](#).

4.28 dollar.c

[Go to the documentation of this file.](#)

```

00001
00006 /* #[] License : */
00007 /*
00008 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00009 * When using this file you are requested to refer to the publication
00010 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011 * This is considered a matter of courtesy as the development was paid
00012 * for by FOM the Dutch physics granting agency and we would like to
00013 * be able to track its scientific use to convince FOM of its value
00014 * for the community.
00015 *
00016 * This file is part of FORM.
00017 *
00018 * FORM is free software: you can redistribute it and/or modify it under the
00019 * terms of the GNU General Public License as published by the Free Software
00020 * Foundation, either version 3 of the License, or (at your option) any later
00021 * version.
00022 *
00023 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00026 * details.
00027 *
00028 * You should have received a copy of the GNU General Public License along
00029 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00030 */
00031 /* #[] License : */
00032 /*
00033 #[] Includes :
00034 */
00035
00036 #include "form3.h"
00037
00038 /* EXTERNLOCK(dummylock) */
00039
00040 static UBYTE underscore[2] = {'_',0};
00041
00042 /*
00043 #[] Includes :
00044 #[] CatchDollar :
00045
00046 Works out a dollar expression during compile type.
00047 Steals it from the buffer and puts it in an assignment.
00048 At the moment we should keep this inside the small buffer.
00049 Later with more sort buffers we can do this better.
00050 Par == 0 : regular assignment
00051 par == -1: after error. Just make zero for now.
00052 */
00053
00054 int CatchDollar(int par)
00055 {
00056 GETIDENTITY
00057 CBUF *C = cbuf + AC.cbufnum;
00058 int error = 0, numterms = 0, numdollar, resetmods = 0;
00059 LONG newsize, retval;
00060 WORD *w, *t, n, nsize, *oldwork = AT.WorkPointer, *dbuffer;
00061 WORD oldncmod = AN.ncmod;
00062 DOLLARS d;
00063 if ( AN.ncmod && ( ( AC.modmode & ALSODOLLARS ) == 0 ) ) AN.ncmod = 0;
00064 if ( AN.ncmod && AN.cmod == 0 ) { SetMods(); resetmods = 1; }
00065
00066 numdollar = C->lhs[C->numlhs][2];
00067
00068 d = Dollars+numdollar;
00069 if ( par == -1 ) {
00070 d->type = DOLUNDEFINED;
00071 cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00072 cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
00073 if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"$-buffer old");
00074 d->size = 0; d->where = &(AM.dollarzero);
00075 cbuf[AM.dbufnum].rhs[numdollar] = d->where;
00076 AN.ncmod = oldncmod;
00077 if ( resetmods ) UnSetMods();
00078 return(0);
00079 }
00080 #ifdef WITHMPI
00081 /*
00082 * The problem here is that only the master can make an assignment
00083 * like $a=g; where g is an expression: only the master has an access to
00084 * the expression. So, in cases where the RHS contains expression names,
00085 * only the master invokes Generator() and then broadcasts the result to
00086 * the all slaves.

```

```

00087      * Broadcasting must be performed immediately; one cannot postpone it
00088      * to the end of the module because the dollar variable is visible
00089      * in the current module. For the same reason, this should be done
00090      * regardless of on/off parallel status.
00091      * If the RHS does not contain any expression names, it can be processed
00092      * in each slave.
00093      */
00094      if ( PF.me == MASTER || !AC.RhsExprInModuleFlag ) {
00095 #endif
00096
00097      EXCHINOUT
00098
00099      if ( NewSort(BHEAD0) ) { if ( !error ) error = 1; goto onerror; }
00100      if ( NewSort(BHEAD0) ) {
00101          LowerSortLevel();
00102          if ( !error ) error = 1;
00103          goto onerror;
00104      }
00105      AN.RepPoint = AT.RepCount + 1;
00106      w = C->rhs[C->lhs[C->numlhs][5]];
00107      while ( *w ) {
00108          n = *w; t = oldwork;
00109          NCOPY(t,w,n);
00110          AT.WorkPointer = t;
00111          AR.Cnumlhs = C->numlhs;
00112          if ( Generator(BHEAD oldwork,C->numlhs) ) { error = 1; break; }
00113      }
00114      AT.WorkPointer = oldwork;
00115      AN.tryterm = 0; /* for now */
00116      dbuffer = 0;
00117      if ( ( retval = EndSort(BHEAD (WORD *)((void *)&dbuffer)),2) < 0 ) { error = 1; }
00118      LowerSortLevel();
00119      if ( retval <= 1 || dbuffer == 0 ) {
00120          d->type = DOLZERO;
00121          if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"$-buffer old");
00122          d->size = 0; d->where = &(AM.dollarzero);
00123          cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00124          cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
00125          goto docopy2;
00126      }
00127      w = dbuffer;
00128      if ( error == 0 )
00129          while ( *w ) { w += *w; numterms++; }
00130      else
00131          goto onerror;
00132      newsize = (w-dbuffer)+1;
00133 #ifdef WITHMPI
00134 }
00135 if ( AC.RhsExprInModuleFlag )
00136     /* PF_BroadcastPreDollar allocates dbuffer for slaves! */
00137     if ( (error = PF_BroadcastPreDollar(&dbuffer, &newsize, &numterms)) != 0 )
00138         goto onerror;
00139 #endif
00140 if ( newsize < MINALLOC ) newsize = MINALLOC;
00141 newsize = (newsize+7)/8*8;
00142 if ( numterms == 0 ) {
00143     d->type = DOLZERO;
00144     goto docopy;
00145 }
00146 else if ( numterms == 1 ) {
00147     t = dbuffer;
00148     n = *t;
00149     nsize = t[n-1];
00150     if ( nsize < 0 ) { nsize = -nsize; }
00151     if ( nsize == (n-1) ) { /* numerical */
00152         nsize = (nsize-1)/2;
00153         w = t + 1 + nsize;
00154         if ( *w != 1 ) goto doterms;
00155         w++; while ( w < ( t + n - 1 ) ) { if ( *w ) break; w++; }
00156         if ( w < ( t + n - 1 ) ) goto doterms;
00157         d->type = DOLNUMBER;
00158         goto docopy;
00159     }
00160     else if ( n == 7 && t[6] == 3 && t[5] == 1 && t[4] == 1
00161         && t[1] == INDEX && t[2] == 3 ) {
00162         d->type = DOLINDEX;
00163         d->index = t[3];
00164         goto docopy;
00165     }
00166     else goto doterms;
00167 }
00168 else {
00169 doterms:;
00170     d->type = DOLTERMS;
00171     cbuf[AM.dbufnum].CanCommu[numdollar] = numcommute(dbuffer,
00172         &(cbuf[AM.dbufnum].NumTerms[numdollar]));
00173 docopy:;

```

```

00174         if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"$-buffer old");
00175         d->size = newsize; d->where = dbuffer;
00176 docopy2;;
00177         cbuf[AM.dbufnum].rhs[numdollar] = d->where;
00178     }
00179     if ( C->Pointer > C->rhs[C->numrhs] ) C->Pointer = C->rhs[C->numrhs];
00180     C->numlhs--; C->numrhs--;
00181 onerror:
00182 #ifdef WITHMPI
00183     if ( PF.me == MASTER || !AC.RhsExprInModuleFlag )
00184 #endif
00185     BACKINOUT
00186     AN.ncmod = oldncmod;
00187     if ( resetmods ) UnSetMods();
00188     return(error);
00189 }
00190
00191 /*
00192  #] CatchDollar :
00193  #[ AssignDollar :
00194
00195  To be called from Generator. Assigns an expression to a $ variable.
00196  This one is slightly different from CatchDollar.
00197  We have no easy buffer this time.
00198  We will have to hack our way using what we normally use for functions.
00199
00200  Note that in the threaded case we trust the user. That means that
00201  we are not going to recheck whether there is a maximum, minimum or sum.
00202  If the user says it is like that, we treat it like that.
00203  We only check that in this centralized version MODLOCAL isn't used.
00204
00205  In a later stage dtype could be used for actually checking MODMAX
00206  and MODMIN cases.
00207 */
00208
00209 int AssignDollar(PHEAD WORD *term, WORD level)
00210 {
00211     GETBIDENTITY
00212     CBUF *C = cbuf+AM.rbufnum;
00213     int numterms = 0, numdollar = C->lhs[level][2];
00214     LONG newsize;
00215     DOLLARS d = Dollars + numdollar;
00216     WORD *w, *t, n, nsize, *rh = cbuf[C->lhs[level][7]].rhs[C->lhs[level][5]];
00217     WORD *ss, *ww;
00218     WORD olddefer, oldcompress, oldncmod = AN.ncmod;
00219 #ifdef WITHPTHREADS
00220     int nummodopt, dtype = -1, dw;
00221     WORD numvalue;
00222     if ( AN.ncmod && ( ( AC.modmode & ALSODOLLARS ) == 0 ) ) AN.ncmod = 0;
00223     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
00224 /*
00225         Here we come only when the module runs with more than one thread.
00226         This must be a variable with a special module option.
00227         For the multi-threaded version we only allow MODSUM, MODMAX and MODMIN.
00228 */
00229         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
00230             if ( numdollar == ModOptdollars[nummodopt].number ) break;
00231         }
00232         if ( nummodopt >= NumModOptdollars ) {
00233             MLOCK(ErrorMessageLock);
00234             MesPrint("Illegal attempt to change $-variable in multi-threaded module %1",AC.CModule);
00235             MUNLOCK(ErrorMessageLock);
00236             Terminate(-1);
00237         }
00238         dtype = ModOptdollars[nummodopt].type;
00239         if ( dtype == MODLOCAL ) {
00240             d = ModOptdollars[nummodopt].dstruct+AT.identity;
00241         }
00242     }
00243 #endif
00244     DUMMYUSE(term);
00245     w = rh;
00246 /*
00247     First some shortcuts
00248 */
00249     if ( *w == 0 ) {
00250 /*
00251         #[ Thread version : Zero case
00252 */
00253 #ifdef WITHPTHREADS
00254         if ( dtype > 0 ) {
00255             /* LOCK(d->pthreadlockwrite); */
00256             LOCK(d->pthreadlockread);
00257 NewValIsZero;;
00258             switch ( d->type ) {
00259                 case DOLZERO: goto NoChangeZero;
00260                 case DOLNUMBER:

```

```

00261         case DOLTERMS:
00262             if ( ( dw = d->where[0] ) > 0 && d->where[dw] != 0 ) {
00263                 break; /* was not a single number. Trust the user */
00264             }
00265             if ( dtype == MODMAX && d->where[dw-1] >= 0 ) goto NoChangeZero;
00266             if ( dtype == MODMIN && d->where[dw-1] <= 0 ) goto NoChangeZero;
00267             break;
00268         default:
00269             numvalue = DolToNumber(BHEAD numdollar);
00270             if ( AN.ErrorInDollar != 0 ) break;
00271             if ( dtype == MODMAX && numvalue >= 0 ) goto NoChangeZero;
00272             if ( dtype == MODMIN && numvalue <= 0 ) goto NoChangeZero;
00273             break;
00274     }
00275     d->type = DOLZERO;
00276     d->where[0] = 0;
00277     cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00278     cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
00279 NoChangeZero:;
00280     CleanDollarFactors(d);
00281     /* UNLOCK(d->pthreadlockwrite); */
00282     UNLOCK(d->pthreadlockread);
00283     AN.ncmod = oldncmod;
00284     return(0);
00285 }
00286 #endif
00287 /*
00288     #] Thread version :
00289 */
00290     d->type = DOLZERO;
00291     d->where[0] = 0;
00292     cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00293     cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
00294     CleanDollarFactors(d);
00295     AN.ncmod = oldncmod;
00296     return(0);
00297 }
00298 else if ( *w == 4 && w[4] == 0 && w[2] == 1 ) {
00299     /*
00300         #[ Thread version : New value is 'single precision'
00301     */
00302     #ifdef WITHPTHREADS
00303         if ( dtype > 0 ) {
00304             /* LOCK(d->pthreadlockwrite); */
00305             LOCK(d->pthreadlockread);
00306             if ( d->size < MINALLOC ) {
00307                 WORD oldsize, *oldwhere, i;
00308                 oldsize = d->size; oldwhere = d->where;
00309                 d->size = MINALLOC;
00310                 d->where = (WORD *)Malloc1(d->size*sizeof(WORD), "dollar contents");
00311                 cbuf[AM.dbufnum].rhs[numdollar] = d->where;
00312                 if ( oldsize > 0 ) {
00313                     for ( i = 0; i < oldsize; i++ ) d->where[i] = oldwhere[i];
00314                 }
00315                 else d->where[0] = 0;
00316                 if ( oldwhere && oldwhere != &(AM.dollarzero) ) M_free(oldwhere, "dollar contents");
00317             }
00318             switch ( d->type ) {
00319                 case DOLZERO:
00320 HandleDolZero:;
00321                     if ( dtype == MODMAX && w[3] <= 0 ) goto NoChangeOne;
00322                     if ( dtype == MODMIN && w[3] >= 0 ) goto NoChangeOne;
00323                     break;
00324                     case DOLNUMBER:
00325                     case DOLTERMS:
00326                         if ( ( dw = d->where[0] ) > 0 && d->where[dw] != 0 ) {
00327                             break; /* was not a single number. Trust the user */
00328                         }
00329                         if ( dtype == MODMAX && CompCoef(d->where,w) >= 0 ) goto NoChangeOne;
00330                         if ( dtype == MODMIN && CompCoef(d->where,w) <= 0 ) goto NoChangeOne;
00331                         break;
00332                     default:
00333                         {
00334                             /*
00335                                 Note that we convert the type for the next time around.
00336                             */
00337                             WORD extraterm[4];
00338                             numvalue = DolToNumber(BHEAD numdollar);
00339                             if ( AN.ErrorInDollar != 0 ) break;
00340                             if ( numvalue == 0 ) {
00341                                 d->type = DOLZERO;
00342                                 d->where[0] = 0;
00343                                 cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00344                                 cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
00345                                 goto HandleDolZero;
00346                             }
00347                             d->where[0] = extraterm[0] = 4;

```



```

00348             d->where[1] = extraterm[1] = ABS(numvalue);
00349             d->where[2] = extraterm[2] = 1;
00350             d->where[3] = extraterm[3] = numvalue > 0 ? 3 : -3;
00351             d->where[4] = 0;
00352             d->type = DOLNUMBER;
00353             if ( dtype == MODMAX && CompCoef(extraterm,w) >= 0 ) goto NoChangeOne;
00354             if ( dtype == MODMIN && CompCoef(extraterm,w) <= 0 ) goto NoChangeOne;
00355             break;
00356         }
00357     }
00358     d->where[0] = w[0];
00359     d->where[1] = w[1];
00360     d->where[2] = w[2];
00361     d->where[3] = w[3];
00362     d->where[4] = 0;
00363     d->type = DOLNUMBER;
00364     cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00365     cbuf[AM.dbufnum].NumTerms[numdollar] = 1;
00366     NoChangeOne;
00367     CleanDollarFactors(d);
00368     /* UNLOCK(d->pthreadlockwrite); */
00369     UNLOCK(d->pthreadlockread);
00370     AN.ncmod = oldncmod;
00371     return(0);
00372 }
00373 #endif
00374 /*
00375     #] Thread version :
00376 */
00377     if ( d->size < MINALLOC ) {
00378         if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollar contents");
00379         d->size = MINALLOC;
00380         d->where = (WORD *)Malloc1(d->size*sizeof(WORD),"dollar contents");
00381         cbuf[AM.dbufnum].rhs[numdollar] = d->where;
00382     }
00383     d->where[0] = w[0];
00384     d->where[1] = w[1];
00385     d->where[2] = w[2];
00386     d->where[3] = w[3];
00387     d->where[4] = 0;
00388     d->type = DOLNUMBER;
00389     cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00390     cbuf[AM.dbufnum].NumTerms[numdollar] = 1;
00391     CleanDollarFactors(d);
00392     AN.ncmod = oldncmod;
00393     return(0);
00394 }
00395 /*
00396     Now the real evaluation.
00397     In the case of threads and MODSUM this requires an immediate lock.
00398     Otherwise the lock could be placed later.
00399 */
00400 #ifdef WITHPTHREADS
00401     if ( dtype == MODSUM ) {
00402         /* LOCK(d->pthreadlockwrite); */
00403         LOCK(d->pthreadlockread);
00404     }
00405 #endif
00406     CleanDollarFactors(d);
00407     /*
00408         The following case cannot occur. We treated it already
00409     */
00410     if ( *w == 0 ) {
00411         ss = 0; numterms = 0; newsize = 0;
00412         olddefer = AR.DeferFlag; AR.DeferFlag = 0;
00413         oldcompress = AR.NoCompress; AR.NoCompress = 1;
00414     }
00415     else
00416     /*
00417         {
00418         /*
00419             New value is an expression that has to be evaluated first
00420             This is all generic. It won't foliate due to the sort level
00421         */
00422         if ( NewSort(BHEAD0) ) {
00423             AN.ncmod = oldncmod;
00424             return(1);
00425         }
00426         olddefer = AR.DeferFlag; AR.DeferFlag = 0;
00427         oldcompress = AR.NoCompress; AR.NoCompress = 1;
00428         while ( *w ) {
00429             n = *w; t = ww = AT.WorkPointer;
00430             NCOPY(t,w,n);
00431             AT.WorkPointer = t;
00432             if ( Generator(BHEAD ww,AR.Cnumlhs) ) {
00433                 AT.WorkPointer = ww;
00434                 LowerSortLevel();

```

```

00435         AR.DeferFlag = olddefer;
00436         AN.ncmod = oldncmod;
00437         return(1);
00438     }
00439     AT.WorkPointer = ww;
00440 }
00441 AN.tryterm = 0; /* for now */
00442 if ( ( newsize = EndSort(BHEAD (WORD *) ((void *) (&ss)), 2) ) < 0 ) {
00443     AN.ncmod = oldncmod;
00444     return(1);
00445 }
00446 numterms = 0; t = ss; while ( *t ) { numterms++; t += *t; }
00447 }
00448 #ifdef WITHPTHREADS
00449     if ( dtype != MODSUM ) {
00450         /* LOCK(d->pthreadlockwrite); */
00451         LOCK(d->pthreadlockread);
00452     }
00453 #endif
00454     if ( numterms == 0 ) {
00455         /*
00456             the new value evaluates to zero
00457         */
00458         #ifdef WITHPTHREADS
00459             if ( dtype == MODMAX || dtype == MODMIN ) {
00460                 if ( ss ) { M_free(ss, "Sort of $"); ss = 0; }
00461                 AR.DeferFlag = olddefer; AR.NoCompress = oldcompress;
00462                 goto NewValIsZero;
00463             }
00464             else
00465         #endif
00466         {
00467             if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where, "dollar contents");
00468             d->where = &(AM.dollarzero);
00469             d->size = 0;
00470             cbuf[AM.dbufnum].rhs[numdollar] = 0;
00471             cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00472             cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
00473             d->type = DOLZERO;
00474         }
00475         if ( ss ) { M_free(ss, "Sort of $"); ss = 0; }
00476     }
00477     else {
00478         /*
00479             #! Thread version :
00480         */
00481         #ifdef WITHPTHREADS
00482             if ( dtype == MODMAX || dtype == MODMIN ) {
00483                 if ( numterms == 1 && ( *ss-1 == ABS(ss[*ss-1]) ) ) { /* is number */
00484                     switch ( d->type ) {
00485                         case DOLZERO:
00486                             HandleDolZero1;
00487                             if ( dtype == MODMAX && ss[*ss-1] > 0 ) break;
00488                             if ( dtype == MODMIN && ss[*ss-1] < 0 ) break;
00489                             if ( ss ) { M_free(ss, "Sort of $"); ss = 0; }
00490                             AR.DeferFlag = olddefer; AR.NoCompress = oldcompress;
00491                             goto NoChange;
00492                         case DOLTERMS:
00493                         case DOLNUMBER:
00494                             if ( ( dw = d->where[0] ) > 0 && d->where[dw] != 0 ) break;
00495                             if ( dtype == MODMAX && CompCoef(ss, d->where) > 0 ) break;
00496                             if ( dtype == MODMIN && CompCoef(ss, d->where) < 0 ) break;
00497                             if ( ss ) { M_free(ss, "Sort of $"); ss = 0; }
00498                             AR.DeferFlag = olddefer; AR.NoCompress = oldcompress;
00499                             goto NoChange;
00500                         default: {
00501                             WORD extraterm[4];
00502                             numvalue = DolToNumber(BHEAD numdollar);
00503                             if ( AN.ErrorInDollar != 0 ) break;
00504                             if ( numvalue == 0 ) {
00505                                 d->type = DOLZERO;
00506                                 d->where[0] = 0;
00507                                 cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00508                                 cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
00509                                 goto HandleDolZero1;
00510                             }
00511                             d->where[0] = extraterm[0] = 4;
00512                             d->where[1] = extraterm[1] = ABS(numvalue);
00513                             d->where[2] = extraterm[2] = 1;
00514                             d->where[3] = extraterm[3] = numvalue > 0 ? 3 : -3;
00515                             d->where[4] = 0;
00516                             d->type = DOLNUMBER;
00517                             if ( dtype == MODMAX && CompCoef(ss, extraterm) > 0 ) break;
00518                             if ( dtype == MODMIN && CompCoef(ss, extraterm) < 0 ) break;
00519                             if ( ss ) { M_free(ss, "Sort of $"); ss = 0; }
00520                             AR.DeferFlag = olddefer; AR.NoCompress = oldcompress;
00521                             goto NoChange;

```

```

00522     }
00523     }
00524     }
00525     else {
00526         if ( ss ) { M_free(ss,"Sort of $"); ss = 0; }
00527         AR.DeferFlag = olddefer; AR.NoCompress = oldcompress;
00528         goto NoChange;
00529     }
00530 }
00531 #endif
00532 /*
00533     #] Thread version :
00534 */
00535     d->type = DOLTERMS;
00536     if ( d->where && d->where != &(AM.dollarzero) ) { M_free(d->where,"dollar contents"); d->where
= 0; }
00537     d->size = newsize + 1;
00538     d->where = ss;
00539     cbuf[AM.dbufnum].rhs[numdollar] = w = d->where;
00540 }
00541     AR.DeferFlag = olddefer; AR.NoCompress = oldcompress;
00542 /*
00543     Now find the special cases
00544 */
00545     if ( numterms == 0 ) {
00546         d->type = DOLZERO;
00547     }
00548     else if ( numterms == 1 ) {
00549         t = d->where;
00550         n = *t;
00551         nsize = t[n-1];
00552         if ( nsize < 0 ) { nsize = -nsize; }
00553         if ( nsize == (n-1) ) {
00554             nsize = (nsize-1)/2;
00555             w = t + 1 + nsize;
00556             if ( *w == 1 ) {
00557                 w++; while ( w < ( t + n - 1 ) ) { if ( *w ) break; w++; }
00558                 if ( w >= ( t + n - 1 ) ) d->type = DOLNUMBER;
00559             }
00560         }
00561         else if ( n == 7 && t[6] == 3 && t[5] == 1 && t[4] == 1
&& t[1] == INDEX && t[2] == 3 ) {
00562             d->type = DOLINDEX;
00563             d->index = t[3];
00564         }
00565     }
00566 }
00567     if ( d->type == DOLTERMS ) {
00568         cbuf[AM.dbufnum].CanCommu[numdollar] = numcommute(d->where,
&cbuf[AM.dbufnum].NumTerms[numdollar]);
00569     }
00570     else {
00571         cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
00572         cbuf[AM.dbufnum].NumTerms[numdollar] = 1;
00573     }
00574 }
00575 #ifdef WITHPTHREADS
00576 NoChange;;
00577 /* UNLOCK(d->pthreadlockwrite); */
00578 UNLOCK(d->pthreadlockread);
00579 #endif
00580     AN.ncmod = oldncmod;
00581     return(0);
00582 }
00583
00584 /*
00585     #] AssignDollar :
00586     #[ WriteDollarToBuffer :
00587
00588     Takes the numbered dollar expression and writes it to output.
00589     We catch however the output in a buffer and return its address.
00590     This routine is needed when we need a text representation of
00591     a dollar expression like for the construction '$name' in the preprocessor.
00592     If par==0 we leave the current printing mode.
00593     If par==1 we insist on normal mode
00594 */
00595
00596 UBYTE *WriteDollarToBuffer(WORD numdollar, WORD par)
00597 {
00598     DOLLARS d = Dollars+numdollar;
00599     UBYTE *s, *oldcurbufwrt = AO.CurBufWrt;
00600     WORD *t, lbrac = 0, first = 0, arg[2], oldOutputMode = AC.OutputMode;
00601     WORD oldinfrack = AO.InFbrack;
00602     int error = 0;
00603     int dict = AO.CurrentDictionary;
00604
00605     AO.DollarOutSizeBuffer = 32;
00606     AO.DollarOutBuffer = (UBYTE *)Malloc1(AO.DollarOutSizeBuffer,"DollarOutBuffer");
00607     AO.DollarInOutBuffer = 1;

```

```

00608     AO.PrintType = 1;
00609     AO.InFbrack = 0;
00610     s = AO.DollarOutBuffer;
00611     *s = 0;
00612     if ( par > 0 && AO.CurDictInDollars == 0 ) {
00613         AC.OutputMode = NORMALFORMAT;
00614         AO.CurrentDictionary = 0;
00615     }
00616     else {
00617         AO.CurBufWrt = (UBYTE *)underscore;
00618     }
00619     AO.OutInBuffer = 1;
00620     switch ( d->type ) {
00621     case DOLARGUMENT:
00622         WriteArgument(d->where);
00623         break;
00624     case DOLSUBTERM:
00625         WriteSubTerm(d->where,1);
00626         break;
00627     case DOLNUMBER:
00628     case DOLTERMS:
00629         t = d->where;
00630         while ( *t ) {
00631             if ( WriteTerm(t,&lbrac,first,PRINTON,0) ) {
00632                 error = 1; break;
00633             }
00634             t += *t;
00635         }
00636         break;
00637     case DOLWILDARGS:
00638         t = d->where+1;
00639         while ( *t ) {
00640             WriteArgument(t);
00641             NEXTARG(t)
00642             if ( *t ) TokenToLine((UBYTE *)(","));
00643         }
00644         break;
00645     case DOLINDEX:
00646         arg[0] = -INDEX; arg[1] = d->index;
00647         WriteArgument(arg);
00648         break;
00649     case DOLZERO:
00650         *s++ = '0'; *s = 0;
00651         AO.DollarInOutBuffer = 1;
00652         break;
00653     case DOLUNDEFINED:
00654         *s = 0;
00655         AO.DollarInOutBuffer = 1;
00656         break;
00657     }
00658     AC.OutputMode = oldOutputMode;
00659     AO.OutInBuffer = 0;
00660     AO.InFbrack = oldinfbrack;
00661     AO.CurBufWrt = oldcurbufwrt;
00662     AO.CurrentDictionary = dict;
00663     if ( error ) {
00664         MLOCK(ErrorMessageLock);
00665         MesPrint("&Illegal dollar object for writing");
00666         MUNLOCK(ErrorMessageLock);
00667         M_free(AO.DollarOutBuffer,"DollarOutBuffer");
00668         AO.DollarOutBuffer = 0;
00669         AO.DollarOutSizeBuffer = 0;
00670         return(0);
00671     }
00672     return(AO.DollarOutBuffer);
00673 }
00674
00675 /*
00676 #] WriteDollarToBuffer :
00677 #[ WriteDollarFactorToBuffer :
00678
00679 Takes the numbered dollar expression and writes it to output.
00680 We catch however the output in a buffer and return its address.
00681 This routine is needed when we need a text representation of
00682 a dollar expression like for the construction '$name' in the preprocessor.
00683 If par==0 we leave the current printing mode.
00684 If par==1 we insist on normal mode
00685 */
00686
00687 UBYTE *WriteDollarFactorToBuffer(WORD numdollar, WORD numfac, WORD par)
00688 {
00689     DOLLARS d = Dollars+numdollar;
00690     UBYTE *s, *oldcurbufwrt = AO.CurBufWrt;
00691     WORD *t, lbrac = 0, first = 0, n[5], oldOutputMode = AC.OutputMode;
00692     WORD oldinfbrack = AO.InFbrack;
00693     int error = 0;
00694     int dict = AO.CurrentDictionary;

```

```

00695
00696     if ( numfac > d->nfactors || numfac < 0 ) {
00697         MLOCK(ErrorMessageLock);
00698         MesPrint("&Illegal factor number for this dollar variable: %d",numfac);
00699         MesPrint("&There are %d factors",d->nfactors);
00700         MUNLOCK(ErrorMessageLock);
00701         return(0);
00702     }
00703
00704     AO.DollarOutSizeBuffer = 32;
00705     AO.DollarOutBuffer = (UBYTE *)Malloca1(AO.DollarOutSizeBuffer,"DollarOutBuffer");
00706     AO.DollarInOutBuffer = 1;
00707     AO.PrintType = 1;
00708     AO.InFbrack = 0;
00709     s = AO.DollarOutBuffer;
00710     *s = 0;
00711     if ( par > 0 ) {
00712         AC.OutputMode = NORMALFORMAT;
00713         AO.CurrentDictionary = 0;
00714     }
00715     else {
00716         AO.CurBufWrt = (UBYTE *)underscore;
00717     }
00718     AO.OutInBuffer = 1;
00719     if ( numfac == 0 ) { /* write the number d->nfactors */
00720         n[0] = 4; n[1] = d->nfactors; n[2] = 1; n[3] = 3; n[4] = 0; t = n;
00721     }
00722     else if ( numfac == 1 && d->factors == 0 ) { /* Here d->factors is zero and d->where is fine */
00723         t = d->where;
00724     }
00725     else if ( d->factors[numfac-1].where == 0 ) { /* write the value */
00726         if ( d->factors[numfac-1].value < 0 ) {
00727             n[0] = 4; n[1] = -d->factors[numfac-1].value; n[2] = 1; n[3] = -3; n[4] = 0; t = n;
00728         }
00729         else {
00730             n[0] = 4; n[1] = d->factors[numfac-1].value; n[2] = 1; n[3] = 3; n[4] = 0; t = n;
00731         }
00732     }
00733     else { t = d->factors[numfac-1].where; }
00734     while ( *t ) {
00735         if ( WriteTerm(t,&lbrack,first,PRINTON,0) ) {
00736             error = 1; break;
00737         }
00738         t += *t;
00739     }
00740     AC.OutputMode = oldOutputMode;
00741     AO.OutInBuffer = 0;
00742     AO.InFbrack = oldinfbrack;
00743     AO.CurBufWrt = oldcurbufwrt;
00744     AO.CurrentDictionary = dict;
00745     if ( error ) {
00746         MLOCK(ErrorMessageLock);
00747         MesPrint("&Illegal dollar object for writing");
00748         MUNLOCK(ErrorMessageLock);
00749         M_free(AO.DollarOutBuffer,"DollarOutBuffer");
00750         AO.DollarOutBuffer = 0;
00751         AO.DollarOutSizeBuffer = 0;
00752         return(0);
00753     }
00754     return(AO.DollarOutBuffer);
00755 }
00756
00757 /*
00758 #] WriteDollarFactorToBuffer :
00759 #[ AddToDollarBuffer :
00760 */
00761
00762 void AddToDollarBuffer(UBYTE *s)
00763 {
00764     int i;
00765     UBYTE *t = s, *u, *newdob;
00766     LONG j;
00767     while ( *t ) { t++; }
00768     i = t - s;
00769     while ( i + AO.DollarInOutBuffer >= AO.DollarOutSizeBuffer ) {
00770         j = AO.DollarInOutBuffer;
00771         AO.DollarOutSizeBuffer -= 2;
00772         t = AO.DollarOutBuffer;
00773         newdob = (UBYTE *)Malloca1(AO.DollarOutSizeBuffer,"DollarOutBuffer");
00774         u = newdob;
00775         while ( --j >= 0 ) *u++ = *t++;
00776         M_free(AO.DollarOutBuffer,"DollarOutBuffer");
00777         AO.DollarOutBuffer = newdob;
00778     }
00779     t = AO.DollarOutBuffer + AO.DollarInOutBuffer-1;
00780     while ( t == AO.DollarOutBuffer && ( *s == '+' || *s == ' ' ) ) s++;
00781     i = 0;

```

```

00782     if ( AO.CurrentDictionary == 0 ) {
00783         while ( *s ) {
00784             if ( *s == ' ' ) { s++; continue; }
00785             *t++ = *s++; i++;
00786         }
00787     }
00788     else {
00789         while ( *s ) { *t++ = *s++; i++; }
00790     }
00791     *t = 0;
00792     AO.DollarInOutBuffer += i;
00793 }
00794
00795 /*
00796 #] AddToDollarBuffer :
00797 #[ TermAssign :
00798
00799 This routine is called from a piece of code in Normalize that has been
00800 commented out.
00801 */
00802
00803 void TermAssign(WORD *term)
00804 {
00805     DOLLARS d;
00806     WORD *t, *tstop, *astop, *w, *m;
00807     WORD i, newsize;
00808     for (;;) {
00809         astop = term + *term;
00810         tstop = astop - ABS(astop[-1]);
00811         t = term + 1;
00812         while ( t < tstop ) {
00813             if ( *t == AM.termfunnum && t[1] == FUNHEAD+2
00814                 && t[FUNHEAD] == -DOLLAREXPRESSION ) {
00815                 d = Dollars + t[FUNHEAD+1];
00816                 newsize = *term - FUNHEAD - 1;
00817                 if ( newsize < MINALLOC ) newsize = MINALLOC;
00818                 newsize = (newsize+7)/8*8;
00819                 if ( d->size > 2*newsize && d->size > 1000 ) {
00820                     if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollar
00821 contents");
00822                     d->size = 0;
00823                     d->where = &(AM.dollarzero);
00824                 }
00825                 if ( d->size < newsize ) {
00826                     if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollar
00827 contents");
00828                     d->size = newsize;
00829                     d->where = (WORD *)Malloc1(newsize*sizeof(WORD),"dollar contents");
00830                 }
00831                 cbuf[AM.dbufnum].rhs[t[FUNHEAD+1]] = w = d->where;
00832                 m = term;
00833                 while ( m < t ) *w++ = *m++;
00834                 m += t[1];
00835                 while ( m < tstop ) {
00836                     if ( *m == AM.termfunnum && m[1] == FUNHEAD+2
00837                         && m[FUNHEAD] == -DOLLAREXPRESSION ) { m += m[1]; }
00838                     else {
00839                         i = m[1];
00840                         while ( --i >= 0 ) *w++ = *m++;
00841                     }
00842                 }
00843                 while ( m < astop ) *w++ = *m++;
00844                 *(d->where) = w - d->where;
00845                 *w = 0;
00846                 d->type = DOLTERMS;
00847                 w = t; m = t + t[1];
00848                 while ( m < astop ) *w++ = *m++;
00849                 *term = w - term;
00850                 break;
00851             }
00852             t += t[1];
00853         }
00854         if ( t >= tstop ) return;
00855     }
00856 }
00857 /*
00858 #] TermAssign :
00859 #[ PutTermInDollar :
00860
00861 We assume here that the dollar is local.
00862 */
00863 int PutTermInDollar(WORD *term, WORD numdollar)
00864 {
00865     DOLLARS d = Dollars+numdollar;
00866     WORD i;

```

```

00867     if ( term == 0 || *term == 0 ) {
00868         d->type = DOLZERO;
00869         return(0);
00870     }
00871     if ( d->size < *term || d->size > 2*term[0] || d->where == 0 ) {
00872         if ( d->size > 0 && d->where ) {
00873             M_free(d->where,"dollar contents");
00874         }
00875         d->where = Malloc1((term[0]+1)*sizeof(WORD),"dollar contents");
00876         d->size = term[0]+1;
00877     }
00878     d->type = DOLTERMS;
00879     for ( i = 0; i < term[0]; i++ ) d->where[i] = term[i];
00880     d->where[i] = 0;
00881     return(0);
00882 }
00883
00884 /*
00885  #] PutTermInDollar :
00886  #[ WildDollars :
00887
00888      Note that we cannot upload wildcards into dollar variables when WITHPTHREADS.
00889  */
00890
00891 void WildDollars(PHEAD WORD *term)
00892 {
00893     GETBIDENTITY
00894     DOLLARS d;
00895     WORD *m, *t, *w, *ww, *orig = 0, *wildvalue, *wildstop;
00896     int numdollar;
00897     LONG weneed, i;
00898     #ifdef WITHPTHREADS
00899         int dtype = -1;
00900     #endif
00901     if ( term == 0 ) {
00902         m = wildvalue = AN.WildValue;
00903         wildstop = AN.WildStop;
00904     }
00905     else {
00906         ww = term + *term; ww -= ABS(ww[-1]); w = term+1;
00907         while ( w < ww && *w != SUBEXPRESSION ) w += w[1];
00908         if ( w >= ww ) return;
00909         wildstop = w + w[1];
00910         w += SUBEXPSIZE;
00911         wildvalue = m = w;
00912     }
00913     while ( m < wildstop ) {
00914         if ( *m != LOADDOLLAR ) { m += m[1]; continue; }
00915         t = m - 4;
00916         while ( *t == LOADDOLLAR || *t == FROMSET || *t == SETTONUM ) t -= 4;
00917         if ( t < wildvalue ) {
00918             MLOCK(ErrorMessageLock);
00919             MesPrint("&Serious bug in wildcard prototype. Found in WildDollars");
00920             MUNLOCK(ErrorMessageLock);
00921             Terminate(-1);
00922         }
00923         numdollar = m[2];
00924         d = Dollars + numdollar;
00925         #ifdef WITHPTHREADS
00926         {
00927             int nummodopt;
00928             dtype = -1;
00929             if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
00930                 for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
00931                     if ( numdollar == ModOptdollars[nummodopt].number ) break;
00932                 }
00933                 if ( nummodopt < NumModOptdollars ) {
00934                     dtype = ModOptdollars[nummodopt].type;
00935                     if ( dtype == MODLOCAL ) {
00936                         d = ModOptdollars[nummodopt].dstruct+AT.identity;
00937                     }
00938                     else {
00939                         MLOCK(ErrorMessageLock);
00940                         MesPrint("&Illegal attempt to use $-variable %s in module %1",
00941                             DOLLARNAME(Dollars,numdollar),AC.CModule);
00942                         MUNLOCK(ErrorMessageLock);
00943                         Terminate(-1);
00944                     }
00945                 }
00946             }
00947         }
00948     #endif
00949     /*
00950         The value of this wildcard goes into our $-variable
00951         First compute the space we need.
00952     */
00953     switch ( *t ) {

```

```

00954         case SYMTONUM:
00955             weneed = 5;
00956             break;
00957         case SYMTOSYM:
00958             weneed = 9;
00959             break;
00960         case SYMTOSUB:
00961         case VECTOSUB:
00962         case INDTOSUB:
00963             orig = cbuf[AT.ebufnum].rhs[t[3]];
00964             w = orig; while ( *w ) w += *w;
00965             weneed = w - orig + 1;
00966             break;
00967         case VECTOMIN:
00968         case VECTOVEC:
00969         case INDTOIND:
00970             weneed = 8;
00971             break;
00972         case FUNTOFUN:
00973             weneed = FUNHEAD+5;
00974             break;
00975         case ARGTOARG:
00976             orig = cbuf[AT.ebufnum].rhs[t[3]];
00977             if ( *orig > 0 ) weneed = *orig+2;
00978             else {
00979                 w = orig+1; while ( *w ) { NEXTARG(w) }
00980                 weneed = w - orig + 1;
00981             }
00982             break;
00983         default:
00984             weneed = MINALLOC;
00985             break;
00986     }
00987     if ( weneed < MINALLOC ) weneed = MINALLOC;
00988     weneed = ((weneed+7)/8)*8;
00989     if ( d->size > 2*weneed && d->size > 1000 ) {
00990         if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollarspace");
00991         d->where = &(AM.dollarzero);
00992         d->size = 0;
00993     }
00994     if ( d->size < weneed ) {
00995         if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollarspace");
00996         d->where = (WORD *)Malloc1(weneed*sizeof(WORD),"dollarspace");
00997         d->size = weneed;
00998     }
00999 /*
01000     It is not clear what the following code does for TFORM
01001
01002     if ( dtype != MODLOCAL ) {
01003 /*
01004         cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
01005         cbuf[AM.dbufnum].NumTerms[numdollar] = 1;
01006 /*
01007         cbuf[AM.dbufnum].rhs[numdollar] = d->where; */
01008         cbuf[AM.dbufnum].rhs[numdollar] = (WORD *) (1);
01009     }
01010     Now load up the value of the wildcard in compiler buffer format
01011 */
01012     w = d->where;
01013     d->type = DOLTERMS;
01014     switch ( *t ) {
01015         case SYMTONUM:
01016             d->where[0] = 4; d->where[2] = 1;
01017             if ( t[3] >= 0 ) { d->where[1] = t[3]; d->where[3] = 3; }
01018             else { d->where[1] = -t[3]; d->where[3] = -3; }
01019             if ( t[3] == 0 ) { d->type = DOLZERO; d->where[0] = 0; }
01020             else { d->type = DOLNUMBER; d->where[4] = 0; }
01021             break;
01022         case SYMTOSYM:
01023             *w++ = 8;
01024             *w++ = SYMBOL;
01025             *w++ = 4;
01026             *w++ = t[3];
01027             *w++ = 1;
01028             *w++ = 1;
01029             *w++ = 1;
01030             *w++ = 3;
01031             *w = 0;
01032             break;
01033         case SYMTOSUB:
01034         case VECTOSUB:
01035         case INDTOSUB:
01036             while ( *orig ) {
01037                 i = *orig; while ( --i >= 0 ) *w++ = *orig++;
01038             }
01039             *w = 0;
01040 /*

```



```

01041             And then we have to fix up CanCommu
01042  */
01043             break;
01044         case VECTOMIN:
01045             *w++ = 7; *w++ = INDEX; *w++ = 3; *w++ = t[3];
01046             *w++ = 1; *w++ = 1; *w++ = -3; *w = 0;
01047             break;
01048         case VECTOVEC:
01049             *w++ = 7; *w++ = INDEX; *w++ = 3; *w++ = t[3];
01050             *w++ = 1; *w++ = 1; *w++ = 3; *w = 0;
01051             break;
01052         case INDTOIND:
01053             d->type = DOLINDEX; d->index = t[3]; *w = 0;
01054             break;
01055         case FUNTOFUN:
01056             *w++ = FUNHEAD+4; *w++ = t[3]; *w++ = FUNHEAD;
01057             FILLFUN(w)
01058             *w++ = 1; *w++ = 1; *w++ = 3; *w = 0;
01059             break;
01060         case ARGTOARG:
01061             if ( *orig > 0 ) ww = orig + *orig + 1;
01062             else {
01063                 ww = orig+1; while ( *ww ) { NEXTARG(ww) }
01064             }
01065             while ( orig < ww ) *w++ = *orig++;
01066             *w = 0;
01067             d->type = DOLWILDARGS;
01068             break;
01069         default:
01070             d->type = DOLUNDEFINED;
01071             break;
01072     }
01073     m += m[1];
01074 }
01075 }
01076
01077 /*
01078  #] WildDollars :
01079  #[ DolToTensor :      with LOCK
01080  */
01081
01082 WORD DolToTensor(PHEAD WORD numdollar)
01083 {
01084     GETBIDENTITY
01085     DOLLARS d = Dollars + numdollar;
01086     WORD retval;
01087     #ifdef WITHPTHREADS
01088     int nummodopt, dtype = -1;
01089     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01090         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01091             if ( numdollar == ModOptdollars[nummodopt].number ) break;
01092         }
01093         if ( nummodopt < NumModOptdollars ) {
01094             dtype = ModOptdollars[nummodopt].type;
01095             if ( dtype == MODLOCAL ) {
01096                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01097             }
01098             else {
01099                 LOCK(d->pthreadlockread);
01100             }
01101         }
01102     }
01103     #endif
01104     AN.ErrorInDollar = 0;
01105     if ( d->type == DOLTERMS && d->where[0] == FUNHEAD+4 &&
01106         d->where[FUNHEAD+4] == 0 && d->where[FUNHEAD+3] == 3 &&
01107         d->where[FUNHEAD+2] == 1 && d->where[FUNHEAD+1] == 1 &&
01108         d->where[1] >= FUNCTION && d->where[1] < FUNCTION+WILDOFFSET
01109         && functions[d->where[1]-FUNCTION].spec >= TENSORFUNCTION ) {
01110         retval = d->where[1];
01111     }
01112     else if ( d->type == DOLARGUMENT &&
01113         d->where[0] <= -FUNCTION && d->where[0] > -FUNCTION-WILDOFFSET
01114         && functions[-d->where[0]-FUNCTION].spec >= TENSORFUNCTION ) {
01115         retval = -d->where[0];
01116     }
01117     else if ( d->type == DOLWILDARGS && d->where[0] == 0
01118         && d->where[1] <= -FUNCTION && d->where[1] > -FUNCTION-WILDOFFSET
01119         && d->where[2] == 0
01120         && functions[-d->where[1]-FUNCTION].spec >= TENSORFUNCTION ) {
01121         retval = -d->where[1];
01122     }
01123     else if ( d->type == DOLSUBTERM &&
01124         d->where[0] >= FUNCTION && d->where[0] < FUNCTION+WILDOFFSET
01125         && functions[d->where[0]-FUNCTION].spec >= TENSORFUNCTION ) {
01126         retval = d->where[0];
01127     }

```

```

01128     else {
01129         AN.ErrorInDollar = 1;
01130         retval = 0;
01131     }
01132 #ifdef WITHPTHREADS
01133     if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
01134 #endif
01135     return(retval);
01136 }
01137
01138 /*
01139 #] DolToTensor :
01140 #[ DolToFunction : with LOCK
01141 */
01142
01143 WORD DolToFunction(PHEAD WORD numdollar)
01144 {
01145     GETBIDENTITY
01146     DOLLARS d = Dollars + numdollar;
01147     WORD retval;
01148 #ifdef WITHPTHREADS
01149     int nummodopt, dtype = -1;
01150     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01151         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01152             if ( numdollar == ModOptdollars[nummodopt].number ) break;
01153         }
01154         if ( nummodopt < NumModOptdollars ) {
01155             dtype = ModOptdollars[nummodopt].type;
01156             if ( dtype == MODLOCAL ) {
01157                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01158             }
01159             else {
01160                 LOCK(d->pthreadlockread);
01161             }
01162         }
01163     }
01164 #endif
01165     AN.ErrorInDollar = 0;
01166     if ( d->type == DOLTERMS && d->where[0] == FUNHEAD+4 &&
01167         d->where[FUNHEAD+4] == 0 && d->where[FUNHEAD+3] == 3 &&
01168         d->where[FUNHEAD+2] == 1 && d->where[FUNHEAD+1] == 1 &&
01169         d->where[1] >= FUNCTION && d->where[1] < FUNCTION+WILDOFFSET ) {
01170         retval = d->where[1];
01171     }
01172     else if ( d->type == DOLARGUMENT &&
01173         d->where[0] <= -FUNCTION && d->where[0] > -FUNCTION-WILDOFFSET ) {
01174         retval = -d->where[0];
01175     }
01176     else if ( d->type == DOLWILDARGS && d->where[0] == 0
01177         && d->where[1] <= -FUNCTION && d->where[1] > -FUNCTION-WILDOFFSET
01178         && d->where[2] == 0 ) {
01179         retval = -d->where[1];
01180     }
01181     else if ( d->type == DOLSUBTERM &&
01182         d->where[0] >= FUNCTION && d->where[0] < FUNCTION+WILDOFFSET ) {
01183         retval = d->where[0];
01184     }
01185     else {
01186         AN.ErrorInDollar = 1;
01187         retval = 0;
01188     }
01189 #ifdef WITHPTHREADS
01190     if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
01191 #endif
01192     return(retval);
01193 }
01194
01195 /*
01196 #] DolToFunction :
01197 #[ DolToVector : with LOCK
01198 */
01199
01200 WORD DolToVector(PHEAD WORD numdollar)
01201 {
01202     GETBIDENTITY
01203     DOLLARS d = Dollars + numdollar;
01204     WORD retval;
01205 #ifdef WITHPTHREADS
01206     int nummodopt, dtype = -1;
01207     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01208         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01209             if ( numdollar == ModOptdollars[nummodopt].number ) break;
01210         }
01211         if ( nummodopt < NumModOptdollars ) {
01212             dtype = ModOptdollars[nummodopt].type;
01213             if ( dtype == MODLOCAL ) {
01214                 d = ModOptdollars[nummodopt].dstruct+AT.identity;

```

```

01215         }
01216         else {
01217             LOCK(d->pthreadlockread);
01218         }
01219     }
01220 }
01221 #endif
01222 AN.ErrorInDollar = 0;
01223 if ( d->type == DOLINDEX && d->index < 0 ) {
01224     retval = d->index;
01225 }
01226 else if ( d->type == DOLARGUMENT && ( d->where[0] == -VECTOR
01227 || d->where[0] == -MINVECTOR ) ) {
01228     retval = d->where[1];
01229 }
01230 else if ( d->type == DOLSUBTERM && d->where[0] == INDEX
01231 && d->where[1] == 3 && d->where[2] < 0 ) {
01232     retval = d->where[2];
01233 }
01234 else if ( d->type == DOLTERMS && d->where[0] == 7 &&
01235 d->where[7] == 0 && d->where[6] == 3 &&
01236 d->where[5] == 1 && d->where[4] == 1 &&
01237 d->where[1] >= INDEX && d->where[3] < 0 ) {
01238     retval = d->where[3];
01239 }
01240 else if ( d->type == DOLWILDARGS && d->where[0] == 0
01241 && ( d->where[1] == -VECTOR || d->where[1] == -MINVECTOR )
01242 && d->where[3] == 0 ) {
01243     retval = d->where[2];
01244 }
01245 else if ( d->type == DOLWILDARGS && d->where[0] == 1
01246 && d->where[1] < 0 ) {
01247     retval = d->where[1];
01248 }
01249 else {
01250     AN.ErrorInDollar = 1;
01251     retval = 0;
01252 }
01253 #ifdef WITHPTHREADS
01254 if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
01255 #endif
01256 return(retval);
01257 }
01258
01259 /*
01260 #] DolToVector :
01261 #[ DolToNumber :
01262 */
01263
01264 WORD DolToNumber(PHEAD WORD numdollar)
01265 {
01266     GETBIDENTITY
01267     DOLLARS d = Dollars + numdollar;
01268 #ifdef WITHPTHREADS
01269     int nummodopt, dtype = -1;
01270     if ( AS.MultiThreaded && ( AC.mparallelfld == PARALLELFIELD ) ) {
01271         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01272             if ( numdollar == ModOptdollars[nummodopt].number ) break;
01273         }
01274         if ( nummodopt < NumModOptdollars ) {
01275             dtype = ModOptdollars[nummodopt].type;
01276             if ( dtype == MODLOCAL ) {
01277                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01278             }
01279         }
01280     }
01281 #endif
01282 AN.ErrorInDollar = 0;
01283 if ( ( d->type == DOLTERMS || d->type == DOLNUMBER )
01284 && d->where[0] == 4 &&
01285 d->where[4] == 0 && ( d->where[3] == 3 || d->where[3] == -3 )
01286 && d->where[2] == 1 && ( d->where[1] & TOPBITONLY ) == 0 ) {
01287     if ( d->where[3] > 0 ) return(d->where[1]);
01288     else return(-d->where[1]);
01289 }
01290 else if ( d->type == DOLARGUMENT && d->where[0] == -SNUMBER ) {
01291     return(d->where[1]);
01292 }
01293 else if ( d->type == DOLARGUMENT && d->where[0] == -INDEX
01294 && d->where[1] >= 0 && d->where[1] < AM.OffsetIndex ) {
01295     return(d->where[1]);
01296 }
01297 else if ( d->type == DOLZERO ) return(0);
01298 else if ( d->type == DOLWILDARGS && d->where[0] == 0
01299 && d->where[1] == -SNUMBER && d->where[3] == 0 ) {
01300     return(d->where[2]);
01301 }

```

```

01302     else if ( d->type == DOLINDEX && d->index >= 0 && d->index < AM.OffsetIndex ) {
01303         return(d->index);
01304     }
01305     else if ( d->type == DOLWILDARGS && d->where[0] == 1
01306 && d->where[1] >= 0 && d->where[1] < AM.OffsetIndex ) {
01307         return(d->where[1]);
01308     }
01309     else if ( d->type == DOLWILDARGS && d->where[0] == 0
01310 && d->where[1] == -INDEX && d->where[3] == 0 && d->where[2] >= 0
01311 && d->where[2] < AM.OffsetIndex ) {
01312         return(d->where[2]);
01313     }
01314     AN.ErrorInDollar = 1;
01315     return(0);
01316 }
01317
01318 /*
01319 #] DolToNumber :
01320 #[ DolToSymbol :      with LOCK
01321 */
01322
01323 WORD DolToSymbol(PHEAD WORD numdollar)
01324 {
01325     GETBIDENTITY
01326     DOLLARS d = Dollars + numdollar;
01327     WORD retval;
01328 #ifdef WITHPTHREADS
01329     int nummodopt, dtype = -1;
01330     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01331         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01332             if ( numdollar == ModOptdollars[nummodopt].number ) break;
01333         }
01334         if ( nummodopt < NumModOptdollars ) {
01335             dtype = ModOptdollars[nummodopt].type;
01336             if ( dtype == MODLOCAL ) {
01337                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01338             }
01339             else {
01340                 LOCK(d->pthreadlockread);
01341             }
01342         }
01343     }
01344 #endif
01345     AN.ErrorInDollar = 0;
01346     if ( d->type == DOLTERMS && d->where[0] == 8 &&
01347 d->where[8] == 0 && d->where[7] == 3 && d->where[6] == 1
01348 && d->where[5] == 1 && d->where[4] == 1 && d->where[1] == SYMBOL ) {
01349         retval = d->where[3];
01350     }
01351     else if ( d->type == DOLARGUMENT && d->where[0] == -SYMBOL ) {
01352         retval = d->where[1];
01353     }
01354     else if ( d->type == DOLSUBTERM && d->where[0] == SYMBOL
01355 && d->where[1] == 4 && d->where[3] == 1 ) {
01356         retval = d->where[2];
01357     }
01358     else if ( d->type == DOLWILDARGS && d->where[0] == 0
01359 && d->where[1] == -SYMBOL && d->where[3] == 0 ) {
01360         retval = d->where[2];
01361     }
01362     else {
01363         AN.ErrorInDollar = 1;
01364         retval = -1;
01365     }
01366 #ifdef WITHPTHREADS
01367     if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
01368 #endif
01369     return(retval);
01370 }
01371
01372 /*
01373 #] DolToSymbol :
01374 #[ DolToIndex :      with LOCK
01375 */
01376
01377 WORD DolToIndex(PHEAD WORD numdollar)
01378 {
01379     GETBIDENTITY
01380     DOLLARS d = Dollars + numdollar;
01381     WORD retval;
01382 #ifdef WITHPTHREADS
01383     int nummodopt, dtype = -1;
01384     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01385         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01386             if ( numdollar == ModOptdollars[nummodopt].number ) break;
01387         }
01388         if ( nummodopt < NumModOptdollars ) {

```

```

01389         dtype = ModOptdollars[nummodopt].type;
01390         if ( dtype == MODLOCAL ) {
01391             d = ModOptdollars[nummodopt].dstruct+AT.identity;
01392         }
01393         else {
01394             LOCK(d->pthreadlockread);
01395         }
01396     }
01397 }
01398 #endif
01399 AN.ErrorInDollar = 0;
01400 if ( d->type == DOLTERMS && d->where[0] == 7 &&
01401     d->where[7] == 0 && d->where[6] == 3 && d->where[5] == 1
01402     && d->where[4] == 1 && d->where[1] == INDEX && d->where[3] >= 0 ) {
01403     retval = d->where[3];
01404 }
01405 else if ( d->type == DOLARGUMENT && d->where[0] == -SNUMBER
01406     && d->where[1] >= 0 && d->where[1] < AM.OffsetIndex ) {
01407     retval = d->where[1];
01408 }
01409 else if ( d->type == DOLARGUMENT && d->where[0] == -INDEX
01410     && d->where[1] >= 0 ) {
01411     retval = d->where[1];
01412 }
01413 else if ( d->type == DOLZERO ) return(0);
01414 else if ( d->type == DOLWILDARGS && d->where[0] == 0
01415     && d->where[1] == -SNUMBER && d->where[3] == 0 && d->where[2] >= 0
01416     && d->where[2] < AM.OffsetIndex ) {
01417     retval = d->where[2];
01418 }
01419 else if ( d->type == DOLINDEX && d->index >= 0 ) {
01420     retval = d->index;
01421 }
01422 else if ( d->type == DOLNUMBER && d->where[0] == 4 && d->where[2] == 1
01423     && d->where[3] == 3 && d->where[4] == 0 && d->where[1] < AM.OffsetIndex ) {
01424     retval = d->where[1];
01425 }
01426 else if ( d->type == DOLWILDARGS && d->where[0] == 1
01427     && d->where[1] >= 0 ) {
01428     retval = d->where[1];
01429 }
01430 else if ( d->type == DOLSUBTERM && d->where[0] == INDEX
01431     && d->where[1] == 3 && d->where[2] >= 0 ) {
01432     retval = d->where[2];
01433 }
01434 else if ( d->type == DOLWILDARGS && d->where[0] == 0
01435     && d->where[1] == -INDEX && d->where[3] == 0 && d->where[2] >= 0 ) {
01436     retval = d->where[2];
01437 }
01438 else {
01439     AN.ErrorInDollar = 1;
01440     retval = 0;
01441 }
01442 #ifdef WITHPTHREADS
01443 if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
01444 #endif
01445 return(retval);
01446 }
01447
01448 /*
01449 #] DolToIndex :
01450 #[ DolToTerms :
01451
01452 Returns a struct of type DOLLARS which contains a copy of the
01453 original dollar variable, provided it can be expressed in terms of
01454 an expression (type = DOLTERMS). Otherwise it returns zero.
01455 The dollar is expressed in terms in the buffer "where"
01456 */
01457
01458 DOLLARS DolToTerms(PHEAD WORD numdollar)
01459 {
01460     GETBIDENTITY
01461     LONG size;
01462     DOLLARS d = Dollars + numdollar, newd;
01463     WORD *t, *w, i;
01464 #ifdef WITHPTHREADS
01465     int nummodopt, dtype = -1;
01466     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01467         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01468             if ( numdollar == ModOptdollars[nummodopt].number ) break;
01469         }
01470         if ( nummodopt < NumModOptdollars ) {
01471             dtype = ModOptdollars[nummodopt].type;
01472             if ( dtype == MODLOCAL ) {
01473                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01474             }
01475         }

```

```

01476     }
01477 #endif
01478     AN.ErrorInDollar = 0;
01479     switch ( d->type ) {
01480     case DOLARGUMENT:
01481         t = d->where;
01482         if ( t[0] < 0 ) {
01483     ShortArgument:
01484             w = AT.WorkPointer;
01485             if ( t[0] <= -FUNCTION ) {
01486                 *w++ = FUNHEAD+4; *w++ = -t[0];
01487                 *w++ = FUNHEAD; FILLFUN(w)
01488                 *w++ = 1; *w++ = 1; *w++ = 3;
01489             }
01490             else if ( t[0] == -SYMBOL ) {
01491                 *w++ = 8; *w++ = SYMBOL; *w++ = 4; *w++ = t[1];
01492                 *w++ = 1; *w++ = 1; *w++ = 1; *w++ = 3;
01493             }
01494             else if ( t[0] == -VECTOR || t[0] == -INDEX ) {
01495                 *w++ = 7; *w++ = INDEX; *w++ = 3; *w++ = t[1];
01496                 *w++ = 1; *w++ = 1; *w++ = 3;
01497             }
01498             else if ( t[0] == -MINVECTOR ) {
01499                 *w++ = 7; *w++ = INDEX; *w++ = 3; *w++ = t[1];
01500                 *w++ = 1; *w++ = 1; *w++ = -3;
01501             }
01502             else if ( t[0] == -SNUMBER ) {
01503                 *w++ = 4;
01504                 if ( t[1] < 0 ) {
01505                     *w++ = -t[1]; *w++ = 1; *w++ = -3;
01506                 }
01507                 else {
01508                     *w++ = t[1]; *w++ = 1; *w++ = 3;
01509                 }
01510             }
01511             *w = 0; size = w - AT.WorkPointer;
01512             w = AT.WorkPointer;
01513             break;
01514         }
01515         /* fall through */
01516     case DOLNUMBER:
01517     case DOLTERMS:
01518         t = d->where;
01519         while ( *t ) t += *t;
01520         size = t - d->where;
01521         w = d->where;
01522         break;
01523     case DOLSUBTERM:
01524         w = AT.WorkPointer;
01525         size = d->where[1];
01526         *w++ = size+4; t = d->where; NCOPY(w,t,size)
01527         *w++ = 1; *w++ = 1; *w++ = 3;
01528         w = AT.WorkPointer; size = d->where[1]+4;
01529         break;
01530     case DOLINDEX:
01531         w = AT.WorkPointer;
01532         *w++ = 7; *w++ = INDEX; *w++ = 3; *w++ = d->index;
01533         *w++ = 1; *w++ = 1; *w++ = 3; *w = 0;
01534         w = AT.WorkPointer; size = 7;
01535         break;
01536     case DOLWILDARGS:
01537     /*
01538         In some cases we can make a copy
01539     */
01540         t = d->where+1;
01541         if ( *t == 0 ) return(0);
01542         NEXTARG(t);
01543         if ( *t ) { /* More than one argument in here */
01544             MLOCK(ErrorMessageLock);
01545             MesPrint("Trying to convert a $ with an argument field into an expression");
01546             MUNLOCK(ErrorMessageLock);
01547             Terminate(-1);
01548         }
01549     /*
01550         Now we have a single argument
01551     */
01552         t = d->where+1;
01553         if ( *t < 0 ) goto ShortArgument;
01554         size = *t - ARGHEAD;
01555         w = t + ARGHEAD;
01556         break;
01557     case DOLUNDEFINED:
01558         MLOCK(ErrorMessageLock);
01559         MesPrint("Trying to use an undefined $ in an expression");
01560         MUNLOCK(ErrorMessageLock);
01561         Terminate(-1);
01562         /* fall through */

```

```

01563         case DOLZERO:
01564             if ( d->where ) { d->where[0] = 0; }
01565             else d->where = &(AM.dollarzero);
01566             size = 0;
01567             w = d->where;
01568             break;
01569         default:
01570             return(0);
01571     }
01572     newd = (DOLLARS)Malloc1(sizeof(struct DoLLaRS)+(size+1)*sizeof(WORD),
01573         "Copy of dollar variable");
01574     t = (WORD *) (newd+1);
01575     newd->where = t;
01576     newd->name = d->name;
01577     newd->node = d->node;
01578     newd->type = DOLTERMS;
01579     newd->size = size;
01580     newd->numdummies = d->numdummies;
01581 #ifdef WITHPTHREADS
01582     newd->pthreadlockread = dummylock;
01583     newd->pthreadlockwrite = dummylock;
01584 #endif
01585     size++;
01586     NCOPY(t,w,size);
01587     newd->nfactors = d->nfactors;
01588     if ( d->nfactors > 1 ) {
01589         newd->factors = (FACDOLLAR *)Malloc1(d->nfactors*sizeof(FACDOLLAR),"Dollar factors");
01590         for ( i = 0; i < d->nfactors; i++ ) {
01591             newd->factors[i].where = 0;
01592             newd->factors[i].size = 0;
01593             newd->factors[i].type = DOLUNDEFINED;
01594             newd->factors[i].value = d->factors[i].value;
01595         }
01596     }
01597     else { newd->factors = 0; }
01598     return(newd);
01599 }
01600
01601 /*
01602  #] DolToTerms :
01603  #[ DolToLong :
01604  */
01605
01606 LONG DolToLong(PHEAD WORD numdollar)
01607 {
01608     GETBIDENTITY
01609     DOLLARS d = Dollars + numdollar;
01610     LONG x;
01611 #ifdef WITHPTHREADS
01612     int nummodopt, dtype = -1;
01613     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01614         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01615             if ( numdollar == ModOptdollars[nummodopt].number ) break;
01616         }
01617         if ( nummodopt < NumModOptdollars ) {
01618             dtype = ModOptdollars[nummodopt].type;
01619             if ( dtype == MODLOCAL ) {
01620                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01621             }
01622         }
01623     }
01624 #endif
01625     AN.ErrorInDollar = 0;
01626     if ( ( d->type == DOLTERMS || d->type == DOLNUMBER )
01627         && d->where[0] == 4 &&
01628         d->where[4] == 0 && ( d->where[3] == 3 || d->where[3] == -3 )
01629         && d->where[2] == 1 && ( d->where[1] & TOPBITONLY ) == 0 ) {
01630         x = d->where[1];
01631         if ( d->where[3] > 0 ) return(x);
01632         else return(-x);
01633     }
01634     else if ( ( d->type == DOLTERMS || d->type == DOLNUMBER )
01635         && d->where[0] == 6 &&
01636         d->where[6] == 0 && ( d->where[5] == 5 || d->where[5] == -5 )
01637         && d->where[3] == 1 && d->where[4] == 1 && ( d->where[2] & TOPBITONLY ) == 0 ) {
01638         x = d->where[1] + ( (LONG) (d->where[2]) << BITSINWORD );
01639         if ( d->where[5] > 0 ) return(x);
01640         else return(-x);
01641     }
01642     else if ( d->type == DOLARGUMENT && d->where[0] == -SNUMBER ) {
01643         x = d->where[1];
01644         return(x);
01645     }
01646     else if ( d->type == DOLARGUMENT && d->where[0] == -INDEX
01647         && d->where[1] >= 0 && d->where[1] < AM.OffsetIndex ) {
01648         x = d->where[1];
01649         return(x);

```

```

01650     }
01651     else if ( d->type == DOLZERO ) return(0);
01652     else if ( d->type == DOLWILDARGS && d->where[0] == 0
01653     && d->where[1] == -SNUMBER && d->where[3] == 0 ) {
01654         x = d->where[2];
01655         return(x);
01656     }
01657     else if ( d->type == DOLINDEX && d->index >= 0 && d->index < AM.OffsetIndex ) {
01658         x = d->index;
01659         return(x);
01660     }
01661     else if ( d->type == DOLWILDARGS && d->where[0] == 1
01662     && d->where[1] >= 0 && d->where[1] < AM.OffsetIndex ) {
01663         x = d->where[1];
01664         return(x);
01665     }
01666     else if ( d->type == DOLWILDARGS && d->where[0] == 0
01667     && d->where[1] == -INDEX && d->where[3] == 0 && d->where[2] >= 0
01668     && d->where[2] < AM.OffsetIndex ) {
01669         x = d->where[2];
01670         return(x);
01671     }
01672     AN.ErrorInDollar = 1;
01673     return(0);
01674 }
01675
01676 /*
01677  #] DolToLong :
01678  #[ ExecInside :
01679  */
01680
01681 int ExecInside(UBYTE *s)
01682 {
01683     GETIDENTITY
01684     UBYTE *t, c;
01685     WORD *w, number;
01686     int error = 0;
01687     w = AT.WorkPointer;
01688     if ( AC.insidelevel >= MAXNEST ) {
01689         MLOCK(ErrorMessageLock);
01690         MesPrint("@Nesting of inside statements more than %d levels", (WORD)MAXNEST);
01691         MUNLOCK(ErrorMessageLock);
01692         return(-1);
01693     }
01694     AC.insidesumcheck[AC.insidelevel] = NestingChecksum();
01695     AC.insidestack[AC.insidelevel] = cbuf[AC.cbunum].Pointer
01696         - cbuf[AC.cbunum].Buffer + 2;
01697     AC.insidelevel++;
01698     *w++ = TYPEINSIDE;
01699     w++; w++;
01700     for(;;) { /* Look for a (comma separated) list of dollar variables */
01701         while ( *s == ',' ) s++;
01702         if ( *s == 0 ) break;
01703         if ( *s == '$' ) {
01704             s++; t = s;
01705             if ( FG.cTable[*s] != 0 ) {
01706                 MLOCK(ErrorMessageLock);
01707                 MesPrint("Illegal name for $ variable: %s", s-1);
01708                 MUNLOCK(ErrorMessageLock);
01709                 goto skipdol;
01710             }
01711             while ( FG.cTable[*s] == 0 || FG.cTable[*s] == 1 ) s++;
01712             c = *s; *s = 0;
01713             if ( ( number = GetDollar(t) ) < 0 ) {
01714                 number = AddDollar(t, 0, 0, 0);
01715             }
01716             *s = c;
01717             *w++ = number;
01718             AddPotModdollar(number);
01719         }
01720         else {
01721             MLOCK(ErrorMessageLock);
01722             MesPrint("&Illegal object in Inside statement");
01723             MUNLOCK(ErrorMessageLock);
01724             skipdol: error = 1;
01725             while ( *s && *s != ',' && s[1] != '$' ) s++;
01726             if ( *s == 0 ) break;
01727         }
01728     }
01729     AT.WorkPointer[1] = w - AT.WorkPointer;
01730     AddNtoL(AT.WorkPointer[1], AT.WorkPointer);
01731     return(error);
01732 }
01733
01734 /*
01735  #] ExecInside :
01736  #[ InsideDollar :

```



```

01737
01738     Execution part of Inside $a;
01739     We have to take the variables one by one and then
01740     convert them into proper terms and call Generator for the proper levels.
01741     The conversion copies the whole dollar into a new buffer, making us
01742     insensitive to redefinitions of $a inside the Inside.
01743     In the end we sort and redefine $a.
01744 */
01745
01746 int InsideDollar(PHEAD WORD *ll, WORD level)
01747 {
01748     GETBIDENTITY
01749     int numvar = (int)(ll[1]-3), j, error = 0;
01750     WORD numdol, *oldcterm, *oldwork = AT.WorkPointer, olddefer, *r, *m;
01751     WORD oldnumlhs, *dbuffer;
01752     DOLLARS d, newd;
01753     oldcterm = AN.cTerm; AN.cTerm = 0;
01754     oldnumlhs = AR.Cnumlhs; AR.Cnumlhs = ll[2];
01755     ll += 3;
01756     olddefer = AR.DeferFlag;
01757     AR.DeferFlag = 0;
01758     while ( --numvar >= 0 ) {
01759         numdol = *ll++;
01760         d = Dollars + numdol;
01761         {
01762 #ifdef WITHPTHREADS
01763             int nummodopt, dtype = -1;
01764             if ( AS.MultiThreaded && ( AC.mparallelfld == PARALLELFLAG ) ) {
01765                 for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01766                     if ( numdol == ModOptdollars[nummodopt].number ) break;
01767                 }
01768                 if ( nummodopt < NumModOptdollars ) {
01769                     dtype = ModOptdollars[nummodopt].type;
01770                     if ( dtype == MODLOCAL ) {
01771                         d = ModOptdollars[nummodopt].dstruct+AT.identity;
01772                     }
01773                     else {
01774                         LOCK(d->pthreadlockwrite); /*
01775                         LOCK(d->pthreadlockread);
01776                     }
01777                 }
01778             }
01779 #endif
01780             newd = DolToTerms(BHEAD numdol);
01781             if ( newd == 0 ) {
01782                 continue;
01783             }
01784             if ( newd->where[0] == 0 ) {
01785                 // DolToTerms potentially allocates memory. Free it.
01786                 // The free below is inside the while loop.
01787                 if ( newd->factors ) M_free(newd->factors, "Dollar factors");
01788                 M_free(newd, "Copy of dollar variable");
01789                 continue;
01790             }
01791             r = newd->where;
01792             NewSort(BHEAD0);
01793             while ( *r ) { /* Sum over the terms */
01794                 m = AT.WorkPointer;
01795                 j = *r;
01796                 while ( --j >= 0 ) *m++ = *r++;
01797                 AT.WorkPointer = m;
01798             /*
01799             What to do with dummy indices?
01800             */
01801             if ( Generator(BHEAD oldwork, level) ) {
01802                 LowerSortLevel();
01803                 error = -1; goto idcall;
01804             }
01805             AT.WorkPointer = oldwork;
01806         }
01807         AN.tryterm = 0; /* for now */
01808         if ( EndSort(BHEAD (WORD *) ((void *) (&dbuffer)), 2) < 0 ) { error = 1; break; }
01809         if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where, "old buffer of dollar");
01810         d->where = dbuffer;
01811         if ( dbuffer == 0 || *dbuffer == 0 ) {
01812             d->type = DOLZERO;
01813             if ( dbuffer ) M_free(dbuffer, "buffer of dollar");
01814             d->where = &(AM.dollarzero); d->size = 0;
01815         }
01816         else {
01817             d->type = DOLTERMS;
01818             r = d->where; while ( *r ) r += *r;
01819             d->size = (r-d->where)+1;
01820         }
01821         /* cbuf[AM.dbufnum].rhs[numdol] = d->where; */
01822         cbuf[AM.dbufnum].rhs[numdol] = (WORD *) (1);
01823         /*

```

```

01824         Now we have a little cleaning up to do
01825     */
01826 #ifdef WITHPTHREADS
01827     if ( dtype > 0 && dtype != MODLOCAL ) {
01828     /*         UNLOCK(d->pthreadlockwrite); */
01829         UNLOCK(d->pthreadlockread);
01830     }
01831 #endif
01832     if ( newd->factors ) M_free(newd->factors,"Dollar factors");
01833     M_free(newd,"Copy of dollar variable");
01834 }
01835 }
01836 idcall::
01837     AR.Cnumlhs = oldnumlhs;
01838     AR.DeferFlag = olddefer;
01839     AN.cTerm = oldcTerm;
01840     AT.WorkPointer = oldwork;
01841     return(error);
01842 }
01843
01844 /*
01845     #] InsideDollar :
01846     #[ ExchangeDollars :
01847 */
01848
01849 void ExchangeDollars(int num1, int num2)
01850 {
01851     DOLLARS d1, d2;
01852     WORD node1, node2;
01853     LONG nam;
01854     d1 = Dollars + num1; node1 = d1->node;
01855     d2 = Dollars + num2; node2 = d2->node;
01856     nam = d1->name; d1->name = d2->name; d2->name = nam;
01857     d1->node = node2; d2->node = node1;
01858     AC.dollarnames->namenode[node1].number = num2;
01859     AC.dollarnames->namenode[node2].number = num1;
01860 }
01861
01862 /*
01863     #] ExchangeDollars :
01864     #[ TermsInDollar :
01865 */
01866
01867 LONG TermsInDollar(WORD num)
01868 {
01869     GETIDENTITY
01870     DOLLARS d = Dollars + num;
01871     WORD *t;
01872     LONG n;
01873 #ifdef WITHPTHREADS
01874     int nummodopt, dtype = -1;
01875     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01876         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01877             if ( num == ModOptdollars[nummodopt].number ) break;
01878         }
01879         if ( nummodopt < NumModOptdollars ) {
01880             dtype = ModOptdollars[nummodopt].type;
01881             if ( dtype == MODLOCAL ) {
01882                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01883             }
01884             else {
01885                 LOCK(d->pthreadlockread);
01886             }
01887         }
01888     }
01889 #endif
01890     if ( d->type == DOLTERMS ) {
01891         n = 0;
01892         t = d->where;
01893         while ( *t ) { t += *t; n++; }
01894     }
01895     else if ( d->type == DOLWILDARGS ) {
01896         n = 0;
01897         if ( d->where[0] == 0 ) {
01898             t = d->where+1;
01899             while ( *t != 0 ) { NEXTARG(t); n++; }
01900         }
01901         else if ( d->where[0] == 1 ) n = 1;
01902     }
01903     else if ( d->type == DOLZERO ) n = 0;
01904     else n = 1;
01905 #ifdef WITHPTHREADS
01906     if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
01907 #endif
01908     return(n);
01909 }
01910

```

```

01911 /*
01912     #] TermsInDollar :
01913     #[ SizeOfDollar :
01914 */
01915
01916 LONG SizeOfDollar(WORD num)
01917 {
01918     GETIDENTITY
01919     DOLLARS d = Dollars + num;
01920     WORD *t;
01921     LONG n;
01922 #ifndef WITHPTHREADS
01923     int nummodopt, dtype = -1;
01924     if ( AS.MultiThreaded && ( AC.mparallelfld == PARALLELFLAG ) ) {
01925         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01926             if ( num == ModOptdollars[nummodopt].number ) break;
01927         }
01928         if ( nummodopt < NumModOptdollars ) {
01929             dtype = ModOptdollars[nummodopt].type;
01930             if ( dtype == MODLOCAL ) {
01931                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01932             }
01933             else {
01934                 LOCK(d->pthreadlockread);
01935             }
01936         }
01937     }
01938 #endif
01939     if ( d->type == DOLTERMS ) {
01940         t = d->where;
01941         while ( *t ) t += *t;
01942         t++;
01943         n = (LONG)(t - d->where);
01944     }
01945     else if ( d->type == DOLWILDARGS ) {
01946         n = 0;
01947         if ( d->where[0] == 0 ) {
01948             t = d->where+1;
01949             while ( *t != 0 ) { NEXTARG(t); n++; }
01950             t++;
01951             n = (LONG)(t - d->where);
01952         }
01953         else if ( d->where[0] == 1 ) n = 1;
01954     }
01955     else if ( d->type == DOLZERO ) n = 0;
01956     else n = 1;
01957 #ifdef WITHPTHREADS
01958     if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
01959 #endif
01960     return(n);
01961 }
01962
01963 /*
01964     #] SizeOfDollar :
01965     #[ PreIfDollarEval :
01966
01967     Routine is invoked in #if etc after $( is encountered.
01968     $(expr1 operator expr2) makes compares between expressions,
01969     $(expr1 operator _keyword) makes compares between expressions,
01970     interpreted as expressions. We are here mainly looking at $variables.
01971     First we look for the operator:
01972         >, <, ==, >=, <=, != : < means that it comes before.
01973     _keywords can be:
01974         _set(setname)    (does the expr belong to the set (only with == or !=))
01975         _productof(expr)
01976 */
01977
01978 UBYTE *PreIfDollarEval(UBYTE *s, int *value)
01979 {
01980     GETIDENTITY
01981     UBYTE *s1,*s2,*s3,*s4,*s5,*t,c,c1,c2,c3;
01982     int oprtr, type;
01983     WORD *buf1 = 0, *buf2 = 0, numset, *oldwork = AT.WorkPointer;
01984     EXCHINOUT
01985 /*
01986     Find the three composing objects (epxpression, operator, expression or keyw
01987 */
01988     while ( *s == ' ' || *s == '\t' || *s == '\n' || *s == '\r' ) s++;
01989     s1 = t = s;
01990     while ( *t != '=' && *t != '!' && *t != '>' && *t != '<' ) {
01991         if ( *t == '[' ) { SKIPBRA1(t) }
01992         else if ( *t == '{' ) { SKIPBRA2(t) }
01993         else if ( *t == '(' ) { SKIPBRA3(t) }
01994         else if ( *t == ']' || *t == '}' || *t == ')' ) {
01995             MLOCK(ErrorMessageLock);
01996             MesPrint("@Improper bracketting in #if");
01997             MUNLOCK(ErrorMessageLock);

```

```

01998         goto onerror;
01999     }
02000     t++;
02001 }
02002 s2 = t;
02003 while ( *t == '=' || *t == '!' || *t == '>' || *t == '<' ) t++;
02004 s3 = t;
02005 while ( *t && *t != ')' ) {
02006     if ( *t == '[' ) { SKIPBRA1(t) }
02007     else if ( *t == '{' ) { SKIPBRA2(t) }
02008     else if ( *t == '(' ) { SKIPBRA3(t) }
02009     else if ( *t == ']' || *t == '}' ) {
02010         MLOCK(ErrorMessageLock);
02011         MesPrint("@Improper brackets in #if");
02012         MUNLOCK(ErrorMessageLock);
02013         goto onerror;
02014     }
02015     t++;
02016 }
02017 if ( *t == 0 ) {
02018     MLOCK(ErrorMessageLock);
02019     MesPrint("@Missing ) to match $( in #if");
02020     MUNLOCK(ErrorMessageLock);
02021     goto onerror;
02022 }
02023 s4 = t; c2 = *s4; *s4 = 0;
02024 if ( s2+2 < s3 || s2 == s3 ) {
02025     IllOp;;
02026     MLOCK(ErrorMessageLock);
02027     MesPrint("@Illegal operator in $( option of #if");
02028     MUNLOCK(ErrorMessageLock);
02029     goto onerror;
02030 }
02031 if ( s2+1 == s3 ) {
02032     if ( *s2 == '=' ) oprtr = EQUAL;
02033     else if ( *s2 == '>' ) oprtr = GREATER;
02034     else if ( *s2 == '<' ) oprtr = LESS;
02035     else goto IllOp;
02036 }
02037 else if ( *s2 == '!' && s2[1] == '=' ) oprtr = NOTEQUAL;
02038 else if ( *s2 == '=' && s2[1] == '=' ) oprtr = EQUAL;
02039 else if ( *s2 == '<' && s2[1] == '=' ) oprtr = LESSEQUAL;
02040 else if ( *s2 == '>' && s2[1] == '=' ) oprtr = GREATEREQUAL;
02041 else goto IllOp;
02042 c1 = *s2; *s2 = 0;
02043 /*
02044     The two expressions are now zero terminated
02045     Look for the special keywords
02046 */
02047 while ( *s3 == ' ' || *s3 == '\t' || *s3 == '\n' || *s3 == '\r' ) s3++;
02048 t = s3;
02049 while ( chartype[*t] == 0 ) t++;
02050 if ( *t == '_' ) {
02051     t++; c = *t; *t = 0;
02052     if ( StrICmp(s3, (UBYTE *) "set_") == 0 ) {
02053         if ( oprtr != EQUAL && oprtr != NOTEQUAL ) {
02054             ImpOp;;
02055             MLOCK(ErrorMessageLock);
02056             MesPrint("@Improper operator for special keyword in $( ) option");
02057             MUNLOCK(ErrorMessageLock);
02058             goto onerror;
02059         }
02060         type = 1;
02061     }
02062     else if ( StrICmp(s3, (UBYTE *) "multipleof_") == 0 ) {
02063         if ( oprtr != EQUAL && oprtr != NOTEQUAL ) goto ImpOp;
02064         type = 2;
02065     }
02066     /*
02067     else if ( StrICmp(s3, (UBYTE *) "productof_") == 0 ) {
02068         if ( oprtr != EQUAL && oprtr != NOTEQUAL ) goto ImpOp;
02069         type = 3;
02070     }
02071 */
02072     else type = 0;
02073 }
02074 else { type = 0; c = *t; }
02075 if ( type > 0 ) {
02076     *t++ = c; s3 = t; s5 = s4-1;
02077     while ( *s5 != ')' ) {
02078         if ( *s5 == ' ' || *s5 == '\t' || *s5 == '\n' || *s5 == '\r' ) s5--;
02079         else {
02080             MLOCK(ErrorMessageLock);
02081             MesPrint("@Improper use of special keyword in $( ) option");
02082             MUNLOCK(ErrorMessageLock);
02083             goto onerror;
02084         }

```

```

02085     }
02086     c3 = *s5; *s5 = 0;
02087 }
02088 else { c3 = c2; s5 = s4; }
02089 /*
02090 Expand the first expression.
02091 */
02092 if ( ( buf1 = TranslateExpression(s1) ) == 0 ) {
02093     AT.WorkPointer = oldwork;
02094     goto onerror;
02095 }
02096 if ( type == 1 ) { /* determine the set */
02097     if ( *s3 == '{' ) {
02098         t = s3+1;
02099         SKIPBRA2(s3)
02100         numset = DoTempSet(t,s3);
02101         s3++;
02102         if ( numset < 0 ) {
02103 noset::
02104             MLOCK(ErrorMessageLock);
02105             MesPrint("@Argument of set_ is not a valid set");
02106             MUNLOCK(ErrorMessageLock);
02107             goto onerror;
02108         }
02109     }
02110     else {
02111         t = s3;
02112         while ( FG.cTable[*s3] == 0 || FG.cTable[*s3] == 1
02113             || *s3 == '_' ) s3++;
02114         c = *s3; *s3 = 0;
02115         if ( GetName(AC.varnames,t,&numset,NOAUTO) != CSET ) {
02116             *s3 = c; goto noset;
02117         }
02118         *s3 = c;
02119     }
02120     while ( *s3 == ' ' || *s3 == '\t' || *s3 == '\n' || *s3 == '\r' ) s3++;
02121     if ( s3 != s5 ) goto noset;
02122     *value = IsSetMember(buf1,numset);
02123     if ( oprtr == NOTEQUAL ) *value ^= 1;
02124 }
02125 else {
02126     if ( ( buf2 = TranslateExpression(s3) ) == 0 ) goto onerror;
02127 }
02128 if ( type == 0 ) {
02129     *value = TwoExprCompare(buf1,buf2,oprtr);
02130 }
02131 else if ( type == 2 ) {
02132     *value = IsMultipleOf(buf1,buf2);
02133     if ( oprtr == NOTEQUAL ) *value ^= 1;
02134 }
02135 /*
02136 else if ( type == 3 ) {
02137     *value = IsProductOf(buf1,buf2);
02138     if ( oprtr == NOTEQUAL ) *value ^= 1;
02139 }
02140 */
02141 if ( buf1 ) M_free(buf1,"Buffer in $()");
02142 if ( buf2 ) M_free(buf2,"Buffer in $()");
02143 *s5 = c3; *s4++ = c2; *s2 = c1;
02144 AT.WorkPointer = oldwork;
02145 BACKINOUT
02146 return(s4);
02147 onerror:
02148 if ( buf1 ) M_free(buf1,"Buffer in $()");
02149 if ( buf2 ) M_free(buf2,"Buffer in $()");
02150 AT.WorkPointer = oldwork;
02151 BACKINOUT
02152 return(0);
02153 }
02154
02155 /*
02156 #] PreIfDollarEval :
02157 #[ TranslateExpression :
02158 */
02159
02160 WORD *TranslateExpression(UBYTE *s)
02161 {
02162     GETIDENTITY
02163     CBUF *C = cbuf+AC.cbunum;
02164     WORD oldnumrhs = C->numrhs;
02165     LONG oldcpointer = C->Pointer - C->Buffer;
02166     WORD *w = AT.WorkPointer;
02167     WORD retcode, oldEside;
02168     WORD *outbuffer;
02169     *w++ = SUBEXPSize + 4;
02170     AC.ProtoType = w;
02171     *w++ = SUBEXPRESSION;

```

```

02172     *w++ = SUBEXPSIZE;
02173     *w++ = C->numrhs+1;
02174     *w++ = 1;
02175     *w++ = AC.cbufnum;
02176     FILLSUB(w)
02177     *w++ = 1; *w++ = 1; *w++ = 3; *w++ = 0;
02178     AT.WorkPointer = w;
02179     if ( ( retcode = CompileAlgebra(s,RHSIDE,AC.ProtoType) ) < 0 ) {
02180         MLOCK(ErrorMessageLock);
02181         MesPrint("@Error translating first expression in $( ) option");
02182         MUNLOCK(ErrorMessageLock);
02183         return(0);
02184     }
02185     else { AC.ProtoType[2] = retcode; }
02186 /*
02187     Evaluate this expression
02188 */
02189     if ( NewSort(BHEAD0) || NewSort(BHEAD0) ) { return(0); }
02190     AN.RepPoint = AT.RepCount + 1;
02191     oldEside = AR.Eside; AR.Eside = RHSIDE;
02192     AR.Cnumlhs = C->numlhs;
02193     if ( Generator(BHEAD AC.ProtoType-1,C->numlhs) ) {
02194         AR.Eside = oldEside;
02195         LowerSortLevel(); LowerSortLevel(); return(0);
02196     }
02197     AR.Eside = oldEside;
02198     AT.WorkPointer = w;
02199     AN.tryterm = 0; /* for now */
02200     if ( EndSort(BHEAD (WORD *)((void *)&outbuffer)),2) < 0 ) { LowerSortLevel(); return(0); }
02201     LowerSortLevel();
02202     C->Pointer = C->Buffer + oldcpointer;
02203     C->numrhs = oldnumrhs;
02204     AT.WorkPointer = AC.ProtoType - 1;
02205     return(outbuffer);
02206 }
02207
02208 /*
02209     #] TranslateExpression :
02210     #[ IsSetMember :
02211
02212     Checks whether the expression in the buffer can be seen as an element
02213     of the given set.
02214     For the special sets: if more than one term: no match!!!
02215 */
02216
02217 int IsSetMember(WORD *buffer, WORD numset)
02218 {
02219     WORD *t = buffer, *tt, num, csize, num1;
02220     WORD bufterm[4];
02221     int i, j, type;
02222     if ( numset < AM.NumFixedSets ) {
02223         if ( t[*t] != 0 ) return(0); /* More than one term */
02224         if ( *t == 0 ) {
02225             if ( numset == POS0_ || numset == NEG0_ || numset == EVEN_
02226                 || numset == Z_ || numset == Q_ ) return(1);
02227             else return(0);
02228         }
02229         if ( numset == SYMBOL_ ) {
02230             if ( *t == 8 && t[1] == SYMBOL && t[7] == 3 && t[6] == 1
02231                 && t[5] == 1 && t[4] == 1 ) return(1);
02232             else return(0);
02233         }
02234         if ( numset == INDEX_ ) {
02235             if ( *t == 7 && t[1] == INDEX && t[6] == 3 && t[5] == 1
02236                 && t[4] == 1 && t[3] > 0 ) return(1);
02237             if ( *t == 4 && t[3] == 3 && t[2] == 1 && t[1] < AM.OffsetIndex )
02238                 return(1);
02239             return(0);
02240         }
02241         if ( numset == FIXED_ ) {
02242             if ( *t == 7 && t[1] == INDEX && t[6] == 3 && t[5] == 1
02243                 && t[4] == 1 && t[3] > 0 && t[3] < AM.OffsetIndex ) return(1);
02244             if ( *t == 4 && t[3] == 3 && t[2] == 1 && t[1] < AM.OffsetIndex )
02245                 return(1);
02246             return(0);
02247         }
02248         if ( numset == DUMMYINDEX_ ) {
02249             if ( *t == 7 && t[1] == INDEX && t[6] == 3 && t[5] == 1
02250                 && t[4] == 1 && t[3] >= AM.IndDum && t[3] < AM.IndDum+MAXDUMMIES ) return(1);
02251             if ( *t == 4 && t[3] == 3 && t[2] == 1
02252                 && t[1] >= AM.IndDum && t[1] < AM.IndDum+MAXDUMMIES ) return(1);
02253             return(0);
02254         }
02255         if ( numset == VECTOR_ ) {
02256             if ( *t == 7 && t[1] == INDEX && t[6] == 3 && t[5] == 1
02257                 && t[4] == 1 && t[3] < (AM.OffsetVector+WILDOFFSET) && t[3] >= AM.OffsetVector )
02258                 return(1);

```

```

02258         return(0);
02259     }
02260     tt = t + *t - 1;
02261     if ( ABS(tt[0]) != *t-1 ) return(0);
02262     if ( numset == Q_ ) return(1);
02263     if ( numset == POS_ || numset == POS0_ ) return(tt[0]>0);
02264     else if ( numset == NEG_ || numset == NEG0_ ) return(tt[0]<0);
02265     i = (ABS(tt[0])-1)/2;
02266     tt -= i;
02267     if ( tt[0] != 1 ) return(0);
02268     for ( j = 1; j < i; j++ ) { if ( tt[j] != 0 ) return(0); }
02269     if ( numset == Z_ ) return(1);
02270     if ( numset == ODD_ ) return(t[1]&1);
02271     if ( numset == EVEN_ ) return(1-(t[1]&1));
02272     return(0);
02273 }
02274 if ( t[*t] != 0 ) return(0); /* More than one term */
02275 type = Sets[numset].type;
02276 switch ( type ) {
02277     case CSYMBOL:
02278         if ( t[0] == 8 && t[1] == SYMBOL && t[7] == 3 && t[6] == 1
02279             && t[5] == 1 && t[4] == 1 ) {
02280             num = t[3];
02281         }
02282         else if ( t[0] == 4 && t[2] == 1 && t[1] <= MAXPOWER ) {
02283             num = t[1];
02284             if ( t[3] < 0 ) num = -num;
02285             num += 2*MAXPOWER;
02286         }
02287         else return(0);
02288         break;
02289     case CVECTOR:
02290         if ( t[0] == 7 && t[1] == INDEX && t[6] == 3 && t[5] == 1
02291             && t[4] == 1 && t[3] < 0 ) {
02292             num = t[3];
02293         }
02294         else return(0);
02295         break;
02296     case CINDEX:
02297         if ( t[0] == 7 && t[1] == INDEX && t[6] == 3 && t[5] == 1
02298             && t[4] == 1 && t[3] > 0 ) {
02299             num = t[3];
02300         }
02301         else if ( t[0] == 4 && t[3] == 3 && t[2] == 1 && t[1] < AM.OffsetIndex ) {
02302             num = t[1];
02303         }
02304         else return(0);
02305         break;
02306     case CFUNCTION:
02307         if ( t[0] == 4+FUNHEAD && t[3+FUNHEAD] == 3 && t[2+FUNHEAD] == 1
02308             && t[1+FUNHEAD] == 1 && t[1] >= FUNCTION ) {
02309             num = t[1];
02310         }
02311         else return(0);
02312         break;
02313     case CNUMBER:
02314         if ( t[0] == 4 && t[2] == 1 && t[1] <= AM.OffsetIndex && t[3] == 3 ) {
02315             num = t[1];
02316         }
02317         else return(0);
02318         break;
02319     case CRANGE:
02320         csize = t[t[0]-1];
02321         csize = ABS(csize);
02322         if ( csize != t[0]-1 ) return(0);
02323         if ( Sets[numset].first < 3*MAXPOWER ) {
02324             num1 = num = Sets[numset].first;
02325             if ( num >= MAXPOWER ) num -= 2*MAXPOWER;
02326             if ( num == 0 ) {
02327                 if ( num1 < MAXPOWER ) {
02328                     if ( t[t[0]-1] >= 0 ) return(0);
02329                 }
02330                 else if ( t[t[0]-1] > 0 ) return(0);
02331             }
02332             else {
02333                 bufterm[0] = 4; bufterm[1] = ABS(num);
02334                 bufterm[2] = 1;
02335                 if ( num < 0 ) bufterm[3] = -3;
02336                 else bufterm[3] = 3;
02337                 num = CompCoef(t,bufterm);
02338                 if ( num1 < MAXPOWER ) {
02339                     if ( num >= 0 ) return(0);
02340                 }
02341                 else if ( num > 0 ) return(0);
02342             }
02343         }
02344         if ( Sets[numset].last > -3*MAXPOWER ) {

```

```

02345         num1 = num = Sets[numset].last;
02346         if ( num <= -MAXPOWER ) num += 2*MAXPOWER;
02347         if ( num == 0 ) {
02348             if ( num1 > -MAXPOWER ) {
02349                 if ( t[t[0]-1] <= 0 ) return(0);
02350             }
02351             else if ( t[t[0]-1] < 0 ) return(0);
02352         }
02353         else {
02354             bufterm[0] = 4; bufterm[1] = ABS(num);
02355             bufterm[2] = 1;
02356             if ( num < 0 ) bufterm[3] = -3;
02357             else bufterm[3] = 3;
02358             num = CompCof(t,bufterm);
02359             if ( num1 > -MAXPOWER ) {
02360                 if ( num <= 0 ) return(0);
02361             }
02362             else if ( num < 0 ) return(0);
02363         }
02364     }
02365     return(1);
02366     break;
02367     default: return(0);
02368 }
02369 t = SetElements + Sets[numset].first;
02370 tt = SetElements + Sets[numset].last;
02371 do {
02372     if ( num == *t ) return(1);
02373     t++;
02374 } while ( t < tt );
02375 return(0);
02376 }
02377
02378 /*
02379 #] IsSetMember :
02380 #[ IsProductOf :
02381
02382 Checks whether the expression in buf1 is a single term multiple of
02383 the expression in buf2.
02384
02385 int IsProductOf(WORD *buf1, WORD *buf2)
02386 {
02387     return(0);
02388 }
02389
02390
02391 #] IsProductOf :
02392 #[ IsMultipleOf :
02393
02394 Checks whether the expression in buf1 is a numerical multiple of
02395 the expression in buf2.
02396 */
02397
02398 int IsMultipleOf(WORD *buf1, WORD *buf2)
02399 {
02400     GETIDENTITY
02401     LONG num1, num2;
02402     WORD *t1, *t2, *m1, *m2, *r1, *r2, nc1, nc2, ni1, ni2;
02403     UWORD *IfScrat1, *IfScrat2;
02404     int i, j;
02405     if ( *buf1 == 0 && *buf2 == 0 ) return(1);
02406 /*
02407     First count terms
02408 */
02409     t1 = buf1; t2 = buf2; num1 = 0; num2 = 0;
02410     while ( *t1 ) { t1 += *t1; num1++; }
02411     while ( *t2 ) { t2 += *t2; num2++; }
02412     if ( num1 != num2 ) return(0);
02413 /*
02414     Test similarity of terms. Difference up to a number.
02415 */
02416     t1 = buf1; t2 = buf2;
02417     while ( *t1 ) {
02418         m1 = t1+1; m2 = t2+1; t1 += *t1; t2 += *t2;
02419         r1 = t1 - ABS(t1[-1]); r2 = t2 - ABS(t2[-1]);
02420         if ( r1-m1 != r2-m2 ) return(0);
02421         while ( m1 < r1 ) {
02422             if ( *m1 != *m2 ) return(0);
02423             m1++; m2++;
02424         }
02425     }
02426 /*
02427     Now we have to test the constant factor
02428 */
02429     IfScrat1 = (UWORD *) (TermMalloc("IsMultipleOf")); IfScrat2 = (UWORD
02430 *) (TermMalloc("IsMultipleOf"));
02431     t1 = buf1; t2 = buf2;

```



```

02431     t1 += *t1; t2 += *t2;
02432     if ( *t1 == 0 && *t2 == 0 ) return(1);
02433     r1 = t1 - ABS(t1[-1]); r2 = t2 - ABS(t2[-1]);
02434     nc1 = REDLENG(t1[-1]); nc2 = REDLENG(t2[-1]);
02435     if ( DivRat(BHEAD (UWORD *)r1,nc1,(UWORD *)r2,nc2,IfScrat1,&ni1) ) {
02436         MLOCK(ErrorMessageLock);
02437         MesPrint("@Called from MultipleOf in $( )");
02438         MUNLOCK(ErrorMessageLock);
02439         TermFree(IfScrat1,"IsMultipleOf"); TermFree(IfScrat2,"IsMultipleOf");
02440         Terminate(-1);
02441     }
02442     while ( *t1 ) {
02443         t1 += *t1; t2 += *t2;
02444         r1 = t1 - ABS(t1[-1]); r2 = t2 - ABS(t2[-1]);
02445         nc1 = REDLENG(t1[-1]); nc2 = REDLENG(t2[-1]);
02446         if ( DivRat(BHEAD (UWORD *)r1,nc1,(UWORD *)r2,nc2,IfScrat2,&ni2) ) {
02447             MLOCK(ErrorMessageLock);
02448             MesPrint("@Called from MultipleOf in $( )");
02449             MUNLOCK(ErrorMessageLock);
02450             TermFree(IfScrat1,"IsMultipleOf"); TermFree(IfScrat2,"IsMultipleOf");
02451             Terminate(-1);
02452         }
02453         if ( ni1 != ni2 ) return(0);
02454         i = 2*ABS(ni1);
02455         for ( j = 0; j < i; j++ ) {
02456             if ( IfScrat1[j] != IfScrat2[j] ) {
02457                 TermFree(IfScrat1,"IsMultipleOf"); TermFree(IfScrat2,"IsMultipleOf");
02458                 return(0);
02459             }
02460         }
02461     }
02462     TermFree(IfScrat1,"IsMultipleOf"); TermFree(IfScrat2,"IsMultipleOf");
02463     return(1);
02464 }
02465
02466 /*
02467 #] IsMultipleOf :
02468 #[ TwoExprCompare :
02469
02470 Compares the expressions in buf1 and buf2 according to oprtr
02471 */
02472
02473 int TwoExprCompare(WORD *buf1, WORD *buf2, int oprtr)
02474 {
02475     GETIDENTITY
02476     WORD *t1, *t2, cond;
02477     t1 = buf1; t2 = buf2;
02478     while ( *t1 && *t2 ) {
02479         cond = CompareTerms(BHEAD t1,t2,1);
02480         if ( cond != 0 ) {
02481             if ( cond > 0 ) { /* t1 comes first */
02482                 switch ( oprtr ) { /* t1 is less */
02483                     case EQUAL: return(0);
02484                     case NOTEQUAL: return(1);
02485                     case GREATEREQUAL: return(0);
02486                     case GREATER: return(0);
02487                     case LESS: return(1);
02488                     case LESSEQUAL: return(1);
02489                 }
02490             }
02491             else {
02492                 switch ( oprtr ) {
02493                     case EQUAL: return(0);
02494                     case NOTEQUAL: return(1);
02495                     case GREATEREQUAL: return(1);
02496                     case GREATER: return(1);
02497                     case LESS: return(0);
02498                     case LESSEQUAL: return(0);
02499                 }
02500             }
02501         }
02502         t1 += *t1; t2 += *t2;
02503     }
02504     if ( *t1 == *t2 ) { /* They are equal */
02505         switch ( oprtr ) {
02506             case EQUAL: return(1);
02507             case NOTEQUAL: return(0);
02508             case GREATEREQUAL: return(1);
02509             case GREATER: return(0);
02510             case LESS: return(0);
02511             case LESSEQUAL: return(1);
02512         }
02513     }
02514     else if ( *t1 ) { /* t1 is greater */
02515         switch ( oprtr ) {
02516             case EQUAL: return(0);
02517             case NOTEQUAL: return(1);

```

```

02518         case GREATEREQUAL: return(1);
02519         case GREATER: return(1);
02520         case LESS: return(0);
02521         case LESSEQUAL: return(0);
02522     }
02523 }
02524 else {
02525     switch ( oprtr ) { /* t1 is less */
02526         case EQUAL: return(0);
02527         case NOTEQUAL: return(1);
02528         case GREATEREQUAL: return(0);
02529         case GREATER: return(0);
02530         case LESS: return(1);
02531         case LESSEQUAL: return(1);
02532     }
02533 }
02534 MLOCK(ErrorMessageLock);
02535 MesPrint("@Internal problems with operator in $( )");
02536 MUNLOCK(ErrorMessageLock);
02537 Terminate(-1);
02538 return(0);
02539 }
02540
02541 /*
02542  #] TwoExprCompare :
02543  #[ DollarRaiseLow :
02544
02545  Raises or lowers the numerical value of a dollar variable
02546  Not to be used in parallel.
02547 */
02548
02549 static UWORD *dscrat = 0;
02550 static WORD ndscrat;
02551
02552 int DollarRaiseLow(UBYTE *name, LONG value)
02553 {
02554     GETIDENTITY
02555     int num;
02556     DOLLARS d;
02557     int sgn = 1;
02558     WORD lnum[4], nnum, *t1, *t2, i;
02559     UBYTE *s, c;
02560     s = name; while ( *s ) s++;
02561     if ( s[-1] == '-' && s[-2] == '-' && s > name+2 ) s -= 2;
02562     else if ( s[-1] == '+' && s[-2] == '+' && s > name+2 ) s -= 2;
02563     c = *s; *s = 0;
02564     num = GetDollar(name);
02565     *s = c;
02566     d = Dollars + num;
02567     if ( value < 0 ) { value = -value; sgn = -1; }
02568     if ( d->type == DOLZERO ) {
02569         if ( d->where ) M_free(d->where,"DollarRaiseLow");
02570         d->size = MINALLOC;
02571         d->where = (WORD *)Malloc1(d->size*sizeof(WORD),"DollarRaiseLow");
02572         if ( ( value & AWORDMASK ) != 0 ) {
02573             d->where[0] = 6; d->where[1] = value » BITSINWORD;
02574             d->where[2] = (WORD)value; d->where[3] = 1; d->where[4] = 0;
02575             d->where[5] = 5*sgn; d->where[6] = 0;
02576             d->type = DOLTERMS;
02577         }
02578     }
02579     else {
02580         d->where[0] = 4; d->where[1] = (WORD)value; d->where[2] = 1;
02581         d->where[3] = 3*sgn; d->where[4] = 0;
02582         d->type = DOLNUMBER;
02583     }
02584     else if ( d->type == DOLNUMBER || ( d->type == DOLTERMS
02585     && d->where[d->where[0]] == 0
02586     && d->where[0] == ABS(d->where[d->where[0]-1])+1 ) ) {
02587         if ( ( value & AWORDMASK ) != 0 ) {
02588             lnum[0] = value » BITSINWORD;
02589             lnum[1] = (WORD)value; lnum[2] = 1; lnum[3] = 0;
02590             nnum = 2*sgn;
02591         }
02592         else {
02593             lnum[0] = (WORD)value; lnum[1] = 1; nnum = sgn;
02594         }
02595         i = d->where[d->where[0]-1];
02596         i = REDLENG(i);
02597         if ( dscrat == 0 ) {
02598             dscrat = (UWORD *)Malloc1((AM.MaxTal+2)*sizeof(UWORD),"DollarRaiseLow");
02599         }
02600         if ( AddRat(BHEAD (UWORD *) (d->where+1), i,
02601         (UWORD *)lnum, nnum, dscrat, &ndscrat) ) {
02602             MLOCK(ErrorMessageLock);
02603             MesCall("DollarRaiseLow");
02604             MUNLOCK(ErrorMessageLock);

```

```

02605         Terminate(-1);
02606     }
02607     ndscrat = INCLENG(ndscrat);
02608     i = ABS(ndscrat);
02609     if ( i == 0 ) {
02610         M_free(d->where,"DollarRaiseLow");
02611         d->where = 0;
02612         d->type = DOLZERO;
02613         d->size = 0;
02614         return(0);
02615     }
02616     if ( i+2 > d->size ) {
02617         M_free(d->where,"DollarRaiseLow");
02618         d->size = i+2;
02619         if ( d->size < MINALLOC ) d->size = MINALLOC;
02620         d->size = ((d->size+7)/8)*8;
02621         d->where = (WORD *)Malloc1(d->size*sizeof(WORD),"DollarRaiseLow");
02622     }
02623     t1 = d->where; *t1++ = i+1; t2 = (WORD *)dscrat;
02624     while ( --i > 0 ) *t1++ = *t2++;
02625     *t1++ = ndscrat; *t1 = 0;
02626     d->type = DOLTERMS;
02627 }
02628 return(0);
02629 }
02630
02631 /*
02632  #] DollarRaiseLow :
02633  #[ EvalDoLoopArg :
02634  */
02651 WORD EvalDoLoopArg(PHEAD WORD *arg, WORD par)
02652 {
02653     WORD num, type, *td;
02654     DOLLARS d;
02655     if ( *arg == SNUMBER ) return(arg[1]);
02656     if ( *arg == DOLLAREXPR2 && arg[1] < 0 ) return(-arg[1]-1);
02657     d = Dollars + arg[1];
02658     #ifdef WITHPTHREADS
02659     {
02660         int nummodopt, dtype = -1;
02661         if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
02662             for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
02663                 if ( arg[1] == ModOptdollars[nummodopt].number ) break;
02664             }
02665             if ( nummodopt < NumModOptdollars ) {
02666                 dtype = ModOptdollars[nummodopt].type;
02667                 if ( dtype == MODLOCAL ) {
02668                     d = ModOptdollars[nummodopt].dstruct+AT.identity;
02669                 }
02670             }
02671         }
02672     }
02673     #endif
02674     if ( *arg == DOLLAREXPRESSION ) {
02675         if ( arg[2] != DOLLAREXPR2 ) { /* end of chain */
02676             endofchain:
02677             type = d->type;
02678             if ( type == DOLZERO ) {}
02679             else if ( type == DOLNUMBER ) {
02680                 td = d->where;
02681                 if ( ( td[0] != 4 ) || ( (td[1]&SPECMASK) != 0 ) || ( td[2] != 1 ) ) {
02682                     MLOCK(ErrorMessageLock);
02683                     if ( par == -1 ) {
02684                         MesPrint("$-variable is not a short number in print statement");
02685                     }
02686                     else {
02687                         MesPrint("$-variable is not a short number in do loop");
02688                     }
02689                     MUNLOCK(ErrorMessageLock);
02690                     Terminate(-1);
02691                 }
02692                 return( td[3] > 0 ? td[1]: -td[1] );
02693             }
02694             else {
02695                 MLOCK(ErrorMessageLock);
02696                 if ( par == -1 ) {
02697                     MesPrint("$-variable is not a number in print statement");
02698                 }
02699                 else {
02700                     MesPrint("$-variable is not a number in do loop");
02701                 }
02702                 MUNLOCK(ErrorMessageLock);
02703                 Terminate(-1);
02704             }
02705             return(0);
02706         }
02707         num = EvalDoLoopArg(BHEAD arg+2,par);

```

```

02708     }
02709     else if ( *arg == DOLLAREXP2 ) {
02710         if ( arg[1] < 0 ) { num = -arg[1]-1; }
02711         else if ( arg[2] != DOLLAREXP2 && par == -1 ) {
02712             goto endofchain;
02713         }
02714         else { num = EvalDoLoopArg(BHEAD arg+2,par); }
02715     }
02716     else {
02717         MLOCK(ErrorMessageLock);
02718         if ( par == -1 ) {
02719             MesPrint("Invalid $-variable in print statement");
02720         }
02721         else {
02722             MesPrint("Invalid $-variable in do loop");
02723         }
02724         MUNLOCK(ErrorMessageLock);
02725         Terminate(-1);
02726         return(0);
02727     }
02728     if ( num == 0 ) return(d->nfactors);
02729     if ( num > d->nfactors || num < 1 ) {
02730         MLOCK(ErrorMessageLock);
02731         if ( par == -1 ) {
02732             MesPrint("Not a valid factor number for $-variable in print statement");
02733         }
02734         else {
02735             MesPrint("Not a valid factor number for $-variable in do loop");
02736         }
02737         MUNLOCK(ErrorMessageLock);
02738         Terminate(-1);
02739         return(0);
02740     }
02741     if ( d->factors[num].type == DOLNUMBER )
02742         return(d->factors[num].value);
02743     else { /* If correct, type can only be DOLNUMBER or DOLTERMS */
02744         MLOCK(ErrorMessageLock);
02745         if ( par == -1 ) {
02746             MesPrint("$-variable in print statement is not a number");
02747         }
02748         else {
02749             MesPrint("$-variable in do loop is not a number");
02750         }
02751         MUNLOCK(ErrorMessageLock);
02752         Terminate(-1);
02753         return(0);
02754     }
02755 }
02756
02757 /*
02758 #] EvalDoLoopArg :
02759 #[ TestDoLoop :
02760 */
02761
02762 WORD TestDoLoop(PHEAD WORD *lhsbuf, WORD level)
02763 {
02764     GETBIDENTITY
02765     WORD start,finish,incr;
02766     WORD *h;
02767     DOLLARS d;
02768     h = lhsbuf + 4; /* address of the start value */
02769     start = EvalDoLoopArg(BHEAD h,0);
02770     while ( ( *h == DOLLAREXP2 || *h == DOLLAREXP2 )
02771         && ( h[2] == DOLLAREXP2 ) ) h += 2;
02772     h += 2;
02773     finish = EvalDoLoopArg(BHEAD h,0);
02774     while ( ( *h == DOLLAREXP2 || *h == DOLLAREXP2 )
02775         && ( h[2] == DOLLAREXP2 ) ) h += 2;
02776     h += 2;
02777     incr = EvalDoLoopArg(BHEAD h,0);
02778
02779     if ( ( finish == start ) || ( finish > start && incr > 0 )
02780         || ( finish < start && incr < 0 ) ) {}
02781     else { level = lhsbuf[3]; } /* skips the loop */
02782 /*
02783     Put start in the dollar variable indicated by lhsbuf[2]
02784 */
02785     d = Dollars + lhsbuf[2];
02786 #ifdef WITHPTHREADS
02787     {
02788         int nummodopt, dtype = -1;
02789         if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
02790             for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
02791                 if ( lhsbuf[2] == ModOptdollars[nummodopt].number ) break;
02792             }
02793             if ( nummodopt < NumModOptdollars ) {
02794                 dtype = ModOptdollars[nummodopt].type;

```

```

02795             if ( dtype == MODLOCAL ) {
02796                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
02797             }
02798         }
02799     }
02800 }
02801 #endif
02802
02803     if ( d->size < MINALLOC ) {
02804         if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollar contents");
02805         d->size = MINALLOC;
02806         d->where = (WORD *)Malloc1(d->size*sizeof(WORD),"dollar contents");
02807     }
02808     if ( start > 0 ) {
02809         d->where[0] = 4;
02810         d->where[1] = start;
02811         d->where[2] = 1;
02812         d->where[3] = 3;
02813         d->where[4] = 0;
02814         d->type = DOLNUMBER;
02815     }
02816     else if ( start < 0 ) {
02817         d->where[0] = 4;
02818         d->where[1] = -start;
02819         d->where[2] = 1;
02820         d->where[3] = -3;
02821         d->where[4] = 0;
02822         d->type = DOLNUMBER;
02823     }
02824     else
02825         d->type = DOLZERO;
02826
02827     if ( d == Dollars + lhsbuf[2] ) {
02828         cbuf[AM.dbufnum].CanCommu[lhsbuf[2]] = 0;
02829         cbuf[AM.dbufnum].NumTerms[lhsbuf[2]] = 1;
02830         cbuf[AM.dbufnum].rhs[lhsbuf[2]] = d->where;
02831     }
02832     return(level);
02833 }
02834
02835 /*
02836 #] TestDoLoop :
02837 #[ TestEndDoLoop :
02838 */
02839
02840 WORD TestEndDoLoop(PHEAD WORD *lhsbuf, WORD level)
02841 {
02842     GETBIDENTITY
02843     WORD start,finish,incr,value;
02844     WORD *h;
02845     DOLLARS d;
02846     h = lhsbuf + 4; /* address of the start value */
02847     start = EvalDoLoopArg(BHEAD h,0);
02848     while ( ( *h == DOLLAREXPRESSION || *h == DOLLAREXPR2 )
02849         && ( h[2] == DOLLAREXPR2 ) ) h += 2;
02850     h += 2;
02851     finish = EvalDoLoopArg(BHEAD h,0);
02852     while ( ( *h == DOLLAREXPRESSION || *h == DOLLAREXPR2 )
02853         && ( h[2] == DOLLAREXPR2 ) ) h += 2;
02854     h += 2;
02855     incr = EvalDoLoopArg(BHEAD h,0);
02856
02857     if ( ( finish == start ) || ( finish > start && incr > 0 )
02858         || ( finish < start && incr < 0 ) ) {}
02859     else { level = lhsbuf[3]; } /* skips the loop */
02860 /*
02861 Put start in the dollar variable indicated by lhsbuf[2]
02862 */
02863     d = Dollars + lhsbuf[2];
02864 #ifdef WITHPTHREADS
02865     {
02866         int nummodopt, dtype = -1;
02867         if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
02868             for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
02869                 if ( lhsbuf[2] == ModOptdollars[nummodopt].number ) break;
02870             }
02871             if ( nummodopt < NumModOptdollars ) {
02872                 dtype = ModOptdollars[nummodopt].type;
02873                 if ( dtype == MODLOCAL ) {
02874                     d = ModOptdollars[nummodopt].dstruct+AT.identity;
02875                 }
02876             }
02877         }
02878     }
02879 #endif
02880 /*
02881 Get the value

```

```

02882 */
02883     if ( d->type == DOLZERO ) {
02884         value = 0;
02885     }
02886     else if ( ( d->type == DOLNUMBER || d->type == DOLTERMS )
02887         && ( d->where[4] == 0 ) && ( d->where[0] == 4 )
02888         && ( d->where[1] > 0 ) && ( d->where[2] == 1 ) ) {
02889         value = ( d->where[3] < 0 ) ? -d->where[1]: d->where[1];
02890     }
02891     else {
02892         MLOCK(ErrorMessageLock);
02893         MesPrint("Wrong type of object in do loop parameter");
02894         MUNLOCK(ErrorMessageLock);
02895         Terminate(-1);
02896         return(level);
02897     }
02898     value += incr;
02899     if ( ( finish > start && value <= finish ) ||
02900         ( finish < start && value >= finish ) ||
02901         ( finish == start && value == finish ) ) {}
02902     else level = lhsbuf[3];
02903
02904     if ( d->size < MINALLOC ) {
02905         if ( d->where && d->where != &(AM.dollarzero) ) M_free(d->where,"dollar contents");
02906         d->size = MINALLOC;
02907         d->where = (WORD *)Malloc1(d->size*sizeof(WORD),"dollar contents");
02908     }
02909     if ( value > 0 ) {
02910         d->where[0] = 4;
02911         d->where[1] = value;
02912         d->where[2] = 1;
02913         d->where[3] = 3;
02914         d->where[4] = 0;
02915         d->type = DOLNUMBER;
02916     }
02917     else if ( start < 0 ) {
02918         d->where[0] = 4;
02919         d->where[1] = -value;
02920         d->where[2] = 1;
02921         d->where[3] = -3;
02922         d->where[4] = 0;
02923         d->type = DOLNUMBER;
02924     }
02925     else
02926         d->type = DOLZERO;
02927
02928     if ( d == Dollars + lhsbuf[2] ) {
02929         cbuf[AM.dbufnum].CanCommu[lhsbuf[2]] = 0;
02930         cbuf[AM.dbufnum].NumTerms[lhsbuf[2]] = 1;
02931         cbuf[AM.dbufnum].rhs[lhsbuf[2]] = d->where;
02932     }
02933     return(level);
02934 }
02935
02936 /*
02937     #] TestEndDoLoop :
02938     #[ DollarFactorize :
02939 */
02940 /* #define STEP2 */
02941 #define STEP2
02942
02943 int DollarFactorize(PHEAD WORD numdollar)
02944 {
02945     GETBIDENTITY
02946     DOLLARS d = Dollars + numdollar;
02947     CBUF *C, *CC;
02948     WORD *oldworkpointer;
02949     WORD *buf1, *t, *term, *buf1content, *buf2, *termextra;
02950     WORD *buf3, *argextra;
02951 #ifdef STEP2
02952     WORD *tstop, pow, *r;
02953 #endif
02954     int i, j, jj, action = 0, sign = 1;
02955     LONG insize, ii;
02956     WORD startebuf = cbuf[AT.ebufnum].numrhs;
02957     WORD nfactors, factorsincontent, extrafactor = 0;
02958     WORD oldsorttype = AR.SortType;
02959 #ifdef WITHPTHREADS
02960     int nummodopt, dtype;
02961     dtype = -1;
02962     if ( AS.MultiThreaded ) {
02963         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
02964             if ( numdollar == ModOptdollars[nummodopt].number ) break;
02965         }
02966         if ( nummodopt < NumModOptdollars ) {
02967             dtype = ModOptdollars[nummodopt].type;
02968         }
02969     }
02970 #endif

```

```

02981         if ( dtype == MODLOCAL ) {
02982             d = ModOptdollars[nummodopt].dstruct+AT.identity;
02983         }
02984         else {
02985             LOCK(d->pthreadlockread);
02986         }
02987     }
02988 }
02989 #endif
02990 CleanDollarFactors(d);
02991 #ifdef WITHPTHREADS
02992     if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
02993 #endif
02994     if ( d->type != DOLTERMS ) { /* only one term */
02995         if ( d->type != DOLZERO ) d->nfactors = 1;
02996         return(0);
02997     }
02998     if ( d->where[d->where[0]] == 0 ) { /* only one term. easy */
02999     }
03000 /*
03001     Here should come the code for the factorization
03002     We copied the routine ArgFactorize in argument.c and changed the
03003     memory management completely. For the actual factorization it
03004     calls WORD *DoFactorizeDollar(PHEAD WORD *expr) which allocates
03005     space for the answer. Notation:
03006     term,...,term,0,term,...,term,0,term,...,term,0,0
03007
03008     #] Step 1: sort the terms properly and/or make copy --> bufl,insize
03009 */
03010     term = d->where;
03011     AR.SortType = SORTHIGHFIRST;
03012     if ( oldsorttype != AR.SortType ) {
03013         NewSort(BHEAD0);
03014         while ( *term ) {
03015             t = term + *term;
03016             if ( AN.ncmod != 0 ) {
03017                 if ( AN.ncmod != 1 || ( (WORD)AN.cmod[0] < 0 ) ) {
03018                     AR.SortType = oldsorttype;
03019                     MLOCK(ErrorMessageLock);
03020                     MesPrint("Factorization modulus a number, greater than a WORD not implemented.");
03021                     MUNLOCK(ErrorMessageLock);
03022                     Terminate(-1);
03023                 }
03024                 if ( Modulus(term) ) {
03025                     AR.SortType = oldsorttype;
03026                     MLOCK(ErrorMessageLock);
03027                     MesCall("DollarFactorize");
03028                     MUNLOCK(ErrorMessageLock);
03029                     Terminate(-1);
03030                 }
03031                 if ( !*term ) { term = t; continue; }
03032             }
03033             StoreTerm(BHEAD term);
03034             term = t;
03035         }
03036         AN.tryterm = 0; /* for now */
03037         EndSort(BHEAD (WORD *)((void *)(&bufl)),2);
03038         t = bufl; while ( *t ) t += *t;
03039         insize = t - bufl;
03040     }
03041     else {
03042         t = term; while ( *t ) t += *t;
03043         ii = insize = t - term;
03044         bufl = (WORD *)Malloc1((insize+1)*sizeof(WORD),"DollarFactorize-1");
03045         t = bufl;
03046         NCOPY(t,term,ii);
03047         *t++ = 0;
03048     }
03049 /*
03050     #] Step 1:
03051     #] Step 2: take out the 'content'.
03052 */
03053 #ifdef STEP2
03054     buflcontent = TermMalloc("DollarContent");
03055     AN.tryterm = -1;
03056     if ( ( bufl2 = TakeContent(BHEAD bufl,buflcontent) ) == 0 ) {
03057         AN.tryterm = 0;
03058         TermFree(buflcontent,"DollarContent");
03059         M_free(bufl,"DollarFactorize-1");
03060         AR.SortType = oldsorttype;
03061         MLOCK(ErrorMessageLock);
03062         MesCall("DollarFactorize");
03063         MUNLOCK(ErrorMessageLock);
03064         Terminate(-1);
03065         return(1);
03066     }
03067     else if ( ( buflcontent[0] == 4 ) && ( buflcontent[1] == 1 ) &&

```

```

03068         ( buf1content[2] == 1 ) && ( buf1content[3] == 3 ) ) { /* Nothing happened */
03069     AN.tryterm = 0;
03070     if ( buf2 != buf1 ) {
03071         M_free(buf2,"DollarFactorize-2");
03072         buf2 = buf1;
03073     }
03074     factorsincontent = 0;
03075 }
03076 else {
03077 /*
03078     The way we took out objects is rather brutish. We have to normalize
03079 */
03080     AN.tryterm = 0;
03081     if ( buf2 != buf1 ) M_free(buf1,"DollarFactorize-1");
03082     buf1 = buf2;
03083     t = buf1; while ( *t ) t += *t;
03084     insize = t - buf1;
03085 /*
03086     Now analyse how many factors there are in the content
03087 */
03088     factorsincontent = 0;
03089     term = buf1content;
03090     tstop = term + *term;
03091     if ( tstop[-1] < 0 ) factorsincontent++;
03092     if ( ABS(tstop[-1]) == 3 && tstop[-2] == 1 && tstop[-3] == 1 ) {
03093         tstop -= ABS(tstop[-1]);
03094     }
03095     else {
03096         factorsincontent++;
03097         tstop -= ABS(tstop[-1]);
03098     }
03099     term++;
03100     while ( term < tstop ) {
03101         switch ( *term ) {
03102             case SYMBOL:
03103                 t = term+2; i = (term[1]-2)/2;
03104                 while ( i > 0 ) {
03105                     factorsincontent += ABS(t[1]);
03106                     i--; t += 2;
03107                 }
03108                 break;
03109             case DOTPRODUCT:
03110                 t = term+2; i = (term[1]-2)/3;
03111                 while ( i > 0 ) {
03112                     factorsincontent += ABS(t[2]);
03113                     i--; t += 3;
03114                 }
03115                 break;
03116             case VECTOR:
03117             case DELTA:
03118                 factorsincontent += (term[1]-2)/2;
03119                 break;
03120             case INDEX:
03121                 factorsincontent += term[1]-2;
03122                 break;
03123             default:
03124                 if ( *term >= FUNCTION ) factorsincontent++;
03125                 break;
03126         }
03127         term += term[1];
03128     }
03129 }
03130 #else
03131     factorsincontent = 0;
03132     buf1content = 0;
03133 #endif
03134 /*
03135     #] Step 2: take out the 'content'.
03136     #[ Step 3: ConvertToPoly
03137         if there are objects that are not SYMBOLs,
03138         invoke ConvertToPoly
03139         We keep the original in buf1 in case there are no factors
03140 */
03141     t = buf1;
03142     while ( *t ) {
03143         if ( ( t[1] != SYMBOL ) && ( *t != (ABS(t[*t-1])+1) ) ) {
03144             action = 1; break;
03145         }
03146         t += *t;
03147     }
03148     if ( DetCommu(buf1) > 1 ) {
03149         MesPrint("Cannot factorize a $-expression with more than one noncommuting object");
03150         AR.SortType = oldsorttype;
03151         M_free(buf1,"DollarFactorize-2");
03152         if ( buf1content ) TermFree(buf1content,"DollarContent");
03153         MesCall("DollarFactorize");
03154         Terminate(-1);

```



```

03155         return(-1);
03156     }
03157     if ( action ) {
03158         t = buf1;
03159         termextra = AT.WorkPointer;
03160         NewSort(BHEAD0);
03161         NewSort(BHEAD0);
03162         while ( *t ) {
03163             if ( LocalConvertToPoly(BHEAD t,termextra,startebuf,0) < 0 ) {
03164 getout:
03165                 AR.SortType = oldsorttype;
03166                 M_free(buf1,"DollarFactorize-2");
03167                 if ( buf1content ) TermFree(buf1content,"DollarContent");
03168                 MesCall("DollarFactorize");
03169                 Terminate(-1);
03170                 return(-1);
03171             }
03172             StoreTerm(BHEAD termextra);
03173             t += *t;
03174         }
03175         AN.tryterm = 0; /* for now */
03176         if ( EndSort(BHEAD (WORD *)((void *)&buf2)),2) < 0 ) { goto getout; }
03177         LowerSortLevel();
03178         t = buf2; while ( *t > 0 ) t += *t;
03179     }
03180     else {
03181         buf2 = buf1;
03182     }
03183     /*
03184         #] Step 3: ConvertToPoly
03185         #[ Step 4: Now the hard work.
03186     */
03187     if ( ( buf3 = poly_factorize_dollar(BHEAD buf2) ) == 0 ) {
03188         MesCall("DollarFactorize");
03189         AR.SortType = oldsorttype;
03190         if ( buf2 != buf1 && buf2 ) M_free(buf2,"DollarFactorize-3");
03191         M_free(buf1,"DollarFactorize-3");
03192         if ( buf1content ) TermFree(buf1content,"DollarContent");
03193         Terminate(-1);
03194         return(-1);
03195     }
03196     if ( buf2 != buf1 && buf2 ) {
03197         M_free(buf2,"DollarFactorize-3");
03198         buf2 = 0;
03199     }
03200     term = buf3;
03201     AR.SortType = oldsorttype;
03202     /*
03203     Count the factors and strip a factor -1
03204     */
03205     nfactors = 0;
03206     while ( *term ) {
03207 #ifdef STEP2
03208         if ( *term == 4 && term[4] == 0 && term[3] == -3 && term[2] == 1
03209             && term[1] == 1 ) {
03210             WORD *tt1, *tt2, *ttstop;
03211             sign = -sign;
03212             tt1 = term; tt2 = term + *term + 1;
03213             ttstop = tt2;
03214             while ( *ttstop ) {
03215                 while ( *ttstop ) ttstop += *ttstop;
03216                 ttstop++;
03217             }
03218             while ( tt2 < ttstop ) *tt1++ = *tt2++;
03219             *tt1 = 0;
03220             factorsincontent++;
03221             extrafactor++;
03222         }
03223     else
03224 #endif
03225     {
03226         term += *term;
03227         while ( *term ) { term += *term; }
03228         nfactors++; term++;
03229     }
03230 }
03231 /*
03232     We have now:
03233     buf1: the original before ConvertToPoly for if only one factor
03234     buf3: the factored expression with nfactors factors
03235
03236     #] Step 4:
03237     #[ Step 5: ConvertFromPoly
03238         If ConvertToPoly was used, use now ConvertFromPoly
03239         Be careful: there should be more than one factor now.
03240     */
03241 #ifdef WITHPTHREADS

```

```

03242     if ( dtype > 0 && dtype != MODLOCAL ) { LOCK(d->pthreadlockread); }
03243 #endif
03244     if ( nfactors == 1 && extrafactor == 0 ) { /* we can use the buf1 contents */
03245         if ( factorsincontent == 0 ) {
03246             d->nfactors = 1;
03247 #ifdef WITHPTHREADS
03248             if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
03249 #endif
03250 /*
03251     We used here (before 3-sep-2015) the original and did not make
03252     provisions for having a factors struct, figuring that all info
03253     is identical to the full dollar. This makes things too
03254     complicated at later stages.
03255 */
03256     d->factors = (FACDOLLAR *)Malloc1(sizeof(FACDOLLAR),"factors in dollar");
03257     term = buf1; while ( *term ) term += *term;
03258     d->factors[0].size = i = term - buf1;
03259     d->factors[0].where = t = (WORD *)Malloc1(sizeof(WORD)*(i+1),"DollarFactorize-5");
03260     term = buf1; NCOPY(t,term,i); *t = 0;
03261     AR.SortType = oldsorttype;
03262     M_free(buf3,"DollarFactorize-4");
03263     if ( buf2 != buf1 && buf2 ) M_free(buf2,"DollarFactorize-4");
03264     M_free(buf1,"DollarFactorize-4");
03265     if ( buf1content ) TermFree(buf1content,"DollarContent");
03266     return(0);
03267 }
03268     else {
03269         d->factors = (FACDOLLAR *)Malloc1(sizeof(FACDOLLAR)*(nfactors+factorsincontent),"factors
in dollar");
03270         term = buf1; while ( *term ) term += *term;
03271         d->factors[0].size = i = term - buf1;
03272         d->factors[0].where = t = (WORD *)Malloc1(sizeof(WORD)*(i+1),"DollarFactorize-5");
03273         term = buf1; NCOPY(t,term,i); *t = 0;
03274         M_free(buf3,"DollarFactorize-4");
03275         buf3 = 0;
03276         if ( buf2 != buf1 && buf2 ) {
03277             M_free(buf2,"DollarFactorize-4");
03278             buf2 = 0;
03279         }
03280     }
03281 }
03282     else if ( action ) {
03283         C = cbuf+AC.cbunum;
03284         CC = cbuf+AT.ebunum;
03285         oldworkpointer = AT.WorkPointer;
03286         d->factors = (FACDOLLAR *)Malloc1(sizeof(FACDOLLAR)*(nfactors+factorsincontent),"factors in
dollar");
03287         term = buf3;
03288         for ( i = 0; i < nfactors; i++ ) {
03289             argextra = AT.WorkPointer;
03290             NewSort(BHEAD0);
03291             NewSort(BHEAD0);
03292             while ( *term ) {
03293                 if ( ConvertFromPoly(BHEAD term,argextra,numxsymbol,CC->numrhs-startebuf+numxsymbol
,startebuf-numxsymbol,1) <= 0 ) {
03294                     LowerSortLevel();
03295                     AR.SortType = oldsorttype;
03296 getout2:
03297                     M_free(d->factors,"factors in dollar");
03298                     d->factors = 0;
03299 #ifdef WITHPTHREADS
03300                     if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
03301 #endif
03302                     M_free(buf3,"DollarFactorize-4");
03303                     if ( buf2 != buf1 && buf2 ) M_free(buf2,"DollarFactorize-4");
03304                     M_free(buf1,"DollarFactorize-4");
03305                     if ( buf1content ) TermFree(buf1content,"DollarContent");
03306                     return(-3);
03307                 }
03308                 AT.WorkPointer = argextra + *argextra;
03309 /*
03310             ConvertFromPoly leaves terms with subexpressions. Hence:
03311 */
03312                 if ( Generator(BHEAD argextra,C->numlhs+1) ) {
03313                     goto getout2;
03314                 }
03315                 term += *term;
03316             }
03317             term++;
03318             AT.WorkPointer = oldworkpointer;
03319             AN.tryterm = 0; /* for now */
03320             EndSort(BHEAD (WORD *)((void *)(&(d->factors[i].where))),2);
03321             LowerSortLevel();
03322             d->factors[i].type = DOLTERMS;
03323             t = d->factors[i].where;
03324             while ( *t ) t += *t;
03325             d->factors[i].size = t - d->factors[i].where;
03326         }

```

```

03327     CC->numrhs = startebuf;
03328 }
03329 else {
03330     C = cbuf+AC.cbunum;
03331     oldworkpointer = AT.WorkPointer;
03332     d->factors = (FACDOLLAR *)Malloc1(sizeof(FACDOLLAR)*(nfactors+factorsincontent),"factors in
dollar");
03333     term = buf3;
03334     for ( i = 0; i < nfactors; i++ ) {
03335         NewSort(BHEAD0);
03336         while ( *term ) {
03337             argextra = oldworkpointer;
03338             j = *term;
03339             NCOPY(argextra,term,j)
03340             AT.WorkPointer = argextra;
03341             if ( Generator(BHEAD oldworkpointer,C->numlhs+1) ) {
03342                 goto getout2;
03343             }
03344         }
03345         term++;
03346         AT.WorkPointer = oldworkpointer;
03347         AN.tryterm = 0; /* for now */
03348         EndSort(BHEAD (WORD *)((void *)(&(d->factors[i].where))),2);
03349         d->factors[i].type = DOLTERMS;
03350         t = d->factors[i].where;
03351         while ( *t ) t += *t;
03352         d->factors[i].size = t - d->factors[i].where;
03353     }
03354 }
03355 d->nfactors = nfactors + factorsincontent;
03356 /*
03357     #] Step 5: ConvertFromPoly
03358     #[ Step 6: The factors of the content
03359 */
03360 if ( buf3 ) M_free(buf3,"DollarFactorize-5");
03361 if ( buf2 != buf1 && buf2 ) M_free(buf2,"DollarFactorize-5");
03362 M_free(buf1,"DollarFactorize-5");
03363 j = nfactors;
03364 #ifdef STEP2
03365 term = buf1content;
03366 tstop = term + *term;
03367 if ( tstop[-1] < 0 ) { tstop[-1] = -tstop[-1]; sign = -sign; }
03368 tstop -= tstop[-1];
03369 term++;
03370 while ( term < tstop ) {
03371     switch ( *term ) {
03372     case SYMBOL:
03373         t = term+2; i = (term[1]-2)/2;
03374         while ( i > 0 ) {
03375             if ( t[1] < 0 ) { t[1] = -t[1]; pow = -1; }
03376             else { pow = 1; }
03377             for ( jj = 0; jj < t[1]; jj++ ) {
03378                 r = d->factors[j].where = (WORD *)Malloc1(9*sizeof(WORD),"factor");
03379                 r[0] = 8; r[1] = SYMBOL; r[2] = 4; r[3] = *t; r[4] = pow;
03380                 r[5] = 1; r[6] = 1; r[7] = 3; r[8] = 0;
03381                 d->factors[j].type = DOLTERMS;
03382                 d->factors[j].size = 8;
03383                 j++;
03384             }
03385             i--; t += 2;
03386         }
03387         break;
03388     case DOTPRODUCT:
03389         t = term+2; i = (term[1]-2)/3;
03390         while ( i > 0 ) {
03391             if ( t[2] < 0 ) { t[2] = -t[2]; pow = -1; }
03392             else { pow = 1; }
03393             for ( jj = 0; jj < t[2]; jj++ ) {
03394                 r = d->factors[j].where = (WORD *)Malloc1(10*sizeof(WORD),"factor");
03395                 r[0] = 9; r[1] = DOTPRODUCT; r[2] = 5; r[3] = t[0]; r[4] = t[1];
03396                 r[5] = pow; r[6] = 1; r[7] = 1; r[8] = 3; r[9] = 0;
03397                 d->factors[j].type = DOLTERMS;
03398                 d->factors[j].size = 9;
03399                 j++;
03400             }
03401             i--; t += 3;
03402         }
03403         break;
03404     case VECTOR:
03405     case DELTA:
03406         t = term+2; i = (term[1]-2)/2;
03407         while ( i > 0 ) {
03408             for ( jj = 0; jj < t[1]; jj++ ) {
03409                 r = d->factors[j].where = (WORD *)Malloc1(9*sizeof(WORD),"factor");
03410                 r[0] = 8; r[1] = *term; r[2] = 4; r[3] = *t; r[4] = t[1];
03411                 r[5] = 1; r[6] = 1; r[7] = 3; r[8] = 0;
03412                 d->factors[j].type = DOLTERMS;

```

```

03413             d->factors[j].size = 8;
03414             j++;
03415         }
03416         i--; t += 2;
03417     }
03418     break;
03419 case INDEX:
03420     t = term+2; i = term[1]-2;
03421     while ( i > 0 ) {
03422         for ( jj = 0; jj < t[1]; jj++ ) {
03423             r = d->factors[j].where = (WORD *)Malloca1(8*sizeof(WORD), "factor");
03424             r[0] = 7; r[1] = *term; r[2] = 3; r[3] = *t;
03425             r[4] = 1; r[5] = 1; r[6] = 3; r[7] = 0;
03426             d->factors[j].type = DOLTERMS;
03427             d->factors[j].size = 7;
03428             j++;
03429         }
03430         i--; t++;
03431     }
03432     break;
03433 default:
03434     if ( *term >= FUNCTION ) {
03435         r = d->factors[j].where = (WORD *)Malloca1((term[1]+5)*sizeof(WORD), "factor");
03436         *r++ = d->factors[j].size = term[1]+4;
03437         for ( jj = 0; jj < t[1]; jj++ ) *r++ = term[jj];
03438         *r++ = 1; *r++ = 1; *r++ = 3; *r = 0;
03439         j++;
03440     }
03441     break;
03442 }
03443 term += term[1];
03444 }
03445 #endif
03446 /*
03447     #] Step 6:
03448     #[ Step 7: Numerical factors
03449 */
03450 #ifdef STEP2
03451     term = buf1content;
03452     tstop = term + *term;
03453     if ( tstop[-1] == 3 && tstop[-2] == 1 && tstop[-3] == 1 ) {}
03454     else if ( tstop[-1] == 3 && tstop[-2] == 1 && (UWORD)(tstop[-3]) <= MAXPOSITIVE ) {
03455         d->factors[j].where = 0;
03456         d->factors[j].size = 0;
03457         d->factors[j].type = DOLNUMBER;
03458         d->factors[j].value = sign*tstop[-3];
03459         sign = 1;
03460         j++;
03461     }
03462     else {
03463         d->factors[j].where = r = (WORD *)Malloca1((tstop[-1]+2)*sizeof(WORD), "numfactor");
03464         d->factors[j].size = tstop[-1]+1;
03465         d->factors[j].type = DOLTERMS;
03466         d->factors[j].value = 0;
03467         i = tstop[-1];
03468         t = tstop - i;
03469         *r++ = tstop[-1]+1;
03470         NCOPY(r,t,i);
03471         *r = 0;
03472         if ( sign < 0 ) {
03473             r = d->factors[j].where;
03474             while ( *r ) {
03475                 r += *r; r[-1] = -r[-1];
03476             }
03477             sign = 1;
03478         }
03479         j++;
03480     }
03481 #endif
03482     if ( sign < 0 ) { /* Note that this guy should come first */
03483         for ( jj = j; jj > 0; jj-- ) {
03484             d->factors[jj] = d->factors[jj-1];
03485         }
03486         d->factors[0].where = 0;
03487         d->factors[0].size = 0;
03488         d->factors[0].type = DOLNUMBER;
03489         d->factors[0].value = -1;
03490         j++;
03491     }
03492     d->nfactors = j;
03493     if ( buf1content ) TermFree(buf1content, "DollarContent");
03494 /*
03495     #] Step 7:
03496     #[ Step 8: Sorting the factors
03497
03498     There are d->nfactors factors. Look which ones have a 'where'
03499     Sort them by bubble sort

```

```

03500 /*
03501     if ( d->nfactors > 1 ) {
03502         WORD ***fac, j1, j2, k, ret, *s1, *s2, *s3;
03503         LONG **facsize, x;
03504         facsize = (LONG **)Malloc1((sizeof(WORD **)+sizeof(LONG *))d->nfactors,"SortDollarFactors");
03505         fac = (WORD **)(facsize+d->nfactors);
03506         k = 0;
03507         for ( j = 0; j < d->nfactors; j++ ) {
03508             if ( d->factors[j].where ) {
03509                 fac[k] = &(d->factors[j].where);
03510                 facsize[k] = &(d->factors[j].size);
03511                 k++;
03512             }
03513         }
03514         if ( k > 1 ) {
03515             for ( j = 1; j < k; j++ ) { /* bubble sort */
03516                 j1 = j; j2 = j1-1;
03517 nextj1:;
03518                 s1 = *(fac[j1]); s2 = *(fac[j2]);
03519                 while ( *s1 && *s2 ) {
03520                     if ( ( ret = CompareTerms(BHEAD s2, s1, (WORD)2) ) == 0 ) {
03521                         s1 += *s1; s2 += *s2;
03522                     }
03523                     else if ( ret > 0 ) goto nextj;
03524                     else {
03525 exch:
03526                         s3 = *(fac[j1]); *(fac[j1]) = *(fac[j2]); *(fac[j2]) = s3;
03527                         x = *(facsize[j1]); *(facsize[j1]) = *(facsize[j2]); *(facsize[j2]) = x;
03528                         j1--; j2--;
03529                         if ( j1 > 0 ) goto nextj1;
03530                         goto nextj;
03531                     }
03532                 }
03533                 if ( *s1 ) goto nextj;
03534                 if ( *s2 ) goto exch;
03535 nextj:;
03536             }
03537         }
03538         M_free(facsize,"SortDollarFactors");
03539     }
03540     /*
03541         #] Step 8:
03542     */
03543     #ifdef WITHPTHREADS
03544     if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
03545     #endif
03546     return(0);
03547 }
03548
03549 /*
03550     #] DollarFactorize :
03551     #[ CleanDollarFactors :
03552 */
03553
03554 void CleanDollarFactors(DOLLARS d)
03555 {
03556     int i;
03557     if ( d->nfactors >= 1 ) {
03558         for ( i = 0; i < d->nfactors; i++ ) {
03559             if ( d->factors[i].where )
03560                 if ( d->factors[i].where )
03561                     M_free(d->factors[i].where,"dollar factors");
03562             }
03563         }
03564         if ( d->factors ) {
03565             M_free(d->factors,"dollar factors");
03566             d->factors = 0;
03567         }
03568         d->nfactors = 0;
03569     }
03570
03571     /*
03572         #] CleanDollarFactors :
03573         #[ TakeDollarContent :
03574     */
03575
03576     WORD *TakeDollarContent(PHEAD WORD *dollarbuffer, WORD **factor)
03577     {
03578         WORD *remain, *t;
03579         int pow;
03580         /*
03581             We force the sign of the first term to be positive.
03582         */
03583         t = dollarbuffer; pow = 1;
03584         t += *t;
03585         if ( t[-1] < 0 ) {
03586             pow = 0;

```

```

03587         t[-1] = -t[-1];
03588         while ( *t ) {
03589             t += *t; t[-1] = -t[-1];
03590         }
03591     }
03592     /*
03593     Now the GCD of the numerators and the LCM of the denominators:
03594     */
03595     if ( AN.cmod != 0 ) {
03596         if ( ( *factor = MakeDollarMod(BHEAD dollarbuffer,&remain) ) == 0 ) {
03597             Terminate(-1);
03598         }
03599         if ( pow == 0 ) {
03600             (*factor)[**factor-1] = -(*factor)[**factor-1];
03601             (*factor)[**factor-1] += AN.cmod[0];
03602         }
03603     }
03604     else {
03605         if ( ( *factor = MakeDollarInteger(BHEAD dollarbuffer,&remain) ) == 0 ) {
03606             Terminate(-1);
03607         }
03608         if ( pow == 0 ) {
03609             (*factor)[**factor-1] = -(*factor)[**factor-1];
03610         }
03611     }
03612     return(remain);
03613 }
03614
03615 /*
03616 #] TakeDollarContent :
03617 #[ MakeDollarInteger :
03618 */
03628 WORD *MakeDollarInteger(PHEAD WORD *bufin,WORD **bufout)
03629 {
03630     GETBIDENTITY
03631     UWORD *GCDBuffer, *GCDBuffer2, *LCMbuffer, *LCMb, *LCMc;
03632     WORD *r, *r1, *r2, *r3, *rnext, i, k, j, *oldworkpointer, *factor;
03633     WORD kGCD, kLCM, kGCD2, kLCM2, jLCM, jGCD;
03634     CBUF *C = cbuf+AC.cbunum;
03635
03636     GCDBuffer = NumberMalloc("MakeDollarInteger");
03637     GCDBuffer2 = NumberMalloc("MakeDollarInteger");
03638     LCMbuffer = NumberMalloc("MakeDollarInteger");
03639     LCMb = NumberMalloc("MakeDollarInteger");
03640     LCMc = NumberMalloc("MakeDollarInteger");
03641     r = bufin;
03642     /*
03643     First take the first term to load up the LCM and the GCD
03644     */
03645     r2 = r + *r;
03646     j = r2[-1];
03647     r3 = r2 - ABS(j);
03648     k = REDLENG(j);
03649     if ( k < 0 ) k = -k;
03650     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
03651     for ( kGCD = 0; kGCD < k; kGCD++ ) GCDBuffer[kGCD] = r3[kGCD];
03652     k = REDLENG(j);
03653     if ( k < 0 ) k = -k;
03654     r3 += k;
03655     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
03656     for ( kLCM = 0; kLCM < k; kLCM++ ) LCMbuffer[kLCM] = r3[kLCM];
03657     r1 = r2;
03658     /*
03659     Now go through the rest of the terms in this argument.
03660     */
03661     while ( *r1 ) {
03662         r2 = r1 + *r1;
03663         j = r2[-1];
03664         r3 = r2 - ABS(j);
03665         k = REDLENG(j);
03666         if ( k < 0 ) k = -k;
03667         while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
03668         if ( ( ( GCDBuffer[0] == 1 ) && ( kGCD == 1 ) ) ) {
03669             /*
03670             GCD is already 1
03671             */
03672         }
03673         else if ( ( ( k != 1 ) || ( r3[0] != 1 ) ) ) {
03674             if ( GcdLong(BHEAD GCDBuffer,kGCD,(UWORD *)r3,k,GCDBuffer2,&kGCD2) ) {
03675                 goto MakeDollarIntegerErr;
03676             }
03677             kGCD = kGCD2;
03678             for ( i = 0; i < kGCD; i++ ) GCDBuffer[i] = GCDBuffer2[i];
03679         }
03680         else {
03681             kGCD = 1; GCDBuffer[0] = 1;
03682         }

```

```

03683     k = REDLENG(j);
03684     if ( k < 0 ) k = -k;
03685     r3 += k;
03686     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
03687     if ( ( ( LCMbuffer[0] == 1 ) && ( kLCM == 1 ) ) ) {
03688         for ( kLCM = 0; kLCM < k; kLCM++ )
03689             LCMbuffer[kLCM] = r3[kLCM];
03690     }
03691     else if ( ( k != 1 ) || ( r3[0] != 1 ) ) {
03692         if ( GcdLong(BHEAD LCMbuffer,kLCM, (UWORD *)r3,k,LCMb,&kkLCM) ) {
03693             goto MakeDollarIntegerErr;
03694         }
03695         DivLong((UWORD *)r3,k,LCMb,kkLCM,LCMb,&kkLCM,LCMc,&jLCM);
03696         Mullong(LCMbuffer,kLCM,LCMb,kkLCM,LCMc,&jLCM);
03697         for ( kLCM = 0; kLCM < jLCM; kLCM++ )
03698             LCMbuffer[kLCM] = LCMc[kLCM];
03699     }
03700     else {} /* LCM doesn't change */
03701     r1 = r2;
03702 }
03703 /*
03704 Now put the factor together: GCD/LCM
03705 */
03706 r3 = (WORD *) (GCDbuffer);
03707 if ( kGCD == kLCM ) {
03708     for ( jGCD = 0; jGCD < kGCD; jGCD++ )
03709         r3[jGCD+kGCD] = LCMbuffer[jGCD];
03710     k = kGCD;
03711 }
03712 else if ( kGCD > kLCM ) {
03713     for ( jGCD = 0; jGCD < kLCM; jGCD++ )
03714         r3[jGCD+kGCD] = LCMbuffer[jGCD];
03715     for ( jGCD = kLCM; jGCD < kGCD; jGCD++ )
03716         r3[jGCD+kGCD] = 0;
03717     k = kGCD;
03718 }
03719 else {
03720     for ( jGCD = kGCD; jGCD < kLCM; jGCD++ )
03721         r3[jGCD] = 0;
03722     for ( jGCD = 0; jGCD < kLCM; jGCD++ )
03723         r3[jGCD+kLCM] = LCMbuffer[jGCD];
03724     k = kLCM;
03725 }
03726 j = 2*k+1;
03727 /*
03728 Now we have to write this to factor
03729 */
03730 factor = r1 = (WORD *)Malloc1((j+2)*sizeof(WORD),"MakeDollarInteger");
03731 *r1++ = j+1; r2 = r3;
03732 for ( i = 0; i < k; i++ ) { *r1++ = *r2++; *r1++ = *r2++; }
03733 *r1++ = j;
03734 *r1 = 0;
03735 /*
03736 Next we have to take the factor out from the argument.
03737 This cannot be done in location, because the denominator stuff can make
03738 coefficients longer.
03739
03740 We do this via a sort because the things may be jumbled any way and we
03741 do not know in advance how much space we need.
03742 */
03743 NewSort(BHEAD0);
03744 r = bufin;
03745 oldworkpointer = AT.WorkPointer;
03746 while ( *r ) {
03747     rnext = r + *r;
03748     j = ABS(rnext[-1]);
03749     r3 = rnext - j;
03750     r2 = oldworkpointer;
03751     while ( r < r3 ) *r2++ = *r++;
03752     j = (j-1)/2; /* reduced length. Remember, k is the other red length */
03753     if ( DivRat(BHEAD (UWORD *)r3,j,GCDbuffer,k,(UWORD *)r2,&i) ) {
03754         goto MakeDollarIntegerErr;
03755     }
03756     i = 2*i+1;
03757     r2 = r2 + i;
03758     if ( rnext[-1] < 0 ) r2[-1] = -i;
03759     else r2[-1] = i;
03760     *oldworkpointer = r2-oldworkpointer;
03761     AT.WorkPointer = r2;
03762     if ( Generator(BHEAD oldworkpointer,C->numlhs) ) {
03763         goto MakeDollarIntegerErr;
03764     }
03765     r = rnext;
03766 }
03767 AT.WorkPointer = oldworkpointer;
03768 AN.tryterm = 0; /* for now */
03769 EndSort(BHEAD (WORD *)bufout,2);

```

```

03770 /*
03771 Cleanup
03772 */
03773 NumberFree(LCMc,"MakeDollarInteger");
03774 NumberFree(LCmb,"MakeDollarInteger");
03775 NumberFree(LCmbuffer,"MakeDollarInteger");
03776 NumberFree(GCDBuffer2,"MakeDollarInteger");
03777 NumberFree(GCDBuffer,"MakeDollarInteger");
03778 return(factor);
03779
03780 MakeDollarIntegerErr:
03781 NumberFree(LCMc,"MakeDollarInteger");
03782 NumberFree(LCmb,"MakeDollarInteger");
03783 NumberFree(LCmbuffer,"MakeDollarInteger");
03784 NumberFree(GCDBuffer2,"MakeDollarInteger");
03785 NumberFree(GCDBuffer,"MakeDollarInteger");
03786 MesCall("MakeDollarInteger");
03787 Terminate(-1);
03788 return(0);
03789 }
03790
03791 /*
03792 #] MakeDollarInteger :
03793 #[ MakeDollarMod :
03794 */
03802 WORD *MakeDollarMod(PHEAD WORD *buffer, WORD **bufout)
03803 {
03804     GETBIDENTITY
03805     WORD *r, *r1, x, xx, ix, ip;
03806     WORD *factor, *oldworkpointer;
03807     int i;
03808     CBUF *C = cbuf+AC.cbunum;
03809     r = buffer;
03810     x = r[*r-3];
03811     if ( r[*r-1] < 0 ) x += AN.cmod[0];
03812     if ( GetModInverses(x, (WORD) (AN.cmod[0]), &ix, &ip) ) {
03813         Terminate(-1);
03814     }
03815     factor = (WORD *)Malloca(5*sizeof(WORD), "MakeDollarMod");
03816     factor[0] = 4; factor[1] = x; factor[2] = 1; factor[3] = 3; factor[4] = 0;
03817 /*
03818 Now we have to multiply all coefficients by ix.
03819 This does not make things longer, but we should keep to the conventions
03820 of MakeDollarInteger.
03821 */
03822 NewSort(BHEAD0);
03823 r = buffer;
03824 oldworkpointer = AT.WorkPointer;
03825 while ( *r ) {
03826     r1 = oldworkpointer; i = *r;
03827     NCOPY(r1, r, i);
03828     xx = r1[-3]; if ( r1[-1] < 0 ) xx += AN.cmod[0];
03829     r1[-1] = (WORD) (((LONG)xx)*ix) % AN.cmod[0];
03830     *r1 = 0; AT.WorkPointer = r1;
03831     if ( Generator(BHEAD oldworkpointer, C->numlhs) ) {
03832         Terminate(-1);
03833     }
03834 }
03835 AT.WorkPointer = oldworkpointer;
03836 AN.tryterm = 0; /* for now */
03837 EndSort(BHEAD (WORD *)bufout, 2);
03838 return(factor);
03839 }
03840 /*
03841 #] MakeDollarMod :
03842 #[ GetDolNum :
03843
03844 Evaluates a chain of DOLLAREXPR2 into a number
03845 */
03846
03847 int GetDolNum(PHEAD WORD *t, WORD *tstop)
03848 {
03849     DOLLARS d;
03850     WORD num, *w;
03851     if ( t+3 < tstop && t[3] == DOLLAREXPR2 ) {
03852         d = Dollars + t[2];
03853 #ifdef WITHPTHREADS
03854     {
03855         int nummodopt, dtype;
03856         dtype = -1;
03857         if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
03858             for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
03859                 if ( t[2] == ModOptdollars[nummodopt].number ) break;
03860             }
03861             if ( nummodopt < NumModOptdollars ) {
03862                 dtype = ModOptdollars[nummodopt].type;
03863                 if ( dtype == MODLOCAL ) {

```



```

03864         d = ModOptdollars[nummodopt].dstruct+AT.identity;
03865     }
03866     else {
03867         MLOCK(ErrorMessageLock);
03868         MesPrint("&Illegal attempt to use $-variable %s in module %1",
03869             DOLLARNAME(Dollars,t[2]),AC.CModule);
03870         MUNLOCK(ErrorMessageLock);
03871         Terminate(-1);
03872     }
03873 }
03874 }
03875 }
03876 #endif
03877 if ( d->factors == 0 ) {
03878     MLOCK(ErrorMessageLock);
03879     MesPrint("Attempt to use a factor of an unfactored $-variable");
03880     MUNLOCK(ErrorMessageLock);
03881     Terminate(-1);
03882 }
03883 num = GetDolNum(BHEAD t+t[1],tstop);
03884 if ( num == 0 ) return(d->nfactors);
03885 if ( num > d->nfactors ) {
03886     MLOCK(ErrorMessageLock);
03887     MesPrint("Attempt to use an nonexistent factor %d of a $-variable",num);
03888     MUNLOCK(ErrorMessageLock);
03889     Terminate(-1);
03890 }
03891 w = d->factors[num-1].where;
03892 if ( w == 0 ) return(d->factors[num-1].value);
03893 if ( w[0] == 4 && w[4] == 0 && w[3] == 3 && w[2] == 1 && w[1] > 0
03894     && w[1] < MAXPOSITIVE ) return(w[1]);
03895 else {
03896     MLOCK(ErrorMessageLock);
03897     MesPrint("Illegal type of factor number of a $-variable");
03898     MUNLOCK(ErrorMessageLock);
03899     Terminate(-1);
03900 }
03901 }
03902 else if ( t[2] < 0 ) {
03903     return(-t[2]-1);
03904 }
03905 else {
03906     d = Dollars + t[2];
03907 #ifdef WITHPTHREADS
03908 {
03909     int nummodopt, dtype;
03910     dtype = -1;
03911     if ( AS.MultiThreaded && ( AC.mparallelfld == PARALLELFIELD ) ) {
03912         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
03913             if ( t[2] == ModOptdollars[nummodopt].number ) break;
03914         }
03915         if ( nummodopt < NumModOptdollars ) {
03916             dtype = ModOptdollars[nummodopt].type;
03917             if ( dtype == MODLOCAL ) {
03918                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
03919             }
03920             else {
03921                 MLOCK(ErrorMessageLock);
03922                 MesPrint("&Illegal attempt to use $-variable %s in module %1",
03923                     DOLLARNAME(Dollars,t[2]),AC.CModule);
03924                 MUNLOCK(ErrorMessageLock);
03925                 Terminate(-1);
03926             }
03927         }
03928     }
03929 }
03930 #endif
03931 if ( d->type == DOLZERO ) return(0);
03932 if ( d->type == DOLTERMS || d->type == DOLNUMBER ) {
03933     if ( d->where[0] == 4 && d->where[4] == 0 && d->where[3] == 3
03934         && d->where[2] == 1 && d->where[1] > 0
03935         && d->where[1] < MAXPOSITIVE ) return(d->where[1]);
03936     MLOCK(ErrorMessageLock);
03937     MesPrint("Attempt to use an nonexistent factor of a $-variable");
03938     MUNLOCK(ErrorMessageLock);
03939     Terminate(-1);
03940 }
03941 MLOCK(ErrorMessageLock);
03942 MesPrint("Illegal type of factor number of a $-variable");
03943 MUNLOCK(ErrorMessageLock);
03944 Terminate(-1);
03945 }
03946 return(0);
03947 }
03948
03949 /*
03950 #] GetDolNum :

```

```

03951     #[ AddPotModdollar :
03952     */
03953
03960 void AddPotModdollar(WORD numdollar)
03961 {
03962     int i, n = NumPotModdollars;
03963     for ( i = 0; i < n; i++ ) {
03964         if ( numdollar == PotModdollars[i] ) break;
03965     }
03966     if ( i >= n ) {
03967         *(WORD *)FromList (&AC.PotModDolList) = numdollar;
03968     }
03969 }
03970
03971 /*
03972     #] AddPotModdollar :
03973     */

```

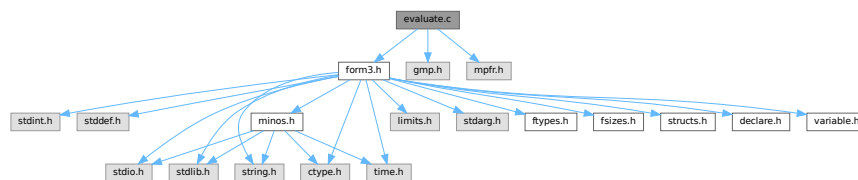
4.29 evaluate.c File Reference

```

#include "form3.h"
#include <gmp.h>
#include <mpfr.h>

```

Include dependency graph for evaluate.c:



Macros

- `#define EXTRAPRECISION 4`
- `#define RND MPFR_RNDN`
- `#define auxr1 (((mpfr_t *) (AT.auxr_))[0])`
- `#define auxr2 (((mpfr_t *) (AT.auxr_))[1])`
- `#define auxr3 (((mpfr_t *) (AT.auxr_))[2])`
- `#define auxr4 (((mpfr_t *) (AT.auxr_))[3])`
- `#define auxr5 (((mpfr_t *) (AT.auxr_))[4])`

Functions

- `int PackFloat (WORD *, mpfr_t)`
- `int UnpackFloat (mpfr_t, WORD *)`
- `void RatToFloat (mpfr_t, UWORD *, int)`
- `void FormtoZ (mpz_t, UWORD *, WORD)`
- `void IntegerToFloat (mpfr_t result, UWORD *formlong, int longsize)`
- `void RatToFloat (mpfr_t result, UWORD *formrat, int ratsize)`
- `void SetFloatPrecision (LONG prec)`
- `void ClearFloat (void)`
- `int GetFloatArgument (PHEAD mpfr_t f_out, WORD *fun, int par)`
- `int GetPiArgument (PHEAD WORD *arg)`
- `int EvaluateFun (PHEAD WORD *term, WORD level, WORD *pars)`

Variables

- `mpf_t ln2`

4.29.1 Detailed Description

Evaluation of functions for the floating-point system, by interfacing with MPFR.

Definition in file [evaluate.c](#).

4.29.2 Macro Definition Documentation

4.29.2.1 EXTRAPRECISION

```
#define EXTRAPRECISION 4
```

Definition at line 39 of file [evaluate.c](#).

4.29.2.2 RND

```
#define RND MPFR_RNDN
```

Definition at line 41 of file [evaluate.c](#).

4.29.2.3 auxr1

```
#define auxr1 (((mpfr_t *) (AT.auxr_)) [0])
```

Definition at line 43 of file [evaluate.c](#).

4.29.2.4 auxr2

```
#define auxr2 (((mpfr_t *) (AT.auxr_)) [1])
```

Definition at line 44 of file [evaluate.c](#).

4.29.2.5 auxr3

```
#define auxr3 (((mpfr_t *) (AT.auxr_)) [2])
```

Definition at line 45 of file [evaluate.c](#).

4.29.2.6 auxr4

```
#define auxr4 (((mpfr_t *) (AT.auxr_)) [3])
```

Definition at line 46 of file [evaluate.c](#).

4.29.2.7 auxr5

```
#define auxr5 ((mpfr_t *) (AT.auxr_))[4])
```

Definition at line 47 of file [evaluate.c](#).

4.29.3 Function Documentation

4.29.3.1 PackFloat()

```
int PackFloat (
    WORD * fun,
    mpf_t infloat )
```

Definition at line 271 of file [float.c](#).

4.29.3.2 UnpackFloat()

```
int UnpackFloat (
    mpf_t outfloat,
    WORD * fun )
```

Definition at line 366 of file [float.c](#).

4.29.3.3 RatToFloat()

```
void RatToFloat (
    mpf_t result,
    UWORD * formrat,
    int ratsize )
```

Definition at line 624 of file [float.c](#).

4.29.3.4 FormtoZ()

```
void FormtoZ (
    mpz_t z,
    UWORD * a,
    WORD na )
```

Definition at line 515 of file [float.c](#).

4.29.3.5 IntegerToFloatr()

```
void IntegerToFloatr (
    mpfr_t result,
    UWORD * formlong,
    int longsize )
```

Definition at line 65 of file [evaluate.c](#).

4.29.3.6 RatToFloatr()

```
void RatToFloatr (
    mpfr_t result,
    UWORD * format,
    int ratsize )
```

Definition at line 81 of file [evaluate.c](#).

4.29.3.7 SetFloatPrecision()

```
void SetFloatPrecision (
    LONG prec )
```

Definition at line 108 of file [evaluate.c](#).

4.29.3.8 ClearFloat()

```
void ClearFloat (
    void )
```

Definition at line 141 of file [evaluate.c](#).

4.29.3.9 GetFloatArgument()

```
int GetFloatArgument (
    PHEAD mpfr_t f_out,
    WORD * fun,
    int par )
```

Definition at line 180 of file [evaluate.c](#).

4.29.3.10 GetPiArgument()

```
int GetPiArgument (
    PHEAD WORD * arg )
```

Definition at line 275 of file [evaluate.c](#).

4.29.3.11 EvaluateFun()

```
int EvaluateFun (
    PHEAD WORD * term,
    WORD level,
    WORD * pars )
```

Definition at line 368 of file [evaluate.c](#).

4.29.4 Variable Documentation

4.29.4.1 ln2

mpf_t ln2

Definition at line 49 of file [evaluate.c](#).

4.30 evaluate.c

[Go to the documentation of this file.](#)

```

00001
00005 /* #[ License : */
00006 /*
00007 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008 *   When using this file you are requested to refer to the publication
00009 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010 *   This is considered a matter of courtesy as the development was paid
00011 *   for by FOM the Dutch physics granting agency and we would like to
00012 *   be able to track its scientific use to convince FOM of its value
00013 *   for the community.
00014 *
00015 *   This file is part of FORM.
00016 *
00017 *   FORM is free software: you can redistribute it and/or modify it under the
00018 *   terms of the GNU General Public License as published by the Free Software
00019 *   Foundation, either version 3 of the License, or (at your option) any later
00020 *   version.
00021 *
00022 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00025 *   details.
00026 *
00027 *   You should have received a copy of the GNU General Public License along
00028 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00029 */
00030 /* #[ License : */
00031 /*
00032     #[ includes :
00033 */
00034
00035 #include "form3.h"
00036 #include <gmp.h>
00037 #include <mpfr.h>
00038
00039 #define EXTRAPRECISION 4
00040
00041 #define RND MPFR_RNDN
00042
00043 #define auxr1 (((mpfr_t *) (AT.auxr_))[0])
00044 #define auxr2 (((mpfr_t *) (AT.auxr_))[1])
00045 #define auxr3 (((mpfr_t *) (AT.auxr_))[2])
00046 #define auxr4 (((mpfr_t *) (AT.auxr_))[3])
00047 #define auxr5 (((mpfr_t *) (AT.auxr_))[4])
00048
00049 mpf_t ln2;
00050
00051 int PackFloat (WORD *, mpf_t);
00052 int UnpackFloat (mpf_t, WORD *);
00053 void RatToFloat (mpf_t, UWORD *, int);
00054 void FormtoZ (mpz_t, UWORD *, WORD);
00055
00056 /*
00057     #[ includes :
00058     #[ mpfr_ :
00059         #[ IntegerToFloatr :
00060
00061         Converts a Form long integer to a mpfr_ float of default size.
00062         We assume that sizeof(unsigned long int) = 2*sizeof(UWORD).
00063 */
00064
00065 void IntegerToFloatr (mpfr_t result, UWORD *formlong, int longsize)
00066 {
00067     mpz_t z;
00068     mpz_init(z);
00069     FormtoZ(z, formlong, longsize);

```

```

00070     mpfr_set_z(result,z,RND);
00071     mpz_clear(z);
00072 }
00073
00074 /*
00075     #] IntegerToFloat :
00076     #[ RatToFloat :
00077
00078     Converts a Form rational to a gmp float of default size.
00079 */
00080
00081 void RatToFloat(mpfr_t result, UWORD *formrat, int ratsize)
00082 {
00083     GETIDENTITY
00084     int nnum, nden;
00085     UWORD *num, *den;
00086     int sgn = 0;
00087     if ( ratsize < 0 ) { ratsize = -ratsize; sgn = 1; }
00088     nnum = nden = (ratsize-1)/2;
00089     num = formrat; den = formrat+nnum;
00090     while ( nnum > 1 && num[nnum-1] == 0 ) { nnum--; }
00091     while ( nden > 1 && den[nden-1] == 0 ) { nden--; }
00092     IntegerToFloat(auxr4,num,nnum);
00093     IntegerToFloat(auxr5,den,nden);
00094     mpfr_div(result,auxr4,auxr5,RND);
00095     if ( sgn > 0 ) mpfr_neg(result,result,RND);
00096 }
00097
00098 /*
00099     #] RatToFloat :
00100     #[ SetFloatPrecision :
00101
00102     The AC.DefaultPrecision is in bits.
00103     prec is in limbs.
00104     fprec is NOT in bits for mpfr_ which is slightly less than in mpf_
00105     We also set the auxr1,auxr2,auxr3,auxr4,auxr5 variables.
00106 */
00107
00108 void SetFloatPrecision(LONG prec)
00109 {
00110     /*
00111         mpfr_prec_t fprec = prec * mp_bits_per_limb - EXTRAPRECISION;
00112     */
00113     mpfr_prec_t fprec = prec + 1;
00114     mpfr_set_default_prec(fprec);
00115     #ifdef WITHPTHREADS
00116         int totnum = AM.totalnumberofthreads, id;
00117         mpfr_t *a;
00118         #ifdef WITHSORTBOTS
00119             totnum = MaX(2*AM.totalnumberofthreads-3,AM.totalnumberofthreads);
00120         #endif
00121         for ( id = 0; id < totnum; id++ ) {
00122             AB[id]->T.auxr_ = (void *)Malloca(sizeof(mpfr_t)*5,"AB[id]->T.auxr_");
00123             a = (mpfr_t *)AB[id]->T.auxr_;
00124         /*
00125             We work here with a[0] etc because the aux1 etc contain B which
00126             in the current routine would be AB[0] only
00127         */
00128             mpfr_inits2(fprec,a[0],a[1],a[2],a[3],a[4],(mpfr_ptr)0);
00129         }
00130     #else
00131         AT.auxr_ = (void *)Malloca(sizeof(mpfr_t)*5,"AT.auxr_");
00132         mpfr_inits2(fprec,auxr1,auxr2,auxr3,auxr4,auxr5,(mpfr_ptr)0);
00133     #endif
00134 }
00135
00136 /*
00137     #] SetFloatPrecision :
00138     #[ ClearFloat :
00139 */
00140
00141 void ClearFloat(void)
00142 {
00143     #ifdef WITHPTHREADS
00144         int totnum = AM.totalnumberofthreads, id;
00145         mpfr_t *a;
00146         #ifdef WITHSORTBOTS
00147             totnum = MaX(2*AM.totalnumberofthreads-3,AM.totalnumberofthreads);
00148         #endif
00149         if ( AB[0]->T.auxr_ ) {
00150             for ( id = 0; id < totnum; id++ ) {
00151                 a = (mpfr_t *)AB[id]->T.auxr_;
00152                 mpfr_clears(a[0],a[1],a[2],a[3],a[4],(mpfr_ptr)0);
00153                 M_free(AB[id]->T.auxr_,"AB[id]->T.auxr_");
00154                 AB[id]->T.auxr_ = 0;
00155             }
00156         }

```

```

00157 #else
00158     if ( AT.auxr_ ) {
00159         mpfr_clears(auxr1,auxr2,auxr3,auxr4,auxr5,(mpfr_ptr)0);
00160         M_free(AT.auxr_,"AT.auxr_");
00161         AT.auxr_ = 0;
00162     }
00163 #endif
00164 }
00165
00166 /*
00167     #] ClearfFloat :
00168     #[ GetFloatArgument :
00169
00170     Convert an argument to an mpfr float if possible.
00171     Return value: 0: was converted successfully.
00172                 -1: could not convert.
00173     Note: arguments with more than one term should be evaluated first
00174     inside an Argument environment. If there is still more than one
00175     term remaining we get the code: "could not convert".
00176     par tells which argument we need. If it is negative it should also
00177     be the last argument.
00178 */
00179
00180 int GetFloatArgument(PHEAD mpfr_t f_out,WORD *fun,int par)
00181 {
00182     WORD *term, *tn, *t, *tstop, first, ncoef, *arg, *argp, abspar;
00183     arg = fun+FUNHEAD;
00184     abspar = ABS(par);
00185     while ( arg < fun+fun[1] && abspar > 1 ) { abspar--; NEXTARG(arg) }
00186     if ( arg >= fun+fun[1] || abspar!= 1 ) return(-1);
00187     if ( par < 0 ) {
00188         argp = arg; NEXTARG(argp); if ( argp != fun+fun[1] ) return(-1);
00189     }
00190     if ( *arg < 0 ) {
00191         if ( *arg == -SNUMBER ) {
00192             mpfr_set_si(f_out,(LONG)(arg[1]),RND);
00193         }
00194         else if ( *arg == -SYMBOL && ( arg[1] == PISYMBOL ) ) {
00195             mpfr_const_pi(f_out,RND);
00196         }
00197         else if ( *arg == -INDEX && arg[1] >= 0 && arg[1] < AM.OffsetIndex ) {
00198             mpfr_set_ui(f_out,(ULONG)(arg[1]),RND);
00199         }
00200         else { return(-1); }
00201         return(0);
00202     }
00203     else if ( arg[0] != arg[ARGHEAD]+ARGHEAD ) { /* more than one term */
00204         return(-1);
00205     }
00206     term = arg+ARGHEAD;
00207     tn = term+*term;
00208     tstop = tn-ABS(tn[-1]);
00209     t = term+1;
00210     first = 1;
00211     while ( t <= tstop ) {
00212         if ( t == tstop ) { /* Fraction */
00213             if ( first ) RatToFloat(f_out,(UWORD *)tstop,tn[-1]);
00214             else {
00215                 RatToFloat(auxr4,(UWORD *)tstop,tn[-1]);
00216                 mpfr_mul(f_out,f_out,auxr4,RND);
00217             }
00218             return(0);
00219         }
00220         else if ( *t == FLOATFUN ) {
00221             if ( t+t[1] != tstop ) return(-1);
00222             if ( UnpackFloat(aux5,t) < 0 ) return(-1);
00223             if ( first ) {
00224                 mpfr_set_f(f_out,aux5,RND);
00225                 first = 0;
00226             }
00227             else {
00228                 mpfr_set_f(auxr4,aux5,RND);
00229                 mpfr_mul(f_out,f_out,auxr4,RND);
00230             }
00231             first = 0;
00232             if ( tn[-1] < 0 ) { /* change sign */
00233                 ncoef = -tn[-1];
00234                 mpfr_neg(f_out,f_out,RND);
00235             }
00236             else ncoef = tn[-1];
00237             if ( ncoef == 3 && tn[-2] == 1 && tn[-3] == 1 ) return(0);
00238         }
00239         /*
00240         If the argument was properly normalized we are not supposed to come here.
00241         MLOCK(ErrorMessageLock);
00242         MesPrint("Unnormalized argument in GetFloatArgument: %a",*term,term);
00243         MUNLOCK(ErrorMessageLock);

```



```

00244         Terminate(-1);
00245     }
00246     else if ( t[0] == SYMBOL && t[1] == 4 && t[2] == PISYMBOL && t[3] == 1 ) {
00247         if ( first ) {
00248             mpfr_const_pi(f_out,RND);
00249             first = 0;
00250         }
00251         else {
00252             mpfr_const_pi(auxr5,RND);
00253             mpfr_mul(f_out,f_out,auxr5,RND);
00254         }
00255     }
00256     else { /* This we do not / cannot do */
00257         return(-1);
00258     }
00259     t += t[1];
00260 }
00261 return(0);
00262 }
00263
00264 /*
00265     #] GetFloatArgument :
00266     #[ GetPiArgument :
00267
00268     Tests for sin,cos,tan whether the argument is a simple
00269     multiple of pi or can be reduced to such.
00270     Return value: -1: the answer is no.
00271     0-23: we have 0, 1/12*pi,...,23/12*pi_
00272     With success most values can be worked out by simple means.
00273 */
00274
00275 int GetPiArgument(PHEAD WORD *arg)
00276 {
00277     UWORD *coef, *co, *tco, twelve, *number, *denom, *c, *d;
00278     WORD i, ii, i2, iflip, nc, nd, *t, *tn, *tstop;
00279     int rem;
00280     /*
00281     One: determine whether there is a rational coefficient and a pi_:
00282     */
00283     if ( *arg == -SNUMBER && arg[1] == 0 ) return(0);
00284     if ( *arg == -SYMBOL && arg[1] == PISYMBOL ) return(12);
00285     if ( *arg < 0 ) return(-1);
00286     if ( arg[ARGHEAD] != *arg-ARGHEAD ) return(-1);
00287     t = arg+ARGHEAD+1;
00288     tn = arg+*arg; tstop = tn - ABS(tn[-1]);
00289     if ( *t != SYMBOL || t[1] != 4 || t[2] != PISYMBOL || t[3] != 1
00290         || t+t[1] != tstop ) return(-1);
00291     /*
00292     The denominator must be a divisor of 12
00293     1: copy the coefficient
00294     */
00295     co = coef = NumberMalloc("GetPiArgument");
00296     tco = (UWORD *)tstop;
00297     i = tn[-1]; if ( i < 0 ) { i = -i; iflip = 1; } else iflip = 0;
00298     ii = (i-1)/2;
00299     twelve = 12;
00300     NCOPY(co,tco,i);
00301     co = coef;
00302     Mully(BHEAD co,&ii,&twelve,1);
00303     /*
00304     Now the denominator should be 1
00305     */
00306     i = ii; i2 = i-1;
00307     number = co; denom = number + i;
00308     if ( i > 1 ) {
00309         while ( i2 > 0 ) {
00310             if ( denom[i2--] != 0 ) {
00311                 NumberFree(co,"GetPiArgument");
00312                 return(-1);
00313             }
00314         }
00315     }
00316     if ( *denom != 1 ) {
00317         NumberFree(co,"GetPiArgument");
00318         return(-1);
00319     }
00320     /*
00321     Now we need the numerator modulus 24.
00322     */
00323     if ( i == 1 ) {
00324         rem = *number % 24;
00325     }
00326     else {
00327         c = NumberMalloc("GetPiArgument");
00328         d = NumberMalloc("GetPiArgument");
00329         twelve *= 2;
00330         DivLong(number,i,&twelve,1,c,&nc,d,&nd);

```

```

00331         rem = *d % 24;
00332         NumberFree(d,"GetPiArgument");
00333         NumberFree(c,"GetPiArgument");
00334     }
00335     NumberFree(co,"GetPiArgument");
00336     if ( iflip && rem != 0 ) rem = 24-rem;
00337     return(rem);
00338 }
00339
00340 /*
00341     #] GetPiArgument :
00342     #[ EvaluateFun :
00343
00344     What we need to do is:
00345     1: look for a function to be treated.
00346     2: make sure its argument is treatable.
00347     3: call the proper mpfr_ function.
00348     4: accumulate the result.
00349     For some functions we have to insert 'smart' shortcuts as is
00350     the case with sin_(pi_) or sin_(pi_/6) of sqrt(4/9) etc.
00351     Otherwise we may have to insert a value for pi_ first.
00352     There are several types of arguments:
00353     a: (short) integers.
00354     b: rationals.
00355     c: floats.
00356     We accumulate the result(s) in auxr2. The argument comes in aux5
00357     and a result in auxr3 which then gets multiplied into auxr2.
00358     In the end we have to combine auxr2 with whatever coefficient
00359     existed already.
00360
00361     When the float system is started we need for aux only 3 variables
00362     per thread. auxr1-auxr3. This should be done separately.
00363     The main problem is the conversion of mpfr_t to float_ and/or mpf_t
00364     and back.
00365 */
00366
00367 int EvaluateFun(PHEAD WORD *term, WORD level, WORD *pars)
00368 {
00369     WORD *t, *tstop, *tt, *tnext, *newterm, i;
00370     WORD *oldworkpointer = AT.WorkPointer, nsize, nsgn, nsgn2;
00371     int retval = 0, first = 1, pimul;
00372
00373     tstop = term + *term; tstop -= ABS(tstop[-1]);
00374     if ( AT.WorkPointer < term+*term ) AT.WorkPointer = term + *term;
00375     t = term+1;
00376     mpfr_set_ui(auxr2,1L,RND);
00377     while ( t < tstop ) {
00378         if ( pars[2] == *t ) { /* have to do this one if possible */
00379             TestArgument:
00380             /*
00381                 There must be a single argument, except for the AGM or atan2 functions
00382             */
00383             tnext = t+t[1]; tt = t+FUNHEAD; NEXTARG(tt);
00384             if( *t == SYMBOL ) {
00385                 for ( WORD* ti = t+2; ti < t+t[1]; ti+=2 ) {
00386                     if( ( *ti == PISYMBOL || *ti == EESYMBOL || *ti == EMSYMBOL )
00387                         && ( pars[2] == ALLFUNCTIONS || pars[3] == *ti ) ) {
00388                         if ( *ti == PISYMBOL )
00389                             mpfr_const_pi(auxr3,RND);
00390                         else if ( *ti == EESYMBOL ) {
00391                             mpfr_set_ui(auxr3,1,RND);
00392                             mpfr_exp(auxr3,auxr3,RND);
00393                         }
00394                         else if ( *ti == EMSYMBOL )
00395                             mpfr_const_euler(auxr3,RND);
00396                         if ( ti[1] != 1 )
00397                             mpfr_pow_si(auxr3,auxr3,ti[1],RND);
00398                         mpfr_mul(auxr2,auxr2,auxr3,RND);
00399                         ti[1] = 0;
00400                         first = 0;
00401                     }
00402                 }
00403             }
00404             goto nextfun;
00405         }
00406         if ( tt != tnext && *t != AGMFUNTION && *t != ATAN2FUNCTION ) goto nextfun;
00407         if ( *t == SINFUNTION ) {
00408             pimul = GetPiArgument(BHEAD t+FUNHEAD);
00409             if ( pimul >= 0 && pimul < 24 ) {
00410                 if ( pimul == 0 || pimul == 12 ) goto getout;
00411                 if ( pimul > 12 ) { pimul = 24-pimul; nsgn = -1; }
00412                 else nsgn = 1;
00413                 if ( pimul > 6 ) pimul = 12-pimul;
00414                 switch ( pimul ) {
00415                     case 1:
00416 label1:
00417                         mpfr_sqrt_ui(auxr3,3L,RND);

```

```

00418             mpfr_sub_ui (auxr3, auxr3, 1L, RND);
00419             mpfr_sqrt_ui (auxr1, 2L, RND);
00420             mpfr_div_ui (auxr1, auxr1, 4L, RND);
00421             mpfr_mul (auxr3, auxr3, auxr1, RND);
00422             if ( nsgn < 0 ) mpfr_neg (auxr3, auxr3, RND);
00423             mpfr_mul (auxr2, auxr2, auxr3, RND);
00424             break;
00425         case 2:
00426 label2:
00427             mpfr_div_ui (auxr2, auxr2, 2L, RND);
00428             if ( nsgn < 0 ) mpfr_neg (auxr2, auxr2, RND);
00429             break;
00430         case 3:
00431 label3:
00432             mpfr_sqrt_ui (auxr3, 2L, RND);
00433             mpfr_div_ui (auxr3, auxr3, 2L, RND);
00434             if ( nsgn < 0 ) mpfr_neg (auxr3, auxr3, RND);
00435             mpfr_mul (auxr2, auxr2, auxr3, RND);
00436             break;
00437         case 4:
00438 label4:
00439             mpfr_sqrt_ui (auxr3, 3L, RND);
00440             mpfr_div_ui (auxr3, auxr3, 2L, RND);
00441             if ( nsgn < 0 ) mpfr_neg (auxr3, auxr3, RND);
00442             mpfr_mul (auxr2, auxr2, auxr3, RND);
00443             break;
00444         case 5:
00445 label5:
00446             mpfr_sqrt_ui (auxr3, 3L, RND);
00447             mpfr_add_ui (auxr3, auxr3, 1L, RND);
00448             mpfr_sqrt_ui (auxr1, 2L, RND);
00449             mpfr_div_ui (auxr1, auxr1, 4L, RND);
00450             mpfr_mul (auxr3, auxr3, auxr1, RND);
00451             if ( nsgn < 0 ) mpfr_neg (auxr3, auxr3, RND);
00452             mpfr_mul (auxr2, auxr2, auxr3, RND);
00453             break;
00454         case 6:
00455 label6:
00456             if ( nsgn < 0 ) mpfr_neg (auxr2, auxr2, RND);
00457             break;
00458     }
00459     *t = 0; first = 0;
00460     goto nextfun;
00461 }
00462 }
00463 else if ( *t == COSFUNCTION ) {
00464     pimul = GetPiArgument (BHEAD t+FUNHEAD);
00465     if ( pimul >= 0 && pimul < 24 ) {
00466         if ( pimul > 12 ) pimul = 24-pimul;
00467         if ( pimul > 6 ) { pimul = 12-pimul; nsgn = -1; }
00468         else nsgn = 1;
00469         if ( pimul == 6 ) goto getout;
00470         switch ( pimul ) {
00471             case 0: goto label6;
00472             case 1: goto label5;
00473             case 2: goto label4;
00474             case 3: goto label3;
00475             case 4: goto label2;
00476             case 5: goto label1;
00477         }
00478     }
00479 }
00480 else if ( *t == TANFUNCTION ) {
00481     pimul = GetPiArgument (BHEAD t+FUNHEAD);
00482     if ( pimul >= 0 && pimul < 24 ) {
00483         if ( pimul == 6 || pimul == 18 ) goto nextfun;
00484         if ( pimul == 0 || pimul == 12 ) goto getout;
00485         if ( pimul > 12 ) { pimul = 24-pimul; nsgn = -1; }
00486         else nsgn = 1;
00487         if ( pimul > 6 ) { pimul = 12-pimul; nsgn = -nsgn; }
00488         switch ( pimul ) {
00489             case 1:
00490                 mpfr_sqrt_ui (auxr3, 3L, RND);
00491                 mpfr_sub_ui (auxr3, auxr3, 2L, RND);
00492                 if ( nsgn > 0 ) mpfr_neg (auxr3, auxr3, RND);
00493                 mpfr_mul (auxr2, auxr2, auxr3, RND);
00494                 break;
00495             case 2:
00496                 mpfr_sqrt_ui (auxr3, 3L, RND);
00497                 mpfr_div_ui (auxr3, auxr3, 3L, RND);
00498                 if ( nsgn < 0 ) mpfr_neg (auxr3, auxr3, RND);
00499                 mpfr_mul (auxr2, auxr2, auxr3, RND);
00500                 break;
00501             case 3:
00502                 break;
00503             case 4:
00504                 mpfr_sqrt_ui (auxr3, 3L, RND);

```

```

00505             if ( nsgn < 0 ) mpfr_neg(auxr3,auxr3,RND);
00506             mpfr_mul(auxr2,auxr2,auxr3,RND);
00507             break;
00508         case 5:
00509             mpfr_sqrt_ui(auxr3,3L,RND);
00510             mpfr_add_ui(auxr3,auxr3,2L,RND);
00511             if ( nsgn < 0 ) mpfr_neg(auxr3,auxr3,RND);
00512             mpfr_mul(auxr2,auxr2,auxr3,RND);
00513             break;
00514     }
00515     *t = 0; first = 0;
00516     goto nextfun;
00517 }
00518 }
00519
00520 if ( *t == AGMFUNCTION || *t == ATAN2FUNCTION ) {
00521     if ( GetFloatArgument(BHEAD auxr1,t,1) < 0 ) goto nextfun;
00522     if ( GetFloatArgument(BHEAD auxr3,t,-2) < 0 ) goto nextfun;
00523 }
00524 else if ( GetFloatArgument(BHEAD auxr1,t,-1) < 0 ) goto nextfun;
00525 nsgn = mpfr_sgn(auxr1);
00526
00527 switch ( *t ) {
00528     case SQRTFUNCTION:
00529         if ( nsgn < 0 ) goto nextfun;
00530         if ( nsgn == 0 ) goto getout;
00531         else mpfr_sqrt(auxr3,auxr1,RND);
00532         mpfr_mul(auxr2,auxr2,auxr3,RND);
00533         break;
00534     case LNFUNCTION:
00535         if ( nsgn <= 0 ) goto nextfun;
00536         if ( mpfr_cmp_ui(auxr1,1L) == 0 ) goto getout;
00537         else mpfr_log(auxr3,auxr1,RND);
00538         mpfr_mul(auxr2,auxr2,auxr3,RND);
00539         break;
00540     case EXPFUNCTION:
00541         mpfr_exp(auxr3,auxr1,RND);
00542         mpfr_mul(auxr2,auxr2,auxr3,RND);
00543         break;
00544     case LI2FUNCTION: /* should be between -1 and +1 */
00545         if ( nsgn == 0 ) goto getout;
00546         mpfr_abs(auxr3,auxr1,RND);
00547         if ( mpfr_cmp_ui(auxr3,1L) > 0 ) goto nextfun;
00548         mpfr_li2(auxr3,auxr1,RND);
00549         mpfr_mul(auxr2,auxr2,auxr3,RND);
00550         break;
00551     case GAMMAFUN:
00552         /*
00553          * We cannot do this when the argument is a non-positive integer
00554          */
00555         if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] <= 0 ) goto nextfun;
00556         if ( t[FUNHEAD] == t[1]-FUNHEAD && ABS(t[t[1]-1]) ==
00557             t[FUNHEAD]-ARGHEAD-1 && t[t[1]-1] < 0 ) {
00558             nsize = (-t[t[1]-1]-1)/2;
00559             if ( t[t[1]-1-nsize] == 1 ) {
00560                 for ( i = 1; i < nsize; i++ ) {
00561                     if ( t[t[1]-1-nsize+i] != 0 ) break;
00562                 }
00563                 if ( i >= nsize ) goto nextfun;
00564             }
00565         }
00566         mpfr_gamma(auxr3,auxr1,RND);
00567         mpfr_mul(auxr2,auxr2,auxr3,RND);
00568         break;
00569     case AGMFUNCTION:
00570         nsgn = mpfr_sgn(auxr1);
00571         nsgn2 = mpfr_sgn(auxr3);
00572         if ( nsgn == 0 || nsgn2 == 0 ) goto getout;
00573         if ( nsgn < 0 || nsgn2 < 0 ) goto nextfun;
00574         mpfr_agm(auxr3,auxr1,auxr3,RND);
00575         mpfr_mul(auxr2,auxr2,auxr3,RND);
00576         break;
00577     case SINHFUNCTION:
00578         if ( nsgn == 0 ) goto getout;
00579         mpfr_sinh(auxr3,auxr1,RND);
00580         mpfr_mul(auxr2,auxr2,auxr3,RND);
00581         break;
00582     case COSHFUNCTION:
00583         mpfr_cosh(auxr3,auxr1,RND);
00584         mpfr_mul(auxr2,auxr2,auxr3,RND);
00585         break;
00586     case TANHFUNCTION:
00587         if ( nsgn == 0 ) goto getout;
00588         mpfr_tanh(auxr3,auxr1,RND);
00589         mpfr_mul(auxr2,auxr2,auxr3,RND);
00590         break;
00591     case ASINHFUNCTION:

```

```

00592         if ( nsgn == 0 ) goto getout;
00593         mpfr_asinh(auxr3,auxr1,RND);
00594         mpfr_mul(auxr2,auxr2,auxr3,RND);
00595     break;
00596     case ACOSHFUNCTION:
00597         if ( mpfr_cmp_ui(auxr1,1L) < 0 ) goto nextfun;
00598         if ( mpfr_cmp_ui(auxr1,1L) == 0 ) goto getout;
00599         mpfr_acosh(auxr3,auxr1,RND);
00600         mpfr_mul(auxr2,auxr2,auxr3,RND);
00601     break;
00602     case ATANHFUNCTION:
00603         if ( nsgn == 0 ) goto getout;
00604         mpfr_abs(auxr3,auxr1,RND);
00605         if ( mpfr_cmp_ui(auxr3,1L) >= 0 ) goto nextfun;
00606         mpfr_atanh(auxr3,auxr1,RND);
00607         mpfr_mul(auxr2,auxr2,auxr3,RND);
00608     break;
00609     case ASINFUNCTION:
00610         if ( nsgn == 0 ) goto getout;
00611         mpfr_abs(auxr3,auxr1,RND);
00612         if ( mpfr_cmp_ui(auxr3,1L) > 0 ) goto nextfun;
00613         mpfr_asin(auxr3,auxr1,RND);
00614         mpfr_mul(auxr2,auxr2,auxr3,RND);
00615     break;
00616     case ACOSFUNCTION:
00617         if ( mpfr_cmp_ui(auxr1,1L) == 0 ) goto getout;
00618         mpfr_abs(auxr3,auxr1,RND);
00619         if ( mpfr_cmp_ui(auxr3,1L) > 0 ) goto nextfun;
00620         mpfr_acos(auxr3,auxr1,RND);
00621         mpfr_mul(auxr2,auxr2,auxr3,RND);
00622     break;
00623     case ATANFUNCTION:
00624         if ( nsgn == 0 ) goto getout;
00625         mpfr_atan(auxr3,auxr1,RND);
00626         mpfr_mul(auxr2,auxr2,auxr3,RND);
00627     break;
00628     case ATAN2FUNCTION:
00629         nsgn = mpfr_sgn(auxr1);
00630         nsgn2 = mpfr_sgn(auxr3);
00631         // We follow the conventions of mpfr here:
00632         if ( nsgn == 0 && nsgn2 >= 0 ) goto getout;
00633         mpfr_atan2(auxr3,auxr1,auxr3,RND);
00634         mpfr_mul(auxr2,auxr2,auxr3,RND);
00635     break;
00636     case SINFUNCTION:
00637         mpfr_sin(auxr3,auxr1,RND);
00638         mpfr_mul(auxr2,auxr2,auxr3,RND);
00639     break;
00640     case COSFUNCTION:
00641         mpfr_cos(auxr3,auxr1,RND);
00642         mpfr_mul(auxr2,auxr2,auxr3,RND);
00643     break;
00644     case TANFUNCTION:
00645         mpfr_tan(auxr3,auxr1,RND);
00646         mpfr_mul(auxr2,auxr2,auxr3,RND);
00647     break;
00648     default:
00649         goto nextfun;
00650     break;
00651 }
00652 first = 0;
00653 *t = 0;
00654 goto nextfun;
00655 }
00656 else if ( pars[2] == ALLFUNCTIONS ) {
00657     /*
00658         Now we have to test whether this is one of our functions
00659     */
00660     if ( t[1] == FUNHEAD ) goto nextfun;
00661     switch ( *t ) {
00662     case SQRTFUNCTION:
00663     case LNFUNCTION:
00664     case LI2FUNCTION:
00665     case LINFUNCTION:
00666     case EXPFUNCTION:
00667     case ASINFUNCTION:
00668     case ACOSFUNCTION:
00669     case ATANFUNCTION:
00670     case ATAN2FUNCTION:
00671     case SINHFUNCTION:
00672     case COSHFUNCTION:
00673     case TANHFUNCTION:
00674     case ASINHFUNCTION:
00675     case ACOSHFUNCTION:
00676     case ATANHFUNCTION:
00677     case SINFUNCTION:
00678     case COSFUNCTION:

```

```

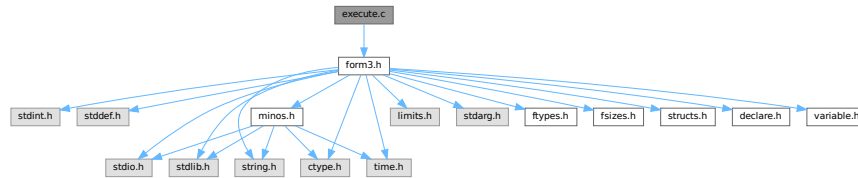
00679             case TANFUNCTION:
00680             case AGMFUNTION:
00681             case SYMBOL:
00682                 goto TestArgument;
00683             case MZV:
00684             case EULER:
00685             case MZVHALF:
00686             default:
00687                 goto nextfun;
00688         }
00689     }
00690     else goto nextfun;
00691 nextfun:
00692     t += t[1];
00693 }
00694 if ( first == 1 ) return(Generator(BHEAD term,level));
00695 mpfr_get_f(aux4,auxr2,RND);
00696 /*
00697 Step 3:
00698 Now the regular coefficient, if it is not 1/1.
00699 We have two cases: size +- 3, or bigger.
00700 */
00701 nsize = term[*term-1];
00702 if ( nsize < 0 ) { nsize = -nsize; nsgn = -1; }
00703 else nsgn = 1;
00704 if ( aux4->_mp_size < 0 ) {
00705     aux4->_mp_size = -aux4->_mp_size;
00706     nsgn = -nsgn;
00707 }
00708 }
00709 if ( nsize == 3 ) {
00710     if ( tstop[0] != 1 ) {
00711         mpfr_mul_ui(aux4,aux4,(ULONG)((UWORD)tstop[0]));
00712     }
00713     if ( tstop[1] != 1 ) {
00714         mpfr_div_ui(aux4,aux4,(ULONG)((UWORD)tstop[1]));
00715     }
00716 }
00717 else {
00718     RatToFloat(aux5,(UWORD *)tstop,nsize);
00719     mpfr_mul(aux4,aux4,aux5);
00720 }
00721 }
00722 /*
00723 Now we have to locate possible other float_ functions.
00724 Note possible incompatibilities between the mpfr and mpfr formats.
00725 */
00726 t = term+1;
00727 while ( t < tstop ) {
00728     if ( *t == FLOATFUN ) {
00729         UnpackFloat(aux5,t);
00730         mpfr_mul(aux4,aux4,aux5);
00731     }
00732     t += t[1];
00733 }
00734 /*
00735 Now we should compose the new term in the Workspace.
00736 */
00737 t = term+1;
00738 newterm = AT.WorkPointer;
00739 tt = newterm+1;
00740 while ( t < tstop ) {
00741     if ( *t == 0 || *t == FLOATFUN ) t += t[1];
00742     else {
00743         i = t[1]; NCOPY(tt,t,i);
00744     }
00745 }
00746 PackFloat(tt,aux4);
00747 tt += tt[1];
00748 *tt++ = 1; *tt++ = 1; *tt++ = 3*nsgn;
00749 *newterm = tt-newterm;
00750 AT.WorkPointer = tt;
00751 retval = Generator(BHEAD newterm,level);
00752 getout:
00753 AT.WorkPointer = oldworkpointer;
00754 return(retval);
00755 }
00756 /*
00757 #] EvaluateFun :
00758 #] mpfr_ :
00759 */

```

4.31 execute.c File Reference

```
#include "form3.h"
```

Include dependency graph for execute.c:



Macros

- `#define` [CURRENTBRACKET](#) 1
- `#define` [BRACKETCURRENTEXPR](#) 2
- `#define` [BRACKETOTHEREXPR](#) 3
- `#define` [NOBRACKETACTIVE](#) 4

Functions

- `int` [CleanExpr](#) (WORD par)
- `int` [PopVariables](#) (void)
- `void` [MakeGlobal](#) (void)
- `void` [TestDrop](#) (void)
- `void` [PutInVflags](#) (WORD nexpr)
- `int` [DoExecute](#) (WORD par, WORD skip)
- `int` [PutBracket](#) (PHEAD WORD *termin)
- `void` [SpecialCleanup](#) (PHEAD0)
- `void` [SetMods](#) (void)
- `void` [UnSetMods](#) (void)
- `void` [ExchangeExpressions](#) (int num1, int num2)
- `int` [GetFirstBracket](#) (WORD *term, int num)
- `int` [GetFirstTerm](#) (WORD *term, int num, int pre)
- `int` [GetContent](#) (WORD *content, int num)
- `int` [CleanupTerm](#) (WORD *term)
- `WORD` [ContentMerge](#) (PHEAD WORD *content, WORD *term)
- `LONG` [TermsInExpression](#) (WORD num)
- `LONG` [SizeOfExpression](#) (WORD num)
- `void` [UpdatePositions](#) (void)
- `LONG` [CountTerms1](#) (PHEAD0)
- `LONG` [TermsInBracket](#) (PHEAD WORD *term, WORD level)

4.31.1 Detailed Description

The routines that start the execution phase of a module. It also contains the routines for placing the bracket subterm.

Definition in file [execute.c](#).

4.31.2 Macro Definition Documentation

4.31.2.1 CURRENTBRACKET

```
#define CURRENTBRACKET 1
```

Definition at line [2566](#) of file [execute.c](#).

4.31.2.2 BRACKETCURRENTEXPR

```
#define BRACKETCURRENTEXPR 2
```

Definition at line [2567](#) of file [execute.c](#).

4.31.2.3 BRACKETOTHEREXPR

```
#define BRACKETOTHEREXPR 3
```

Definition at line [2568](#) of file [execute.c](#).

4.31.2.4 NOBRACKETACTIVE

```
#define NOBRACKETACTIVE 4
```

Definition at line [2569](#) of file [execute.c](#).

4.31.3 Function Documentation

4.31.3.1 CleanExpr()

```
int CleanExpr (  
    WORD par )
```

Definition at line [47](#) of file [execute.c](#).

4.31.3.2 PopVariables()

```
int PopVariables (  
    void )
```

Definition at line [211](#) of file [execute.c](#).

4.31.3.3 MakeGlobal()

```
void MakeGlobal (  
    void )
```

Definition at line [383](#) of file [execute.c](#).

4.31.3.4 TestDrop()

```
void TestDrop (
    void )
```

Definition at line [484](#) of file [execute.c](#).

4.31.3.5 PutInVflags()

```
void PutInVflags (
    WORD nexpr )
```

Definition at line [562](#) of file [execute.c](#).

4.31.3.6 DoExecute()

```
int DoExecute (
    WORD par,
    WORD skip )
```

Definition at line [616](#) of file [execute.c](#).

4.31.3.7 PutBracket()

```
int PutBracket (
    PHEAD WORD * termin )
```

Definition at line [1112](#) of file [execute.c](#).

4.31.3.8 SpecialCleanup()

```
void SpecialCleanup (
    PHEAD0 )
```

Definition at line [1626](#) of file [execute.c](#).

4.31.3.9 SetMods()

```
void SetMods (
    void )
```

Definition at line [1640](#) of file [execute.c](#).

4.31.3.10 UnSetMods()

```
void UnSetMods (
    void )
```

Definition at line [1658](#) of file [execute.c](#).

4.31.3.11 ExchangeExpressions()

```
void ExchangeExpressions (
    int num1,
    int num2 )
```

Definition at line 1673 of file [execute.c](#).

4.31.3.12 GetFirstBracket()

```
int GetFirstBracket (
    WORD * term,
    int num )
```

Definition at line 1747 of file [execute.c](#).

4.31.3.13 GetFirstTerm()

```
int GetFirstTerm (
    WORD * term,
    int num,
    int pre )
```

Gets the first term of an expression. Routine should be thread safe.

Parameters

out	<i>term</i>	Pointer to the buffer where the term should be written.
in	<i>num</i>	The expression number from which to get the term.
in	<i>pre</i>	Denotes a pre-processor call (1) or not (0). Used to determine the correct source file for the expression.

Returns

First WORD of term, a negative value denotes an error.

Definition at line 1866 of file [execute.c](#).

References [FiLe::handle](#), [VaRrEnUm::lo](#), and [ReNuMbEr::symb](#).

4.31.3.14 GetContent()

```
int GetContent (
    WORD * content,
    int num )
```

Definition at line 1974 of file [execute.c](#).

4.31.3.15 CleanupTerm()

```
int CleanupTerm (
    WORD * term )
```

Definition at line 2091 of file [execute.c](#).

4.31.3.16 ContentMerge()

```
WORD ContentMerge (
    PHEAD WORD * content,
    WORD * term )
```

Definition at line 2115 of file [execute.c](#).

4.31.3.17 TermsInExpression()

```
LONG TermsInExpression (
    WORD num )
```

Definition at line 2379 of file [execute.c](#).

4.31.3.18 SizeOfExpression()

```
LONG SizeOfExpression (
    WORD num )
```

Definition at line 2391 of file [execute.c](#).

4.31.3.19 UpdatePositions()

```
void UpdatePositions (
    void )
```

Definition at line 2403 of file [execute.c](#).

4.31.3.20 CountTerms1()

```
LONG CountTerms1 (
    PHEAD0 )
```

Definition at line 2463 of file [execute.c](#).

4.31.3.21 TermsInBracket()

```
LONG TermsInBracket (
    PHEAD WORD * term,
    WORD level )
```

Definition at line 2571 of file [execute.c](#).

4.32 execute.c

[Go to the documentation of this file.](#)

```

00001
00006 /* #[] License : */
00007 /*
00008 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009 *   When using this file you are requested to refer to the publication
00010 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011 *   This is considered a matter of courtesy as the development was paid
00012 *   for by FOM the Dutch physics granting agency and we would like to
00013 *   be able to track its scientific use to convince FOM of its value
00014 *   for the community.
00015 *
00016 *   This file is part of FORM.
00017 *
00018 *   FORM is free software: you can redistribute it and/or modify it under the
00019 *   terms of the GNU General Public License as published by the Free Software
00020 *   Foundation, either version 3 of the License, or (at your option) any later
00021 *   version.
00022 *
00023 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00026 *   details.
00027 *
00028 *   You should have received a copy of the GNU General Public License along
00029 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00030 */
00031 /* #[] License : */
00032 /*
00033 *   #[] Includes : execute.c
00034 */
00035
00036 #include "form3.h"
00037
00038 /*
00039 *   #[] Includes :
00040 *   #[] DoExecute :
00041 *   #[] CleanExpr :
00042 *
00043 *   par == 1 after .store or .clear
00044 *   par == 0 after .sort
00045 */
00046
00047 int CleanExpr(WORD par)
00048 {
00049     GETIDENTITY
00050     WORD j, n, i;
00051     POSITION length;
00052     EXPRESSIONS e_in, e_out, e;
00053     int numhid = 0;
00054     NAMENODE *node;
00055     n = NumExpressions;
00056     j = 0;
00057     e_in = e_out = Expressions;
00058     if ( n > 0 ) { do {
00059         e_in->vflags &= ~( TOBEFACTORED | TOBEUNFACTORED );
00060         if ( par ) {
00061             if ( e_in->renumlists ) {
00062                 if ( e_in->renumlists != AN.dummyrenumlist )
00063                     M_free(e_in->renumlists, "Renumlist");
00064                 e_in->renumlists = 0;
00065             }
00066             if ( e_in->renum ) {
00067                 M_free(e_in->renum, "Renum"); e_in->renum = 0;
00068             }
00069         }
00070         if ( e_in->status == HIDDENEXPRESSION
00071             || e_in->status == HIDDENEXPRESSION ) numhid++;
00072         switch ( e_in->status ) {
00073             case SPECTATOREXPRESSION:
00074             case LOCALEXPRESSION:
00075             case HIDDENEXPRESSION:
00076                 if ( par ) {
00077                     AC.exprnames->namenode[e_in->node].type = CDELETE;
00078                     AC.DidClean = 1;
00079                     if ( e_in->status != HIDDENEXPRESSION )
00080                         ClearBracketIndex(e_in->Expressions);
00081                     break;
00082                 }
00083                 /* fall through */
00084             case GLOBALEXPRESSION:
00085             case HIDDENEXPRESSION:
00086                 if ( par ) {

```

```

00087 #ifdef WITHMPI
00088     /*
00089      * Broadcast the global expression from the master to the all workers.
00090      */
00091     if ( PF_BroadcastExpr(e_in, e_in->status == HIDDENEXPRESSION ? AR.hidefile :
AR.outfile) ) return -1;
00092     if ( PF.me == MASTER ) {
00093 #endif
00094         e = e_in;
00095         i = n-1;
00096         while ( --i >= 0 ) {
00097             e++;
00098             if ( e_in->status == HIDDENEXPRESSION ) {
00099                 if ( e->status == HIDDENEXPRESSION
00100                     || e->status == HIDDENLEXPRESSION ) break;
00101             }
00102             else {
00103                 if ( e->status == GLOBALEXPRESSION
00104                     || e->status == LOCALEXPRESSION ) break;
00105             }
00106         }
00107 #ifdef WITHMPI
00108     }
00109     else {
00110         /*
00111          * On the slaves, the broadcast expression is sitting at the end of the file.
00112          */
00113         e = e_in;
00114         i = -1;
00115     }
00116 #endif
00117     if ( i >= 0 ) {
00118         DIFPOS(length,e->onfile,e_in->onfile);
00119     }
00120     else {
00121         FILEHANDLE *f = e_in->status == HIDDENEXPRESSION ? AR.hidefile : AR.outfile;
00122         if ( f->handle < 0 ) {
00123             SETBASELENGTH(length,TOLONG(f->Pofull)
00124                 - TOLONG(f->Pobuffer)
00125                 - BASEPOSITION(e_in->onfile));
00126         }
00127         else {
00128             SeekFile(f->handle,&(f->filesize),SEEK_SET);
00129             DIFPOS(length,f->filesize,e_in->onfile);
00130         }
00131     }
00132     if ( ToStorage(e_in,&length) ) {
00133         return(MesCall("CleanExpr"));
00134     }
00135     e_in->status = STOREDEXPRESSION;
00136     if ( e_in->status != HIDDENEXPRESSION )
00137         ClearBracketIndex(e_in->Expressions);
00138 }
00139 /* fall through */
00140 case SKIPLEXPRESSION:
00141 case DROPLEXPRESSION:
00142 case DROPHLEXPRESSION:
00143 case DROPGEXPRESSION:
00144 case DROPHGEXPRESSION:
00145 case STOREDEXPRESSION:
00146 case DROPSPECTATOREXPRESSION:
00147     if ( e_out != e_in ) {
00148         node = AC.exprnames->namenode + e_in->node;
00149         node->number = e_out - Expressions;
00150
00151         e_out->onfile = e_in->onfile;
00152         e_out->size = e_in->size;
00153         e_out->printflag = 0;
00154         if ( par ) e_out->status = STOREDEXPRESSION;
00155         else e_out->status = e_in->status;
00156         e_out->name = e_in->name;
00157         e_out->node = e_in->node;
00158         e_out->renum = e_in->renum;
00159         e_out->renumlists = e_in->renumlists;
00160         e_out->counter = e_in->counter;
00161         e_out->hidelevel = e_in->hidelevel;
00162         e_out->inmem = e_in->inmem;
00163         e_out->bracketinfo = e_in->bracketinfo;
00164         e_out->newbracketinfo = e_in->newbracketinfo;
00165         e_out->numdummies = e_in->numdummies;
00166         e_out->numfactors = e_in->numfactors;
00167         e_out->vflags = e_in->vflags;
00168         e_out->uflags = e_in->uflags;
00169         e_out->sizeprototype = e_in->sizeprototype;
00170     }
00171 #ifdef PARALLELCODE
00172     e_out->partodo = 0;

```

```

00173 #endif
00174         e_out++;
00175         j++;
00176         break;
00177     case DROPEXPRESSION:
00178         break;
00179     default:
00180         AC.exprnames->namenode[e_in->node].type = CDELETE;
00181         AC.DidClean = 1;
00182         break;
00183     }
00184     e_in++;
00185 } while ( --n > 0 ); }
00186 UpdateMaxSize();
00187 NumExpressions = j;
00188 if ( numhid == 0 && AR.hidefile->PObuffer ) {
00189     if ( AR.hidefile->handle >= 0 ) {
00190         CloseFile(AR.hidefile->handle);
00191         remove(AR.hidefile->name);
00192         AR.hidefile->handle = -1;
00193     }
00194     AR.hidefile->POfull =
00195     AR.hidefile->POfill = AR.hidefile->PObuffer;
00196     PUTZERO(AR.hidefile->POposition);
00197 }
00198 FlushSpectators();
00199 return(0);
00200 }
00201
00202 /*
00203     #] CleanExpr :
00204     #[ PopVariables :
00205
00206     Pops the local variables from the tables.
00207     The Expressions are reprocessed and their tables are compactified.
00208
00209 */
00210 int PopVariables(void)
00211 {
00212     GETIDENTITY
00213     WORD i, j;
00214     int retval;
00215     UBYTE *s;
00216
00217     retval = CleanExpr(1);
00218     ResetVariables(1);
00219
00220     if ( AC.DidClean ) CompactifyTree(AC.exprnames,EXPRNAMES);
00221
00222     AC.CodesFlag = AM.gCodesFlag;
00223     AC.NamesFlag = AM.gNamesFlag;
00224     AC.StatsFlag = AM.gStatsFlag;
00225 #ifdef WITHFLOAT
00226     AC.MaxWeight = AM.gMaxWeight;
00227     AC.DefaultPrecision = AM.gDefaultPrecision;
00228 #endif
00229     AC.OldFactArgFlag = AM.gOldFactArgFlag;
00230     AC.TokensWriteFlag = AM.gTokensWriteFlag;
00231     AC.extrasymbols = AM.gextrasymbols;
00232     if ( AC.extrasym ) { M_free(AC.extrasym,"extrasym"); AC.extrasym = 0; }
00233     i = 1; s = AM.gextrasym; while ( *s ) { s++; i++; }
00234     AC.extrasym = (UBYTE *)Malloc1(i*sizeof(UBYTE),"extrasym");
00235     for ( j = 0; j < i; j++ ) AC.extrasym[j] = AM.gextrasym[j];
00236     AO.NoSpacesInNumbers = AM.gNoSpacesInNumbers;
00237     AC.IndentSpace = AM.gIndentSpace;
00238     AC.lUnitTrace = AM.gUnitTrace;
00239     AC.lDefDim = AM.gDefDim;
00240     AC.lDefDim4 = AM.gDefDim4;
00241     if ( AC.halfmod ) {
00242         if ( AC.ncmod == AM.gncmod && AC.modmode == AM.gmodmode ) {
00243             j = ABS(AC.ncmod);
00244             while ( --j >= 0 ) {
00245                 if ( AC.cmod[j] != AM.gcmod[j] ) break;
00246             }
00247             if ( j >= 0 ) {
00248                 M_free(AC.halfmod,"halfmod");
00249                 AC.halfmod = 0; AC.nhalfmod = 0;
00250             }
00251         }
00252         else {
00253             M_free(AC.halfmod,"halfmod");
00254             AC.halfmod = 0; AC.nhalfmod = 0;
00255         }
00256     }
00257     if ( AC.modinverses ) {
00258         if ( AC.ncmod == AM.gncmod && AC.modmode == AM.gmodmode ) {

```

```

00260         j = ABS(AC.ncmod);
00261         while ( --j >= 0 ) {
00262             if ( AC.cmod[j] != AM.gcmmod[j] ) break;
00263         }
00264         if ( j >= 0 ) {
00265             M_free(AC.modinverses,"modinverses");
00266             AC.modinverses = 0;
00267         }
00268     }
00269     else {
00270         M_free(AC.modinverses,"modinverses");
00271         AC.modinverses = 0;
00272     }
00273 }
00274 AN.ncmod = AC.ncmod = AM.gncmod;
00275 AC.npowmod = AM.gnpowmod;
00276 AC.modmode = AM.gmodmode;
00277 if ( ( ( AC.modmode & INVERSETABLE ) != 0 ) && ( AC.modinverses == 0 ) )
00278     MakeInverses();
00279 AC.funpowers = AM.gfunpowers;
00280 AC.lPolyFun = AM.gPolyFun;
00281 AC.lPolyFunInv = AM.gPolyFunInv;
00282 AC.lPolyFunType = AM.gPolyFunType;
00283 AC.lPolyFunExp = AM.gPolyFunExp;
00284 AR.PolyFunVar = AC.lPolyFunVar = AM.gPolyFunVar;
00285 AC.lPolyFunPow = AM.gPolyFunPow;
00286 AC.parallelflag = AM.gparallelflag;
00287 AC.ProcessBucketSize = AC.mProcessBucketSize = AM.gProcessBucketSize;
00288 AC.properorderflag = AM.gproperorderflag;
00289 AC.ThreadBucketSize = AM.gThreadBucketSize;
00290 AC.ThreadStats = AM.gThreadStats;
00291 AC.FinalStats = AM.gFinalStats;
00292 AC.OldGCDflag = AM.gOldGCDflag;
00293 AC.WTimeStatsFlag = AM.gWTimeStatsFlag;
00294 AC.ThreadsFlag = AM.gThreadsFlag;
00295 AC.ThreadBalancing = AM.gThreadBalancing;
00296 AC.ThreadSortFileSynch = AM.gThreadSortFileSynch;
00297 AC.ProcessStats = AM.gProcessStats;
00298 AC.OldParallelStats = AM.gOldParallelStats;
00299 AC.IsFortran90 = AM.gIsFortran90;
00300 AC.SizeCommuteInSet = AM.gSizeCommuteInSet;
00301 PruneExtraSymbols(AM.gnumextrasyms);
00302
00303 if ( AC.Fortran90Kind ) {
00304     M_free(AC.Fortran90Kind,"Fortran90 Kind");
00305     AC.Fortran90Kind = 0;
00306 }
00307 if ( AM.gFortran90Kind ) {
00308     AC.Fortran90Kind = strdup(AM.gFortran90Kind,"Fortran90 Kind");
00309 }
00310 if ( AC.ThreadsFlag && AM.totalnumberofthreads > 1 ) AS.MultiThreaded = 1;
00311 {
00312     UWORD *p, *m;
00313     p = AM.gcmmod;
00314     m = AC.cmod;
00315     j = ABS(AC.ncmod);
00316     NCOPY(m,p,j);
00317     p = AM.gpowmod;
00318     m = AC.powmod;
00319     j = AC.npowmod;
00320     NCOPY(m,p,j);
00321     if ( AC.DirtPow ) {
00322         if ( MakeModTable() ) {
00323             MesPrint("===No printing in powers of generator");
00324         }
00325         AC.DirtPow = 0;
00326     }
00327 }
00328 {
00329     WORD *p, *m;
00330     p = AM.gUniTrace;
00331     m = AC.lUniTrace;
00332     j = 4;
00333     NCOPY(m,p,j);
00334 }
00335 AC.Cnumpows = AM.gCnumpows;
00336 AC.OutputMode = AM.gOutputMode;
00337 AC.OutputSpaces = AM.gOutputSpaces;
00338 AC.OutNumberType = AM.gOutNumberType;
00339 AR.SortType = AC.SortType = AM.gSortType;
00340 AC.ShortStatsMax = AM.gShortStatsMax;
00341 /*
00342 Now we have to clean up the commutation properties
00343 */
00344 for ( i = 0; i < NumFunctions; i++ ) functions[i].flags &= ~COULDCOMMUTE;
00345 if ( AC.CommuteInSet ) {
00346     WORD *g, *gg;

```

```

00347     g = AC.CommuteInSet;
00348     while ( *g ) {
00349         gg = g+1; g += *g;
00350         while ( gg < g ) {
00351             if ( *gg <= GAMMASEVEN && *gg >= GAMMA ) {
00352                 functions[GAMMA-FUNCTION].flags |= COULDCOMMUTE;
00353                 functions[GAMMAI-FUNCTION].flags |= COULDCOMMUTE;
00354                 functions[GAMMAFIVE-FUNCTION].flags |= COULDCOMMUTE;
00355                 functions[GAMMASIX-FUNCTION].flags |= COULDCOMMUTE;
00356                 functions[GAMMASEVEN-FUNCTION].flags |= COULDCOMMUTE;
00357             }
00358             else {
00359                 functions[*gg-FUNCTION].flags |= COULDCOMMUTE;
00360             }
00361         }
00362     }
00363 }
00364 /*
00365 Clean up the dictionaries.
00366 */
00367 for ( i = AO.NumDictionaries-1; i >= AO.gNumDictionaries; i-- ) {
00368     RemoveDictionary(AO.Dictionaries[i]);
00369     M_free(AO.Dictionaries[i], "Dictionary");
00370 }
00371 for( ; i >= 0; i-- ) {
00372     ShrinkDictionary(AO.Dictionaries[i]);
00373 }
00374 AO.NumDictionaries = AO.gNumDictionaries;
00375 return(retval);
00376 }
00377
00378 /*
00379     #] PopVariables :
00380     #[ MakeGlobal :
00381 */
00382
00383 void MakeGlobal(void)
00384 {
00385     WORD i, j, *pp, *mm;
00386     UWORD *p, *m;
00387     UBYTE *s;
00388     Globalize(0);
00389
00390     AM.gCodesFlag = AC.CodesFlag;
00391     AM.gNamesFlag = AC.NamesFlag;
00392     AM.gStatsFlag = AC.StatsFlag;
00393     #ifdef WITHFLOAT
00394     AM.gMaxWeight = AC.MaxWeight;
00395     AM.gDefaultPrecision = AC.DefaultPrecision;
00396     #endif
00397     AM.gOldFactArgFlag = AC.OldFactArgFlag;
00398     AM.gextrasymsymbols = AC.gextrasymsymbols;
00399     if ( AM.gextrasym ) { M_free(AM.gextrasym, "extrasym"); AM.gextrasym = 0; }
00400     i = 1; s = AC.gextrasym; while ( *s ) { s++; i++; }
00401     AM.gextrasym = (UBYTE *)Malloc1(i*sizeof(UBYTE), "extrasym");
00402     for ( j = 0; j < i; j++ ) AM.gextrasym[j] = AC.gextrasym[j];
00403     AM.gTokensWriteFlag = AC.TokensWriteFlag;
00404     AM.gNoSpacesInNumbers = AO.NoSpacesInNumbers;
00405     AM.gIndentSpace = AO.IndentSpace;
00406     AM.gUnitTrace = AC.lUnitTrace;
00407     AM.gDefDim = AC.lDefDim;
00408     AM.gDefDim4 = AC.lDefDim4;
00409     AM.gncmod = AC.ncmod;
00410     AM.gnpowmod = AC.npowmod;
00411     AM.gmodmode = AC.modmode;
00412     AM.gCnumpows = AC.Cnumpows;
00413     AM.gOutputMode = AC.OutputMode;
00414     AM.gOutputSpaces = AC.OutputSpaces;
00415     AM.gOutNumberType = AC.OutNumberType;
00416     AM.gfunpowers = AC.funpowers;
00417     AM.gPolyFun = AC.lPolyFun;
00418     AM.gPolyFunInv = AC.lPolyFunInv;
00419     AM.gPolyFunType = AC.lPolyFunType;
00420     AM.gPolyFunExp = AC.lPolyFunExp;
00421     AM.gPolyFunVar = AC.lPolyFunVar;
00422     AM.gPolyFunPow = AC.lPolyFunPow;
00423     AM.gparallelflag = AC.parallelflag;
00424     AM.gProcessBucketSize = AC.ProcessBucketSize;
00425     AM.gproperorderflag = AC.properorderflag;
00426     AM.gThreadBucketSize = AC.ThreadBucketSize;
00427     AM.gThreadStats = AC.ThreadStats;
00428     AM.gFinalStats = AC.FinalStats;
00429     AM.gOldGCDflag = AC.OldGCDflag;
00430     AM.gWTimeStatsFlag = AC.WTimeStatsFlag;
00431     AM.gThreadsFlag = AC.ThreadsFlag;
00432     AM.gThreadBalancing = AC.ThreadBalancing;
00433     AM.gThreadSortFileSynch = AC.ThreadSortFileSynch;

```



```

00434     AM.gProcessStats = AC.ProcessStats;
00435     AM.gOldParallelStats = AC.OldParallelStats;
00436     AM.gIsFortran90 = AC.IsFortran90;
00437     AM.gSizeCommuteInSet = AC.SizeCommuteInSet;
00438     AM.gnumextrasyms = (cbuf+AM.sbufnum)->numrhs;
00439     if ( AM.gFortran90Kind ) {
00440         M_free(AM.gFortran90Kind,"Fortran 90 Kind");
00441         AM.gFortran90Kind = 0;
00442     }
00443     if ( AC.Fortran90Kind ) {
00444         AM.gFortran90Kind = strdup(AC.Fortran90Kind,"Fortran 90 Kind");
00445     }
00446     p = AM.gcmmod;
00447     m = AC.cmod;
00448     i = ABS(AC.ncmod);
00449     NCOPY(p,m,i);
00450     p = AM.gpowmod;
00451     m = AC.powmod;
00452     i = AC.npowmod;
00453     NCOPY(p,m,i);
00454     pp = AM.gUniTrace;
00455     mm = AC.lUniTrace;
00456     i = 4;
00457     NCOPY(pp,mm,i);
00458     AM.gSortType = AC.SortType;
00459     AM.gShortStatsMax = AC.ShortStatsMax;
00460
00461     if ( AO.CurrentDictionary > 0 || AP.OpenDictionary > 0 ) {
00462         Warning("You cannot have an open or selected dictionary at a .global. Dictionary closed.");
00463         AP.OpenDictionary = 0;
00464         AO.CurrentDictionary = 0;
00465     }
00466
00467     AO.gNumDictionaries = AO.NumDictionaries;
00468     for ( i = 0; i < AO.NumDictionaries; i++ ) {
00469         AO.Dictionaries[i]->gnumelements = AO.Dictionaries[i]->numelements;
00470     }
00471     if ( AM.NumSpectatorFiles > 0 ) {
00472         for ( i = 0; i < AM.SizeForSpectatorFiles; i++ ) {
00473             if ( AM.SpectatorFiles[i].name != 0 )
00474                 AM.SpectatorFiles[i].flags |= GLOBALSPECTATORFLAG;
00475         }
00476     }
00477 }
00478
00479 /*
00480     #] MakeGlobal :
00481     #[ TestDrop :
00482 */
00483
00484 void TestDrop(void)
00485 {
00486     EXPRESSIONS e;
00487     WORD j;
00488     for ( j = 0, e = Expressions; j < NumExpressions; j++, e++ ) {
00489         switch ( e->status ) {
00490             case SKIPLEXPRESSION:
00491                 e->status = LOCALEXPRESSION;
00492                 break;
00493             case UNHIDELEXPRESSION:
00494                 e->status = LOCALEXPRESSION;
00495                 ClearBracketIndex(j);
00496                 e->bracketinfo = e->newbracketinfo; e->newbracketinfo = 0;
00497                 break;
00498             case HIDELEXPRESSION:
00499                 e->status = HIDDENLEXPRESSION;
00500                 break;
00501             case SKIPGEXPRESSION:
00502                 e->status = GLOBALEXPRESSION;
00503                 break;
00504             case UNHIDEGEXPRESSION:
00505                 e->status = GLOBALEXPRESSION;
00506                 ClearBracketIndex(j);
00507                 e->bracketinfo = e->newbracketinfo; e->newbracketinfo = 0;
00508                 break;
00509             case HIDEGEXPRESSION:
00510                 e->status = HIDDENGEXPRESSION;
00511                 break;
00512             case DROPLEXPRESSION:
00513             case DROPGEXPRESSION:
00514             case DROPHLEXPRESSION:
00515             case DROPHGEXPRESSION:
00516             case DROPSPECTATOREXPRESSION:
00517                 e->status = DROPPEDEXPRESSION;
00518                 ClearBracketIndex(j);
00519                 e->bracketinfo = e->newbracketinfo; e->newbracketinfo = 0;
00520                 if ( e->replace >= 0 ) {

```

```

00521         Expressions[e->replace].replace = REGULAREXPRESSION;
00522         AC.exprnames->namenode[e->node].number = e->replace;
00523         e->replace = REGULAREXPRESSION;
00524     }
00525     else {
00526         AC.exprnames->namenode[e->node].type = CDELETE;
00527         AC.DidClean = 1;
00528     }
00529     break;
00530 case LOCALEXPRESSION:
00531 case GLOBALEXPRESSION:
00532     ClearBracketIndex(j);
00533     e->bracketinfo = e->newbracketinfo; e->newbracketinfo = 0;
00534     break;
00535 case HIDDENLEXPRESSION:
00536 case HIDDENGEXPRESSION:
00537     break;
00538 case INTOHIDELEXPRESSION:
00539     ClearBracketIndex(j);
00540     e->bracketinfo = e->newbracketinfo; e->newbracketinfo = 0;
00541     e->status = HIDDENLEXPRESSION;
00542     break;
00543 case INTOHIDEGEXPRESSION:
00544     ClearBracketIndex(j);
00545     e->bracketinfo = e->newbracketinfo; e->newbracketinfo = 0;
00546     e->status = HIDDENGEXPRESSION;
00547     break;
00548 default:
00549     ClearBracketIndex(j);
00550     e->bracketinfo = 0;
00551     break;
00552 }
00553 if ( e->replace == NEWLYDEFINEDEXPRESSION ) e->replace = REGULAREXPRESSION;
00554 }
00555 }
00556
00557 /*
00558     #] TestDrop :
00559     #[ PutInVflags :
00560 */
00561 void PutInVflags(WORD nexpr)
00562 {
00563     EXPRESSIONS e = Expressions + nexpr;
00564     POSITION *old;
00565     WORD *oldw;
00566     int i;
00567 restart:;
00568     if ( AS.OldOnFile == 0 ) {
00569         AS.NumOldOnFile = 20;
00570         AS.OldOnFile = (POSITION *)Malloc1(AS.NumOldOnFile*sizeof(POSITION),"file pointers");
00571     }
00572     else if ( nexpr >= AS.NumOldOnFile ) {
00573         old = AS.OldOnFile;
00574         AS.OldOnFile = (POSITION *)Malloc1(2*AS.NumOldOnFile*sizeof(POSITION),"file pointers");
00575         for ( i = 0; i < AS.NumOldOnFile; i++ ) AS.OldOnFile[i] = old[i];
00576         AS.NumOldOnFile = 2*AS.NumOldOnFile;
00577         M_free(old,"process file pointers");
00578     }
00579     if ( AS.OldNumFactors == 0 ) {
00580         AS.NumOldNumFactors = 20;
00581         AS.OldNumFactors = (WORD *)Malloc1(AS.NumOldNumFactors*sizeof(WORD),"numfactors pointers");
00582         AS.Oldvflags = (WORD *)Malloc1(AS.NumOldNumFactors*sizeof(WORD),"vflags pointers");
00583         AS.Olduflags = (WORD *)Malloc1(AS.NumOldNumFactors*sizeof(WORD),"uflags pointers");
00584     }
00585     else if ( nexpr >= AS.NumOldNumFactors ) {
00586         oldw = AS.OldNumFactors;
00587         AS.OldNumFactors = (WORD *)Malloc1(2*AS.NumOldNumFactors*sizeof(WORD),"numfactors pointers");
00588         for ( i = 0; i < AS.NumOldNumFactors; i++ ) AS.OldNumFactors[i] = oldw[i];
00589         M_free(oldw,"numfactors pointers");
00590         oldw = AS.Oldvflags;
00591         AS.Oldvflags = (WORD *)Malloc1(2*AS.NumOldNumFactors*sizeof(WORD),"vflags pointers");
00592         for ( i = 0; i < AS.NumOldNumFactors; i++ ) AS.Oldvflags[i] = oldw[i];
00593         M_free(oldw,"vflags pointers");
00594         oldw = AS.Olduflags;
00595         AS.Olduflags = (WORD *)Malloc1(2*AS.NumOldNumFactors*sizeof(WORD),"uflags pointers");
00596         for ( i = 0; i < AS.NumOldNumFactors; i++ ) AS.Olduflags[i] = oldw[i];
00597         M_free(oldw,"uflags pointers");
00598         AS.NumOldNumFactors = 2*AS.NumOldNumFactors;
00599     }
00600 }
00601 /*
00602     The next is needed when we Load a .sav file with lots of expressions.
00603 */
00604 if ( nexpr >= AS.NumOldOnFile || nexpr >= AS.NumOldNumFactors ) goto restart;
00605 AS.OldOnFile[nexpr] = e->onfile;
00606 AS.OldNumFactors[nexpr] = e->numfactors;
00607 AS.Oldvflags[nexpr] = e->vflags;

```

```

00608     AS.Olduflags[nexpr] = e->uflags;
00609 }
00610
00611 /*
00612     #] PutInVflags :
00613     #[ DoExecute :
00614 */
00615
00616 int DoExecute(WORD par, WORD skip)
00617 {
00618     GETIDENTITY
00619     int RetCode = 0;
00620     int i, oldmultithreaded = AS.MultiThreaded;
00621 #ifdef PARALLELCODE
00622     int j;
00623 #endif
00624
00625     SpecialCleanup(BHEAD0);
00626     if ( skip ) goto skipexec;
00627     if ( AC.IfLevel > 0 ) {
00628         MesPrint(" %d endif statement(s) missing",AC.IfLevel);
00629         RetCode = 1;
00630     }
00631     if ( AC.WhileLevel > 0 ) {
00632         MesPrint(" %d endwhile statement(s) missing",AC.WhileLevel);
00633         RetCode = 1;
00634     }
00635     if ( AC.arglevel > 0 ) {
00636         MesPrint(" %d endargument statement(s) missing",AC.arglevel);
00637         RetCode = 1;
00638     }
00639     if ( AC.termlevel > 0 ) {
00640         MesPrint(" %d endterm statement(s) missing",AC.termlevel);
00641         RetCode = 1;
00642     }
00643     if ( AC.insidelevel > 0 ) {
00644         MesPrint(" %d endinside statement(s) missing",AC.insidelevel);
00645         RetCode = 1;
00646     }
00647     if ( AC.inexprlevel > 0 ) {
00648         MesPrint(" %d endinexpression statement(s) missing",AC.inexprlevel);
00649         RetCode = 1;
00650     }
00651     if ( AC.NumLabels > 0 ) {
00652         for ( i = 0; i < AC.NumLabels; i++ ) {
00653             if ( AC.Labels[i] < 0 ) {
00654                 MesPrint(" -->Label %s missing",AC.LabelNames[i]);
00655                 RetCode = 1;
00656             }
00657         }
00658     }
00659     if ( AC.SwitchLevel > 0 ) {
00660         MesPrint(" %d endswitch statement(s) missing",AC.SwitchLevel);
00661         RetCode = 1;
00662     }
00663     if ( AC.dolooplevel > 0 ) {
00664         MesPrint(" %d enddo statement(s) missing",AC.dolooplevel);
00665         RetCode = 1;
00666     }
00667     if ( AP.OpenDictionary > 0 ) {
00668         MesPrint(" Dictionary %s has not been closed.",
00669             AO.Dictionaries[AP.OpenDictionary-1]->name);
00670         AP.OpenDictionary = 0;
00671         RetCode = 1;
00672     }
00673     if ( RetCode ) return(RetCode);
00674     AR.Cnumlhs = cbuf[AM.rbufnum].numlhs;
00675
00676     if ( ( AS.ExecMode = par ) == GLOBALMODULE ) AS.ExecMode = 0;
00677 #ifdef PARALLELCODE
00678 /*
00679     Now check whether we have either the regular parallel flag or the
00680     mparallel flag set.
00681     Next check whether any of the expressions has partodo set.
00682     If any of the above we need to check what the dollar status is.
00683 */
00684     AC.partodoflag = -1;
00685     if ( NumPotModdollars >= 0 ) {
00686         for ( i = 0; i < NumExpressions; i++ ) {
00687             if ( Expressions[i].partodo ) { AC.partodoflag = 1; break; }
00688         }
00689     }
00690 #ifdef WITHMPI
00691     if ( AC.partodoflag > 0 && PF.numtasks < 3 ) {
00692         AC.partodoflag = 0;
00693     }
00694 #endif

```

```

00695     if ( AC.partodoflag > 0 || ( NumPotModdollars > 0 && AC.mparallelflag == PARALLELFLAG ) ) {
00696     if ( NumPotModdollars > NumModOptdollars ) {
00697         AC.mparallelflag |= NOPARALLEL_DOLLAR;
00698 #ifdef WITHPTHREADS
00699         AS.MultiThreaded = 0;
00700 #endif
00701         AC.partodoflag = 0;
00702     }
00703     else {
00704         for ( i = 0; i < NumPotModdollars; i++ ) {
00705             for ( j = 0; j < NumModOptdollars; j++ ) {
00706                 if ( PotModdollars[i] == ModOptdollars[j].number ) break;
00707                 if ( j >= NumModOptdollars ) {
00708                     AC.mparallelflag |= NOPARALLEL_DOLLAR;
00709 #ifdef WITHPTHREADS
00710                     AS.MultiThreaded = 0;
00711 #endif
00712                     AC.partodoflag = 0;
00713                     break;
00714                 }
00715                 switch ( ModOptdollars[j].type ) {
00716                     case MODSUM:
00717                     case MODMAX:
00718                     case MODMIN:
00719                     case MODLOCAL:
00720                         break;
00721                     default:
00722                         AC.mparallelflag |= NOPARALLEL_DOLLAR;
00723                         AS.MultiThreaded = 0;
00724                         AC.partodoflag = 0;
00725                         break;
00726                 }
00727             }
00728         }
00729     }
00730     else if ( ( AC.mparallelflag & NOPARALLEL_USER ) != 0 ) {
00731 #ifdef WITHPTHREADS
00732         AS.MultiThreaded = 0;
00733 #endif
00734         AC.partodoflag = 0;
00735     }
00736     if ( AC.partodoflag == 0 ) {
00737         for ( i = 0; i < NumExpressions; i++ ) {
00738             Expressions[i].partodo = 0;
00739         }
00740     }
00741     else if ( AC.partodoflag == -1 ) {
00742         AC.partodoflag = 0;
00743     }
00744 #endif
00745 #ifdef WITHMPI
00746 /*
00747  * Check RHS expressions.
00748  */
00749 if ( AC.RhsExprInModuleFlag && (AC.mparallelflag == PARALLELFLAG || AC.partodoflag) ) {
00750     if ( PF.rhsInParallel ) {
00751         PF.mkSlaveInfile=1;
00752         if (PF.me != MASTER) {
00753             PF.slavebuf.PObuffer=(WORD *)Malloc1(AM.ScratSize*sizeof(WORD),"PF inbuf");
00754             PF.slavebuf.POsize=AM.ScratSize*sizeof(WORD);
00755             PF.slavebuf.POfull = PF.slavebuf.POfill = PF.slavebuf.PObuffer;
00756             PF.slavebuf.POstop= PF.slavebuf.PObuffer+AM.ScratSize;
00757             PUTZERO(PF.slavebuf.POposition);
00758             /*if (PF.me != MASTER)*/
00759         }
00760         else {
00761             AC.mparallelflag |= NOPARALLEL_RHS;
00762             AC.partodoflag = 0;
00763             for ( i = 0; i < NumExpressions; i++ ) {
00764                 Expressions[i].partodo = 0;
00765             }
00766         }
00767     }
00768 /*
00769  * Set $-variables with MODSUM to zero on the slaves.
00770  */
00771 if ( (AC.mparallelflag == PARALLELFLAG || AC.partodoflag) && PF.me != MASTER ) {
00772     for ( i = 0; i < NumModOptdollars; i++ ) {
00773         if ( ModOptdollars[i].type == MODSUM ) {
00774             DOLLARS d = Dollars + ModOptdollars[i].number;
00775             d->type = DOLZERO;
00776             if ( d->where && d->where != &AM.dollarzero ) M_free(d->where, "old content of
dollar");
00777             d->where = &AM.dollarzero;
00778             d->size = 0;
00779             CleanDollarFactors(d);
00780         }

```

```

00781     }
00782 }
00783 #endif
00784 AR.SortType = AC.SortType;
00785 #ifdef WITHMPI
00786     if ( PF.me == MASTER )
00787 #endif
00788 {
00789     if ( AC.SetupFlag ) WriteSetup();
00790     if ( AC.NamesFlag || AC.CodesFlag ) WriteLists();
00791 }
00792 if ( par == GLOBALMODULE ) MakeGlobal();
00793 if ( RevertScratch() ) return(-1);
00794 if ( AC.ncmod ) SetMods();
00795 /*
00796     Warn if the module has to run in sequential mode due to some problems.
00797 */
00798 #ifdef WITHMPI
00799     if ( PF.me == MASTER )
00800 #endif
00801 {
00802     if ( !AC.ThreadsFlag || AC.mparallelflag & NOPARALLEL_USER ) {
00803         /* The user switched off the parallel execution explicitly. */
00804     }
00805     else if ( AC.mparallelflag & NOPARALLEL_DOLLAR ) {
00806         if ( AC.WarnFlag >= 1 ) { /* Warning */
00807             int i, j, k, n;
00808             UBYTE *s, *s1;
00809             s = strdup((UBYTE *)"", "NOPARALLEL_DOLLAR s");
00810             n = 0;
00811             j = NumPotModdollars;
00812             for ( i = 0; i < j; i++ ) {
00813                 for ( k = 0; k < NumModOptdollars; k++ )
00814                     if ( ModOptdollars[k].number == PotModdollars[i] ) break;
00815                 if ( k >= NumModOptdollars ) {
00816                     /* global $-variable */
00817                     if ( n > 0 )
00818                         s = AddToString(s, (UBYTE *)", ", 0);
00819                     s = AddToString(s, (UBYTE *)"$", 0);
00820                     s = AddToString(s, DOLLARNAME(Dollars, PotModdollars[i]), 0);
00821                     n++;
00822                 }
00823             }
00824             s1 = strdup((UBYTE *)"This module is forced to run in sequential mode due to
00825 $-variable", "NOPARALLEL_DOLLAR s1");
00826             if ( n != 1 )
00827                 s1 = AddToString(s1, (UBYTE *)"s", 0);
00828             s1 = AddToString(s1, (UBYTE *)": ", 0);
00829             s1 = AddToString(s1, s, 0);
00830             Warning((char *)s1);
00831             M_free(s, "NOPARALLEL_DOLLAR s");
00832             M_free(s1, "NOPARALLEL_DOLLAR s1");
00833         }
00834         else if ( AC.mparallelflag & NOPARALLEL_RHS ) {
00835             HighWarning("This module is forced to run in sequential mode due to RHS expression
00836 names");
00837         }
00838         else if ( AC.mparallelflag & NOPARALLEL_CONVPOLY ) {
00839             HighWarning("This module is forced to run in sequential mode due to conversion to extra
00840 symbols");
00841         }
00842         else if ( AC.mparallelflag & NOPARALLEL_SPECTATOR ) {
00843             HighWarning("This module is forced to run in sequential mode due to
00844 tospectator/copspectator");
00845         }
00846         else if ( AC.mparallelflag & NOPARALLEL_TBLDOLLAR ) {
00847             HighWarning("This module is forced to run in sequential mode due to $-variable assignments
00848 in tables");
00849         }
00850         else if ( AC.mparallelflag & NOPARALLEL_NPROC ) {
00851             HighWarning("This module is forced to run in sequential mode because there is only one
00852 processor");
00853         }
00854     }
00855 }
00856 /*
00857     Now the actual execution
00858 */
00859 #ifdef WITHMPI
00860     /*
00861         * Turn on AS.printflag to print runtime errors occurring on slaves.
00862         */
00863     AS.printflag = 1;
00864 #endif
00865 if ( AP.preError == 0 && ( Processor() || WriteAll() ) ) RetCode = -1;
00866 #ifdef WITHMPI
00867     AS.printflag = 0;

```

```

00862 #endif
00863 /*
00864     That was it. Next is cleanup.
00865 */
00866     if ( AC.ncmod ) UnSetMods();
00867     AS.MultiThreaded = oldmultithreaded;
00868     TableReset();
00869
00870 /*[28sep2005 mt]:*/
00871 #ifdef WITHMPI
00872     /* Combine and then broadcast modified dollar variables. */
00873     if ( NumPotModdollars > 0 ) {
00874         RetCode = PF_CollectModifiedDollars();
00875         if ( RetCode ) return RetCode;
00876         RetCode = PF_BroadcastModifiedDollars();
00877         if ( RetCode ) return RetCode;
00878     }
00879     /* Broadcast the list of objects converted to symbols in AM.sbufnum. */
00880     if ( AC.topolynomialflag & TOPOLYNOMIALFLAG ) {
00881         RetCode = PF_BroadcastCBuf(AM.sbufnum);
00882         if ( RetCode ) return RetCode;
00883     }
00884     /*
00885     * Broadcast AR.expflags, which may be used on the slaves in the next module
00886     * via ZERO_ or UNCHANGED_. It also broadcasts several flags of each expression.
00887     */
00888     RetCode = PF_BroadcastExpFlags();
00889     if ( RetCode ) return RetCode;
00890     /*
00891     * Clean the hide file on the slaves, which was used for RHS expressions
00892     * broadcast from the master at the beginning of the module.
00893     */
00894     if ( PF.me != MASTER && AR.hidefile->PObuffer ) {
00895         if ( AR.hidefile->handle >= 0 ) {
00896             CloseFile(AR.hidefile->handle);
00897             AR.hidefile->handle = -1;
00898             remove(AR.hidefile->name);
00899         }
00900         AR.hidefile->POfull = AR.hidefile->POfill = AR.hidefile->PObuffer;
00901         PUTZERO(AR.hidefile->POposition);
00902     }
00903 #endif
00904 #ifdef WITHPTHREADS
00905     for ( j = 0; j < NumModOptdollars; j++ ) {
00906         if ( ModOptdollars[j].dstruct ) {
00907             /*
00908                 First clean up dollar values.
00909             */
00910             for ( i = 0; i < AM.totalnumberofthreads; i++ ) {
00911                 if ( ModOptdollars[j].dstruct[i].size > 0 ) {
00912                     CleanDollarFactors(&(ModOptdollars[j].dstruct[i]));
00913                     M_free(ModOptdollars[j].dstruct[i].where,"Local dollar value");
00914                 }
00915             }
00916             /*
00917                 Now clean up the whole array.
00918             */
00919             M_free(ModOptdollars[j].dstruct,"Local DOLLARS");
00920             ModOptdollars[j].dstruct = 0;
00921         }
00922     }
00923 #endif
00924 /*:[28sep2005 mt]:*/
00925
00926 /*
00927     @@@@@@@@@@@@@@@@@@
00928     Now follows the code to invalidate caches for all objects in the
00929     PotModdollars. There are NumPotModdollars of them and PotModdollars
00930     is an array of WORD.
00931 */
00932 /*
00933     Cleanup:
00934 */
00935 #ifdef JV_IS_WRONG
00936 /*
00937     Giving back this memory gives way too much activity with Malloc1
00938     Better to keep it and just put the number of used objects to zero (JV)
00939     If you put the lijst equal to NULL, please also make maxnum = 0
00940 */
00941     if ( ModOptdollars ) M_free(ModOptdollars, "ModOptdollars pointer");
00942     if ( PotModdollars ) M_free(PotModdollars, "PotModdollars pointer");
00943
00944     /* ModOptdollars changed to AC.ModOptDolList.lijst because AIX C compiler complained. MF
00945     30/07/2003. */
00946     AC.ModOptDolList.lijst = NULL;
00947     /* PotModdollars changed to AC.PotModDolList.lijst because AIX C compiler complained. MF
00948     30/07/2003. */

```

```

00947     AC.PotModDolList.lijst = NULL;
00948 #endif
00949     NumPotModdollars = 0;
00950     NumModOptdollars = 0;
00951
00952 skipexec:
00953 /*
00954     Clean up the switch information.
00955     We keep the switch array and heap.
00956 */
00957 if ( AC.SwitchInArray > 0 ) {
00958     for ( i = 0; i < AC.SwitchInArray; i++ ) {
00959         SWITCH *sw = AC.SwitchArray + i;
00960         if ( sw->table ) M_free(sw->table,"Switch table");
00961         sw->table = 0;
00962         sw->defaultcase.ncase = 0;
00963         sw->defaultcase.value = 0;
00964         sw->defaultcase.compbuffer = 0;
00965         sw->endswitch.ncase = 0;
00966         sw->endswitch.value = 0;
00967         sw->endswitch.compbuffer = 0;
00968         sw->typetable = 0;
00969         sw->maxcase = 0;
00970         sw->mincase = 0;
00971         sw->numcases = 0;
00972         sw->tablesize = 0;
00973         sw->caseoffset = 0;
00974         sw->iflevel = 0;
00975         sw->whilelevel = 0;
00976         sw->nestingsum = 0;
00977     }
00978     AC.SwitchInArray = 0;
00979     AC.SwitchLevel = 0;
00980 }
00981 #ifdef PARALLELCODE
00982     AC.numpfirstnum = 0;
00983 #endif
00984     AC.DidClean = 0;
00985     AC.PolyRatFunChanged = 0;
00986     TestDrop();
00987     if ( par == STOREMODULE || par == CLEARMODULE ) {
00988         ClearOptimize();
00989         if ( par == STOREMODULE && PopVariables() ) RetCode = -1;
00990         if ( AR.infile->handle >= 0 ) {
00991             CloseFile(AR.infile->handle);
00992             remove(AR.infile->name);
00993             AR.infile->handle = -1;
00994         }
00995         AR.infile->POfill = AR.infile->PObuffer;
00996         PUTZERO(AR.infile->POposition);
00997         AR.infile->POfull = AR.infile->PObuffer;
00998         if ( AR.outfile->handle >= 0 ) {
00999             CloseFile(AR.outfile->handle);
01000             remove(AR.outfile->name);
01001             AR.outfile->handle = -1;
01002         }
01003         AR.outfile->POfull =
01004         AR.outfile->POfill = AR.outfile->PObuffer;
01005         PUTZERO(AR.outfile->POposition);
01006         if ( AR.hidefile->handle >= 0 ) {
01007             CloseFile(AR.hidefile->handle);
01008             remove(AR.hidefile->name);
01009             AR.hidefile->handle = -1;
01010         }
01011         AR.hidefile->POfull =
01012         AR.hidefile->POfill = AR.hidefile->PObuffer;
01013         PUTZERO(AR.hidefile->POposition);
01014         AC.HideLevel = 0;
01015         if ( par == CLEARMODULE ) {
01016             if ( DeleteStore(0) < 0 ) {
01017                 MesPrint("Cannot restart the storage file");
01018                 RetCode = -1;
01019             }
01020             else RetCode = 0;
01021             CleanUp(1);
01022             ResetVariables(2);
01023             AM.gProcessBucketSize = AM.hProcessBucketSize;
01024             AM.gparallelflag = PARALLELFLAG;
01025             AM.gnumextrasyms = AM.ggnumextrasyms;
01026             PruneExtraSymbols(AM.ggnumextrasyms);
01027             IniVars();
01028         }
01029         ClearSpectators(par);
01030     }
01031     else {
01032         if ( CleanExpr(0) ) RetCode = -1;
01033         if ( AC.DidClean ) CompactifyTree(AC.exprnames,EXPRNAMES);

```

```

01034     ResetVariables(0);
01035     CleanupSort(-1);
01036 }
01037 clearcbuf(AC.cbunum);
01038 if ( AC.MultiBracketBuf != 0 ) {
01039     for ( i = 0; i < MAXMULTIBRACKETLEVELS; i++ ) {
01040         if ( AC.MultiBracketBuf[i] ) {
01041             M_free(AC.MultiBracketBuf[i], "bracket buffer i");
01042             AC.MultiBracketBuf[i] = 0;
01043         }
01044     }
01045     AC.MultiBracketLevels = 0;
01046     M_free(AC.MultiBracketBuf, "multi bracket buffer");
01047     AC.MultiBracketBuf = 0;
01048 }
01049
01050 if ( AC.SortReallocateFlag ) {
01051     /* Reallocate the sort buffers to reduce resident set usage */
01052     /* AT.SS is the same as AT.S0 here */
01053     SORTING* S = AT.S0;
01054     M_free(S->lBuffer, "SortReallocate lBuffer+sBuffer");
01055     S->lBuffer = Malloc1(sizeof(*(S->lBuffer))*(S->LargeSize+S->SmallEsize), "SortReallocate
lBuffer+sBuffer");
01056     S->lTop = S->lBuffer+S->LargeSize;
01057     S->sBuffer = S->lTop;
01058     if ( S->LargeSize == 0 ) { S->lBuffer = 0; S->lTop = 0; }
01059     S->sTop = S->sBuffer + S->SmallSize;
01060     S->sTop2 = S->sBuffer + S->SmallEsize;
01061     S->sHalf = S->sBuffer + (LONG)((S->SmallSize+S->SmallEsize)>>1);
01062
01063 #ifdef WITHPTHREADS
01064     /* We have to re-set the pointers into master lBuffer in the SortBlocks */
01065     UpdateSortBlocks(AM.totalnumberofthreads-1);
01066
01067     /* The SortBots do not have a real sort buffer to reallocate. */
01068     /* AB[0] has been reallocated above already. */
01069     for ( i = 1; i < AM.totalnumberofthreads; i++ ) {
01070         SORTING* S = AB[i]->T.S0;
01071         M_free(S->lBuffer, "SortReallocate lBuffer+sBuffer");
01072         S->lBuffer = Malloc1(sizeof(*(S->lBuffer))*(S->LargeSize+S->SmallEsize), "SortReallocate
lBuffer+sBuffer");
01073         S->lTop = S->lBuffer+S->LargeSize;
01074         S->sBuffer = S->lTop;
01075         if ( S->LargeSize == 0 ) { S->lBuffer = 0; S->lTop = 0; }
01076         S->sTop = S->sBuffer + S->SmallSize;
01077         S->sTop2 = S->sBuffer + S->SmallEsize;
01078         S->sHalf = S->sBuffer + (LONG)((S->SmallSize+S->SmallEsize)>>1);
01079     }
01080 #endif
01081 }
01082 if ( AC.SortReallocateFlag == 2 ) {
01083     /* The Flag was set for a single module by the preprocessor #sortreallocate,
01084        so turn it off again. */
01085     AC.SortReallocateFlag = 0;
01086 }
01087
01088 return(RetCode);
01089 }
01090
01091 /*
01092     #] DoExecute :
01093     #[ PutBracket :
01094
01095     Routine uses the bracket info to split a term into two pieces:
01096     1: the part outside the bracket, and
01097     2: the part inside the bracket.
01098     These parts are separated by a subterm of type HAAKJE.
01099     This subterm looks like: HAAKJE,3,level
01100     The level is used for nestings of brackets. The print routines
01101     cannot handle this yet (31-Mar-1988).
01102
01103     The Bracket selector is in AT.BrackBuf in the form of a regular term,
01104     but without coefficient.
01105     When AR.BracketOn < 0 we have a so-called antibracket. The main effect
01106     is an exchange of the inner and outer part and where the coefficient goes.
01107
01108     Routine recoded to facilitate b p1,p2; etc for dotproducts and tensors
01109     15-oct-1991
01110 */
01111
01112 int PutBracket(PHEAD WORD *termin)
01113 {
01114     GETBIDENTITY
01115     WORD *t, *t1, *b, i, j, *lastfun;
01116     WORD *t2, *s1, *s2;
01117     WORD *bStop, *bb, *bf, *tStop;
01118     WORD *term1, *term2, *m1, *m2, *tStopa;

```



```

01119     WORD *bbb = 0, *bind, *binst = 0, bwild = 0, *bss = 0, *bns = 0, bset = 0;
01120     term1 = AT.WorkPointer+1;
01121     term2 = (WORD *) ((UBYTE *) (term1)) + AM.MaxTer);
01122     if ( ( (WORD *) ((UBYTE *) (term2)) + AM.MaxTer ) > AT.WorkTop ) {
01123         MesWork();
01124     }
01125     if ( AR.BracketOn < 0 ) {
01126         t2 = term1; t1 = term2;      /* AntiBracket */
01127     }
01128     else {
01129         t1 = term1; t2 = term2;      /* Regular bracket */
01130     }
01131     b = AT.BrackBuf; bStop = b+b; b++;
01132     while ( b < bStop ) {
01133         if ( *b == INDEX ) { bwild = 1; bbb = b+2; binst = b + b[1]; }
01134         if ( *b == SETSET ) { bset = 1; bss = b+2; bns = b + b[1]; }
01135         b += b[1];
01136     }
01137
01138     t = termin; tStopa = t + *t; i = *(t + *t -1); i = ABS(i);
01139     if ( AR.PolyFun && AT.PolyAct ) tStop = termin + AT.PolyAct;
01140 #ifdef WITHFLOAT
01141     else if ( AT.FloatPos ) tStop = termin + AT.FloatPos;
01142 #endif
01143     else tStop = tStopa - i;
01144     t++;
01145     if ( AR.BracketOn < 0 ) {
01146         lastfun = 0;
01147         while ( t < tStop && *t >= FUNCTION
01148             && functions[*t-FUNCTION].commute ) {
01149             b = AT.BrackBuf+1;
01150             while ( b < bStop ) {
01151                 if ( *b == *t ) {
01152                     lastfun = t;
01153                     while ( t < tStop && *t >= FUNCTION
01154                         && functions[*t-FUNCTION].commute ) t += t[1];
01155                     goto NextNcom1;
01156                 }
01157                 b += b[1];
01158             }
01159             if ( bset ) {
01160                 b = bss;
01161                 while ( b < bns ) {
01162                     if ( b[1] == CFUNCTION ) { /* Set of functions */
01163                         SETS set = Sets+b[0]; WORD i;
01164                         for ( i = set->first; i < set->last; i++ ) {
01165                             if ( SetElements[i] == *t ) {
01166                                 lastfun = t;
01167                                 while ( t < tStop && *t >= FUNCTION
01168                                     && functions[*t-FUNCTION].commute ) t += t[1];
01169                                 goto NextNcom1;
01170                             }
01171                         }
01172                     }
01173                     b += 2;
01174                 }
01175             }
01176             if ( bwild && *t >= FUNCTION && functions[*t-FUNCTION].spec ) {
01177                 s1 = t + t[1];
01178                 s2 = t + FUNHEAD;
01179                 while ( s2 < s1 ) {
01180                     bind = bbb;
01181                     while ( bind < binst ) {
01182                         if ( *bind == *s2 ) {
01183                             lastfun = t;
01184                             while ( t < tStop && *t >= FUNCTION
01185                                 && functions[*t-FUNCTION].commute ) t += t[1];
01186                             goto NextNcom1;
01187                         }
01188                         bind++;
01189                     }
01190                     s2++;
01191                 }
01192             }
01193             t += t[1];
01194         }
01195     NextNcom1:
01196     s1 = termin + 1;
01197     if ( lastfun ) {
01198         while ( s1 < lastfun ) *t2++ = *s1++;
01199         while ( s1 < t ) *t1++ = *s1++;
01200     }
01201     else {
01202         while ( s1 < t ) *t2++ = *s1++;
01203     }
01204
01205 }

```

```

01206     else {
01207         lastfun = t;
01208         while ( t < tStop && *t >= FUNCTION
01209             && functions[*t-FUNCTION].commute ) {
01210             b = AT.BrackBuf+1;
01211             while ( b < bStop ) {
01212                 if ( *b == *t ) { lastfun = t + t[1]; goto NextNcom; }
01213                 b += b[1];
01214             }
01215             if ( bset ) {
01216                 b = bss;
01217                 while ( b < bns ) {
01218                     if ( b[1] == CFUNCTION ) { /* Set of functions */
01219                         SETS set = Sets+b[0]; WORD i;
01220                         for ( i = set->first; i < set->last; i++ ) {
01221                             if ( SetElements[i] == *t ) {
01222                                 lastfun = t + t[1];
01223                                 goto NextNcom;
01224                             }
01225                         }
01226                     }
01227                     b += 2;
01228                 }
01229             }
01230             if ( bwild && *t >= FUNCTION && functions[*t-FUNCTION].spec ) {
01231                 s1 = t + t[1];
01232                 s2 = t + FUNHEAD;
01233                 while ( s2 < s1 ) {
01234                     bind = bbb;
01235                     while ( bind < binst ) {
01236                         if ( *bind == *s2 ) { lastfun = t + t[1]; goto NextNcom; }
01237                         bind++;
01238                     }
01239                     s2++;
01240                 }
01241             }
01242 NextNcom:
01243             t += t[1];
01244         }
01245         s1 = termin + 1;
01246         while ( s1 < lastfun ) *t1++ = *s1++;
01247         while ( s1 < t ) *t2++ = *s1++;
01248     }
01249     /*
01250     Now we have only commuting functions left. Move the b pointer to them.
01251     */
01252     b = AT.BrackBuf + 1;
01253     while ( b < bStop && *b >= FUNCTION
01254         && ( *b < FUNCTION || functions[*b-FUNCTION].commute ) ) {
01255         b += b[1];
01256     }
01257     bf = b;
01258
01259     while ( t < tStop && ( bf < bStop || bwild || bset ) ) {
01260         b = bf;
01261         while ( b < bStop && *b != *t ) { b += b[1]; }
01262         i = t[1];
01263         if ( *t >= FUNCTION ) { /* We are in function territory */
01264             if ( b < bStop && *b == *t ) goto FunBrac;
01265             if ( bset ) {
01266                 b = bss;
01267                 while ( b < bns ) {
01268                     if ( b[1] == CFUNCTION ) { /* Set of functions */
01269                         SETS set = Sets+b[0]; WORD i;
01270                         for ( i = set->first; i < set->last; i++ ) {
01271                             if ( SetElements[i] == *t ) goto FunBrac;
01272                         }
01273                     }
01274                     b += 2;
01275                 }
01276             }
01277             if ( bwild && *t >= FUNCTION && functions[*t-FUNCTION].spec ) {
01278                 s1 = t + t[1];
01279                 s2 = t + FUNHEAD;
01280                 while ( s2 < s1 ) {
01281                     bind = bbb;
01282                     while ( bind < binst ) {
01283                         if ( *bind == *s2 ) goto FunBrac;
01284                         bind++;
01285                     }
01286                     s2++;
01287                 }
01288             }
01289             NCOPY(t2,t,i);
01290             continue;
01291 FunBrac:
01292             NCOPY(t1,t,i);
01293             continue;

```

```

01293     }
01294     /*
01295     We have left: DELTA, INDEX, VECTOR, DOTPRODUCT, SYMBOL
01296     */
01297     if ( *t == DELTA ) {
01298         if ( b < bStop && *b == DELTA ) {
01299             b += b[1];
01300             NCOPY(t1,t,i);
01301         }
01302         else { NCOPY(t2,t,i); }
01303     }
01304     else if ( *t == INDEX ) {
01305         if ( bwild ) {
01306             m1 = t1; m2 = t2;
01307             *t1++ = *t; t1++; *t2++ = *t; t2++;
01308             bind = bbb;
01309             j = t[1] - 2;
01310             t += 2;
01311             while ( --j >= 0 ) {
01312                 while ( *bind < *t && bind < binst ) bind++;
01313                 if ( *bind == *t && bind < binst ) {
01314                     *t1++ = *t++;
01315                 }
01316                 else if ( bset ) {
01317                     WORD *b3 = bss;
01318                     while ( b3 < bns ) {
01319                         if ( b3[1] == CVECTOR ) {
01320                             SETS set = Sets+b3[0]; WORD i;
01321                             for ( i = set->first; i < set->last; i++ ) {
01322                                 if ( SetElements[i] == *t ) {
01323                                     *t1++ = *t++;
01324                                     goto nextind;
01325                                 }
01326                             }
01327                         }
01328                         b3 += 2;
01329                     }
01330                     *t2++ = *t++;
01331                 }
01332                 else *t2++ = *t++;
01333             nextind;;
01334             }
01335             m1[1] = WORDDIF(t1,m1);
01336             if ( m1[1] == 2 ) t1 = m1;
01337             m2[1] = WORDDIF(t2,m2);
01338             if ( m2[1] == 2 ) t2 = m2;
01339         }
01340         else if ( bset ) {
01341             m1 = t1; m2 = t2;
01342             *t1++ = *t; t1++; *t2++ = *t; t2++;
01343             j = t[1] - 2;
01344             t += 2;
01345             while ( --j >= 0 ) {
01346                 WORD *b3 = bss;
01347                 while ( b3 < bns ) {
01348                     if ( b3[1] == CVECTOR ) {
01349                         SETS set = Sets+b3[0]; WORD i;
01350                         for ( i = set->first; i < set->last; i++ ) {
01351                             if ( SetElements[i] == *t ) {
01352                                 *t1++ = *t++;
01353                                 goto nextind2;
01354                             }
01355                         }
01356                     }
01357                     b3 += 2;
01358                 }
01359                 *t2++ = *t++;
01360             nextind2;;
01361             }
01362             m1[1] = WORDDIF(t1,m1);
01363             if ( m1[1] == 2 ) t1 = m1;
01364             m2[1] = WORDDIF(t2,m2);
01365             if ( m2[1] == 2 ) t2 = m2;
01366         }
01367         else {
01368             NCOPY(t2,t,i);
01369         }
01370     }
01371     else if ( *t == VECTOR ) {
01372         if ( ( b < bStop && *b == VECTOR ) || bwild ) {
01373             if ( b < bStop && *b == VECTOR ) {
01374                 bb = b + b[1]; b += 2;
01375             }
01376             else bb = b;
01377             j = t[1] - 2;
01378             m1 = t1; m2 = t2; *t1++ = *t; *t2++ = *t; t1++; t2++; t += 2;
01379             while ( j > 0 ) {

```

```

01380         j -= 2;
01381         while ( b < bb && ( *b < *t ||
01382         ( *b == *t && b[1] < t[1] ) ) ) b += 2;
01383         if ( b < bb && ( *t == *b && t[1] == b[1] ) ) {
01384             *t1++ = *t++; *t1++ = *t++; goto nextvec;
01385         }
01386         else if ( bwild ) {
01387             bind = bbb;
01388             while ( bind < binst ) {
01389                 if ( *t == *bind || t[1] == *bind ) {
01390                     *t1++ = *t++; *t1++ = *t++;
01391                     goto nextvec;
01392                 }
01393                 bind++;
01394             }
01395         }
01396         if ( bset ) {
01397             WORD *b3 = bss;
01398             while ( b3 < bns ) {
01399                 if ( b3[1] == CVECTOR ) {
01400                     SETS set = Sets+b3[0]; WORD i;
01401                     for ( i = set->first; i < set->last; i++ ) {
01402                         if ( SetElements[i] == *t ) {
01403                             *t1++ = *t++; *t1++ = *t++;
01404                             goto nextvec;
01405                         }
01406                     }
01407                 }
01408                 b3 += 2;
01409             }
01410         }
01411         *t2++ = *t++; *t2++ = *t++;
01412 nextvec;;
01413     }
01414     m1[1] = WORDDIF(t1,m1);
01415     if ( m1[1] == 2 ) t1 = m1;
01416     m2[1] = WORDDIF(t2,m2);
01417     if ( m2[1] == 2 ) t2 = m2;
01418 }
01419 else if ( bset ) {
01420     m1 = t1; *t1++ = *t; t1++;
01421     m2 = t2; *t2++ = *t; t2++;
01422     s2 = t + i; t += 2;
01423     while ( t < s2 ) {
01424         WORD *b3 = bss;
01425         while ( b3 < bns ) {
01426             if ( b3[1] == CVECTOR ) {
01427                 SETS set = Sets+b3[0]; WORD i;
01428                 for ( i = set->first; i < set->last; i++ ) {
01429                     if ( SetElements[i] == *t ) {
01430                         *t1++ = *t++; *t1++ = *t++;
01431                         goto nextvec2;
01432                     }
01433                 }
01434             }
01435             b3 += 2;
01436         }
01437         *t2++ = *t++; *t2++ = *t++;
01438 nextvec2;;
01439     }
01440     m1[1] = WORDDIF(t1,m1);
01441     if ( m1[1] == 2 ) t1 = m1;
01442     m2[1] = WORDDIF(t2,m2);
01443     if ( m2[1] == 2 ) t2 = m2;
01444 }
01445 else {
01446     NCOPY(t2,t,i);
01447 }
01448 }
01449 else if ( *t == DOTPRODUCT ) {
01450     if ( ( b < bStop && *b == *t ) || bwild ) {
01451         m1 = t1; *t1++ = *t; t1++;
01452         m2 = t2; *t2++ = *t; t2++;
01453         if ( b >= bStop || *b != *t ) { bb = b; s1 = b; }
01454         else {
01455             s1 = b + b[1]; bb = b + 2;
01456         }
01457         s2 = t + i; t += 2;
01458         while ( t < s2 && ( bb < s1 || bwild || bset ) ) {
01459             while ( bb < s1 && ( *bb < *t ||
01460             ( *bb == *t && bb[1] < t[1] ) ) ) bb += 3;
01461             if ( bb < s1 && *bb == *t && bb[1] == t[1] ) {
01462                 *t1++ = *t++; *t1++ = *t++; *t1++ = *t++; bb += 3;
01463                 goto nextdot;
01464             }
01465             else if ( bwild ) {
01466                 bind = bbb;

```

```

01467         while ( bind < binst ) {
01468             if ( *bind == *t || *bind == t[1] ) {
01469                 *t1++ = *t++; *t1++ = *t++; *t1++ = *t++;
01470                 goto nextdot;
01471             }
01472             bind++;
01473         }
01474     }
01475     if ( bset ) {
01476         WORD *b3 = bss;
01477         while ( b3 < bns ) {
01478             if ( b3[1] == CVECTOR ) {
01479                 SETS set = Sets+b3[0]; WORD i;
01480                 for ( i = set->first; i < set->last; i++ ) {
01481                     if ( SetElements[i] == *t || SetElements[i] == t[1] ) {
01482                         *t1++ = *t++; *t1++ = *t++; *t1++ = *t++;
01483                         goto nextdot;
01484                     }
01485                 }
01486             }
01487             b3 += 2;
01488         }
01489     }
01490     *t2++ = *t++; *t2++ = *t++; *t2++ = *t++;
01491 nextdot:;
01492     }
01493     while ( t < s2 ) *t2++ = *t++;
01494     m1[1] = WORDDIF(t1,m1);
01495     if ( m1[1] == 2 ) t1 = m1;
01496     m2[1] = WORDDIF(t2,m2);
01497     if ( m2[1] == 2 ) t2 = m2;
01498 }
01499 else if ( bset ) {
01500     m1 = t1; *t1++ = *t; t1++;
01501     m2 = t2; *t2++ = *t; t2++;
01502     s2 = t + i; t += 2;
01503     while ( t < s2 ) {
01504         WORD *b3 = bss;
01505         while ( b3 < bns ) {
01506             if ( b3[1] == CVECTOR ) {
01507                 SETS set = Sets+b3[0]; WORD i;
01508                 for ( i = set->first; i < set->last; i++ ) {
01509                     if ( SetElements[i] == *t || SetElements[i] == t[1] ) {
01510                         *t1++ = *t++; *t1++ = *t++; *t1++ = *t++;
01511                         goto nextdot2;
01512                     }
01513                 }
01514             }
01515             b3 += 2;
01516         }
01517         *t2++ = *t++; *t2++ = *t++; *t2++ = *t++;
01518 nextdot2:;
01519     }
01520     m1[1] = WORDDIF(t1,m1);
01521     if ( m1[1] == 2 ) t1 = m1;
01522     m2[1] = WORDDIF(t2,m2);
01523     if ( m2[1] == 2 ) t2 = m2;
01524 }
01525 else { NCOPY(t2,t,i); }
01526 }
01527 else if ( *t == SYMBOL ) {
01528     if ( b < bStop && *b == *t ) {
01529         m1 = t1; *t1++ = *t; t1++;
01530         m2 = t2; *t2++ = *t; t2++;
01531         s1 = b + b[1]; bb = b+2;
01532         s2 = t + i; t += 2;
01533         while ( bb < s1 && t < s2 ) {
01534             while ( bb < s1 && *bb < *t ) bb += 2;
01535             if ( bb >= s1 ) {
01536                 if ( bset ) goto TrySymbolSet;
01537                 break;
01538             }
01539             if ( *bb == *t ) { *t1++ = *t++; *t1++ = *t++; }
01540             else if ( bset ) {
01541                 WORD *bbb;
01542 TrySymbolSet:
01543                 bbb = bss;
01544                 while ( bbb < bns ) {
01545                     if ( bbb[1] == CSYMBOL ) { /* Set of symbols */
01546                         SETS set = Sets+bbb[0]; WORD i;
01547                         for ( i = set->first; i < set->last; i++ ) {
01548                             if ( SetElements[i] == *t ) {
01549                                 *t1++ = *t++; *t1++ = *t++;
01550                                 goto NextSymbol;
01551                             }
01552                         }
01553                     }

```

```

01554             bbb += 2;
01555         }
01556         *t2++ = *t++; *t2++ = *t++;
01557     }
01558     else { *t2++ = *t++; *t2++ = *t++; }
01559 NextSymbol::
01560     }
01561     while ( t < s2 ) *t2++ = *t++;
01562     m1[1] = WORDDIF(t1,m1);
01563     if ( m1[1] == 2 ) t1 = m1;
01564     m2[1] = WORDDIF(t2,m2);
01565     if ( m2[1] == 2 ) t2 = m2;
01566 }
01567 else if ( bset ) {
01568     WORD *bbb;
01569     m1 = t1; *t1++ = *t; t1++;
01570     m2 = t2; *t2++ = *t; t2++;
01571     s2 = t + i; t += 2;
01572     while ( t < s2 ) {
01573         bbb = bss;
01574         while ( bbb < bns ) {
01575             if ( bbb[1] == CSYMBOL ) { /* Set of symbols */
01576                 SETS set = Sets+bbb[0]; WORD i;
01577                 for ( i = set->first; i < set->last; i++ ) {
01578                     if ( SetElements[i] == *t ) {
01579                         *t1++ = *t++; *t1++ = *t++;
01580                         goto NextSymbol2;
01581                     }
01582                 }
01583             }
01584             bbb += 2;
01585         }
01586         *t2++ = *t++; *t2++ = *t++;
01587 NextSymbol2::
01588     }
01589     m1[1] = WORDDIF(t1,m1);
01590     if ( m1[1] == 2 ) t1 = m1;
01591     m2[1] = WORDDIF(t2,m2);
01592     if ( m2[1] == 2 ) t2 = m2;
01593 }
01594     else { NCOPY(t2,t,i); }
01595 }
01596 else {
01597     NCOPY(t2,t,i);
01598 }
01599 }
01600 if ( ( i = WORDDIF(tStop,t) ) > 0 ) NCOPY(t2,t,i);
01601 if ( AR.BracketOn < 0 ) {
01602     s1 = t1; t1 = t2; t2 = s1;
01603 }
01604 do { *t2++ = *t++; } while ( t < (WORD *)tStopa );
01605 t = AT.WorkPointer;
01606 i = WORDDIF(t1,term1);
01607 *t++ = 4 + i + WORDDIF(t2,term2);
01608 t += i;
01609 *t++ = HAAKJE;
01610 *t++ = 3;
01611 *t++ = 0; /* This feature won't be used for a while */
01612 i = WORDDIF(t2,term2);
01613 t1 = term2;
01614 if ( i > 0 ) NCOPY(t,t1,i);
01615
01616 AT.WorkPointer = t;
01617
01618 return(0);
01619 }
01620
01621 /*
01622     #] PutBracket :
01623     #[ SpecialCleanup :
01624 */
01625
01626 void SpecialCleanup(PHEAD0)
01627 {
01628     GETBIDENTITY
01629     if ( AT.previousEfactor ) M_free(AT.previousEfactor,"Efactor cache");
01630     AT.previousEfactor = 0;
01631 }
01632
01633 /*
01634     #] SpecialCleanup :
01635     #[ SetMods :
01636 */
01637
01638 #ifndef WITHPTHREADS
01639
01640 void SetMods(void)

```

```

01641 {
01642     int i, n;
01643     if ( AN.cmod != 0 ) M_free(AN.cmod,"AN.cmod");
01644     n = ABS(AN.ncmod);
01645     AN.cmod = (UWORD *)Malloc1(sizeof(WORD)*n,"AN.cmod");
01646     for ( i = 0; i < n; i++ ) AN.cmod[i] = AC.cmod[i];
01647 }
01648
01649 #endif
01650
01651 /*
01652     #] SetMods :
01653     #[ UnSetMods :
01654 */
01655
01656 #ifndef WITHPTHREADS
01657 void UnSetMods(void)
01658 {
01659     if ( AN.cmod != 0 ) M_free(AN.cmod,"AN.cmod");
01660     AN.cmod = 0;
01661 }
01662
01663 #endif
01664
01665 /*
01666     #] UnSetMods :
01667     #[ DoExecute :
01668     #[ Expressions :
01669     #[ ExchangeExpressions :
01670 */
01671
01672 void ExchangeExpressions(int num1, int num2)
01673 {
01674     GETIDENTITY
01675     WORD node1, node2, namesize, TMproto[SUBEXPSIZE];
01676     INDEXENTRY *ind;
01677     EXPRESSIONS e1, e2;
01678     LONG a;
01679     SBYTE *s1, *s2;
01680     int i;
01681     e1 = Expressions + num1;
01682     e2 = Expressions + num2;
01683     node1 = e1->node;
01684     node2 = e2->node;
01685     AC.exprnames->namenode[node1].number = num2;
01686     AC.exprnames->namenode[node2].number = num1;
01687     a = e1->name; e1->name = e2->name; e2->name = a;
01688     namesize = e1->namesize; e1->namesize = e2->namesize; e2->namesize = namesize;
01689     e1->node = node2;
01690     e2->node = node1;
01691     if ( e1->status == STOREDEXPRESSION ) {
01692         /*
01693             Find the name in the index and replace by the new name
01694         */
01695         TMproto[0] = EXPRESSION;
01696         TMproto[1] = SUBEXPSIZE;
01697         TMproto[2] = num1;
01698         TMproto[3] = 1;
01699         { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
01700         AT.TMaddr = TMproto;
01701         ind = FindInIndex(num1,&AR.StoreData,0,0);
01702         s1 = (SBYTE *) (AC.exprnames->namebuffer+e1->name);
01703         i = e1->namesize;
01704         s2 = ind->name;
01705         NCOPY(s2,s1,i);
01706         *s2 = 0;
01707         SeekFile(AR.StoreData.Handle,&(e1->onfile),SEEK_SET);
01708         if ( WriteFile(AR.StoreData.Handle,(UBYTE *)ind,
01709             (LONG)(sizeof(INDEXENTRY)) != sizeof(INDEXENTRY) ) {
01710             MesPrint("File error while exchanging expressions");
01711             Terminate(-1);
01712         }
01713         FlushFile(AR.StoreData.Handle);
01714     }
01715     if ( e2->status == STOREDEXPRESSION ) {
01716         /*
01717             Find the name in the index and replace by the new name
01718         */
01719         TMproto[0] = EXPRESSION;
01720         TMproto[1] = SUBEXPSIZE;
01721         TMproto[2] = num2;
01722         TMproto[3] = 1;
01723         { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
01724         AT.TMaddr = TMproto;
01725         ind = FindInIndex(num1,&AR.StoreData,0,0);
01726         s1 = (SBYTE *) (AC.exprnames->namebuffer+e2->name);
01727

```

```

01728         i = e2->namesize;
01729         s2 = ind->name;
01730         NCOPY(s2,s1,i);
01731         *s2 = 0;
01732         SeekFile(AR.StoreData.Handle,&(e2->onfile),SEEK_SET);
01733         if ( WriteFile(AR.StoreData.Handle,(UBYTE *)ind,
01734             (LONG)(sizeof(INDEXENTRY)) != sizeof(INDEXENTRY) ) {
01735             MesPrint("File error while exchanging expressions");
01736             Terminate(-1);
01737         }
01738         FlushFile(AR.StoreData.Handle);
01739     }
01740 }
01741
01742 /*
01743     #] ExchangeExpressions :
01744     #[ GetFirstBracket :
01745 */
01746
01747 int GetFirstBracket(WORD *term, int num)
01748 {
01749     /*
01750         Gets the first bracket of the expression 'num'
01751         Puts it in term. If no brackets the answer is one.
01752         Routine should be thread-safe
01753     */
01754     GETIDENTITY
01755     POSITION position, oldposition;
01756     RENUMBER renumber;
01757     FILEHANDLE *fi;
01758     WORD type, *oldcomppointer, oldonefile, numword;
01759     WORD *t, *tstop;
01760
01761     oldcomppointer = AR.CompressPointer;
01762     type = Expressions[num].status;
01763     if ( type == STOREDEXPRESSION ) {
01764         WORD Tmproto[SUBEXPSIZE];
01765         Tmproto[0] = EXPRESSION;
01766         Tmproto[1] = SUBEXPSIZE;
01767         Tmproto[2] = num;
01768         Tmproto[3] = 1;
01769         { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) Tmproto[ie] = 0; }
01770         AT.Tmaddr = Tmproto;
01771         PUTZERO(position);
01772         if ( ( renumber = GetTable(num,&position,0) ) == 0 ) {
01773             MesCall("GetFirstBracket");
01774             SETERROR(-1)
01775         }
01776         if ( GetFromStore(term,&position,renumber,&numword,num) < 0 ) {
01777             MesCall("GetFirstBracket");
01778             SETERROR(-1)
01779         }
01780     }
01781     #ifdef WITHPTHREADS
01782     /*
01783         if ( renumber->symb.lo != AN.dummyrenumlist )
01784             M_free(renumber->symb.lo,"VarSpace");
01785         M_free(renumber,"Renummer");
01786     */
01787     #endif
01788     /*
01789     }
01790     else {
01791         /* Active expression */
01792         oldonefile = AR.GetOneFile;
01793         if ( type == HIDDENLEXPRESSION || type == HIDDENEXPRESSION ) {
01794             AR.GetOneFile = 2; fi = AR.hidefile;
01795         }
01796         else {
01797             AR.GetOneFile = 0; fi = AR.infile;
01798         }
01799         if ( fi->handle >= 0 ) {
01800             PUTZERO(oldposition);
01801             SeekFile(fi->handle,&oldposition,SEEK_CUR);
01802         }
01803     }
01804     else {
01805         SETBASEPOSITION(oldposition,fi->POfill-fi->PObuffer);
01806     }
01807     position = AS.OldOnFile[num];
01808     if ( GetOneTerm(BHEAD term,fi,&position,1) < 0
01809         || ( GetOneTerm(BHEAD term,fi,&position,1) < 0 ) ) {
01810         MLOCK(ErrorMessageLock);
01811         MesCall("GetFirstBracket");
01812         MUNLOCK(ErrorMessageLock);
01813         SETERROR(-1)
01814     }

```



```

01815         if ( fi->handle >= 0 ) {
01816 /*
01817         SeekFile(fi->handle,&oldposition,SEEK_SET);
01818         if ( ISNEGPOS(oldposition) ) {
01819             MLOCK(ErrorMessageLock);
01820             MesPrint("File error");
01821             MUNLOCK(ErrorMessageLock);
01822             SETERROR(-1)
01823         }
01824 */
01825     }
01826     else {
01827         fi->POfill = fi->PObuffer+BASEPOSITION(oldposition);
01828     }
01829     AR.GetOneFile = oldonefile;
01830 }
01831 AR.CompressPointer = oldcomppointer;
01832 if ( *term ) {
01833     tstop = term + *term; tstop -= ABS(tstop[-1]);
01834     t = term + 1;
01835     while ( t < tstop ) {
01836         if ( *t == HAAKJE ) break;
01837         t += t[1];
01838     }
01839     if ( t >= tstop ) {
01840         term[0] = 4; term[1] = 1; term[2] = 1; term[3] = 3;
01841     }
01842     else {
01843         *t++ = 1; *t++ = 1; *t++ = 3; *term = t - term;
01844     }
01845 }
01846 else {
01847     term[0] = 4; term[1] = 1; term[2] = 1; term[3] = 3;
01848 }
01849 return(*term);
01850 }
01851
01852 /*
01853     #] GetFirstBracket :
01854     #[ GetFirstTerm :
01855 */
01856
01866 int GetFirstTerm(WORD *term, int num, int pre)
01867 {
01868     GETIDENTITY
01869     POSITION position, oldposition;
01870     RENUMBER renumber;
01871     FILEHANDLE *fi;
01872     WORD type, *oldcomppointer, oldonefile, numword;
01873
01874     oldcomppointer = AR.CompressPointer;
01875     type = Expressions[num].status;
01876     if ( type == STOREDEXPRESSION ) {
01877         WORD Tmproto[SUBEXPSIZE];
01878         Tmproto[0] = EXPRESSION;
01879         Tmproto[1] = SUBEXPSIZE;
01880         Tmproto[2] = num;
01881         Tmproto[3] = 1;
01882         { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) Tmproto[ie] = 0; }
01883         AT.TMaddr = Tmproto;
01884         PUTZERO(position);
01885         if ( ( renumber = GetTable(num,&position,0) ) == 0 ) {
01886             MesCall("GetFirstTerm");
01887             SETERROR(-1)
01888         }
01889         if ( GetFromStore(term,&position,renumber,&numword,num) < 0 ) {
01890             MesCall("GetFirstTerm");
01891             SETERROR(-1)
01892         }
01893     }
01894 /*
01895     #ifdef WITHPTHREADS
01896 */
01897     if ( renumber->symb.lo != AN.dummyrenumlist )
01898         M_free(renumber->symb.lo,"VarSpace");
01899     M_free(renumber,"Renumber");
01900 /*
01901 */
01902 }
01903 else {
01904     /* Active expression */
01905     oldonefile = AR.GetOneFile;
01906     if ( type == HIDDENLEXPRESSION || type == HIDDENEXPRESSSION ) {
01907         AR.GetOneFile = 2; fi = AR.hidefile;
01908     }
01909     else {
01910         AR.GetOneFile = 0;
01911         if ( Expressions[num].replace == NEWLYDEFINEDEXPRESSION ) {

```

```

01911         /* During execution, if the expression has already been processed it
01912            will be in the outfile. If it has not, the usage is illegal according
01913            to the manual, though no error is given. */
01914         if ( pre == 0 ) { fi = AR.outfile; }
01915         /* During preprocessing, the expression certainly has not been processed
01916            yet. Print an error and terminate. */
01917         else {
01918             MesPrint("&isnumerical: expression is not yet defined!");
01919             SETERROR(-1);
01920         }
01921     }
01922     else {
01923         /* During execution, we should use the definition as stored at the end
01924            of the previous module. This is in the infile. */
01925         if ( pre == 0 ) { fi = AR.infile; }
01926         /* During preprocessing, this function is called before the RevertScratch
01927            at the beginning of this module's execution. Thus the expressions are
01928            in the outfile of the previous module. */
01929         else { fi = AR.outfile; }
01930     }
01931 }
01932 if ( fi->handle >= 0 ) {
01933     PUTZERO(oldposition);
01934     /*
01935         SeekFile(fi->handle,&oldposition,SEEK_CUR);
01936     */
01937 }
01938 else {
01939     SETBASEPOSITION(oldposition,fi->POfill-fi->PObuffer);
01940 }
01941 position = AS.OldOnFile[num];
01942 if ( GetOneTerm(BHEAD term,fi,&position,1) < 0
01943 || ( GetOneTerm(BHEAD term,fi,&position,1) < 0 ) ) {
01944     MLOCK(ErrorMessageLock);
01945     MesCall("GetFirstTerm");
01946     MUNLOCK(ErrorMessageLock);
01947     SETERROR(-1)
01948 }
01949 if ( fi->handle >= 0 ) {
01950     /*
01951         SeekFile(fi->handle,&oldposition,SEEK_SET);
01952         if ( ISNEGPOS(oldposition) ) {
01953             MLOCK(ErrorMessageLock);
01954             MesPrint("File error");
01955             MUNLOCK(ErrorMessageLock);
01956             SETERROR(-1)
01957         }
01958     */
01959 }
01960 else {
01961     fi->POfill = fi->PObuffer+BASEPOSITION(oldposition);
01962 }
01963 AR.GetOneFile = oldonefile;
01964 }
01965 AR.CompressPointer = oldcomppointer;
01966 return(*term);
01967 }
01968
01969 /*
01970     #] GetFirstTerm :
01971     #[ GetContent :
01972 */
01973
01974 int GetContent(WORD *content, int num)
01975 {
01976     /*
01977         Gets the content of the expression 'num'
01978         Puts it in content.
01979         Routine should be thread-safe
01980         The content is defined as the term that will make the expression 'num'
01981         with integer coefficients, no GCD and all common factors taken out,
01982         all negative powers removed when we divide the expression by this
01983         content.
01984     */
01985     GETIDENTITY
01986     POSITION position, oldposition;
01987     RENUMBER renumber;
01988     FILEHANDLE *fi;
01989     WORD type, *oldcomppointer, oldonefile, numword, *term, i;
01990     WORD *cbuffer = TermMalloc("GetContent");
01991     WORD *oldworkpointer = AT.WorkPointer;
01992
01993     oldcomppointer = AR.CompressPointer;
01994     type = Expressions[num].status;
01995     if ( type == STOREDEXPRESSION ) {
01996         WORD TmpProto[SUBEXPSIZE];
01997         TmpProto[0] = EXPRESSION;

```

```

01998     Tmproto[1] = SUBEXPSIZE;
01999     Tmproto[2] = num;
02000     Tmproto[3] = 1;
02001     { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) Tmproto[ie] = 0; }
02002     AT.Tmaddr = Tmproto;
02003     PUTZERO(position);
02004     if ( ( renumber = GetTable(num,&position,0) ) == 0 ) goto CalledFrom;
02005     if ( GetFromStore(cbuffer,&position,renumber,&numword,num) < 0 ) goto CalledFrom;
02006     for(;;) {
02007         term = oldworkpointer;
02008         AR.CompressPointer = oldcomppointer;
02009         if ( GetFromStore(term,&position,renumber,&numword,num) < 0 ) goto CalledFrom;
02010         if ( *term == 0 ) break;
02011     /*
02012         'merge' the two terms
02013     */
02014         if ( ContentMerge(BHEAD cbuffer,term) < 0 ) goto CalledFrom;
02015     }
02016     /*
02017     #ifdef WITHPTHREADS
02018     */
02019     if ( renumber->symb.lo != AN.dummyrenumlist )
02020         M_free(renumber->symb.lo,"VarSpace");
02021     M_free(renumber,"Renummer");
02022     /*
02023     #endif
02024     */
02025     }
02026     else { /* Active expression */
02027         oldonefile = AR.GetOneFile;
02028         if ( type == HIDDENLEXPRESSION || type == HIDDENEXPRESSSION ) {
02029             AR.GetOneFile = 2; fi = AR.hidefile;
02030         }
02031         else {
02032             AR.GetOneFile = 0;
02033             if ( Expressions[num].replace == NEWLYDEFINEDEXPRESSSION )
02034                 fi = AR.outfile;
02035             else fi = AR.infile;
02036         }
02037         if ( fi->handle >= 0 ) {
02038             PUTZERO(oldposition);
02039         /*
02040             SeekFile(fi->handle,&oldposition,SEEK_CUR);
02041         */
02042         }
02043         else {
02044             SETBASEPOSITION(oldposition,fi->POfill-fi->PObuffer);
02045         }
02046         position = AS.OldOnFile[num];
02047         if ( GetOneTerm(BHEAD cbuffer,fi,&position,1) < 0 ) goto CalledFrom;
02048         AR.CompressPointer = oldcomppointer;
02049         if ( GetOneTerm(BHEAD cbuffer,fi,&position,1) < 0 ) goto CalledFrom;
02050     /*
02051         Now go through the terms. For each term we have to test whether
02052         what is in cbuffer is also in that term. If not, we have to remove
02053         it from cbuffer. Additionally we have to accumulate the GCD of the
02054         numerators and the LCM of the denominators. This is all done in the
02055         routine ContentMerge.
02056     */
02057         for(;;) {
02058             term = oldworkpointer;
02059             AR.CompressPointer = oldcomppointer;
02060             if ( GetOneTerm(BHEAD term,fi,&position,1) < 0 ) goto CalledFrom;
02061             if ( *term == 0 ) break;
02062         /*
02063             'merge' the two terms
02064         */
02065             if ( ContentMerge(BHEAD cbuffer,term) < 0 ) goto CalledFrom;
02066         }
02067         if ( fi->handle < 0 ) {
02068             fi->POfill = fi->PObuffer+BASEPOSITION(oldposition);
02069         }
02070         AR.GetOneFile = oldonefile;
02071     }
02072     AR.CompressPointer = oldcomppointer;
02073     for ( i = 0; i < *cbuffer; i++ ) content[i] = cbuffer[i];
02074     TermFree(cbuffer,"GetContent");
02075     AT.WorkPointer = oldworkpointer;
02076     return(*content);
02077 CalledFrom:
02078     MLOCK(ErrorMessageLock);
02079     MesCall("GetContent");
02080     MUNLOCK(ErrorMessageLock);
02081     SETERROR(-1)
02082 }
02083
02084 /*

```

```

02085         #] GetContent :
02086         #[ CleanupTerm :
02087
02088         Removes noncommuting objects from the term
02089     */
02090
02091 int CleanupTerm(WORD *term)
02092 {
02093     WORD *tstop, *t, *tfill, *tt;
02094     GETSTOP(term,tstop);
02095     t = term+1;
02096     while ( t < tstop ) {
02097         if ( *t >= FUNCTION && ( functions[*t-FUNCTION].commute || *t == DENOMINATOR ) ) {
02098             tfill = t; tt = t + t[1]; tstop = term + *term;
02099             while ( tt < tstop ) *tfill++ = *tt++;
02100             *term = tfill - term;
02101             tstop -= ABS(tfill[-1]);
02102         }
02103         else {
02104             t += t[1];
02105         }
02106     }
02107     return(0);
02108 }
02109
02110 /*
02111     #] CleanupTerm :
02112     #[ ContentMerge :
02113 */
02114
02115 WORD ContentMerge(PHEAD WORD *content, WORD *term)
02116 {
02117     GETBIDENTITY
02118     WORD *cstop, csize, crsize, sign = 1, numsize, densize, i, tnsz, tdsz;
02119     UWORD *num, *den, *tnum, *tden;
02120     WORD *outfill, *outb = TermMalloc("ContentMerge"), *ct;
02121     WORD *t, *tstop, tsz, trsz, *told;
02122     WORD *t1, *t2, *c1, *c2, i1, i2, *out1;
02123     WORD didsymbol = 0, diddotp = 0, tfirst;
02124     cstop = content + *content;
02125     csize = cstop[-1];
02126     if ( csize < 0 ) { sign = -sign; csize = -csize; }
02127     cstop -= csize;
02128     numsize = densize = crsize = (csize-1)/2;
02129     num = NumberMalloc("ContentMerge");
02130     den = NumberMalloc("ContentMerge");
02131     for ( i = 0; i < numsize; i++ ) num[i] = (UWORD)(cstop[i]);
02132     for ( i = 0; i < densize; i++ ) den[i] = (UWORD)(cstop[i+crsize]);
02133     while ( num[numsize-1] == 0 ) numsize--;
02134     while ( den[densize-1] == 0 ) densize--;
02135 /*
02136     First we do the coefficient
02137 */
02138     tstop = term + *term;
02139     tsz = tstop[-1];
02140     if ( tsz < 0 ) tsz = -tsz;
02141 /*
02142     else { sign = 1; } */
02143     tstop = tstop - tsz;
02144     tnsz = tdsz = trsz = (tsz-1)/2;
02145     tnum = (UWORD *)tstop; tden = (UWORD *) (tstop + trsz);
02146     while ( tnum[tnsz-1] == 0 ) tnsz--;
02147     while ( tden[tdsz-1] == 0 ) tdsz--;
02148     GcdLong(BHEAD num, numsize, tnum, tnsz, num, &numsize);
02149     if ( LcmLong(BHEAD den, densize, tden, tdsz, den, &densize) ) goto CalledFrom;
02150     outfill = outb + 1;
02151     ct = content + 1;
02152     t = term + 1;
02153     while ( ct < cstop ) {
02154         switch ( *ct ) {
02155             case SYMBOL:
02156                 didsymbol = 1;
02157                 t = term+1;
02158                 while ( t < tstop && *t != *ct ) t += t[1];
02159                 if ( t >= tstop ) break;
02160                 t1 = t+2; t2 = t+t[1];
02161                 c1 = ct+2; c2 = ct+ct[1];
02162                 out1 = outfill; *outfill++ = *ct; outfill++;
02163                 while ( c1 < c2 && t1 < t2 ) {
02164                     if ( *c1 == *t1 ) {
02165                         if ( t1[1] <= c1[1] ) {
02166                             *outfill++ = *t1++; *outfill++ = *t1++;
02167                             c1 += 2;
02168                         }
02169                     }
02170                     else {
02171                         *outfill++ = *c1++; *outfill++ = *c1++;
02172                         t1 += 2;
02173                     }
02174                 }
02175             }

```

```

02172     }
02173     else if ( *c1 < *t1 ) {
02174         if ( c1[1] < 0 ) {
02175             *outfill++ = *c1++; *outfill++ = *c1++;
02176         }
02177         else { c1 += 2; }
02178     }
02179     else {
02180         if ( t1[1] < 0 ) {
02181             *outfill++ = *t1++; *outfill++ = *t1++;
02182         }
02183         else t1 += 2;
02184     }
02185 }
02186 while ( c1 < c2 ) {
02187     if ( c1[1] < 0 ) { *outfill++ = c1[0]; *outfill++ = c1[1]; }
02188     c1 += 2;
02189 }
02190 while ( t1 < t2 ) {
02191     if ( t1[1] < 0 ) { *outfill++ = t1[0]; *outfill++ = t1[1]; }
02192     t1 += 2;
02193 }
02194 out1[1] = outfill - out1;
02195 if ( out1[1] == 2 ) outfill = out1;
02196 break;
02197 case DOTPRODUCT:
02198     diddotp = 1;
02199     t = term+1;
02200     while ( t < tstop && *t != *ct ) t += t[1];
02201     if ( t >= tstop ) break;
02202     t1 = t+2; t2 = t+t[1];
02203     c1 = ct+2; c2 = ct+ct[1];
02204     out1 = outfill; *outfill++ = *ct; outfill++;
02205     while ( c1 < c2 && t1 < t2 ) {
02206         if ( *c1 == *t1 && c1[1] == t1[1] ) {
02207             if ( t1[2] <= c1[2] ) {
02208                 *outfill++ = *t1++; *outfill++ = *t1++; *outfill++ = *t1++;
02209                 c1 += 3;
02210             }
02211             else {
02212                 *outfill++ = *c1++; *outfill++ = *c1++; *outfill++ = *c1++;
02213                 t1 += 3;
02214             }
02215         }
02216         else if ( *c1 < *t1 || ( *c1 == *t1 && c1[1] < t1[1] ) ) {
02217             if ( c1[2] < 0 ) {
02218                 *outfill++ = *c1++; *outfill++ = *c1++; *outfill++ = *c1++;
02219             }
02220             else { c1 += 3; }
02221         }
02222         else {
02223             if ( t1[2] < 0 ) {
02224                 *outfill++ = *t1++; *outfill++ = *t1++; *outfill++ = *t1++;
02225             }
02226             else t1 += 3;
02227         }
02228     }
02229     while ( c1 < c2 ) {
02230         if ( c1[2] < 0 ) { *outfill++ = c1[0]; *outfill++ = c1[1]; *outfill++ = c1[1]; }
02231         c1 += 3;
02232     }
02233     while ( t1 < t2 ) {
02234         if ( t1[2] < 0 ) { *outfill++ = t1[0]; *outfill++ = t1[1]; *outfill++ = t1[1]; }
02235         t1 += 3;
02236     }
02237     out1[1] = outfill - out1;
02238     if ( out1[1] == 2 ) outfill = out1;
02239     break;
02240 case INDEX:
02241     t = term+1;
02242     while ( t < tstop && *t != *ct ) t += t[1];
02243     if ( t >= tstop ) break;
02244     t1 = t+2; t2 = t+t[1];
02245     c1 = ct+2; c2 = ct+ct[1];
02246     out1 = outfill; *outfill++ = *ct; outfill++;
02247     while ( c1 < c2 && t1 < t2 ) {
02248         if ( *c1 == *t1 ) {
02249             *outfill++ = *c1++;
02250             t1 += 1;
02251         }
02252         else if ( *c1 < *t1 ) { c1 += 1; }
02253         else { t1 += 1; }
02254     }
02255     out1[1] = outfill - out1;
02256     if ( out1[1] == 2 ) outfill = out1;
02257     break;
02258 case VECTOR:

```

```

02259         case DELTA:
02260             t = term+1;
02261             while ( t < tstop && *t != *ct ) t += t[1];
02262             if ( t >= tstop ) break;
02263             t1 = t+2; t2 = t+t[1];
02264             c1 = ct+2; c2 = ct+ct[1];
02265             out1 = outfill; *outfill++ = *ct; outfill++;
02266             while ( c1 < c2 && t1 < t2 ) {
02267                 if ( *c1 == *t1 && c1[1] && t1[1] ) {
02268                     *outfill++ = *c1++; *outfill++ = *c1++;
02269                     t1 += 2;
02270                 }
02271                 else if ( *c1 < *t1 || ( *c1 == *t1 && c1[1] < t1[1] ) ) {
02272                     c1 += 2;
02273                 }
02274                 else {
02275                     t1 += 2;
02276                 }
02277             }
02278             out1[1] = outfill - out1;
02279             if ( out1[1] == 2 ) outfill = out1;
02280             break;
02281         case GAMMA:
02282             default: /* Functions */
02283                 told = t;
02284                 t = term+1;
02285                 while ( t < tstop ) {
02286                     if ( *t != *ct ) { t += t[1]; continue; }
02287                     if ( ct[1] != t[1] ) { t += t[1]; continue; }
02288                     if ( ct[2] != t[2] ) { t += t[1]; continue; }
02289                     t1 = t; t2 = ct; i1 = t1[1]; i2 = t2[1];
02290                     while ( i1 > 0 ) {
02291                         if ( *t1 != *t2 ) break;
02292                         t1++; t2++; i1--;
02293                     }
02294                     if ( i1 != 0 ) { t += t[1]; continue; }
02295                     t1 = t;
02296                     for ( i = 0; i < i2; i++ ) { *outfill++ = *t++; }
02297                 /*
02298                     Mark as 'used'. The flags must be different!
02299                 */
02300                 t1[2] |= SUBTERMUSED1;
02301                 ct[2] |= SUBTERMUSED2;
02302                 t = told;
02303                 break;
02304             }
02305             break;
02306         }
02307         ct += ct[1];
02308     }
02309     if ( diddotp == 0 ) {
02310         t = term+1; while ( t < tstop && *t != DOTPRODUCT ) t += t[1];
02311         if ( t < tstop ) { /* now we need the negative powers */
02312             tfirst = 1; told = outfill;
02313             for ( i = 2; i < t[1]; i += 3 ) {
02314                 if ( t[i+2] < 0 ) {
02315                     if ( tfirst ) { *outfill++ = DOTPRODUCT; *outfill++ = 0; tfirst = 0; }
02316                     *outfill++ = t[i]; *outfill++ = t[i+1]; *outfill++ = t[i+2];
02317                 }
02318             }
02319             if ( outfill > told ) told[1] = outfill-told;
02320         }
02321     }
02322     if ( didsymbol == 0 ) {
02323         t = term+1; while ( t < tstop && *t != SYMBOL ) t += t[1];
02324         if ( t < tstop ) { /* now we need the negative powers */
02325             tfirst = 1; told = outfill;
02326             for ( i = 2; i < t[1]; i += 2 ) {
02327                 if ( t[i+1] < 0 ) {
02328                     if ( tfirst ) { *outfill++ = SYMBOL; *outfill++ = 0; tfirst = 0; }
02329                     *outfill++ = t[i]; *outfill++ = t[i+1];
02330                 }
02331             }
02332             if ( outfill > told ) told[1] = outfill-told;
02333         }
02334     }
02335     /*
02336     Now put the coefficient back.
02337     */
02338     if ( numsize < densize ) {
02339         for ( i = numsize; i < densize; i++ ) num[i] = 0;
02340         numsize = densize;
02341     }
02342     else if ( densize < numsize ) {
02343         for ( i = densize; i < numsize; i++ ) den[i] = 0;
02344         densize = numsize;
02345     }

```

```

02346     for ( i = 0; i < numsize; i++ ) *outfill++ = num[i];
02347     for ( i = 0; i < densize; i++ ) *outfill++ = den[i];
02348     csize = numsize+densize+1;
02349     if ( sign < 0 ) csize = -csize;
02350     *outfill++ = csize;
02351     *outb = outfill-outb;
02352     NumberFree(den,"ContentMerge");
02353     NumberFree(num,"ContentMerge");
02354     for ( i = 0; i < *outb; i++ ) content[i] = outb[i];
02355     TermFree(outb,"ContentMerge");
02356 /*
02357     Now we have to 'restore' the term to its original.
02358     We do not restore the content, because if anything was used the
02359     new content overwrites the old. 6-mar-2018 JV
02360 */
02361     t = term + 1;
02362     while ( t < tstop ) {
02363         if ( *t >= FUNCTION ) t[2] &= ~SUBTERMUDED1;
02364         t += t[1];
02365     }
02366     return(*content);
02367 CalledFrom:
02368     MLOCK(ErrorMessageLock);
02369     MesCall("GetContent");
02370     MUNLOCK(ErrorMessageLock);
02371     SETERROR(-1)
02372 }
02373
02374 /*
02375     #] ContentMerge :
02376     #[ TermsInExpression :
02377 */
02378
02379 LONG TermsInExpression(WORD num)
02380 {
02381     LONG x = Expressions[num].counter;
02382     if ( x >= 0 ) return(x);
02383     return(-1);
02384 }
02385
02386 /*
02387     #] TermsInExpression :
02388     #[ SizeOfExpression :
02389 */
02390
02391 LONG SizeOfExpression(WORD num)
02392 {
02393     LONG x = (LONG) (DIVPOS(Expressions[num].size,sizeof(WORD)));
02394     if ( x >= 0 ) return(x);
02395     return(-1);
02396 }
02397
02398 /*
02399     #] SizeOfExpression :
02400     #[ UpdatePositions :
02401 */
02402
02403 void UpdatePositions(void)
02404 {
02405     EXPRESSIONS e = Expressions;
02406     POSITION *old;
02407     WORD *oldw;
02408     int i;
02409     if ( NumExpressions > 0 &&
02410         ( AS.OldOnFile == 0 || AS.NumOldOnFile < NumExpressions ) ) {
02411         if ( AS.OldOnFile ) {
02412             old = AS.OldOnFile;
02413             AS.OldOnFile = (POSITION *)Malloc1(NumExpressions*sizeof(POSITION),"file pointers");
02414             for ( i = 0; i < AS.NumOldOnFile; i++ ) AS.OldOnFile[i] = old[i];
02415             AS.NumOldOnFile = NumExpressions;
02416             M_free(old,"process file pointers");
02417         }
02418         else {
02419             AS.OldOnFile = (POSITION *)Malloc1(NumExpressions*sizeof(POSITION),"file pointers");
02420             AS.NumOldOnFile = NumExpressions;
02421         }
02422     }
02423     if ( NumExpressions > 0 &&
02424         ( AS.OldNumFactors == 0 || AS.NumOldNumFactors < NumExpressions ) ) {
02425         if ( AS.OldNumFactors ) {
02426             oldw = AS.OldNumFactors;
02427             AS.OldNumFactors = (WORD *)Malloc1(NumExpressions*sizeof(WORD),"numfactors pointers");
02428             for ( i = 0; i < AS.NumOldNumFactors; i++ ) AS.OldNumFactors[i] = oldw[i];
02429             M_free(oldw,"numfactors pointers");
02430             oldw = AS.Oldvflags;
02431             AS.Oldvflags = (WORD *)Malloc1(NumExpressions*sizeof(WORD),"vflags pointers");
02432             for ( i = 0; i < AS.NumOldNumFactors; i++ ) AS.Oldvflags[i] = oldw[i];

```

```

02433     M_free(oldw,"vflags pointers");
02434     oldw = AS.Olduflags;
02435     AS.Olduflags = (WORD *)Malloca1(NumExpressions*sizeof(WORD),"uflags pointers");
02436     for ( i = 0; i < AS.NumOldNumFactors; i++ ) AS.Olduflags[i] = oldw[i];
02437     M_free(oldw,"uflags pointers");
02438     AS.NumOldNumFactors = NumExpressions;
02439 }
02440 else {
02441     AS.OldNumFactors = (WORD *)Malloca1(NumExpressions*sizeof(WORD),"numfactors pointers");
02442     AS.Oldvflags = (WORD *)Malloca1(NumExpressions*sizeof(WORD),"vflags pointers");
02443     AS.Olduflags = (WORD *)Malloca1(NumExpressions*sizeof(WORD),"uflags pointers");
02444     AS.NumOldNumFactors = NumExpressions;
02445 }
02446 }
02447 for ( i = 0; i < NumExpressions; i++ ) {
02448     AS.OldOnFile[i] = e[i].onfile;
02449     AS.OldNumFactors[i] = e[i].numfactors;
02450     AS.Oldvflags[i] = e[i].vflags;
02451     AS.Olduflags[i] = e[i].uflags;
02452 }
02453 }
02454
02455 /*
02456     #] UpdatePositions :
02457     #[ CountTerms1 :      LONG CountTerms1()
02458
02459     Counts the terms in the current deferred bracket
02460     Is mainly an adaptation of the routine Deferred in proces.c
02461 */
02462
02463 LONG CountTerms1(PHEAD0)
02464 {
02465     GETBIDENTITY
02466     POSITION oldposition, startposition;
02467     WORD *t, *m, *mstop, decr, i, *oldwork, retval;
02468     WORD *oldipointer = AR.CompressPointer;
02469     WORD oldGetOneFile = AR.GetOneFile, olddeferflag = AR.DeferFlag;
02470     LONG numterms = 0;
02471     AR.GetOneFile = 1;
02472     oldwork = AT.WorkPointer;
02473     AT.WorkPointer = (WORD *)(((UBYTE *) (AT.WorkPointer)) + AM.MaxTer);
02474     AR.DeferFlag = 0;
02475     startposition = AR.DefPosition;
02476 /*
02477     Store old position
02478 */
02479 if ( AR.infile->handle >= 0 ) {
02480     PUTZERO(oldposition);
02481 /*
02482     SeekFile(AR.infile->handle,&oldposition,SEEK_CUR);
02483 */
02484 }
02485 else {
02486     SETBASEPOSITION(oldposition,AR.infile->POfill-AR.infile->PObuffer);
02487     AR.infile->POfill = (WORD *)(((UBYTE *) (AR.infile->PObuffer)
02488         +BASEPOSITION(startposition)));
02489 }
02490 /*
02491     Look in the CompressBuffer where the bracket contents start
02492 */
02493 t = m = AR.CompressBuffer;
02494 t += *t;
02495 mstop = t - ABS(t[-1]);
02496 m++;
02497 while ( *m != HAAKJE && m < mstop ) m += m[1];
02498 if ( m >= mstop ) { /* No deferred action! */
02499     numterms = 1;
02500     AR.DeferFlag = olddeferflag;
02501     AT.WorkPointer = oldwork;
02502     AR.GetOneFile = oldGetOneFile;
02503     return(numterms);
02504 }
02505 mstop = m + m[1];
02506 decr = WORDDIFF(mstop,AR.CompressBuffer)-1;
02507
02508 m = AR.CompressBuffer;
02509 t = AR.CompressPointer;
02510 i = *m;
02511 NCOPY(t,m,i);
02512 AR.TePos = 0;
02513 AN.TeSuOut = 0;
02514 /*
02515     Status:
02516     First bracket content starts at mstop.
02517     Next term starts at startposition.
02518     Decompression information is in AR.CompressPointer.
02519     The outside of the bracket runs from AR.CompressBuffer+1 to mstop.

```



```

02520 */
02521     AR.CompressPointer = oldipointer;
02522     for(;;) {
02523         numterms++;
02524         retval = GetOneTerm(BHEAD AT.WorkPointer,AR.infile,&startposition,0);
02525         if ( retval >= 0 ) AR.CompressPointer = oldipointer;
02526         if ( retval <= 0 ) break;
02527         t = AR.CompressPointer;
02528         if ( *t < (1 + decr + ABS(*(t+*t-1))) ) break;
02529         t++;
02530         m = AR.CompressBuffer+1;
02531         while ( m < mstop ) {
02532             if ( *m != *t ) goto Thatsit;
02533             m++; t++;
02534         }
02535     }
02536 Thatsit:;
02537 /*
02538     Finished. Reposition the file, restore information and return.
02539 */
02540     AT.WorkPointer = oldwork;
02541     if ( AR.infile->handle >= 0 ) {
02542 /*
02543         SeekFile(AR.infile->handle,&oldposition,SEEK_SET);
02544 */
02545     }
02546     else {
02547         AR.infile->POfill = AR.infile->PObuffer + BASEPOSITION(oldposition);
02548     }
02549     AR.DeferFlag = olddeferflag;
02550     AR.GetOneFile = oldGetOneFile;
02551     return(numterms);
02552 }
02553
02554 /*
02555     #] CountTerms1 :
02556     #[ TermsInBracket :      LONG TermsInBracket(term,level)
02557
02558     The function TermsInBracket_()
02559     Syntax:
02560         TermsInBracket_() : The current bracket in a Keep Brackets
02561         TermsInBracket_(bracket) : This bracket in the current expression
02562         TermsInBracket_(expression,bracket) : This bracket in the given expression
02563     All other specifications don't have any effect.
02564 */
02565
02566 #define CURRENTBRACKET 1
02567 #define BRACKETCURRENTEXPR 2
02568 #define BRACKETOTHEREXPR 3
02569 #define NOBRACKETACTIVE 4
02570
02571 LONG TermsInBracket(PHEAD WORD *term, WORD level)
02572 {
02573     WORD *t, *tstop, *b, *tt, *n1, *n2;
02574     int type = 0, i, num;
02575     LONG numterms = 0;
02576     WORD *bracketbuffer = AT.WorkPointer;
02577     t = term; GETSTOP(t,tstop);
02578     t++; b = bracketbuffer;
02579     while ( t < tstop ) {
02580         if ( *t != TERMSINBRACKET ) { t += t[1]; continue; }
02581         if ( t[1] == FUNHEAD || (
02582             t[1] == FUNHEAD+2
02583             && t[FUNHEAD] == -SNUMBER
02584             && t[FUNHEAD+1] == 0
02585         ) ) {
02586             if ( AC.ComDefer == 0 ) {
02587                 type = NOBRACKETACTIVE;
02588             }
02589             else {
02590                 type = CURRENTBRACKET;
02591             }
02592             *b = 0;
02593             break;
02594         }
02595         if ( t[FUNHEAD] == -EXPRESSION ) {
02596             if ( t[FUNHEAD+2] < 0 ) {
02597                 if ( ( t[FUNHEAD+2] <= -FUNCTION ) && ( t[1] == FUNHEAD+3 ) ) {
02598                     type = BRACKETOTHEREXPR;
02599                     *b++ = FUNHEAD+4; *b++ = -t[FUNHEAD+2]; *b++ = FUNHEAD;
02600                     for ( i = 2; i < FUNHEAD; i++ ) *b++ = 0;
02601                     *b++ = 1; *b++ = 1; *b++ = 3;
02602                     break;
02603                 }
02604                 else if ( ( t[FUNHEAD+2] > -FUNCTION ) && ( t[1] == FUNHEAD+4 ) ) {
02605                     type = BRACKETOTHEREXPR;
02606                     tt = t + FUNHEAD+2;

```

```

02607         switch ( *tt ) {
02608             case -SYMBOL:
02609                 *b++ = 8; *b++ = SYMBOL; *b++ = 4; *b++ = tt[1];
02610                 *b++ = 1; *b++ = 1; *b++ = 1; *b++ = 3;
02611                 break;
02612             case -SNUMBER:
02613                 if ( tt[1] == 1 ) {
02614                     *b++ = 4; *b++ = 1; *b++ = 1; *b++ = 3;
02615                 }
02616                 else goto IllBraReq;
02617                 break;
02618             default:
02619                 goto IllBraReq;
02620         }
02621         break;
02622     }
02623 }
02624 else if ( ( t[FUNHEAD+2] == (t[1]-FUNHEAD-2) ) &&
02625           ( t[FUNHEAD+2+ARGHEAD] == (t[FUNHEAD+2]-ARGHEAD) ) ) {
02626     type = BRACKETOTHEREXPR;
02627     tt = t + FUNHEAD + ARGHEAD; num = *tt;
02628     for ( i = 0; i < num; i++ ) *b++ = *tt++;
02629     break;
02630 }
02631 }
02632 else {
02633     if ( t[FUNHEAD] < 0 ) {
02634         if ( ( t[FUNHEAD] <= -FUNCTION ) && ( t[1] == FUNHEAD+1 ) ) {
02635             type = BRACKETCURRENTEXPR;
02636             *b++ = FUNHEAD+4; *b++ = -t[FUNHEAD+2]; *b++ = FUNHEAD;
02637             for ( i = 2; i < FUNHEAD; i++ ) *b++ = 0;
02638             *b++ = 1; *b++ = 1; *b++ = 3; *b = 0;
02639             break;
02640         }
02641         else if ( ( t[FUNHEAD] > -FUNCTION ) && ( t[1] == FUNHEAD+2 ) ) {
02642             type = BRACKETCURRENTEXPR;
02643             tt = t + FUNHEAD+2;
02644             switch ( *tt ) {
02645                 case -SYMBOL:
02646                     *b++ = 8; *b++ = SYMBOL; *b++ = 4; *b++ = tt[1];
02647                     *b++ = 1; *b++ = 1; *b++ = 1; *b++ = 3;
02648                     break;
02649                 case -SNUMBER:
02650                     if ( tt[1] == 1 ) {
02651                         *b++ = 4; *b++ = 1; *b++ = 1; *b++ = 3;
02652                     }
02653                     else goto IllBraReq;
02654                     break;
02655                 default:
02656                     goto IllBraReq;
02657             }
02658             break;
02659         }
02660     }
02661     else if ( ( t[FUNHEAD] == (t[1]-FUNHEAD) ) &&
02662              ( t[FUNHEAD+ARGHEAD] == (t[FUNHEAD]-ARGHEAD) ) ) {
02663         type = BRACKETCURRENTEXPR;
02664         tt = t + FUNHEAD + ARGHEAD; num = *tt;
02665         for ( i = 0; i < num; i++ ) *b++ = *tt++;
02666         break;
02667     }
02668     else {
02669 IllBraReq;
02670         MLOCK(ErrorMessageLock);
02671         MesPrint("Illegal bracket request in termsinbracket_ function.");
02672         MUNLOCK(ErrorMessageLock);
02673         Terminate(-1);
02674     }
02675 }
02676 t += t[1];
02677 }
02678 AT.WorkPointer = b;
02679 if ( AT.WorkPointer + *term + 4 > AT.WorkTop ) {
02680     MLOCK(ErrorMessageLock);
02681     MesWork();
02682     MesPrint("Called from termsinbracket_ function.");
02683     MUNLOCK(ErrorMessageLock);
02684     return(-1);
02685 }
02686 /*
02687 We are now in the position to look for the bracket
02688 */
02689 switch ( type ) {
02690     case CURRENTBRACKET:
02691 /*
02692     The code here should be rather similar to when we pick up
02693     the contents of the bracket. In our case we only count the

```

```

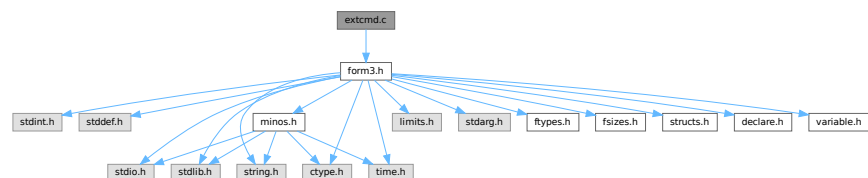
02694         terms though.
02695  */
02696         numterms = CountTerms1(BHEAD0);
02697         break;
02698     case BRACKETCURRENTEXPR:
02699  /*
02700         Not implemented yet.
02701  */
02702         MLOCK(ErrorMessageLock);
02703         MesPrint("termsinbracket_ function currently only handles Keep Brackets.");
02704         MUNLOCK(ErrorMessageLock);
02705         return(-1);
02706     case BRACKETOTHEREXPR:
02707         MLOCK(ErrorMessageLock);
02708         MesPrint("termsinbracket_ function currently only handles Keep Brackets.");
02709         MUNLOCK(ErrorMessageLock);
02710         return(-1);
02711     case NOBRACKETACTIVE:
02712         numterms = 1;
02713         break;
02714 }
02715 /*
02716 Now we have the number in numterms. We replace the function by it.
02717 */
02718 n1 = term; n2 = AT.WorkPointer; tstop = n1 + *n1;
02719 while ( n1 < t ) *n2++ = *n1++;
02720 i = numterms » BITSINWORD;
02721 if ( i == 0 ) {
02722     *n2++ = LNUMBER; *n2++ = 4; *n2++ = 1; *n2++ = (WORD)(numterms & WORDMASK);
02723 }
02724 else {
02725     *n2++ = LNUMBER; *n2++ = 5; *n2++ = 2;
02726     *n2++ = (WORD)(numterms & WORDMASK); *n2++ = i;
02727 }
02728 n1 += n1[1];
02729 while ( n1 < tstop ) *n2++ = *n1++;
02730 AT.WorkPointer[0] = n2 - AT.WorkPointer;
02731 AT.WorkPointer = n2;
02732 if ( Generator(BHEAD n1,level) < 0 ) {
02733     AT.WorkPointer = bracketbuffer;
02734     MLOCK(ErrorMessageLock);
02735     MesPrint("Called from termsinbracket_ function.");
02736     MUNLOCK(ErrorMessageLock);
02737     return(-1);
02738 }
02739 /*
02740 Finished. Reset things and return.
02741 */
02742 AT.WorkPointer = bracketbuffer;
02743 return(numterms);
02744 }
02745 /*
02746 #] TermsInBracket :      LONG TermsInBracket(term,level)
02747 #] Expressions :
02748 */

```

4.33 extcmd.c File Reference

```
#include "form3.h"
```

Include dependency graph for extcmd.c:



Functions

- int [openExternalChannel](#) (UBYTE *cmd, int daemonize, UBYTE *shellname, UBYTE *stderrname)

- int [initPresetExternalChannels](#) (UBYTE *theline, int thetimeout)
- int [closeExternalChannel](#) (int n)
- int [selectExternalChannel](#) (int n)
- int [getCurrentExternalChannel](#) (void)
- void [closeAllExternalChannels](#) (void)

4.33.1 Detailed Description

The system that takes care of communication with external programs.

Definition in file [extcmd.c](#).

4.33.2 Function Documentation

4.33.2.1 [openExternalChannel\(\)](#)

```
int openExternalChannel (
    UBYTE * cmd,
    int daemonize,
    UBYTE * shellname,
    UBYTE * stderrname )
```

Definition at line [230](#) of file [extcmd.c](#).

4.33.2.2 [initPresetExternalChannels\(\)](#)

```
int initPresetExternalChannels (
    UBYTE * theline,
    int thetimeout )
```

Definition at line [232](#) of file [extcmd.c](#).

4.33.2.3 [closeExternalChannel\(\)](#)

```
int closeExternalChannel (
    int n )
```

Definition at line [233](#) of file [extcmd.c](#).

4.33.2.4 [selectExternalChannel\(\)](#)

```
int selectExternalChannel (
    int n )
```

Definition at line [234](#) of file [extcmd.c](#).

4.33.2.5 getCurrentExternalChannel()

```
int getCurrentExternalChannel (
    void )
```

Definition at line 235 of file [extcmd.c](#).

4.33.2.6 closeAllExternalChannels()

```
void closeAllExternalChannels (
    void )
```

Definition at line 236 of file [extcmd.c](#).

4.34 extcmd.c

[Go to the documentation of this file.](#)

```
00001
00005 /* #[] License : */
00006 /*
00007 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00008 * When using this file you are requested to refer to the publication
00009 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010 * This is considered a matter of courtesy as the development was paid
00011 * for by FOM the Dutch physics granting agency and we would like to
00012 * be able to track its scientific use to convince FOM of its value
00013 * for the community.
00014 *
00015 * This file is part of FORM.
00016 *
00017 * FORM is free software: you can redistribute it and/or modify it under the
00018 * terms of the GNU General Public License as published by the Free Software
00019 * Foundation, either version 3 of the License, or (at your option) any later
00020 * version.
00021 *
00022 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00025 * details.
00026 *
00027 * You should have received a copy of the GNU General Public License along
00028 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00029 */
00030 /* #[] License : */
00031 /*
00032 #[] Documentation :
00033
00034 This module is written by M.Tentyukov as a part of implementation of
00035 interaction between FORM and external processes, first release
00036 09.04.2004. A part of this code is copied from the DIANA project, which was
00037 written by M. Tentyukov and published under the GPL version 2 as
00038 published by the Free Software Foundation.
00039
00040 This file is completely re-written by M.Tentyukov in May 2006.
00041 Since the interface was changed, the public function were changed,
00042 also. A new public functions were added: initPresetExternalChannels()
00043 (see comments just before this function in the present file) and
00044 setKillModeForExternalChannel (a pointer, not a function).
00045
00046 If a macro WITHEXTERNALCHANNEL is not defined, all public functions
00047 are stubs returning failure.
00048
00049 The idea is to start an external command swallowing
00050 its stdin and stdout. This can be done by means of the function
00051 int openExternalChannel(cmd,daemonize,shellname,stderrname), where
00052 cmd is a command to run,
00053 daemonize: if !=0 then start the command in the "daemon" mode,
00054 shellname: if !=NULL, execute the command in a subshell,
00055 stderrname: if != NULL, redirect stderr of the command to this file.
00056 The function returns some small positive integer number (the
00057 descriptor of a newly created external channel), or -1 on failure.
00058
```

```

00059     After the command is started, it becomes a _current_ opened external
00060     channel. The buffer can be sent to its stdin by a function
00061     int writeBufToExtChannel(buf, n)
00062     (here buf is a pointer to the buffer, n is the length in bytes; the
00063     function returns 0 in success, or -1 on failure),
00064     or one character can be read from its stdout by
00065     means of the function
00066     int getcFromExtChannel().
00067
00068     The latter returns the
00069     character casted to integer, or something <0. This can be -2 (if there
00070     is no current external channel) or EOF, if the external program closes
00071     its stdout, or if the external program outputs a string coinciding
00072     with a _terminator_.
00073
00074     By default, the terminator if an empty line. For the current external
00075     channel it can be set by means of the function
00076     int setTerminatorForExternalChannel(newterminator).
00077     The function returns 0 in success, or !0 if something is wrong (no
00078     current channel, too long terminator).
00079
00080     After getcFromExtChannel() returns EOF, the current channel becomes
00081     undefined. Any further attempts to read information by
00082     getcFromExtChannel() result in -2. To set (re-set) a current channel,
00083     the function
00084     int selectExternalChannel(n)
00085     can be used. This function accepts the valid external channel
00086     descriptor (returned by openExternalChannel) and returns the
00087     descriptor of a previous current channel (0, if there was no current
00088     channel, or -1, if the external channel descriptor is invalid).
00089     If n == 0, the function undefine the current external channel.
00090
00091     The function
00092     int closeExternalChannel(n)
00093     destroys the opened external channel with the descriptor n. It returns
00094     0 in success, or -1 on failure. If the corresponding external channel
00095     was the current one, the current channel becomes undefined. If n==0,
00096     the function closes the current external channel.
00097
00098     The function
00099     int getCurrentExternalChannel(void)
00100     returns the descriptor if the current external channel, or 0 , if
00101     there is no current external channel.
00102
00103     The function
00104     void closeAllExternalChannels(void)
00105
00106     destroys all opened external channels.
00107
00108     List of all public functions:
00109     int openExternalChannel(UBYTE *cmd,int daemonize,UBYTE *shellname, UBYTE * stderrname);
00110     int initPresetExternalChannels(UBYTE *theline, int thetimeout);
00111     int setTerminatorForExternalChannel(char *newterminator);
00112     int setKillModeForExternalChannel(int signum, int sentToWholeGroup);
00113     int closeExternalChannel(int n);
00114     int selectExternalChannel(int n);
00115     int writeBufToExtChannel(char *buf,int n);
00116     int getcFromExtChannel(void);
00117     int getCurrentExternalChannel(void);
00118     void closeAllExternalChannels(void);
00119
00120     ATTENTION!
00121
00122     Four of them:
00123     1 setTerminatorForExternalChannel
00124     2 setKillModeForExternalChannel
00125     3 writeBufToExtChannel
00126     4 getcFromExtChannel
00127
00128     are NOT functions, but variables (pointers) of a corrsponding type.
00129     They are initialised by proper values to avoid repeated error checking.
00130
00131     All public functions are independent of realization hidden in this module.
00132     All other functions may have a returned type/parameters type local w.r.t.
00133     this module; they are not declared outside of this file.
00134
00135     #] Documentation :
00136     #[ Selftest initializing:
00137 */
00138
00139 /*
00140 Uncomment to get a self-consistent program:
00141 #define SELFTEST 1
00142 */
00143
00144
00145 #ifndef SELFTEST

```

```

00146 #define WITHEXTERNALCHANNEL 1
00147 #ifdef _MSC_VER
00148 #define FORM_INLINE __inline
00149 #else
00150 #define FORM_INLINE inline
00151 #endif
00152
00153 /*
00154     From form3.h:
00155 */
00156 typedef unsigned char UBYTE;
00157
00158 /*The following variables should be defined in variable.h:*/
00159 extern int (*writeBufToExtChannel)(char *buffer, size_t n);
00160 extern int (*getcFromExtChannel)();
00161 extern int (*setTerminatorForExternalChannel)(char *buffer);
00162 extern int (*setKillModeForExternalChannel)(int signum, int sentToWholeGroup);
00163
00164 #else /*ifdef SELFTEST*/
00165 #include "form3.h"
00166 #endif /*ifdef SELFTEST ... else*/
00167 /*
00168 pid_t getExternalChannelPid(void);
00169 */
00170 /*
00171     #] Selftest initializing:
00172     #[ Includes :
00173 */
00174 #ifdef WITHEXTERNALCHANNEL
00175 #include <stdio.h>
00176 #ifndef _MSC_VER
00177 #include <unistd.h>
00178 #endif
00179 #include <fcntl.h>
00180 #include <sys/types.h>
00181 #ifndef _MSC_VER
00182 #include <sys/time.h>
00183 #include <sys/wait.h>
00184 #endif
00185 #include <errno.h>
00186 #include <signal.h>
00187 #include <limits.h>
00188 /*
00189     #] Includes :
00190     #[ FailureFunctions:
00191 */
00192
00193 /*Non-initialized variant of public functions:*/
00194 int writeBufToExtChannelFailure(char *buf, size_t count)
00195 {
00196     DUMMYUSE(buf); DUMMYUSE(count);
00197     return(-1);
00198 } /*writeBufToExtChannelFailure*/
00199
00200 int setTerminatorForExternalChannelFailure(char *newTerminator)
00201 {
00202     DUMMYUSE(newTerminator);
00203     return(-1);
00204 } /*setTerminatorForExternalChannelFailure*/
00205
00206 int setKillModeForExternalChannelFailure(int signum, int sentToWholeGroup)
00207 {
00208     DUMMYUSE(signum); DUMMYUSE(sentToWholeGroup);
00209     return(-1);
00210 } /*setKillModeForExternalChannelFailure*/
00211
00212 int getcFromExtChannelFailure(void)
00213 {
00214     return(-2);
00215 } /*getcFromExtChannelFailure*/
00216
00217 int (*writeBufToExtChannel)(char *buffer, size_t n) = &writeBufToExtChannelFailure;
00218 int (*setTerminatorForExternalChannel)(char *buffer) =
00219     &setTerminatorForExternalChannelFailure;
00220 int (*setKillModeForExternalChannel)(int signum, int sentToWholeGroup) =
00221     &setKillModeForExternalChannelFailure;
00222 int (*getcFromExtChannel)(void) = &getcFromExtChannelFailure;
00223 #endif
00224 /*
00225     #] FailureFunctions:
00226     #[ Stubs :
00227 */
00228 #ifndef WITHEXTERNALCHANNEL
00229 /*Stubs for public functions:*/
00230 int openExternalChannel(UBYTE *cmd, int daemonize, UBYTE *shellname, UBYTE *stderrname)
00231 { DUMMYUSE(cmd); DUMMYUSE(daemonize); DUMMYUSE(shellname); DUMMYUSE(stderrname); return(-1); };
00232 int initPresetExternalChannels(UBYTE *theline, int thetimeout) { DUMMYUSE(theline);

```

```

        DUMMYUSE(thetimeout); return(-1); };
00233 int closeExternalChannel(int n) { DUMMYUSE(n); return(-1); };
00234 int selectExternalChannel(int n) { DUMMYUSE(n); return(-1); };
00235 int getCurrentExternalChannel(void) { return(0); };
00236 void closeAllExternalChannels(void) {};
00237 #else /*ifndef WITHEXTERNALCHANNEL*/
00238 /*
00239     #] Stubs :
00240     #[ Local types :
00241 */
00242 /*First argument for the function signal:*/
00243 #ifndef INTSIGHANDLER
00244 typedef void (*mysighandler_t)(int);
00245 #else
00246 /* Sometimes, this nonsense may occurs:*/
00247 /*typedef int (*mysighandler_t)(int);*/
00248 #endif
00249
00250 /*Input IO buffer size increment -- each time the buffer
00251 is expired it will be increased by this value (in bytes):*/
00252 #define DELTA_EXT_BUF 128
00253
00254 /*Re-allocatable array containing External Channel
00255 handlers increased each time by this value:*/
00256 #define DELTA_EXT_LIST 8
00257
00258 /*How many times I/O routines may attempt to continue their work
00259 in some failures:*/
00260 #define MAX_FAILS_IO 2
00261
00262 /*The external channel handler structure:*/
00263 typedef struct ExternalChannel {
00264     pid_t pid; /*PID of the external process*/
00265     pid_t gpid; /*process group ID of the external process.
00266                 If <=0, not used, if >0, the kill signals
00267                 is sent to the whole group */
00268     FILE *frec; /*stdout of the external process*/
00269     char *INbuf; /*External channel buffer*/
00270     char *IBfill; /*Position in INbuf from which the next input character will be read*/
00271     char *IBfull; /*End of read INbuf*/
00272     char *IBstop; /*end of allocated space for INbuf*/
00273     char *terminator; /* Terminator - when extern. program outputs ONLY this string,
00274                       it is assumed that the answer is ready, and getcFromExtChannel
00275                       returns EOF. Should not be longer then the minimal buffer!*/
00276     /*Info fields, not changable after creating a channel:*/
00277     char *cmd; /*the command*/
00278     char *shellname;
00279     char *stderrname; /*filename to redirect stderr, or NULL*/
00280     int fsend; /*stdin of the external process*/
00281     int killSignal; /*signal to kill*/
00282     int daemonize; /*0 --neither setsid nor daemonize, !=0 -- full daemonization*/
00283 } EXTHANDLE;
00284
00285
00286 static EXTHANDLE *externalChannelsList=0;
00287 /*Here integers are better than pointers: */
00288 static int externalChannelsListStop=0;
00289 static int externalChannelsListFill=0;
00290
00291 /*"current" external channel:*/
00292 static EXTHANDLE *externalChannelsListTop=0;
00293 /*
00294     #] Local types :
00295     #[ Selftest functions :
00296 */
00297 #ifdef SELFTEST
00298
00299 /*For malloc prototype:*/
00300 #include <stdlib.h>
00301
00302 /*StrLen, Malloc1, M_free and strDup1 are defined in tools.c -- here only emulation:*/
00303 int StrLen(char *pattern)
00304 {
00305     register char *p=(char*)pattern;
00306     while(*p)p++;
00307     return((int) ((p-(char*)pattern)) );
00308 }/*StrLen*/
00309 void *Malloc1(int l, char *c)
00310 {
00311     return(malloc(l));
00312 }
00313 void M_free(void *p,char *c)
00314 {
00315     return(free(p));
00316 }
00317
00318 char *strDup1(UBYTE *instring, char *ifwrong)

```



```

00319 {
00320     UBYTE *s = instring, *to;
00321     while ( *s ) s++;
00322     to = s = (UBYTE *)Malloca((s-instring)+1,ifwrong);
00323     while ( *instring ) *to++ = *instring++;
00324     *to = 0;
00325     return(s);
00326 }
00327
00328 /*PutPreVar from pre.c -- just ths stub:*/
00329 int PutPreVar(UBYTE *a,UBYTE *b,UBYTE *c,int i)
00330 {
00331     return(0);
00332 }
00333
00334 #endif
00335 /*
00336     #] Selftest functions :
00337     #[ Local functions :
00338 */
00339
00340 /*Initialize one cell of handler:*/
00341 static FORM_INLINE void extHandlerInit(EXTHANDLE *h)
00342 {
00343     h->pid=-1;
00344     h->gpip=-1;
00345     h->fsend=0;
00346     h->killSignal=SIGKILL;
00347     h->daemonize=1;
00348     h->frec=NULL;
00349     h->INbuf=h->IBfill=h->IBfull=h->IBstop=
00350     h->terminator=h->cmd=h->shellname=h->stderrname=NULL;
00351 }/*extHandlerInit*/
00352
00353 /* Copies each field of handler:*/
00354 static FORM_INLINE void extHandlerSwallowCopy(EXTHANDLE *to, EXTHANDLE *from)
00355 {
00356     to->pid=from->pid;
00357     to->gpip=from->gpip;
00358     to->fsend=from->fsend;
00359     to->killSignal=from->killSignal;
00360     to->daemonize=from->daemonize;
00361     to->frec=from->frec;
00362     to->INbuf=from->INbuf;
00363     to->IBfill=from->IBfill;
00364     to->IBfull=from->IBfull;
00365     to->IBstop=from->IBstop;
00366     to->terminator=from->terminator;
00367     to->cmd=from->cmd;
00368     to->shellname=from->shellname;
00369     to->stderrname=from->stderrname;
00370 }/*extHandlerSwallow*/
00371
00372 /*Allocates memory for fields of handler which have no fixed
00373 storage size and initializes some fields:*/
00374 static FORM_INLINE void
00375 extHandlerAlloc(EXTHANDLE *h, char *cmd, char *shellname, char *stderrname)
00376 {
00377     h->IBfill=h->IBfull=h->INbuf=
00378         Malloca(DELTA_EXT_BUF,"External channel buffer");
00379     h->IBstop=h->INbuf+DELTA_EXT_BUF;
00380     /*Initialize a terminator:*/
00381     *(h->terminator=Malloca(DELTA_EXT_BUF,"External channel terminator"))='\n';
00382     (h->terminator)[1]='\0';/*By default the terminator is '\n'*/
00383     /*Deep copy all strings:*/
00384     if(cmd!=NULL)
00385         h->cmd=(char *)strDup1((UBYTE *)cmd,"External channel command");
00386     else/*cmd cannot be NULL! If this is NULL then force it to be something special*/
00387         h->cmd=(char *)strDup1((UBYTE *)"/","External channel command");
00388     if(shellname!=NULL)
00389         h->shellname=
00390             (char *)strDup1((UBYTE *)shellname,"External channel shell name");
00391     if(stderrname!=NULL)
00392         h->stderrname=
00393             (char *)strDup1((UBYTE *)stderrname,"External channel stderr name");
00394 }/*extHandlerAlloc*/
00395
00396 /*Disallocates dynamically allocated fields of a handler:*/
00397 static FORM_INLINE void extHandlerFree(EXTHANDLE *h)
00398 {
00399     if(h->stderrname) M_free(h->stderrname,"External channel stderr name");
00400     if(h->shellname) M_free(h->shellname,"External channel shell name");
00401     if(h->cmd) M_free(h->cmd,"External channel command");
00402     if(h->terminator)M_free(h->terminator,"External channel terminator");
00403     if(h->INbuf)M_free(h->INbuf,"External channel buffer");
00404     extHandlerInit(h);
00405 }/*extHandlerFree*/

```

```

00406 /* Closes all descriptors, kills the external process, frees all internal fields,
00407 BUT does NOT free the main container:*/
00408 static void destroyExternalChannel(EXTHANDLE *h)
00409 {
00410     /*Note, this function works in parallel mode correctly, see comments below.*/
00411
00412     /*Note, for slaves in a parallel mode h->pid == 0:*/
00413     if( (h->pid > 0) && (h->killSignal > 0) ){
00414         int chstatus;
00415         if( h->gpipid > 0)
00416             chstatus=kill(-h->gpipid,h->killSignal);
00417         else
00418             chstatus=kill(h->pid,h->killSignal);
00419         if(chstatus==0)
00420             /*If the process will not be killed by this signal, FORM hangs up here!*/
00421             waitpid(h->pid, &chstatus, 0);
00422     }/*if( (h->pid > 0) && (h->killSignal > 0) )*/
00423
00424     /*Note, for slaves in a parallel mode h->frec == h->fsend == 0:*/
00425     if(h->frec) fclose(h->frec);
00426     if( h->fsend > 0) close(h->fsend);
00427
00428     extHandlerFree(h);
00429     /*Does not do "free(h)"!*/
00430 }/*destroyExternalChannel*/
00431
00432 /*Wrapper to the read() syscall, to handle possible interrupts by unblocked signals:*/
00433 static FORM_INLINE ssize_t read2b(int fd, char *buf, size_t count)
00434 {
00435     ssize_t res;
00436
00437     if( (res=read(fd,buf,count)) <1 )/*EOF or read is interrupted by a signal?:*/
00438         while( (errno == EINTR)&&(res <1) )
00439             /*The call was interrupted by a signal before any data was read, try again:*/
00440             res=read(fd,buf,count);
00441     return (res);
00442 }/*read2b*/
00443
00444 /*Wrapper to the write() syscall, to handle possible interrupts by unblocked signals:*/
00445 static FORM_INLINE ssize_t writeFromb(int fd, char *buf, size_t count)
00446 {
00447     ssize_t res;
00448     if( (res=write(fd,buf,count)) <1 )/*Is write interrupted by a signal?:*/
00449         while( (errno == EINTR)&&(res <1) )
00450             /*The call was interrupted by a signal before any data was written, try again:*/
00451             res=write(fd,buf,count);
00452     return (res);
00453 }/*writeFromb*/
00454
00455 /* Read one (binary) PID from the file descriptor fd:*/
00456 static FORM_INLINE pid_t readpid(int fd)
00457 {
00458     pid_t tmp;
00459     if(read2b(fd, (char*)&tmp, sizeof(pid_t))!=sizeof(pid_t))
00460         return (pid_t)-1;
00461     return tmp;
00462 }/*readpid*/
00463
00464 /* Write one (binary) PID to the file descriptor fd:*/
00465 static FORM_INLINE pid_t writepid(int fd, pid_t thepid)
00466 {
00467     if(writeFromb(fd, (char*)&thepid, sizeof(pid_t))!=sizeof(pid_t))
00468         return (pid_t)-1;
00469     return (pid_t)0;
00470 }/*writepid*/
00471
00472 /*Writes exactly count bytes from the buffer buf into the descriptor fd, independently on
00473 nonblocked signals and the MPU/buffer hits. Returns 0 or -1:
00474 */
00475 static FORM_INLINE int writexactly(int fd, char *buf, size_t count)
00476 {
00477     ssize_t i;
00478     int j=0,n=0;
00479
00480     for(;;){
00481         if( (i=writeFromb(fd, buf+j, count-j)) < 0 ) return(-1);
00482         j+=i;
00483         if ( ((size_t)j) == count ) break;
00484         if(i==0)n++;
00485         else n=0;
00486         if(n>MAX_FAILS_IO)return (-1);
00487     }/*for(;;)*/
00488     return (0);
00489 }/*writexactly*/
00490
00491 /* Set the FD_CLOEXEC flag of desc if value is nonzero,
00492 or clear the flag if value is 0.

```

```

00493 Return 0 on success, or -1 on error with errno set. */
00494 static int set_cloexec_flag(int desc, int value)
00495 {
00496     int oldflags = fcntl (desc, F_GETFD, 0);
00497     /* If reading the flags failed, return error indication now.*/
00498     if (oldflags < 0)
00499         return (oldflags);
00500     /* Set just the flag we want to set. */
00501     if (value != 0)
00502         oldflags |= FD_CLOEXEC;
00503     else
00504         oldflags &= ~FD_CLOEXEC;
00505     /* Store modified flag word in the descriptor. */
00506     return (fcntl(desc, F_SETFD, oldflags));
00507 } /*set_cloexec_flag*/
00508
00509 /* Adds the integer fd to the array fifo of length top+1 so that
00510 the array is ascendantly ordered. It is supposed that all 0 -- top-1
00511 elements in the array are already ordered:*/
00512 static void pushDescriptor(int *fifo, int top, int fd)
00513 {
00514     if ( top == 0 ) {
00515         fifo[top] = fd;
00516     } else {
00517         int ins=top-1;
00518         if( fifo[ins]<=fd )
00519             fifo[top]=fd;
00520         else{
00521             /*Find the position:*/
00522             while( (ins>=0)&&(fifo[ins]>fd) )ins--;
00523             /*Move all elements starting from the position to the right:*/
00524             for(ins++;top>ins; top--)
00525                 fifo[top]=fifo[top-1];
00526             /*Put the element:*/
00527             fifo[ins]=fd;
00528         }
00529     }
00530 } /*pushDescriptor*/
00531
00532 /*Close all descriptors greater or equal than startFrom except those
00533 listed in the ascendantly ordered array usedFd of length top:*/
00534 static FORM_INLINE void closeAllDescriptors(int startFrom, int *usedFd, int top)
00535 {
00536     int n,maxfd;
00537     for(n=0;n<top; n++){
00538         maxfd=usedFd[n];
00539         for(;startFrom<maxfd;startFrom++)/*Close all less than maxfd*/
00540             close(startFrom);
00541         startFrom++;/*skip maxfd*/
00542     }/*for(;startFrom<maxfd;startFrom++)*/
00543     /*Close all the rest:*/
00544     maxfd=sysconf(_SC_OPEN_MAX);
00545     for(;startFrom<maxfd;startFrom++)
00546         close(startFrom);
00547 } /*closeAllDescriptors*/
00548
00549 typedef int L_APIPE[2];
00550 /*Closes both pipe descriptors if not -1:*/
00551 static void closepipe(L_APIPE *thepipe)
00552 {
00553     if( (*thepipe)[0] != -1) close ((*thepipe)[0]);
00554     if( (*thepipe)[1] != -1) close ((*thepipe)[1]);
00555 } /*closepipe*/
00556
00557 /*Parses the cmd line like "sh -c myprg", passes each option to the
00558 corresponding element of argv, ends agrv by NULL. Returns the
00559 number of stored argv elements, or -1 if fails:*/
00560 static FORM_INLINE int parseline(char **argv, char *cmd)
00561 {
00562     int n=0;
00563     while(*cmd != '\0'){
00564         for(; (*cmd <= ' ') && (*cmd != '\0') ;cmd++);
00565         if(*cmd != '\0'){
00566             argv[n]=cmd;
00567             while(++cmd > ' ');
00568             if(*cmd != '\0')
00569                 *cmd++ = '\0';
00570             n++;
00571         }/*if(*cmd != '\0')*/
00572     }/*while(*cmd != '\0')*/
00573     argv[n]=NULL;
00574     if(n==0)return -1;
00575     return n;
00576 } /*parseline*/
00577
00578 /*Reads positive decimal number (not bigger than maxnum)
00579 from the string and returns it;

```

```

00580  the pointer *b is set to the next non-converted character:*/
00581 static LONG str2i(char *str, char **b, LONG maxnum)
00582 {
00583     LONG n=0;
00584     /*Eat trailing spaces:*/
00585     while(*str<=' '){if(*str++ == '\0')return(-1);
00586     (*b)=str;
00587     while (*str>='0'&&*str<='9')
00588         if( (n=10*n + *str++ - '0')>maxnum )
00589             return(-1);
00590     if ((*b)==str)/*No single number!*/
00591         return(-1);
00592     (*b)=str;
00593     return(n);
00594 }
00595
00596 /*Converts long integer to a decimal representation.
00597 For portability reasons we cannot use LongCopy from tools.c
00598 since theoretically LONG may be smaller than pid_t:*/
00599 static char *l2s(LONG x, char *to)
00600 {
00601     char *s;
00602     int i = 0, j;
00603     s = to;
00604     do { *s++ = (x % 10)+'0'; i++; } while ( ( x /= 10 ) != 0 );
00605     *s-- = '\0';
00606     j = ( i - 1 ) >> 1;
00607     while ( j >= 0 ) {
00608         i = to[j]; to[j] = s[-j]; s[-j] = (char)i; j--;
00609     }
00610     return(s+1);
00611 }
00612
00613 /*like strcat() but returns the pointer to the end of the
00614 resulting string:*/
00615 static FORM_INLINE char *addStr(char *to, char *from)
00616 {
00617     while( (*to++ = *from++)!='\0' );
00618     return(to-1);
00619 }/*addStr*/
00620
00621
00622 /*Try to write (atomically) short buffer (of length count) to fd.
00623 timeout is a timeout in millisecs. Returns number of written bytes or -1:*/
00624 static FORM_INLINE ssize_t writeSome(int fd, char *buf, size_t count, int timeout)
00625 {
00626     ssize_t res = 0;
00627     fd_set wfds;
00628     struct timeval tv;
00629     int nrep=5;/*five attempts it interrupted by a non-blocking signal*/
00630     int flags = fcntl(fd, F_GETFL,0);
00631
00632     /*Add O_NONBLOCK:*/
00633     fcntl(fd,F_SETFL, flags | O_NONBLOCK);
00634     /* important -- in order to avoid blocking of short receiver buffer*/
00635
00636     do{
00637         FD_ZERO(&wfds);
00638         FD_SET(fd, &wfds);
00639         /* Wait up to timeout. */
00640
00641         tv.tv_sec =timeout /1000;
00642         tv.tv_usec = (timeout % 1000)*1000;
00643         nrep--;
00644
00645         switch(select(fd+1, NULL, &wfds, NULL, &tv)){
00646             case -1:
00647                 if((nrep == 0)|| ( errno != EINTR) ){
00648                     perror("select()");
00649                     res=-1;
00650                     nrep=0;
00651                 }/*else -- A non blocked signal was caught, just repeat*/
00652                 break;
00653             case 0:/*timeout*/
00654                 res=-1;
00655                 nrep=0;
00656                 break;
00657             default:
00658                 if( (res=write(fd,buf,count)) <0 )/*Signal?*/
00659                     while( (errno == EINTR)&&(res <0) )
00660                         res=write(fd,buf,count);
00661                 nrep=0;
00662             }/*switch*/
00663         }while(nrep);
00664
00665         /*restore the flags:*/
00666         fcntl(fd,F_SETFL, flags);

```

```

00667     return (res);
00668 }/*writeSome*/
00669
00670 /*Try to read short buffer (of length not more than count)
00671  from fd. timeout is a timeout in millisecs. Returns number
00672  of written bytes or -1: */
00673 static FORM_INLINE ssize_t readSome(int fd, char *buf, size_t count, int timeout)
00674 {
00675     ssize_t res = 0;
00676     fd_set rfd;
00677     struct timeval tv;
00678     int nrep=5; /*five attempts it interrupted by a non-blocking signal*/
00679
00680     do{
00681         FD_ZERO(&rfd);
00682         FD_SET(fd, &rfd);
00683         /* Wait up to timeout. */
00684
00685         tv.tv_sec = timeout/1000;
00686         tv.tv_usec = (timeout % 1000)*1000;
00687         nrep--;
00688
00689         switch(select(fd+1, &rfd, NULL, NULL, &tv)){
00690             case -1:
00691                 if((nrep == 0) || (errno != EINTR) ){
00692                     perror("select()");
00693                     res=-1;
00694                     nrep=0;
00695                     /*else -- A non blocked signal was caught, just repeat*/
00696                     break;
00697                 case 0: /*timeout*/
00698                     res=-1;
00699                     nrep=0;
00700                     break;
00701                 default:
00702                     if( (res=read(fd,buf,count)) <0 ) /*Signal?*/
00703                         while( (errno == EINTR)&&(res <0) )
00704                             res=read(fd,buf,count);
00705                     nrep=0;
00706                 } /*switch*/
00707             } while(nrep);
00708         return (res);
00709 }/*readSome*/
00710
00711 /*
00712  #] Local functions :
00713  #[ Ok functions:
00714  */
00715
00716 /*Copies (deep copy) newTerminator to thehandler->terminator. Returns 0 if
00717  newTerminator fits to the buffer, or !0 if it does not fit. ATT! In the
00718  latter case thehandler->terminator is NOT '\0' terminated! */
00719 int setTerminatorForExternalChannelOk(char *newTerminator)
00720 {
00721     int i=DELTA_EXT_BUF;
00722     /*
00723      No problems with externalChannelsListTop are
00724      possible since this function may be invoked only
00725      when the current channel is defined and externalChannelsListTop
00726      is set properly
00727      */
00728     char *t=externalChannelsListTop->terminator;
00729
00730     for( ; i>1; i-- )
00731         if( (*t++ = *newTerminator++)=='\0' )
00732             break;
00733     /*Add trailing '\n', if absent:*/
00734     if( (i == DELTA_EXT_BUF) /*newTerminator == '\0'*/
00735         || (*t-2)!='\n' ) {
00736         *(t-1)='\n'; *t='\0';
00737     }
00738     return (i==1);
00739 }/*setTerminatorForExternalChannelOk*/
00740
00741 /*Interface to change handler fields "killSignal" and "gpId"*/
00742 int setKillModeForExternalChannelOk(int signum, int sentToWholeGroup)
00743 {
00744     if(signum<0)
00745         return(-1);
00746     /*
00747      No problems with externalChannelsListTop are
00748      possible since this function may be invoked only
00749      when the current channel is defined and externalChannelsListTop
00750      is set properly
00751      */
00752     externalChannelsListTop->killSignal=signum;
00753     if(sentToWholeGroup){ /*gpId must be >0*/

```

```

00754         if(externalChannelsListTop->gpid <= 0)
00755             externalChannelsListTop->gpid=-externalChannelsListTop->gpid;
00756     }else{/*gpid must be <=0*/
00757         if(externalChannelsListTop->gpid>0)
00758             externalChannelsListTop->gpid=-externalChannelsListTop->gpid;
00759     }
00760     return(0);
00761 }/*setKillModeForExternalChannelOk*/
00762
00763 /*
00764     #[] getcFromExtChannelOk
00765 */
00766 /*Returns one character from the external channel. If the input is expired,
00767 returns EOF. If the external process is finished completely, the function closes
00768 the channel (and returns EOF). If the external process was finished, the function
00769 returns EOF:*/
00770 int getcFromExtChannelOk(void)
00771 {
00772     mysighandler_t oldPIPE = 0;
00773     EXTHANDLE *h;
00774     int ret;
00775
00776     if (externalChannelsListTop->IBfill < externalChannelsListTop->IBfull)
00777         /*in buffer*/
00778         return( *(externalChannelsListTop->IBfill++) );
00779     /*else -- the buffer is empty*/
00780     ret=EOF;
00781     h= externalChannelsListTop;
00782 #ifdef WITHMPI
00783     if ( PF.me == MASTER ){
00784 #endif
00785         /* Temporary ignore this signal:*/
00786         /* if compiler fails here, try to change the definition of
00787         mysighandler_t on the beginning of this file
00788         (just define INTSIGHANDLER).*/
00789         oldPIPE=signal(SIGPIPE,SIG_IGN);
00790 #ifdef WITHMPI
00791         if( fgets(h->INbuf,h->IBstop - h->INbuf, h->frec) == 0 )/*Fail! EOF?*/
00792             *(h->INbuf)='\0';/*Empty line may not appear here!*/
00793         #else
00794         if( fgets(h->INbuf,h->IBstop - h->INbuf, h->frec) == 0)/*Fail! EOF?*/
00795             ||( *(h->INbuf) == '\0')/*Empty line? This shouldn't be!*/
00796         ){
00797             closeExternalChannel(externalChannelsListTop-externalChannelsList+1);
00798             /*Note, this code is only for the sequential mode! */
00799             goto getcFromExtChannelReady;
00800             /*Here we assume that fgets is never interrupted by singals*/
00801         }/*if( fgets(h->INbuf,h->IBstop - h->INbuf, h->frec) == 0 )*/
00802         #endif
00803 #ifdef WITHMPI
00804         }/*if ( PF.me == MASTER */
00805
00806         /*Master broadcasts result to slaves, slaves read it from the master:*/
00807         if( PF_BroadcastString((UBYTE *)h->INbuf) ){/*Fail!*/
00808             MesPrint("Fail broadcasting external channel results");
00809             Terminate(-1);
00810         }/*if( PF_BroadcastString((UBYTE *)h->INbuf) )*/
00811
00812         if( *(h->INbuf) == '\0')/*Empty line? This shouldn't be!*/
00813             closeExternalChannel(externalChannelsListTop-externalChannelsList+1);
00814         goto getcFromExtChannelReady;
00815         }/*if( *(h->INbuf) == '\0')*/
00816 #endif
00817
00818     {/*Block*/
00819         char *t=h->terminator;
00820         /*Move IBfull to the end of read line and compare the line with the terminator.
00821         Note, by construction the terminator fits to the first read line, see
00822         the function setTerminatorForExternalChannel.*/
00823         for(h->IBfull=h->INbuf; *(h->IBfull)!='\0'; (h->IBfull)++)
00824             if( *t== *(h->IBfull) )
00825                 t++;
00826             else
00827                 break;/*not a terminator*/
00828         /*Continue moving IBfull to the end of read line:*/
00829         while( *(h->IBfull)!='\0') (h->IBfull)++;
00830
00831         if( (t-h->terminator) == (h->IBfull-h->INbuf) ){
00832             /*Terminator!*/
00833             /*Reset the channel*/
00834             h->IBfull=h->IBfill=h->INbuf;
00835             externalChannelsListTop=0;/*Undefine the current channel*/
00836             writeBufToExtChannel=&writeBufToExtChannelFailure;
00837             getcFromExtChannel=&getcFromExtChannelFailure;
00838             setTerminatorForExternalChannel=&setTerminatorForExternalChannelFailure;
00839             setKillModeForExternalChannel=&setKillModeForExternalChannelFailure;
00840             goto getcFromExtChannelReady;

```

```

00841     }/*if(t == (h->IBfull-h->INbuf) )*/
00842 }/*Block*/
00843 /*Does the buffer have enough capacity?*/
00844 while( *(h->IBfull - 1) != '\n' ){/*Buffer is not enough!*/
00845     /*Extend the buffer:*/
00846     int l= (h->IBstop - h->INbuf)+DELTA_EXT_BUF;
00847     char *newbuf=Malloc1(l,"External channel buffer");
00848     /*We wouldn't like to use realloc.*/
00849     /*Copy the buffer:*/
00850     char *n=newbuf,*o=h->INbuf;
00851     while( (*n++ = *o++)!='\0' );
00852     /*Att! The order of the following operators is important!*/
00853     h->IBfull= newbuf+(h->IBfull-h->INbuf);
00854     M_free(h->INbuf,"External channel buffer");
00855     h->INbuf = newbuf;
00856     h->IBstop = h->INbuf+l;
00857 #ifdef WITHMPI
00858     if ( PF.me == MASTER ){
00859         (h->IBfull)[1]='\0';/*Will mark (h->IBfull)[1] as '!' for failure*/
00860         if( fgets(h->IBfull,h->IBstop - h->IBfull, h->frec) == 0 ){
00861             /*EOF! No trailing '\n'?*/
00862             /*Mark:*/
00863             (h->IBfull)[0]='\0';
00864             (h->IBfull)[1]='!';
00865             (h->IBfull)[2]='\0';
00866             /*The string "\0!\0" is used as an image of NULL.*/
00867             }/*if( fgets(h->IBfull,h->IBstop - h->IBfull, h->frec) == 0 )*/
00868         }/*if ( PF.me == MASTER )*/
00869
00870         /*Master broadcasts results to slaves, slaves read it from the master:*/
00871         if( PF_BroadcastString((UBYTE *)h->IBfull) ){/*Fail!*/
00872             MesPrint("Fail broadcasting external channel results");
00873             Terminate(-1);
00874         }/*if( PF_BroadcastString(h->IBfull) )*/
00875
00876         /*The string "\0!\0" is used as the image of NULL.*/
00877         if(
00878             ( (h->IBfull)[0]=='\0' )
00879             &&( (h->IBfull)[1]=='!' )
00880             &&( (h->IBfull)[2]=='\0' )
00881             )/*EOF! No trailing '\n'?*/
00882             break;
00883     #else
00884         if( fgets(h->IBfull,h->IBstop - h->IBfull, h->frec) == 0 )
00885             /*EOF! No trailing '\n'?*/
00886             break;
00887     #endif
00888     while( *(h->IBfull)!='\0' ) (h->IBfull)++;
00889 }/*while( *(h->IBfull - 1) != '\n' )*/
00890 /*In h->INbuf we have a fresh string.*/
00891 ret=*(h->IBfill=h->INbuf);
00892 h->IBfill++;/*Next time a new, isn't it?*/
00893
00894 getcFromExtChannelReady:
00895 #ifdef WITHMPI
00896     if ( PF.me == MASTER ){
00897 #endif
00898         signal(SIGPIPE,oldPIPE);
00899 #ifdef WITHMPI
00900     }/*if ( PF.me == MASTER )*/
00901 #endif
00902     return(ret);
00903 }/*getcFromExtChannelOk*/
00904 /*
00905     #] getcFromExtChannelOk
00906 */
00907 /*Writes exactly count bytes from the buffer buf to the external channel thehandler
00908 Returns 0 (on success) or -1:
00909 */
00910 int writeBufToExtChannelOk(char *buf, size_t count)
00911 {
00912     int ret;
00913     mysighandler_t oldPIPE;
00914 #ifdef WITHMPI
00915     /*Only master communicates with the external program:*/
00916     if ( PF.me == MASTER ){
00917 #endif
00918         /* Temporary ignore this signal:*/
00919         /* if compiler fails here, try to change the definition of
00920            mysighandler_t on the beginning of this file
00921            (just define INTSIGHANDLER)*/
00922         oldPIPE=signal(SIGPIPE,SIG_IGN);
00923         ret=writeexactly( externalChannelsListTop->fsend, buf, count);
00924         signal(SIGPIPE,oldPIPE);
00925 #ifdef WITHMPI
00926     }else{

```

```

00928         /*Do not wait the master status: this would be too slow!*/
00929         ret=0;
00930     }
00931 #endif
00932     return(ret);
00933 }/*writeBufToExtChannel*/
00934 /*
00935     #] Ok functions:
00936     #[ do_run_cmd :
00937 */
00938 /*The function returns PID of the started command*/
00939 static FORM_INLINE pid_t do_run_cmd(
00940     int *fdsend,
00941     int *fdreceive,
00942     int *gpid, /*returns group process ID*/
00943     int tty mode,
00944     /*
00945         &8 - daemonizeing
00946         &16 - setsid()*/
00947     char *cmd,
00948     char *argv[],
00949     char *stderrname
00950 )
00951 {
00952     int fdin[2]={-1,-1}, fdout[2]={-1,-1}, fdsig[2]={-1,-1};
00953     /*initialised by -1 for possible rollback at failure, see closepipe() above*/
00954     pid_t childpid, fatherchildpid = (pid_t)0;
00955     mysighandler_t oldPIPE=NULL;
00956     if(
00957         (pipe(fdsig)!=0)/*This pipe will be used by a child to tell the father if fail.*/
00958         || (pipe(fdin)!=0)
00959         || (pipe(fdout)!=0)
00960     )goto fail_do_run_cmd;
00961     if((childpid = fork()) == -1){
00962         perror("fork");
00963         goto fail_do_run_cmd;
00964     }/*if((childpid = fork()) == -1)*/
00965     if(childpid == 0){/*Child.*/
00966         int fifo[3], top=0;
00967         /*
00968             To be thread safely we can't rely on ascendant order of opened
00969             file descriptors. So we put each of descriptor we have to
00970             preserve into the array fifo.
00971             Note, in _this_ process there are no any threads but descriptors
00972             were created in frame of the parent process which may have
00973             multiple threads.
00974         */
00975         /*Mark descriptors which will NOT be closed:*/
00976         pushDescriptor(fifo,top++,fdsig[1]);
00977         pushDescriptor(fifo,top++,fdin[0]);
00978         pushDescriptor(fifo,top++,fdout[1]);
00979         /*Close all except stdin, stdout, stderr and placed into fifo:*/
00980         closeAllDescriptors(3,fifo, top);
00981         /*Now reopen stdin and stdout.*/
00982         /*thread-safety is not a problem here since there are no any threads up to now:*/
00983         if(
00984             (close(0) == -1 )||/* Use fdin as stdin :*/
00985             (dup(fdin[0]) == -1 )||
00986             (close(1)==-1)||/* Use fdout as stdout:*/
00987             (dup(fdout[1]) == -1 )
00988         )
00989         {/*Fail!*/
00990             /*Signal to parent:*/
00991             writepid(fdsig[1],(pid_t)-2);
00992             _exit(1);
00993         }
00994         if(stderrname != NULL){
00995             if(
00996                 (close(2) != 0 )||
00997                 (open(stderrname,O_WRONLY)<0)
00998             )
00999             {/*Fail!*/
01000                 writepid(fdsig[1],(pid_t)-2);
01001                 _exit(1);
01002             }
01003             /*if(stderrname != NULL)*/
01004             if( tty mode & 16 )/* create a session and sets the process group ID */
01005                 setsid();
01006             /* */

```



```

01015     if(set_cloexec_flag (fdsig[1], 1)!=0){/*Error?*/
01016         /*Signal to parent:*/
01017         writepid(fdsig[1],(pid_t)-2);
01018         _exit(1);
01019     }/*if(set_cloexec_flag (fdsig[1], 1)!=0)*/
01020
01021     if( ttymode & 8 ){/*Daemonize*/
01022         int fdsig2[2];/*To check exec() success*/
01023         if(
01024             pipe(fdsig2)||
01025             (set_cloexec_flag (fdsig2[1], 1)!=0)
01026         )
01027             {/*Error?*/
01028                 /*Signal to parent:*/
01029                 writepid(fdsig[1],(pid_t)-2);
01030                 _exit(1);
01031             }
01032         set_cloexec_flag (fdsig2[0], 1);
01033         switch(childpid=fork()){
01034             case 0:/*grandchild*/
01035                 /*Execute external command:*/
01036                 execvp(cmd, argv);
01037                 /* Control can reach this point only on error!*/
01038                 writepid(fdsig2[1],(pid_t)-2);
01039                 break;
01040             case -1:
01041                 /* Control can reach this point only on error!*/
01042                 /*Inform the father about the failure*/
01043                 writepid(fdsig[1],(pid_t)-2);
01044                 _exit(1);/*The child, just exit, not return*/
01045             default:/*Son of his father*/
01046                 close(fdsig2[1]);
01047                 /*Ignore SIGPIPE (up to the end of the process):*/
01048                 signal(SIGPIPE,SIG_IGN);
01049
01050                 /*Wait on read() while the grandchild close the pipe
01051                 (on success) or send -2 (if exec() fails).*/
01052                 /*There are two possibilities:
01053                 -1 -- this is ok, the pipe was closed on exec,
01054                 the program was successfully executed;
01055                 -2 -- something is wrong, exec failed since the
01056                 grandchild sends -2 after exec.
01057                 */
01058                 if( readpid(fdsig2[0]) != (pid_t)-1 )/*something is wrong*/
01059                     writepid(fdsig[1],(pid_t)-1);
01060                 else/*ok, send PID of the grandchild to the father:*/
01061                     writepid(fdsig[1],childpid);
01062                 /*Die and free the life space for the grandchild:*/
01063                 _exit(0);/*The child, just exit, not return*/
01064             }/*switch(childpid=fork())*/
01065         }else{/*if( ttymode & 8 )*/
01066             execvp(cmd, argv);
01067             /* Control can reach this point only on error!*/
01068             writepid(fdsig[1],(pid_t)-2);
01069             _exit(2);/*The child, just exit, not return*/
01070         }/*if( ttymode & 8 )...else*/
01071     }else{/* The (grand)father*/
01072         close(fdsig[1]);
01073         /*To prevent closing fdsig in rollback:*/
01074         fdsig[1]=-1;
01075         close(fdin[0]);
01076         close(fdout[1]);
01077         *fdsend = fdin[1];
01078         *fdreceive = fdout[0];
01079
01080         /*Get the process group ID.*/
01081         /*Avoid to use getpgid() which is non-standard.*/
01082         if( ttymode & 16){/*setsid() was invoked, the child is a group leader:*/
01083             *gpid=childpid;
01084         }else/*the child belongs to the same process group as the this process:*/
01085             *gpid=getpgrp();/*if compiler fails here, try getpgrp(0) instead!*/
01086         /*
01087         Rationale: getpgrp conform to POSIX.1 while 4.3BSD provides a
01088         getpgrp() function that returns the process group ID for a
01089         specified process.
01090         */
01091
01092         /* Temporary ignore this signal:*/
01093         /* if compiler fails here, try to change the definition of
01094         mysighandler_t on the beginning of this file
01095         (just define INTSIGHANDLER)*/
01096         oldPIPE=signal(SIGPIPE,SIG_IGN);
01097
01098         if( ttymode & 8 ){/*Daemonize*/
01099             /*Read the grandchild PID from the son.*/
01100             fatherchildpid=childpid;
01101             if( (childpid=readpid(fdsig[0]))<0 ){

```

```

01102             /*Daemonization process fails for some reasons!*/
01103             childpid=fatherchildpid; /*for rollback*/
01104             goto fail_do_run_cmd;
01105         }
01106     }else{
01107         /*fdsig[1] should be closed on exec and this read operation
01108         must fail on success:*/
01109         if( readpid(fdsig[0])!= (pid_t)-1 )
01110             goto fail_do_run_cmd;
01111         /*if( ttymode & 8 ) ... else*/
01112     }/*if(childpid == 0)...else*/
01113
01114     /*Here can be ONLY the father*/
01115     close(fdsig[0]);
01116     /*To prevent closing fdsig in rollback after goto fail_flnk_do_runcmd:*/
01117     fdsig[0]=-1;
01118
01119     if( ttymode & 8 ){ /*Daemonize*/
01120         int i;
01121         /*Wait while the father of a grandchild dies:*/
01122         waitpid(fatherchildpid,&i,0);
01123     }
01124
01125     /*Restore the signal:*/
01126     signal(SIGPIPE,oldPIPE);
01127
01128     return(childpid);
01129
01130     fail_do_run_cmd:
01131         closepipe(&fdout);
01132         closepipe(&fdin);
01133         closepipe(&fdsig);
01134         return((pid_t)-1);
01135 }/*do_run_cmd*/
01136 /*
01137 #] do_run_cmd :
01138 #[ run_cmd :
01139 */
01140 /*Starts the command cmd (directly, if shellpath is NULL, or in a subshell),
01141 swallowing its stdin and stdout;
01142 stderr will be re-directed to stderrname (if !=NULL). Returns PID of
01143 the started process. Stdin will be available as fdsend, and stdout
01144 will be available as fdreceive:*/
01145 static FORM_INLINE pid_t run_cmd(char *cmd,
01146                                 int *fdsend,
01147                                 int *fdreceive,
01148                                 int *gpipid,
01149                                 int daemonize,
01150                                 char *shellpath,
01151                                 char *stderrname )
01152 {
01153     char **argv;
01154     pid_t thepid;
01155
01156     cmd=(char*)strDup1((UBYTE*)cmd, "run_cmd: cmd"); /*detouch cmd*/
01157
01158
01159     /* Prepare arguments for execvp:*/
01160     if(shellpath != NULL){ /*Run in a subshell.*/
01161         int nopt;
01162         /*Allocate space which is definitely enough:*/
01163         argv=Malloc1(StrLen((UBYTE*)shellpath)*sizeof(char*)+2,"run_cmd:argv");
01164         shellpath=(char*)strDup1((UBYTE*)shellpath, "run_cmd: shellpath"); /*detouch shellpath*/
01165         /*Parse a shell (e.g., "/bin/sh -c"):*/
01166         nopt=parseline(argv, shellpath);
01167         /* and add the command as a shell argument:*/
01168         argv[nopt]=cmd;
01169         argv[nopt+1]=NULL;
01170     }else{ /*Run the command directly:*/
01171         /*Allocate space which is definitely enough:*/
01172         argv=Malloc1(StrLen((UBYTE*)cmd)*sizeof(char*)+1,"run_cmd:argv");
01173         parseline(argv, cmd);
01174     }
01175
01176     thepid=do_run_cmd(
01177         fdsend,
01178         fdreceive,
01179         gpipid,
01180         (daemonize)?(8|16):0,
01181         argv[0],
01182         argv,
01183         stderrname
01184     );
01185
01186     M_free(argv,"run_cmd:argv");
01187     if(shellpath)
01188         M_free(shellpath,"run_cmd:argv");

```

```

01189     M_free(cmd, "run_cmd: cmd");
01190
01191     return(theppid);
01192 }/*run_cmd*/
01193 /*
01194     #] run_cmd :
01195     #[ createExternalChannel :
01196 */
01197 /*The structure to pass parameters to createExternalChannel
01198 and openExternalChannel in case of preset channel (instead of
01199 shellname):*/
01200 typedef struct{
01201     int fdin;
01202     int fdout;
01203     pid_t theppid;
01204 }ECINFOSTRUCT;
01205
01206 /* Creates a new external channel starting the command cmd (if cmd !=NULL)
01207 or using information from (ECINFOSTRUCT *)shellname, if cmd ==NULL:*/
01208 static FORM_INLINE void *createExternalChannel(
01209     EXTHANDLE *h,
01210     char *cmd, /*Command to run or NULL*/
01211     /*0 --neither setuid nor daemonize, !=0 -- full daemonization:*/
01212     int daemonize,
01213     char *shellname,/* The shell (like "/bin/sh -c") or NULL*/
01214     char *stderrname/*filename to redirect stderr or NULL*/
01215 )
01216 {
01217     int fdreceive=0;
01218     int gpid = 0;
01219     ECINFOSTRUCT *psetInfo;
01220 #ifdef WITHMPI
01221     char statusbuf[2]={'\0','\0'}; /*'\0' if run_cmd returns ok, '!' otherwise.*/
01222 #endif
01223     extHandlerInit(h);
01224
01225     h->pid=0;
01226
01227     if( cmd==NULL ){/*Instead of starting a new command, use preset channel:*/
01228         psetInfo=(ECINFOSTRUCT *)shellname;
01229         shellname=NULL;
01230         h->killSignal=0;
01231         h->daemonize=0;
01232     }
01233
01234     /*Create a channel:*/
01235 #ifdef WITHMPI
01236     if ( PF.me == MASTER ){
01237 #endif
01238         if(cmd!=NULL)
01239             h->pid=run_cmd (cmd, &(h->fsend),
01240                             &fdreceive,&gpid,daemonize,shellname,stderrname);
01241         else{
01242             gpid=-psetInfo->theppid;
01243             h->pid=psetInfo->theppid;
01244             h->fsend=psetInfo->fdout;
01245             fdreceive=psetInfo->fdin;
01246         }
01247 #ifdef WITHMPI
01248         if(h->pid<0)
01249             statusbuf[0]='!';/*Broadcast fail to slaves*/
01250     }
01251     /*else: Keep h->pid = 0 and h->fsend = 0 for slaves in parallel mode!*/
01252
01253     /*Master broadcasts status to slaves, slaves read it from the master:*/
01254     if( PF_BroadcastString((UBYTE *)statusbuf) ){/*Fail!*/
01255         h->pid=-1;
01256     }else if( statusbuf[0]!='!')/*Master fails*/
01257         h->pid=-1;
01258 #endif
01259
01260     if(h->pid<0)goto createExternalChannelFails;
01261 #ifdef WITHMPI
01262     if ( PF.me == MASTER ){
01263 #endif
01264         h->gpid=gpid;
01265         /*Open stdout of a newly created program as FILE* :*/
01266         if( (h->frec=fdopen(fdreceive,"r")) == 0 )goto createExternalChannelFails;
01267 #ifdef WITHMPI
01268     }
01269 #endif
01270 #endif
01271     /*Initialize buffers:*/
01272     extHandlerAlloc(h,cmd,shellname,stderrname);
01273     return(h);
01274     /*Something is wrong?*/
01275     createExternalChannelFails:

```

```

01276         destroyExternalChannel(h);
01277         return(NULL);
01278     }/*createExternalChannel*/
01280
01281     /*
01282     #] createExternalChannel :
01283     #[ openExternalChannel :
01284     */
01285     int openExternalChannel(UBYTE *cmd, int daemonize, UBYTE *shellname, UBYTE *stderrname)
01286     {
01287         EXTHANDLE *h=externalChannelsListTop;
01288         int i=0;
01289
01290         for (externalChannelsListTop=0;i<externalChannelsListFill;i++)
01291             if (externalChannelsList[i].cmd==0){
01292                 externalChannelsListTop=externalChannelsList+i;
01293                 break;
01294             }/*if (externalChannelsList[i].cmd!=0)*/
01295
01296         if (externalChannelsListTop==0){
01297             /*No free cells, create the new one:*/
01298             if (externalChannelsListFill == externalChannelsListStop){
01299                 /*The list is full, increase and reallocate it:*/
01300                 EXTHANDLE *newbuf=Malloc1(
01301                     (externalChannelsListStop+DELTA_EXT_LIST)*sizeof(EXTHANDLE),
01302                     "External channel list");
01303                 for (i=0;i<externalChannelsListFill;i++)
01304                     extHandlerSwallowCopy(newbuf+i,externalChannelsList+i);
01305                 if (externalChannelsList!=0)
01306                     M_free(externalChannelsList,"External channel list");
01307                 for (;i<externalChannelsListStop;i++)
01308                     extHandlerInit(newbuf+i);
01309                 externalChannelsList=newbuf;
01310             }/*if (externalChannelsListFill == externalChannelsListStop)*/
01311             externalChannelsListTop=externalChannelsList+externalChannelsListFill;
01312             externalChannelsListFill++;
01313         }/*if (externalChannelsListTop==0)*/
01314
01315         if (createExternalChannel(
01316             externalChannelsListTop,
01317             (char*) cmd,
01318             daemonize,
01319             (char*) shellname,
01320             (char*) stderrname
01321             )!=NULL){
01322             writeBufToExtChannel=&writeBufToExtChannelOk;
01323             getcFromExtChannel=&getcFromExtChannelOk;
01324             setTerminatorForExternalChannel=&setTerminatorForExternalChannelOk;
01325             setKillModeForExternalChannel=&setKillModeForExternalChannelOk;
01326             return (externalChannelsListTop-externalChannelsList+1);
01327         }/*else - failure!*/
01328         externalChannelsListTop=h;
01329         return (-1);
01330     }/*openExternalChannel*/
01331
01332     #] openExternalChannel :
01333     #[ initPresetExternalChannels :
01334     */
01335     /*Just simple wrapper to invoke openExternalChannel()
01336     from initPresetExternalChannels():*/
01337     static FORM_INLINE int openPresetExternalChannel(int fdin, int fdout, pid_t theppid)
01338     {
01339         ECINFOSTRUCT inf;
01340         inf.fdin=fdin; inf.fdout=fdout; inf.theppid=theppid;
01341         return ( openExternalChannel(NULL,0,(UBYTE *)&inf,NULL) );
01342     }/*openPresetExternalChannel*/
01343
01344     #define PIDTXTSIZE 23
01345     #define BOTHPIDSIZE 45
01346
01347     #ifndef LONG_MAX
01348     #define LONG_MAX 0x7FFFFFFFL
01349     #endif
01350
01351     /*There is a possibility to start FORM from another program providing
01352     one (or more) communication channels. These channels will be
01353     visible from a FORM program as ``pre-opened" external channels existing
01354     after FORM starts.
01355     Before starting FORM, the parent application must create one or more
01356     pairs of pipes The read-only descriptor of the first pipe in the pair
01357     and the write-only descriptor of the second pipe must be passed to
01358     FORM as an argument of a command line option -pipe in ASCII decimal
01359     format. The argument of the option is a comma-separated list of pairs
01360     r#w# where r# is a read-only descriptor and w# is a write-only
01361     descriptor. Alternatively, the environment variable FORM_PIPES can be
01362     used.

```

```

01363     The following function expects as the first argument
01364 this comma-separated list of the descriptor pairs and tries to
01365 initialize each of channel during thetimeout milliseconds:*/
01366
01367 int initPresetExternalChannels(UBYTE *theline, int thetimeout)
01368 {
01369     int i, nchannels = 0;
01370     int pidln;
01371     /*The length of FORM PID in pidtxt*/
01372     char pidtxt[PIDTXTSIZE], /*64/Log_2[10] = 19.3, this is enough for any integer*/
01373         chdescriptor[PIDTXTSIZE],
01374         bothpidtxt[BOTHPIDSIZE], /*"#,#\n\0"*/
01375         *c,*b = 0;
01376     int pip[2];
01377     pid_t ppid;
01378     if ( theline == NULL ) return(-1);
01379     /*Put into pidtxt PID\n:*/
01380     c = l2s((LONG)getpid(),pidtxt);
01381     *c++='\n';
01382     *c = '\0';
01383     pidln = c-pidtxt;
01384     do {
01385         pip[0] = (int)str2i((char*)theline,&c,0xFFFF);
01386         if( ( pip[0] < 0 ) || ( *c != ',' ) ) goto presetFails;
01387
01388         theline = (UBYTE*)c + 1;
01389         pip[1] = (int)str2i((char*)theline,&c,0xFFFF);
01390         if ( ( pip[1] < 0 ) || ( ( *c != ',' ) && ( *c != '\0' ) ) ) goto presetFails;
01391         theline = (UBYTE *)c + 1;
01392         /*Now we have two descriptors.
01393         According to the protocol, FORM must send to external channel
01394         it's PID with added '\n' and read back two comma-separated
01395         decimals with added '\n'. The first must be repeated FORM PID,
01396         the second must be the parent PID
01397         */
01398         if ( writeSome(pip[1],pidtxt,pidln,thetimeout) != pidln ) goto presetFails;
01399         i = readSome(pip[0],bothpidtxt,BOTHPIDSIZE,thetimeout);
01400         if( ( i < 4 ) /*at least 1,2\n*/
01401             || ( i == BOTHPIDSIZE ) /*No space for trailing '\0'*/
01402         ) goto presetFails;
01403         /*read the FORM PID:*/
01404         ppid = (pid_t)str2i(bothpidtxt,&b,getpid());
01405         if( ( *b != ',' ) || ( ppid != getpid() ) )goto presetFails;
01406         /*read the parent PID:*/
01407         /*The problem is that we do not know the the real type of pid_t.
01408         But long should be enough. On obsolete systems (when LONG_MAX
01409         is not defined) we assume pid_t is 32-bit integer.
01410         This can lead to problem with portability: */
01411         ppid = (pid_t)str2i(b+1,&b, LONG_MAX);
01412         if ( (*b != '\n') || ( ppid<2 ) ) goto presetFails;
01413         i = openPresetExternalChannel(pip[0],pip[1],ppid);
01414         if ( i < 0 ) goto presetFails;
01415         nchannels++;
01416         /*Now use bothpidtxt as a buffer for preprovar, the space is enough:*/
01417         /*"PIPE#" where # is ne order number of the channel:*/
01418         b = l2s(nchannels,addStr(bothpidtxt,"PIPE"));
01419         *b = '_';
01420         b[1] = '\0';
01421         *l2s(i,chdescriptor) = '\0';
01422         PutPreVar((UBYTE*)bothpidtxt, (UBYTE*)chdescriptor,0,0);
01423     } while ( *c != '\0' );
01424     /*Export preprovar "PIPES_":*/
01425     *l2s(nchannels,chdescriptor)='\0';
01426     PutPreVar((UBYTE*)"PIPES_", (UBYTE*)chdescriptor,0,0);
01427
01428     /*success:*/
01429     return (nchannels);
01430
01431 presetFails:
01432     /*Here we assume the descriptors the beginning of the list!*/
01433     for(i=0; i<nchannels; i++)
01434         destroyExternalChannel(externalChannelsList+i);
01435     return(-1);
01436 } /*initPresetExternalChannels*/
01437
01438 #] initPresetExternalChannels :
01439 #[ selectExternalChannel :
01440 */
01441 /*
01442 Accepts the valid external channel descriptor (returned by
01443 openExternalChannel) and returns the descriptor of a previous current
01444 channel (0, if there was no current channel, or -1, if the external
01445 channel descriptor is invalid). If n == 0, the function undefine the
01446 current external channel:
01447 */
01448 int selectExternalChannel(int n)
01449 {

```

```

01450 int ret=0;
01451     if (externalChannelsListTop!=0)
01452         ret=externalChannelsListTop-externalChannelsList+1;
01453
01454     if (--n<0) {
01455         if (n!=-1)
01456             return(-1);
01457         externalChannelsListTop=0;
01458         writeBufToExtChannel=&writeBufToExtChannelFailure;
01459         getCFromExtChannel=&getCFromExtChannelFailure;
01460         setTerminatorForExternalChannel=&setTerminatorForExternalChannelFailure;
01461         setKillModeForExternalChannel=&setKillModeForExternalChannelFailure;
01462         return(ret);
01463     }
01464     if (
01465         (n>=externalChannelsListFill) ||
01466         (externalChannelsList[n].cmd==0)
01467     )
01468         return(-1);
01469
01470     externalChannelsListTop=externalChannelsList+n;
01471     writeBufToExtChannel=&writeBufToExtChannelOk;
01472     getCFromExtChannel=&getCFromExtChannelOk;
01473     setTerminatorForExternalChannel=&setTerminatorForExternalChannelOk;
01474     setKillModeForExternalChannel=&setKillModeForExternalChannelOk;
01475     return(ret);
01476 }/*selectExternalChannel*/
01477 /*
01478     #] selectExternalChannel :
01479     #[ closeExternalChannel :
01480 */
01481
01482 /*
01483 Destroys the opened external channel with the descriptor n. It returns
01484 0 in success, or -1 on failure. If the corresponding external channel
01485 was the current one, the current channel becomes undefined. If n==0,
01486 the function closes the current external channel.
01487 */
01488 int closeExternalChannel(int n)
01489 {
01490     if (n==0)
01491         n=externalChannelsListTop-externalChannelsList;
01492     else
01493         n--;/*Count from 0*/
01494
01495     if (
01496         (n<0) ||
01497         (n>=externalChannelsListFill) ||
01498         (externalChannelsList[n].cmd==0)
01499     )/*No such a channel*/
01500         return(-1);
01501
01502     destroyExternalChannel(externalChannelsList+n);
01503     /*If the current external channel was destroyed, undefine current channel:*/
01504     if (externalChannelsListTop==externalChannelsList+n) {
01505         externalChannelsListTop=NULL;
01506         writeBufToExtChannel=&writeBufToExtChannelFailure;
01507         getCFromExtChannel=&getCFromExtChannelFailure;
01508         setTerminatorForExternalChannel=&setTerminatorForExternalChannelFailure;
01509         setKillModeForExternalChannel=&setKillModeForExternalChannelFailure;
01510     }/*if (externalChannelsListTop==externalChannelsList+n)*/
01511     return(0);
01512 }/*closeExternalChannel*/
01513 /*
01514     #] closeExternalChannel :
01515     #[ closeAllExternalChannels :
01516 */
01517 void closeAllExternalChannels(void)
01518 {
01519     int i;
01520     for (i=0; i<externalChannelsListFill; i++)
01521         destroyExternalChannel(externalChannelsList+i);
01522     externalChannelsListFill=externalChannelsListStop=0;
01523     externalChannelsListTop=NULL;
01524
01525     writeBufToExtChannel=&writeBufToExtChannelFailure;
01526     getCFromExtChannel=&getCFromExtChannelFailure;
01527     setTerminatorForExternalChannel=&setTerminatorForExternalChannelFailure;
01528     setKillModeForExternalChannel=&setKillModeForExternalChannelFailure;
01529
01530     if (externalChannelsList!=NULL) {
01531         M_free(externalChannelsList,"External channel list");
01532         externalChannelsList=NULL;
01533     }
01534 }/*closeAllExternalChannels*/
01535 /*
01536     #] closeAllExternalChannels :

```

```

01537     # [ getExternalChannelPid :
01538     /*
01539     #ifdef SELFTEST
01540     pid_t getExternalChannelPid(void)
01541     {
01542         if (externalChannelsListTop != 0)
01543             return (externalChannelsListTop -> pid);
01544         return (-1);
01545     } /*getExternalChannelPid*/
01546     #endif
01547     /*
01548     # ] getExternalChannelPid :
01549     # [ getCurrentExternalChannel :
01550     /*
01551
01552     int getCurrentExternalChannel(void)
01553     {
01554
01555         if ( externalChannelsListTop != 0 )
01556             return (externalChannelsListTop-externalChannelsList+1);
01557         return (0);
01558     } /*getCurrentExternalChannel*/
01559     /*
01560     # ] getCurrentExternalChannel :
01561     # [ Selftest main :
01562     /*
01563
01564     #ifdef SELFTEST
01565
01566     /*
01567         This is the example of how all these public functions may be used:
01568     */
01569
01570     char buf[1024];
01571     char buf2[1024];
01572
01573     void help(void)
01574     {
01575         printf("String started with a special letter is a command\n");
01576         printf("Known commands are:\n");
01577         printf("H or ? -- this help\n");
01578         printf("Nn<command> -- start a new command\n");
01579         printf("S<command> -- start a new command in a subshell, daemon, stderr>/dev/null\n");
01580         printf("C# -- destroy channel #\n");
01581         printf("R# -- set a new cahhel(number#) as a current one\n");
01582         printf("K#1 #2 -- set signal for kill and kill mode (0 or !=0)\n");
01583         printf(" ^d to quit\n");
01584     } /*help*/
01585
01586     int main (void)
01587     {
01588         int i, j, k, last;
01589         long long sum = 0;
01590
01591         /*openExternalChannel(UBYTE *cmd, int daemonize, UBYTE *shellname, UBYTE *stderrname)*/
01592
01593         help();
01594
01595         printf("Initial channel:%d\n", last=openExternalChannel((UBYTE*)"cat", 0, NULL, NULL));
01596
01597         if ( ( i = setTerminatorForExternalChannel("qu") ) != 0 ) return 1;
01598         printf("Terminator is 'qu'\n");
01599
01600         while ( fgets(buf, 1024, stdin) != NULL ) {
01601             if ( *buf == 'N' ) {
01602                 printf("New channel:%d\n", j=openExternalChannel((UBYTE*)buf+1, 0, NULL, NULL));
01603                 continue;
01604             }
01605             else if ( *buf == 'C' ) {
01606                 int n;
01607                 sscanf(buf+1, "%d", &n);
01608                 printf("Destroy last channel:%d\n", closeExternalChannel(n));
01609                 continue;
01610             }
01611             else if ( *buf == 'R' ) {
01612                 int n = 0;
01613                 sscanf(buf+1, "%d", &n);
01614                 printf("Reopen channel %d:%d\n", n, selectExternalChannel(n));
01615                 continue;
01616             } else if ( *buf == 'K' ) {
01617                 int n=0, g = 0;
01618                 sscanf(buf+1, "%d %d", &n, &g);
01619                 printf("setKillMode %d\n", setKillModeForExternalChannel(n, g));
01620                 continue;
01621             } else if ( *buf == 'S' ) {
01622                 printf("New channel with sh&d&stderr:%d\n",
01623                     j=openExternalChannel((UBYTE*)buf+1, 1, (UBYTE*)" /bin/sh -c", (UBYTE*)" /dev/null"));

```

```

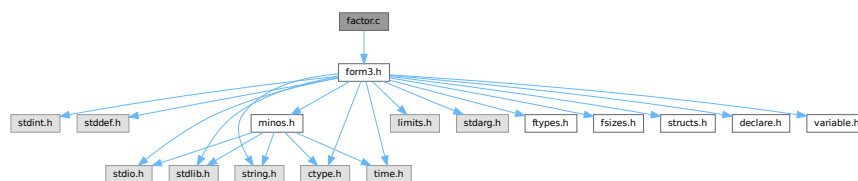
01624         continue;
01625     }
01626     else if( ( *buf == 'H' ) || ( *buf == '?' ) ){
01627         help();
01628         continue;
01629     }
01630
01631     writeBufToExtChannel(buf,k=StrLen(buf));
01632     sum += k;
01633     for ( j = 0; ( i = getcFromExtChannel() ) != '\n'; j++) {
01634         if ( i == EOF ) {
01635             printf("EOF!\n");
01636             break;
01637         }
01638         buf2[j] = i;
01639     }
01640     buf2[j] = '\0';
01641     printf("I got:'%s'; pid=%d\n",buf2,getExternalChannelPid());
01642 }
01643 printf("Total:%lld bytes\n",sum);
01644 closeAllExternalChannels();
01645 return 0;
01646 }
01647 #endif /*ifdef SELFTEST*/
01648 /*
01649 #] Selftest main :
01650 */
01651
01652 #endif /*ifndef WITHEXTERNALCHANNEL ... else*/

```

4.35 factor.c File Reference

#include "form3.h"

Include dependency graph for factor.c:



Functions

- int [FactorIn](#) (PHEAD WORD *term, WORD level)
- int [FactorInExpr](#) (PHEAD WORD *term, WORD level)

4.35.1 Detailed Description

The routines for finding (one term) factors in dollars and expressions.

Definition in file [factor.c](#).

4.35.2 Function Documentation

4.35.2.1 FactorIn()

```

int FactorIn (
    PHEAD WORD * term,
    WORD level )

```

Definition at line 46 of file [factor.c](#).

4.35.2.2 FactorInExpr()

```
int FactorInExpr (
    PHEAD WORD * term,
    WORD level )
```

Definition at line 421 of file [factor.c](#).

4.36 factor.c

[Go to the documentation of this file.](#)

```
00001
00005 /* #[] License : */
00006 /*
00007  * Copyright (C) 1984-2026 J.A.M. Vermaseren
00008  * When using this file you are requested to refer to the publication
00009  * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  * This is considered a matter of courtesy as the development was paid
00011  * for by FOM the Dutch physics granting agency and we would like to
00012  * be able to track its scientific use to convince FOM of its value
00013  * for the community.
00014  *
00015  * This file is part of FORM.
00016  *
00017  * FORM is free software: you can redistribute it and/or modify it under the
00018  * terms of the GNU General Public License as published by the Free Software
00019  * Foundation, either version 3 of the License, or (at your option) any later
00020  * version.
00021  *
00022  * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00025  * details.
00026  *
00027  * You should have received a copy of the GNU General Public License along
00028  * with FORM. If not, see <http://www.gnu.org/licenses/>.
00029 */
00030 /* #[] License : */
00031 /*
00032  #[] Includes : factor.c
00033 */
00034
00035 #include "form3.h"
00036
00037 /*
00038  #[] Includes :
00039  #[] FactorIn :
00040
00041  This routine tests for a factor in a dollar expression.
00042
00043  Note that unlike with regular active or hidden expressions we cannot
00044  add memory as easily as dollars are rather volatile.
00045 */
00046 int FactorIn(PHEAD WORD *term, WORD level)
00047 {
00048     GETBIDENTITY
00049     WORD *t, *tstop, *m, *mm, *oldwork, *mstop, *n1, *n2, *n3, *n4, *n1stop, *n2stop;
00050     WORD *r1, *r2, *r3, *r4, j, k, kGCD, kGCD2, kLCM, jGCD, kkLCM, jLCM, size;
00051     UWORD *GCDBuffer, *GCDBuffer2, *LCMBuffer, *LCMb, *LCMc;
00052     int fromwhere = 0, i;
00053     DOLLARS d;
00054     t = term; GETSTOP(t,tstop); t++;
00055     while ( ( t < tstop ) && ( *t != FACTORIN || ( ( *t == FACTORIN )
00056         && ( t[FUNHEAD] != -DOLLAREXPRESSION || t[1] != FUNHEAD+2 ) ) ) ) t += t[1];
00057     if ( t >= tstop ) {
00058         MLOCK(ErrorMessageLock);
00059         MesPrint("Internal error. Could not find proper factorin_ function.");
00060         MUNLOCK(ErrorMessageLock);
00061         return(-1);
00062     }
00063     oldwork = AT.WorkPointer;
00064     d = Dollars + t[FUNHEAD+1];
00065     #ifndef WITHPTHREADS
00066     {
00067         int nummodopt, dtype = -1;
00068         if ( AS.MultiThreaded ) {
00069             for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
```

```

00070         if ( t[FUNHEAD+1] == ModOptdollars[nummodopt].number ) break;
00071     }
00072     if ( nummodopt < NumModOptdollars ) {
00073         dtype = ModOptdollars[nummodopt].type;
00074         if ( dtype == MODLOCAL ) {
00075             d = ModOptdollars[nummodopt].dstruct+AT.identity;
00076         }
00077     }
00078 }
00079 }
00080 #endif
00081
00082 if ( d->type == DOLTERMS ) {
00083     fromwhere = 1;
00084 }
00085 else if ( ( d = DolToTerms(BHEAD t[FUNHEAD+1]) ) == 0 ) {
00086 /*
00087     The variable cannot convert to an expression
00088     We replace the function by 1.
00089 */
00090     m = oldwork; n1 = term;
00091     while ( n1 < t ) *m++ = *n1++;
00092     n1 = t + t[1]; tstop = term + *term;
00093     while ( n1 < tstop ) *m++ = *n1++;
00094     *oldwork = m - oldwork;
00095     AT.WorkPointer = m;
00096     if ( Generator(BHEAD oldwork,level) ) return(-1);
00097     AT.WorkPointer = oldwork;
00098     return(0);
00099 }
00100 if ( d->where[0] == 0 ) {
00101     if ( fromwhere == 0 ) {
00102         if ( d->factors ) M_free(d->factors,"Dollar factors");
00103         M_free(d,"Dollar in FactorIn_");
00104     }
00105     return(0);
00106 }
00107 /*
00108     Now we have an expression in d->where. Find the symbolic factor that
00109     divides the expression and the numerical factor that makes all
00110     coefficients integer.
00111
00112     For the symbolic factor we make a copy of the first term, and then
00113     go through all terms, scratching in the copy the objects that do not
00114     occur in the terms.
00115 */
00116 m = oldwork;
00117 mm = d->where;
00118 k = *mm - ABS((mm[*mm-1]));
00119 for ( j = 0; j < k; j++ ) *m++ = *mm++;
00120 mstop = m;
00121 *oldwork = k;
00122 /*
00123     The copy is in place. Now search through the terms. Start at the second term
00124 */
00125 mm = d->where + d->where[0];
00126 while ( *mm ) {
00127     m = oldwork+1;
00128     r2 = mm+*mm;
00129     r2 -= ABS(r2[-1]);
00130     r1 = mm+1;
00131     while ( m < mstop ) {
00132         while ( r1 < r2 ) {
00133             if ( *r1 != *m ) {
00134                 r1 += r1[1]; continue;
00135             }
00136         }
00137         /*
00138             Now the various cases
00139             #[ SYMBOL :
00140         */
00141         if ( *m == SYMBOL ) {
00142             n1 = m+2; n1stop = m+m[1];
00143             n2stop = r1+r1[1];
00144             while ( n1 < n1stop ) {
00145                 n2 = r1+2;
00146                 while ( n2 < n2stop ) {
00147                     if ( *n1 != *n2 ) { n2 += 2; continue; }
00148                     if ( n1[1] > 0 ) {
00149                         if ( n2[1] < 0 ) { n2 += 2; continue; }
00150                         if ( n2[1] < n1[1] ) n1[1] = n2[1];
00151                     }
00152                     else {
00153                         if ( n2[1] > 0 ) { n2 += 2; continue; }
00154                         if ( n2[1] > n1[1] ) n1[1] = n2[1];
00155                     }
00156                     break;
00157                 }
00158             }
00159         }
00160     }
}

```

```

00157         if ( n2 >= n2stop ) { /* scratch symbol */
00158             if ( m[1] == 4 ) goto scratch;
00159             m[1] -= 2;
00160             n3 = n1; n4 = n1+2;
00161             while ( n4 < mstop ) *n3++ = *n4++;
00162             *oldwork = n3 - oldwork;
00163             mstop -= 2; n1stop -= 2;
00164             continue;
00165         }
00166         n1 += 2;
00167     }
00168     break;
00169 }
00170 /*
00171 #] SYMBOL :
00172 #[ DOTPRODUCT :
00173 */
00174     else if ( *m == DOTPRODUCT ) {
00175         n1 = m+2; n1stop = m+m[1];
00176         n2stop = r1+r1[1];
00177         while ( n1 < n1stop ) {
00178             n2 = r1+2;
00179             while ( n2 < n2stop ) {
00180                 if ( *n1 != *n2 || n1[1] != n2[1] ) { n2 += 3; continue; }
00181                 if ( n1[2] > 0 ) {
00182                     if ( n2[2] < 0 ) { n2 += 3; continue; }
00183                     if ( n2[2] < n1[2] ) n1[2] = n2[2];
00184                 }
00185                 else {
00186                     if ( n2[2] > 0 ) { n2 += 3; continue; }
00187                     if ( n2[2] > n1[2] ) n1[2] = n2[2];
00188                 }
00189                 break;
00190             }
00191             if ( n2 >= n2stop ) { /* scratch symbol */
00192                 if ( m[1] == 5 ) goto scratch;
00193                 m[1] -= 3;
00194                 n3 = n1; n4 = n1+3;
00195                 while ( n4 < mstop ) *n3++ = *n4++;
00196                 *oldwork = n3 - oldwork;
00197                 mstop -= 3; n1stop -= 3;
00198                 continue;
00199             }
00200             n1 += 3;
00201         }
00202         break;
00203     }
00204 /*
00205 #] DOTPRODUCT :
00206 #[ VECTOR :
00207 */
00208     else if ( *m == VECTOR ) {
00209 /*
00210      Here we have to be careful if there is more than
00211      one of the same
00212 */
00213         n1 = m+2; n1stop = m+m[1];
00214         n2 = r1+2; n2stop = r1+r1[1];
00215         while ( n1 < n1stop ) {
00216             while ( n2 < n2stop ) {
00217                 if ( *n1 == *n2 && n1[1] == n2[1] ) {
00218                     n2 += 2; goto nextn1;
00219                 }
00220                 n2 += 2;
00221             }
00222             if ( n2 >= n2stop ) { /* scratch symbol */
00223                 if ( m[1] == 4 ) goto scratch;
00224                 m[1] -= 2;
00225                 n3 = n1; n4 = n1+2;
00226                 while ( n4 < mstop ) *n3++ = *n4++;
00227                 *oldwork = n3 - oldwork;
00228                 mstop -= 2; n1stop -= 2;
00229                 continue;
00230             }
00231             n2 = r1+2;
00232             n1 += 2;
00233             nextn1:
00234         }
00235         break;
00236 /*
00237 #] VECTOR :
00238 #[ REMAINDER :
00239 */
00240     else {
00241 /*
00242      Again: watch for multiple occurrences of the same object
00243 */

```

```

00244         if ( m[1] != r1[1] ) { r1 += r1[1]; continue; }
00245         for ( j = 2; j < m[1]; j++ ) {
00246             if ( m[j] != r1[j] ) break;
00247         }
00248         if ( j < m[1] ) { r1 += r1[1]; continue; }
00249         r1 += r1[1]; /* to restart at the next potential match */
00250         goto nextm; /* match */
00251     }
00252 /*
00253     #] REMAINDER :
00254 */
00255     }
00256     if ( r1 >= r2 ) { /* no factor! */
00257 scratch:;
00258         r3 = m + m[1]; r4 = m;
00259         while ( r3 < mstop ) *r4++ = *r3++;
00260         *oldwork = r4 - oldwork;
00261         if ( *oldwork == 1 ) goto nofactor;
00262         mstop = r4;
00263         r1 = mm + 1;
00264         continue;
00265     }
00266     r1 = mm + 1;
00267 nextm:
00268     m += m[1];
00269     mm = mm + *mm;
00270 }
00271
00272 nofactor:;
00273 /*
00274     For the coefficient we have to determine the LCM of the denominators
00275     and the GCD of the numerators.
00276 */
00277     GCDbuffer = NumberMalloc("FactorIn"); GCDbuffer2 = NumberMalloc("FactorIn");
00278     LCMbuffer = NumberMalloc("FactorIn"); LCMb = NumberMalloc("FactorIn"); LCMc =
00279     NumberMalloc("FactorIn");
00280     r1 = d->where;
00281 /*
00282     First take the first term to load up the LCM and the GCD
00283 */
00284     r2 = r1 + *r1;
00285     j = r2[-1];
00286     r3 = r2 - ABS(j);
00287     k = REDLENG(j);
00288     if ( k < 0 ) k = -k;
00289     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00290     for ( kGCD = 0; kGCD < k; kGCD++ ) GCDbuffer[kGCD] = r3[kGCD];
00291     k = REDLENG(j);
00292     if ( k < 0 ) k = -k;
00293     r3 += k;
00294     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00295     for ( kLCM = 0; kLCM < k; kLCM++ ) LCMbuffer[kLCM] = r3[kLCM];
00296     r1 = r2;
00297 /*
00298     Now go through the rest of the terms in this dollar buffer.
00299 */
00300     while ( *r1 ) {
00301         r2 = r1 + *r1;
00302         j = r2[-1];
00303         r3 = r2 - ABS(j);
00304         k = REDLENG(j);
00305         if ( k < 0 ) k = -k;
00306         while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00307         if ( ( ( GCDbuffer[0] == 1 ) && ( kGCD == 1 ) ) ) {
00308             /*
00309                 GCD is already 1
00310             */
00311         }
00312         else if ( ( ( k != 1 ) || ( r3[0] != 1 ) ) ) {
00313             if ( GcdLong(BHEAD GCDbuffer, kGCD, (UWORD *)r3, k, GCDbuffer2, &kGCD2) ) {
00314                 goto onerror;
00315             }
00316             kGCD = kGCD2;
00317             for ( i = 0; i < kGCD; i++ ) GCDbuffer[i] = GCDbuffer2[i];
00318         }
00319         else {
00320             kGCD = 1; GCDbuffer[0] = 1;
00321         }
00322         k = REDLENG(j);
00323         if ( k < 0 ) k = -k;
00324         r3 += k;
00325         while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00326         if ( ( ( LCMbuffer[0] == 1 ) && ( kLCM == 1 ) ) ) {
00327             for ( kLCM = 0; kLCM < k; kLCM++ )
00328                 LCMbuffer[kLCM] = r3[kLCM];
00329         }
00330         else if ( ( k != 1 ) || ( r3[0] != 1 ) ) {

```

```

00330         if ( GcdLong(BHEAD LCMbuffer,kLCM, (UWORD *)r3,k,LCMb,&kkLCM) ) {
00331             goto onerror;
00332         }
00333         DivLong((UWORD *)r3,k,LCMb,kkLCM,LCMb,&kkLCM,LCMc,&jLCM);
00334         MulLong(LCMbuffer,kLCM,LCMb,kkLCM,LCMc,&jLCM);
00335         for ( kLCM = 0; kLCM < jLCM; kLCM++ )
00336             LCMbuffer[kLCM] = LCMc[kLCM];
00337     }
00338     else {} /* LCM doesn't change */
00339     r1 = r2;
00340 }
00341 /*
00342 Now put the factor together: GCD/LCM
00343 */
00344 r3 = (WORD *) (GCDbuffer);
00345 if ( kGCD == kLCM ) {
00346     for ( jGCD = 0; jGCD < kGCD; jGCD++ )
00347         r3[jGCD+kGCD] = LCMbuffer[jGCD];
00348     k = kGCD;
00349 }
00350 else if ( kGCD > kLCM ) {
00351     for ( jGCD = 0; jGCD < kLCM; jGCD++ )
00352         r3[jGCD+kGCD] = LCMbuffer[jGCD];
00353     for ( jGCD = kLCM; jGCD < kGCD; jGCD++ )
00354         r3[jGCD+kGCD] = 0;
00355     k = kGCD;
00356 }
00357 else {
00358     for ( jGCD = kGCD; jGCD < kLCM; jGCD++ )
00359         r3[jGCD] = 0;
00360     for ( jGCD = 0; jGCD < kLCM; jGCD++ )
00361         r3[jGCD+kLCM] = LCMbuffer[jGCD];
00362     k = kLCM;
00363 }
00364 j = 2*k+1;
00365 mm = m = oldwork + oldwork[0];
00366 /*
00367 Now compose the new term
00368 */
00369 n1 = term;
00370 while ( n1 < t ) *m++ = *n1++;
00371 n1 += n1[1];
00372 n2 = oldwork+1;
00373 while ( n2 < mm ) *m++ = *n2++;
00374 while ( n1 < tstop ) *m++ = *n1++;
00375 /*
00376 And the coefficient
00377 */
00378 size = term[*term-1];
00379 size = REDLENG(size);
00380 if ( MulRat(BHEAD (UWORD *)tstop,size,(UWORD *)r3,k,
00381             (UWORD *)m,&size) ) goto onerror;
00382 size = INCLENG(size);
00383 k = size < 0 ? -size: size;
00384 m[k-1] = size;
00385 m += k;
00386 *mm = (WORD) (m - mm);
00387 AT.WorkPointer = m;
00388 if ( Generator(BHEAD mm,level) ) goto onerror;
00389 AT.WorkPointer = oldwork;
00390 if ( fromwhere == 0 ) {
00391     if ( d->factors ) M_free(d->factors,"Dollar factors");
00392     M_free(d,"Dollar in FactorIn");
00393 }
00394 NumberFree(GCDbuffer,"FactorIn"); NumberFree(GCDbuffer2,"FactorIn");
00395 NumberFree(LCMbuffer,"FactorIn"); NumberFree(LCMb,"FactorIn"); NumberFree(LCMc,"FactorIn");
00396 return(0);
00397 onerror:
00398 AT.WorkPointer = oldwork;
00399 MLOCK(ErrorMessageLock);
00400 MesCall("FactorIn");
00401 MUNLOCK(ErrorMessageLock);
00402 NumberFree(GCDbuffer,"FactorIn"); NumberFree(GCDbuffer2,"FactorIn");
00403 NumberFree(LCMbuffer,"FactorIn"); NumberFree(LCMb,"FactorIn"); NumberFree(LCMc,"FactorIn");
00404 return(-1);
00405 }
00406
00407 /*
00408 #] FactorIn :
00409 #[ FactorInExpr :
00410
00411 This routine tests for a factor in an active or hidden expression.
00412
00413 The factor from the last call is stored in a cache.
00414 Main problem here is whether the cache is global or private to each thread.
00415 A global cache gives most likely the same traffic jam for the computation
00416 as a local cache. The local cache may stay valid longer.

```

```

00417      In the future we may make it such that threads can look at the cache
00418      of the others, or even whether a certain factor is under construction.
00419  */
00420
00421  int FactorInExpr(PHEAD WORD *term, WORD level)
00422  {
00423      GETBIDENTITY
00424      WORD *t, *tstop, *m, *oldwork, *mstop, *n1, *n2, *n3, *n4, *n1stop, *n2stop;
00425      WORD *r1, *r2, *r3, *r4, j, k, kGCD, kGCD2, kLCM, jGCD, kkLCM, jLCM, size, sign;
00426      WORD *newterm, expr = 0;
00427      WORD olddeferflag = AR.DeferFlag, oldgetfile = AR.GetFile, oldhold = AR.KeptInHold;
00428      WORD newgetfile, newhold;
00429      int i;
00430      EXPRESSIONS e;
00431      FILEHANDLE *file = 0;
00432      POSITION position, oldposition, startposition;
00433      WORD *oldcpointer = AR.CompressPointer;
00434      UWORD *GCDBuffer, *GCDBuffer2, *LCMBuffer, *LCMb, *LCMc;
00435      GCDBuffer = NumberMalloc("FactorInExpr"); GCDBuffer2 = NumberMalloc("FactorInExpr");
00436      LCMBuffer = NumberMalloc("FactorInExpr"); LCMb = NumberMalloc("FactorInExpr"); LCMc =
00437      NumberMalloc("FactorInExpr");
00438      t = term; GETSTOP(t,tstop); t++;
00439      while ( t < tstop ) {
00440          if ( *t == FACTORIN && t[1] == FUNHEAD+2 && t[FUNHEAD] == -EXPRESSION ) {
00441              expr = t[FUNHEAD+1];
00442              break;
00443          }
00444          t += t[1];
00445      }
00446      if ( t >= tstop ) {
00447          MLOCK(ErrorMessageLock);
00448          MesPrint("Internal error. Could not find proper factorin_ function.");
00449          MUNLOCK(ErrorMessageLock);
00450          NumberFree(GCDBuffer,"FactorInExpr"); NumberFree(GCDBuffer2,"FactorInExpr");
00451          NumberFree(LCMBuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
00452          NumberFree(LCMc,"FactorInExpr");
00453          return(-1);
00454      }
00455      oldwork = AT.WorkPointer;
00456      if ( AT.previousEfactor && ( expr == AT.previousEfactor[0] ) ) {
00457          /*
00458              We have a hit in the cache. Construct the new term.
00459              At the moment AT.previousEfactor[1] is reserved for future flags
00460          */
00461          goto PutTheFactor;
00462      }
00463      /*
00464          No hit. We have to do the work. We start with constructing the factor
00465          in the WorkSpace. Later we will move it to the cache.
00466          Finally we will jump to PutTheFactor.
00467      */
00468      e = Expressions + expr;
00469      switch ( e->status ) {
00470          case LOCALEXPRESSION:
00471          case SKIPLEXPRESSION:
00472          case DROPLEXPRESSION:
00473          case GLOBALEXPRESSION:
00474          case SKIPGEXPRESSION:
00475          case DROPGEXPRESSION:
00476          /*
00477              Expression is to be found in the input Scratch file.
00478              Set the file handle and the position.
00479              The rest is done by GetTerm.
00480          */
00481          newhold = 0;
00482          newgetfile = 1;
00483          file = AR.infile;
00484          break;
00485          case HIDDENLEXPRESSION:
00486          case HIDDENGEXPRESSION:
00487          case HIDELEXPRESSION:
00488          case HIDEGEXPRESSION:
00489          case DROPHLEXPRESSION:
00490          case DROPHGEXPRESSION:
00491          case UNHIDELEXPRESSION:
00492          case UNHIDEGEXPRESSION:
00493          /*
00494              Expression is to be found in the hide Scratch file.
00495              Set the file handle and the position.
00496              The rest is done by GetTerm.
00497          */
00498          newhold = 0;
00499          newgetfile = 2;
00500          file = AR.hidefile;
00501          break;
00502          case STOREDEXPRESSION:

```

```

00502         This is an 'illegal' case
00503     */
00504     MLOCK(ErrorMessageLock);
00505     MesPrint("Error: factorin_ cannot determine factors in stored expressions.");
00506     MUNLOCK(ErrorMessageLock);
00507     NumberFree(GCDbuffer, "FactorInExpr"); NumberFree(GCDbuffer2, "FactorInExpr");
00508     NumberFree(LCMbuffer, "FactorInExpr"); NumberFree(LCMb, "FactorInExpr");
00509     NumberFree(LCMc, "FactorInExpr");
00510     return(-1);
00511     case DROPPEDEXPRESSION:
00512     /*
00513         We replace the function by 1.
00514     */
00515     m = oldwork; n1 = term;
00516     while ( n1 < t ) *m++ = *n1++;
00517     n1 = t + t[1]; tstop = term + *term;
00518     while ( n1 < tstop ) *m++ = *n1++;
00519     *oldwork = m - oldwork;
00520     AT.WorkPointer = m;
00521     if ( Generator(BHEAD oldwork, level) ) {
00522         NumberFree(GCDbuffer, "FactorInExpr"); NumberFree(GCDbuffer2, "FactorInExpr");
00523         NumberFree(LCMbuffer, "FactorInExpr"); NumberFree(LCMb, "FactorInExpr");
00524         NumberFree(LCMc, "FactorInExpr");
00525         return(-1);
00526     }
00527     AT.WorkPointer = oldwork;
00528     NumberFree(GCDbuffer, "FactorInExpr"); NumberFree(GCDbuffer2, "FactorInExpr");
00529     NumberFree(LCMbuffer, "FactorInExpr"); NumberFree(LCMb, "FactorInExpr");
00530     NumberFree(LCMc, "FactorInExpr");
00531     return(0);
00532     default:
00533     MLOCK(ErrorMessageLock);
00534     MesPrint("Error: Illegal expression in factorinexpr.");
00535     MUNLOCK(ErrorMessageLock);
00536     NumberFree(GCDbuffer, "FactorInExpr"); NumberFree(GCDbuffer2, "FactorInExpr");
00537     NumberFree(LCMbuffer, "FactorInExpr"); NumberFree(LCMb, "FactorInExpr");
00538     NumberFree(LCMc, "FactorInExpr");
00539     return(-1);
00540 }
00541 /*
00542     Before we start with the file we set the buffers for the coefficient
00543     For the coefficient we have to determine the LCM of the denominators
00544     and the GCD of the numerators.
00545 */
00546 position = AS.OldOnFile[expr];
00547 AR.DeferFlag = 0; AR.KeptInHold = newhold; AR.GetFile = newgetfile;
00548 SeekScratch(file, &oldposition);
00549 SetScratch(file, &position);
00550 if ( GetTerm(BHEAD oldwork) <= 0 ) {
00551     MLOCK(ErrorMessageLock);
00552     MesPrint("(5) Expression %d has problems in scratchfile", expr);
00553     MUNLOCK(ErrorMessageLock);
00554     NumberFree(GCDbuffer, "FactorInExpr"); NumberFree(GCDbuffer2, "FactorInExpr");
00555     NumberFree(LCMbuffer, "FactorInExpr"); NumberFree(LCMb, "FactorInExpr");
00556     NumberFree(LCMc, "FactorInExpr");
00557     return(-1);
00558 }
00559 SeekScratch(file, &startposition);
00560 SeekScratch(file, &position);
00561 /*
00562     Load the first term in the workspace
00563 */
00564 if ( GetTerm(BHEAD oldwork) == 0 ) {
00565     SetScratch(file, &oldposition); /* We still need this until Processor is clean */
00566     AR.DeferFlag = olddeferflag;
00567     oldwork[0] = 4; oldwork[1] = 1; oldwork[2] = 1; oldwork[3] = 3;
00568     goto Complete;
00569 }
00570 SeekScratch(file, &position);
00571 AR.DeferFlag = olddeferflag; AR.KeptInHold = oldhold; AR.GetFile = oldgetfile;
00572 r2 = m = oldwork + *oldwork;
00573 j = m[-1];
00574 m -= ABS(j);
00575 *oldwork = (WORD) (m-oldwork);
00576 AT.WorkPointer = newterm = mstop = m;
00577 /*
00578     Now take the coefficient of the first term to load up the LCM and the GCD
00579 */
00580 r3 = m;
00581 k = REDLENG(j);
00582 if ( k < 0 ) { k = -k; sign = -1; }
00583 else { sign = 1; }
00584 while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00585 for ( kGCD = 0; kGCD < k; kGCD++ ) GCDbuffer[kGCD] = r3[kGCD];
00586 k = REDLENG(j);
00587 if ( k < 0 ) k = -k;

```

```

00584     r3 += k;
00585     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00586     for ( kLCM = 0; kLCM < k; kLCM++ ) LCMbuffer[kLCM] = r3[kLCM];
00587 /*
00588     The copy and the coefficient are in place. Now search through the terms.
00589 */
00590     for ( ;; ) {
00591         AR.DeferFlag = 0; AR.KeptInHold = newhold; AR.GetFile = newgetfile;
00592         SetScratch(file,&position);
00593         size = GetTerm(BHEAD newterm);
00594         SeekScratch(file,&position);
00595         AR.DeferFlag = olddeferflag; AR.KeptInHold = oldhold; AR.GetFile = oldgetfile;
00596         if ( size == 0 ) break;
00597         m = oldwork+1;
00598         r2 = newterm + *newterm;
00599         r2 -= ABS(r2[-1]);
00600         r1 = newterm+1;
00601         while ( m < mstop ) {
00602             while ( r1 < r2 ) {
00603                 if ( *r1 != *m ) {
00604                     r1 += r1[1]; continue;
00605                 }
00606             /*
00607                 Now the various cases
00608             #[ SYMBOL :
00609             */
00610                 if ( *m == SYMBOL ) {
00611                     n1 = m+2; n1stop = m+m[1];
00612                     n2stop = r1+r1[1];
00613                     while ( n1 < n1stop ) {
00614                         n2 = r1+2;
00615                         while ( n2 < n2stop ) {
00616                             if ( *n1 != *n2 ) { n2 += 2; continue; }
00617                             if ( n1[1] > 0 ) {
00618                                 if ( n2[1] < 0 ) { n2 += 2; continue; }
00619                                 if ( n2[1] < n1[1] ) n1[1] = n2[1];
00620                             }
00621                             else {
00622                                 if ( n2[1] > 0 ) { n2 += 2; continue; }
00623                                 if ( n2[1] > n1[1] ) n1[1] = n2[1];
00624                             }
00625                             break;
00626                         }
00627                     if ( n2 >= n2stop ) { /* scratch symbol */
00628                         if ( m[1] == 4 ) goto scratch;
00629                         m[1] -= 2;
00630                         n3 = n1; n4 = n1+2;
00631                         while ( n4 < mstop ) *n3++ = *n4++;
00632                         *oldwork = n3 - oldwork;
00633                         mstop -= 2; n1stop -= 2;
00634                         continue;
00635                     }
00636                     n1 += 2;
00637                 }
00638                 break;
00639             }
00640             /*
00641             #[ SYMBOL :
00642             #[ DOTPRODUCT :
00643             */
00644                 else if ( *m == DOTPRODUCT ) {
00645                     n1 = m+2; n1stop = m+m[1];
00646                     n2stop = r1+r1[1];
00647                     while ( n1 < n1stop ) {
00648                         n2 = r1+2;
00649                         while ( n2 < n2stop ) {
00650                             if ( *n1 != *n2 || n1[1] != n2[1] ) { n2 += 3; continue; }
00651                             if ( n1[2] > 0 ) {
00652                                 if ( n2[2] < 0 ) { n2 += 3; continue; }
00653                                 if ( n2[2] < n1[2] ) n1[2] = n2[2];
00654                             }
00655                             else {
00656                                 if ( n2[2] > 0 ) { n2 += 3; continue; }
00657                                 if ( n2[2] > n1[2] ) n1[2] = n2[2];
00658                             }
00659                             break;
00660                         }
00661                     if ( n2 >= n2stop ) { /* scratch dotproduct */
00662                         if ( m[1] == 5 ) goto scratch;
00663                         m[1] -= 3;
00664                         n3 = n1; n4 = n1+3;
00665                         while ( n4 < mstop ) *n3++ = *n4++;
00666                         *oldwork = n3 - oldwork;
00667                         mstop -= 3; n1stop -= 3;
00668                         continue;
00669                     }
00670                     n1 += 3;

```



```

00671         }
00672         break;
00673     }
00674 /*
00675     #] DOTPRODUCT :
00676     #[ VECTOR :
00677 */
00678     else if ( *m == VECTOR ) {
00679 /*
00680         Here we have to be careful if there is more than
00681         one of the same
00682 */
00683         n1 = m+2; n1stop = m+m[1];
00684         n2 = r1+2; n2stop = r1+r1[1];
00685         while ( n1 < n1stop ) {
00686             while ( n2 < n2stop ) {
00687                 if ( *n1 == *n2 && n1[1] == n2[1] ) {
00688                     n2 += 2; goto nextn1;
00689                 }
00690                 n2 += 2;
00691             }
00692             if ( n2 >= n2stop ) { /* scratch vector */
00693                 if ( m[1] == 4 ) goto scratch;
00694                 m[1] -= 2;
00695                 n3 = n1; n4 = n1+2;
00696                 while ( n4 < mstop ) *n3++ = *n4++;
00697                 *oldwork = n3 - oldwork;
00698                 mstop -= 2; n1stop -= 2;
00699                 continue;
00700             }
00701             n2 = r1+2;
00702 nextn1:         n1 += 2;
00703         }
00704         break;
00705     }
00706 /*
00707     #] VECTOR :
00708     #[ REMAINDER :
00709 */
00710     else {
00711 /*
00712         Again: watch for multiple occurrences of the same object
00713 */
00714         if ( m[1] != r1[1] ) { r1 += r1[1]; continue; }
00715         for ( j = 2; j < m[1]; j++ ) {
00716             if ( m[j] != r1[j] ) break;
00717         }
00718         if ( j < m[1] ) { r1 += r1[1]; continue; }
00719         r1 += r1[1]; /* to restart at the next potential match */
00720         goto nextm; /* match */
00721     }
00722 /*
00723     #] REMAINDER :
00724 */
00725     }
00726     if ( r1 >= r2 ) { /* no factor! */
00727 scratch:;
00728         r3 = m + m[1]; r4 = m;
00729         while ( r3 < mstop ) *r4++ = *r3++;
00730         *oldwork = r4 - oldwork;
00731         if ( *oldwork == 1 ) goto nofactor;
00732         mstop = r4;
00733         r1 = newterm + 1;
00734         continue;
00735     }
00736     r1 = newterm + 1;
00737 nextm:     m += m[1];
00738     }
00739 nofactor:;
00740 /*
00741     Now the coefficient
00742 */
00743     r2 = newterm + *newterm;
00744     j = r2[-1];
00745     r3 = r2 - ABS(j);
00746     k = REDLENG(j);
00747     if ( k < 0 ) k = -k;
00748     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00749     if ( ( ( GCDBuffer[0] == 1 ) && ( kGCD == 1 ) ) ) {
00750 /*
00751         GCD is already 1
00752 */
00753     }
00754     else if ( ( ( k != 1 ) || ( r3[0] != 1 ) ) ) {
00755         if ( GcdLong(BHEAD GCDBuffer, kGCD, (UWORD *)r3, k, GCDBuffer2, &kGCD2) ) {
00756             goto onerror;
00757         }
00758     }

```

```

00758         kGCD = kGCD2;
00759         for ( i = 0; i < kGCD; i++ ) GCDBuffer[i] = GCDBuffer2[i];
00760     }
00761     else {
00762         kGCD = 1; GCDBuffer[0] = 1;
00763     }
00764     k = REDLENG(j);
00765     if ( k < 0 ) k = -k;
00766     r3 += k;
00767     while ( ( k > 1 ) && ( r3[k-1] == 0 ) ) k--;
00768     if ( ( ( LCMbuffer[0] == 1 ) && ( kLCM == 1 ) ) ) {
00769         for ( kLCM = 0; kLCM < k; kLCM++ )
00770             LCMbuffer[kLCM] = r3[kLCM];
00771     }
00772     else if ( ( k != 1 ) || ( r3[0] != 1 ) ) {
00773         if ( GcdLong(BHEAD LCMbuffer,kLCM,(UWORD *)r3,k,LCMb,&kLCM) ) {
00774             goto onerror;
00775         }
00776         DivLong((UWORD *)r3,k,LCMb,kLCM,LCMb,&kLCM,LCMc,&jLCM);
00777         MulLong(LCMbuffer,kLCM,LCMb,kLCM,LCMc,&jLCM);
00778         for ( kLCM = 0; kLCM < jLCM; kLCM++ )
00779             LCMbuffer[kLCM] = LCMc[kLCM];
00780     }
00781     else {} /* LCM doesn't change */
00782 }
00783 SetScratch(file,&oldposition); /* Needed until Processor is thread safe */
00784 AR.DeferFlag = olddeferflag;
00785 /*
00786 Now put the term together in oldwork: GCD/LCM
00787 We have already the algebraic contents there.
00788 */
00789 r3 = (WORD *) (GCDBuffer);
00790 r4 = (WORD *) (LCMbuffer);
00791 r1 = oldwork + *oldwork;
00792 if ( kGCD == kLCM ) {
00793     for ( jGCD = 0; jGCD < kGCD; jGCD++ ) *r1++ = *r3++;
00794     for ( jGCD = 0; jGCD < kGCD; jGCD++ ) *r1++ = *r4++;
00795     k = 2*kGCD+1;
00796 }
00797 else if ( kGCD > kLCM ) {
00798     for ( jGCD = 0; jGCD < kGCD; jGCD++ ) *r1++ = *r3++;
00799     for ( jGCD = 0; jGCD < kLCM; jGCD++ ) *r1++ = *r4++;
00800     for ( jGCD = kLCM; jGCD < kGCD; jGCD++ ) *r1++ = 0;
00801     k = 2*kGCD+1;
00802 }
00803 else {
00804     for ( jGCD = 0; jGCD < kGCD; jGCD++ ) *r1++ = *r3++;
00805     for ( jGCD = kGCD; jGCD < kLCM; jGCD++ ) *r1++ = 0;
00806     for ( jGCD = 0; jGCD < kLCM; jGCD++ ) *r1++ = *r4++;
00807     k = 2*kLCM+1;
00808 }
00809 if ( sign < 0 ) *r1++ = -k;
00810 else *r1++ = k;
00811 *oldwork = (WORD) (r1-oldwork);
00812 /*
00813 Now put the new term in the cache
00814 */
00815 Complete;;
00816 if ( AT.previousEfactor ) M_free(AT.previousEfactor,"Efactor cache");
00817 AT.previousEfactor = (WORD *) Malloc1(( *oldwork+2)*sizeof(WORD),"Efactor cache");
00818 AT.previousEfactor[0] = expr;
00819 r1 = oldwork; r2 = AT.previousEfactor + 2; k = *oldwork;
00820 NCOPY(r2,r1,k)
00821 AT.previousEfactor[1] = 0;
00822 /*
00823 Now we construct the new term in the workspace.
00824 */
00825 PutTheFactor;;
00826 if ( AT.WorkPointer + AT.previousEfactor[2] >= AT.WorkTop ) {
00827     MLOCK(ErrorMessageLock);
00828     MesWork();
00829     MesPrint("Called from factorin_");
00830     MUNLOCK(ErrorMessageLock);
00831     NumberFree(GCDBuffer,"FactorInExpr"); NumberFree(GCDBuffer2,"FactorInExpr");
00832     NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
00833     NumberFree(LCMc,"FactorInExpr");
00834     return(-1);
00835 }
00836 n1 = oldwork; n2 = term; while ( n2 < t ) *n1++ = *n2++;
00837 n2 = AT.previousEfactor+2; GETSTOP(n2,n2stop); n3 = n2 + *n2;
00838 n2++; while ( n2 < n2stop ) *n1++ = *n2++;
00839 n2 = t + t[1]; while ( n2 < tstop ) *n1++ = *n2++;
00840 size = term[*term-1];
00841 size = REDLENG(size);
00842 k = n3[-1]; k = REDLENG(k);
00843 if ( MulRat(BHEAD (UWORD *)tstop,size,(UWORD *)n2stop,k,
(UWORD *)n1,&size) ) goto onerror;

```

```

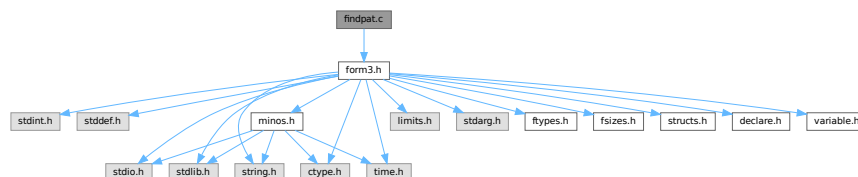
00844     size = INCLENG(size);
00845     k = size < 0 ? -size: size;
00846     n1 += k; n1[-1] = size;
00847     *oldwork = n1 - oldwork;
00848     AT.WorkPointer = n1;
00849     if ( Generator(BHEAD oldwork,level) ) {
00850         NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
00851         NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
00852         NumberFree(LCMc,"FactorInExpr");
00853         return(-1);
00854     }
00855     AT.WorkPointer = oldwork;
00856     AR.CompressPointer = oldcpointer;
00857     NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
00858     NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
00859     NumberFree(LCMc,"FactorInExpr");
00860     return(0);
00861 onerror:
00862     AT.WorkPointer = oldwork;
00863     AR.CompressPointer = oldcpointer;
00864     MLOCK(ErrorMessageLock);
00865     MesCall("FactorInExpr");
00866     MUNLOCK(ErrorMessageLock);
00867     NumberFree(GCDbuffer,"FactorInExpr"); NumberFree(GCDbuffer2,"FactorInExpr");
00868     NumberFree(LCMbuffer,"FactorInExpr"); NumberFree(LCMb,"FactorInExpr");
00869     NumberFree(LCMc,"FactorInExpr");
00870     return(-1);
00871 }
00872 /*
00873 #] FactorInExpr :
00874 */

```

4.37 findpat.c File Reference

```
#include "form3.h"
```

Include dependency graph for findpat.c:



Functions

- int [FindOnly](#) (PHEAD WORD *term, WORD *pattern)
- int [FindOnce](#) (PHEAD WORD *term, WORD *pattern)
- WORD [FindMulti](#) (PHEAD WORD *term, WORD *pattern)
- int [FindRest](#) (PHEAD WORD *term, WORD *pattern)

4.37.1 Detailed Description

Pattern matching of symbols and dotproducts. There are various routines because of the options in the id-statements like once, only, multi and many. These are among the oldest routines in FORM and that can be noticed, because the interplay with the function matching is not complete. When we match functions and halfway we fail we can backtrack properly. With the symbols, the dotproducts and the vectors (in [pattern.c](#)) there is no proper backtracking. Hence the routines here need still quite some work or may even have to be rewritten.

Definition in file [findpat.c](#).

4.37.2 Function Documentation

4.37.2.1 FindOnly()

```
int FindOnly (
    PHEAD WORD * term,
    WORD * pattern )
```

Definition at line 61 of file [findpat.c](#).

4.37.2.2 FindOnce()

```
int FindOnce (
    PHEAD WORD * term,
    WORD * pattern )
```

Definition at line 419 of file [findpat.c](#).

4.37.2.3 FindMulti()

```
WORD FindMulti (
    PHEAD WORD * term,
    WORD * pattern )
```

Definition at line 954 of file [findpat.c](#).

4.37.2.4 FindRest()

```
int FindRest (
    PHEAD WORD * term,
    WORD * pattern )
```

Definition at line 1070 of file [findpat.c](#).

4.38 findpat.c

[Go to the documentation of this file.](#)

```
00001
00013 /* #[] License : */
00014 /*
00015 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00016 * When using this file you are requested to refer to the publication
00017 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00018 * This is considered a matter of courtesy as the development was paid
00019 * for by FOM the Dutch physics granting agency and we would like to
00020 * be able to track its scientific use to convince FOM of its value
00021 * for the community.
00022 *
00023 * This file is part of FORM.
00024 *
00025 * FORM is free software: you can redistribute it and/or modify it under the
00026 * terms of the GNU General Public License as published by the Free Software
00027 * Foundation, either version 3 of the License, or (at your option) any later
00028 * version.
00029 *
00030 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
```

```

00031 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00032 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00033 *   details.
00034 *
00035 *   You should have received a copy of the GNU General Public License along
00036 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00037 */
00038 /* #] License : */
00039 /*
00040     #[ Includes : findpat.c
00041 */
00042
00043 #include "form3.h"
00044
00045 /*
00046     #] Includes :
00047     #[ Patterns :
00048         #[ FindOnly :           WORD FindOnly(term,pattern)
00049
00050     The current version doesn't scan function arguments yet. 10-Apr-1988
00051
00052     This routine searches for an exact match. This means in particular:
00053     1: x^# must match exactly.
00054     2: x^n? must have a single value for n that cannot be adapted.
00055
00056     When setp != 0 it points to a collection of sets
00057     A match can occur only if no object will be left that belongs
00058     to any of these sets.
00059 */
00060
00061 int FindOnly(PHEAD WORD *term, WORD *pattern)
00062 {
00063     GETBIDENTITY
00064     WORD *t, *m;
00065     WORD *tstop, *mstop;
00066     WORD *xstop, *ystop, *setp = AN.ForFindOnly;
00067     WORD n, nt, *p, nq;
00068     WORD older[NORMSIZE], *q, newval1, newval2, newval3;
00069     AN.UsedOtherFind = 0;
00070     m = pattern;
00071     mstop = m + *m;
00072     m++;
00073     t = term;
00074     t += *term - 1;
00075     tstop = t - ABS(*t) + 1;
00076     t = term;
00077     t++;
00078     while ( t < tstop && *t > DOTPRODUCT ) t += t[1];
00079     while ( m < mstop && *m > DOTPRODUCT ) m += m[1];
00080     if ( m < mstop ) { do {
00081 /*
00082         #[ SYMBOLS :
00083 */
00084         if ( *m == SYMBOL ) {
00085             ystop = m + m[1];
00086             m += 2;
00087             n = 0;
00088             p = older;
00089             if ( t < tstop ) while ( *t != SYMBOL ) {
00090                 t += t[1];
00091                 if ( t >= tstop ) {
00092 OnlyZer1:
00093                     do {
00094                         if ( *m >= 2*MAXPOWER ) return(0);
00095                         if ( m[1] >= 2*MAXPOWER ) nt = m[1];
00096                         else if ( m[1] <= -2*MAXPOWER ) nt = -m[1];
00097                         else return(0);
00098                         nt -= 2*MAXPOWER;
00099                         if ( CheckWild(BHEAD nt,SYMTONUM,0,&newval3) ) return(0);
00100                         AddWild(BHEAD nt,SYMTONUM,0);
00101                         m += 2;
00102                     } while ( m < ystop );
00103                     goto EndLoop;
00104                 }
00105             }
00106             else goto OnlyZer1;
00107             xstop = t + t[1];
00108             t += 2;
00109             do {
00110                 if ( *m == *t && t < xstop ) {
00111                     if ( m[1] == t[1] ) { m += 2; t += 2; }
00112                     else if ( m[1] >= 2*MAXPOWER ) {
00113                         nt = t[1];
00114                         nq = m[1];
00115                         goto OnlyL2;
00116                     }
00117                     else if ( m[1] <= -2*MAXPOWER ) {

```

```

00118             nt = -t[1];
00119             nq = -m[1];
00120 OnlyL2:         nq -= 2*MAXPOWER;
00121                 if ( CheckWild(BHEAD nq, SYMTONUM, nt, &newval3) ) return(0);
00122                 AddWild(BHEAD nq, SYMTONUM, nt);
00123                 m += 2;
00124                 t += 2;
00125             }
00126             else {
00127                 *p++ = *t++; *p++ = *t++; n += 2;
00128             }
00129         }
00130     else if ( *m >= 2*MAXPOWER ) {
00131         while ( t < xstop ) { *p++ = *t++; *p++ = *t++; n += 2; }
00132         nq = n;
00133         p = older;
00134         while ( nq > 0 ) {
00135             if ( !CheckWild(BHEAD *m-2*MAXPOWER, SYMTOSYM, *p, &newval1) ) {
00136                 if ( m[1] == p[1] ) {
00137                     AddWild(BHEAD *m-2*MAXPOWER, SYMTOSYM, newval1);
00138                     break;
00139                 }
00140                 else if ( m[1] >= 2*MAXPOWER && m[1] != *m ) {
00141                     if ( !CheckWild(BHEAD m[1]-2*MAXPOWER, SYMTONUM, p[1], &newval3) ) {
00142                         AddWild(BHEAD m[1]-2*MAXPOWER, SYMTONUM, p[1]);
00143                         AddWild(BHEAD *m-2*MAXPOWER, SYMTOSYM, newval1);
00144                         break;
00145                     }
00146                 }
00147                 else if ( m[1] <= -2*MAXPOWER && m[1] != -( *m ) ) {
00148                     if ( !CheckWild(BHEAD -m[1]-2*MAXPOWER, SYMTONUM, -p[1], &newval3) ) {
00149                         AddWild(BHEAD -m[1]-2*MAXPOWER, SYMTONUM, -p[1]);
00150                         AddWild(BHEAD *m-2*MAXPOWER, SYMTOSYM, newval1);
00151                         break;
00152                     }
00153                 }
00154             }
00155             nq -= 2;
00156             p += 2;
00157         }
00158         if ( nq <= 0 ) return(0);
00159         nq -= 2;
00160         n -= 2;
00161         q = p + 2;
00162         while ( --nq >= 0 ) *p++ = *q++;
00163         m += 2;
00164     }
00165     else {
00166         if ( t >= xstop || *m < *t ) {
00167             if ( m[1] >= 2*MAXPOWER ) nt = m[1];
00168             else if ( m[1] <= -2*MAXPOWER ) nt = -m[1];
00169             else return(0);
00170             nt -= 2*MAXPOWER;
00171             if ( CheckWild(BHEAD nt, SYMTONUM, 0, &newval3) ) return(0);
00172             AddWild(BHEAD nt, SYMTONUM, 0);
00173             m += 2;
00174         }
00175         else {
00176             *p++ = *t++; *p++ = *t++; n += 2;
00177         }
00178     }
00179 } while ( m < ystop );
00180 if ( setp ) {
00181     while ( t < xstop ) { *p++ = *t++; *p++ = *t++; n += 2; }
00182     p = older;
00183     while ( n > 0 ) {
00184         nq = setp[1] - 2;
00185         m = setp + 2;
00186         while ( --nq >= 0 ) {
00187             if ( Sets[*m].type != CSYMBOL ) { m++; continue; }
00188             t = SetElements + Sets[*m].first;
00189             tstop = SetElements + Sets[*m].last;
00190             while ( t < tstop ) {
00191                 if ( *t++ == *p ) return(0);
00192             }
00193             m++;
00194         }
00195         n -= 2;
00196         p += 2;
00197     }
00198 }
00199 return(1);
00200 }
00201 /*
00202     #] SYMBOLS :
00203     #[ DOTPRODUCTS :
00204 */

```

```

00205     else if ( *m == DOTPRODUCT ) {
00206         ystop = m + m[1];
00207         m += 2;
00208         n = 0;
00209         p = older;
00210         if ( t < tstop ) {
00211             if ( *t < DOTPRODUCT ) goto OnlyZer2;
00212             while ( *t > DOTPRODUCT ) {
00213                 t += t[1];
00214                 if ( t >= tstop || *t < DOTPRODUCT ) {
00215 OnlyZer2:
00216                     do {
00217                         if ( *m >= (AM.OffsetVector+WILDOFFSET)
00218                             || m[1] >= (AM.OffsetVector+WILDOFFSET) ) return(0);
00219                         if ( m[2] >= 2*MAXPOWER ) nq = m[2];
00220                         else if ( m[2] <= -2*MAXPOWER ) nq = -m[2];
00221                         else return(0);
00222                         nq -= 2*MAXPOWER;
00223                         if ( CheckWild(BHEAD nq, SYMTONUM, 0, &newval3) ) return(0);
00224                         AddWild(BHEAD nq, SYMTONUM, 0);
00225                         m += 3;
00226                     } while ( m < ystop );
00227                     goto EndLoop;
00228                 }
00229             }
00230         }
00231     else goto OnlyZer2;
00232     xstop = t + t[1];
00233     t += 2;
00234     do {
00235         if ( *m == *t && m[1] == t[1] && t < xstop ) {
00236             if ( t[2] != m[2] ) {
00237                 if ( m[2] >= 2*MAXPOWER ) {
00238                     nq = m[2];
00239                     nt = t[2];
00240                 }
00241                 else if ( m[2] <= -2*MAXPOWER ) {
00242                     nq = -m[2];
00243                     nt = -t[2];
00244                 }
00245                 else return(0);
00246                 nq -= 2*MAXPOWER;
00247                 if ( CheckWild(BHEAD nq, SYMTONUM, nt, &newval3) ) return(0);
00248                 AddWild(BHEAD nq, SYMTONUM, nt);
00249             }
00250             t += 3; m += 3;
00251         }
00252     else if ( *m >= (AM.OffsetVector+WILDOFFSET) ) {
00253         while ( t < xstop ) {
00254             *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
00255         }
00256         nq = n;
00257         p = older;
00258         while ( nq > 0 ) {
00259             if ( *m == m[1] ) {
00260                 if ( *p != p[1] ) goto NextInDot;
00261             }
00262             if ( !CheckWild(BHEAD *m-WILDOFFSET, VECTOVEC, *p, &newval1) &&
00263                 !CheckWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, p[1], &newval2) ) {
00264                 if ( p[2] == m[2] ) {
00265 OnlyL9:
00266                     AddWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, newval2);
00267                     AddWild(BHEAD *m-WILDOFFSET, VECTOVEC, newval1);
00268                     break;
00269                 }
00270                 if ( m[2] >= 2*MAXPOWER ) {
00271                     if ( !CheckWild(BHEAD m[2]-2*MAXPOWER, SYMTONUM, p[2], &newval3) ) {
00272                         AddWild(BHEAD m[2]-2*MAXPOWER, SYMTONUM, newval3);
00273                         goto OnlyL9;
00274                     }
00275                 }
00276                 else if ( m[2] <= -2*MAXPOWER ) {
00277                     if ( !CheckWild(BHEAD -m[2]-2*MAXPOWER, SYMTONUM, -p[2], &newval3) ) {
00278                         AddWild(BHEAD -m[2]-2*MAXPOWER, SYMTONUM, -p[2]);
00279                         goto OnlyL9;
00280                     }
00281                 }
00282             }
00283             if ( !CheckWild(BHEAD *m-WILDOFFSET, VECTOVEC, p[1], &newval1) &&
00284                 !CheckWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, *p, &newval2) ) {
00285                 if ( p[2] == m[2] ) {
00286 OnlyL10:
00287                     AddWild(BHEAD *m-WILDOFFSET, VECTOVEC, newval1);
00288                     AddWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, newval2);
00289                     break;
00290                 }
00291                 if ( m[2] >= 2*MAXPOWER ) {
00292                     if ( !CheckWild(BHEAD m[2]-2*MAXPOWER, SYMTONUM, p[2], &newval3) ) {
00293                         AddWild(BHEAD m[2]-2*MAXPOWER, SYMTONUM, p[2]);
00294                     }
00295                 }
00296                 else if ( m[2] <= -2*MAXPOWER ) {
00297                     if ( !CheckWild(BHEAD -m[2]-2*MAXPOWER, SYMTONUM, -p[2], &newval3) ) {
00298                         AddWild(BHEAD -m[2]-2*MAXPOWER, SYMTONUM, -p[2]);
00299                     }
00300                 }
00301             }
00302         }
00303     }
00304 }

```

```

00292                                goto OnlyL10;
00293                                }
00294                                }
00295                                else if ( m[2] <= -2*MAXPOWER ) {
00296                                    if ( !CheckWild(BHEAD -m[2]-2*MAXPOWER, SYMTONUM, -p[2], &newval3) ) {
00297                                        AddWild(BHEAD -m[2]-2*MAXPOWER, SYMTONUM, -p[2]);
00298                                        goto OnlyL10;
00299                                    }
00300                                }
00301                                }
00302 NextInDot:
00303                                p += 3; nq -= 3;
00304                                }
00305                                if ( nq <= 0 ) return(0);
00306                                q = p+3;
00307                                nq -= 3;
00308                                n -= 3;
00309                                while ( --nq >= 0 ) *p++ = *q++;
00310                                m += 3;
00311                            }
00312                            else if ( m[1] >= (AM.OffsetVector+WILDOFFSET) ) {
00313                                while ( *m >= *t && t < xstop ) {
00314                                    *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
00315                                }
00316                                nq = n;
00317                                p = older;
00318                                while ( nq > 0 ) {
00319                                    if ( *m == *p && !CheckWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, p[1], &newval1) ) {
00320                                        if ( p[2] == m[2] ) {
00321                                            AddWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, newval1);
00322                                            break;
00323                                        }
00324                                        else if ( m[2] >= 2*MAXPOWER ) {
00325                                            if ( !CheckWild(BHEAD m[2]-2*MAXPOWER, SYMTONUM, p[2], &newval3) ) {
00326                                                AddWild(BHEAD m[2]-2*MAXPOWER, SYMTONUM, p[2]);
00327                                                AddWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, newval1);
00328                                                break;
00329                                            }
00330                                        }
00331                                        else if ( m[2] <= -2*MAXPOWER ) {
00332                                            if ( !CheckWild(BHEAD -m[2]-2*MAXPOWER, SYMTONUM, -p[2], &newval3) ) {
00333                                                AddWild(BHEAD -m[2]-2*MAXPOWER, SYMTONUM, -p[2]);
00334                                                AddWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, newval1);
00335                                                break;
00336                                            }
00337                                        }
00338                                    }
00339                                    if ( *m == p[1] && !CheckWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, *p, &newval1) ) {
00340                                        if ( p[2] == m[2] ) {
00341                                            AddWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, newval1);
00342                                            break;
00343                                        }
00344                                        if ( m[2] >= 2*MAXPOWER ) {
00345                                            if ( !CheckWild(BHEAD m[2]-2*MAXPOWER, SYMTONUM, p[2], &newval3) ) {
00346                                                AddWild(BHEAD m[2]-2*MAXPOWER, SYMTONUM, p[2]);
00347                                                AddWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, newval1);
00348                                                break;
00349                                            }
00350                                        }
00351                                        else if ( m[2] <= -2*MAXPOWER ) {
00352                                            if ( !CheckWild(BHEAD -m[2]-2*MAXPOWER, SYMTONUM, -p[2], &newval3) ) {
00353                                                AddWild(BHEAD -m[2]-2*MAXPOWER, SYMTONUM, -p[2]);
00354                                                AddWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, newval1);
00355                                                break;
00356                                            }
00357                                        }
00358                                    }
00359                                    p += 3; nq -= 3;
00360                                }
00361                                if ( nq <= 0 ) return(0);
00362                                q = p+3;
00363                                nq -= 3;
00364                                n -= 3;
00365                                while ( --nq >= 0 ) *p++ = *q++;
00366                                m += 3;
00367                            }
00368                            else {
00369                                if ( t >= xstop || *m < *t || ( *m == *t && m[1] < t[1] ) ) {
00370                                    if ( m[2] > 2*MAXPOWER ) nt = m[2];
00371                                    else if ( m[2] <= -2*MAXPOWER ) nt = -m[2];
00372                                    else return(0);
00373                                    nt -= 2*MAXPOWER;
00374                                    if ( !CheckWild(BHEAD nt, SYMTONUM, 0, &newval3) ) return(0);
00375                                    AddWild(BHEAD nt, SYMTONUM, 0);
00376                                    m += 3;
00377                                }
00378                                else {

```



```

00379             *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
00380         }
00381     }
00382     } while ( m < ystop );
00383     t = xstop;
00384 }
00385 /*
00386     #] DOTPRODUCTS :
00387 */
00388     else {
00389         MLOCK(ErrorMessageLock);
00390         MesPrint("Error in pattern(1)");
00391         MUNLOCK(ErrorMessageLock);
00392         Terminate(-1);
00393     }
00394 EndLoop;;
00395     } while ( m < mstop ); }
00396     if ( setp ) {
00397 /*
00398         while ( t < tstop && *t > SYMBOL ) t += t[1];
00399         if ( t < tstop && setp[1] > 2 ) return(0);
00400 */
00401         /* There were nonempty sets */
00402         /* Empty sets are rejected by the compiler */
00403     }
00404     return(1);
00405 }
00406
00407 /*
00408     #] FindOnly :
00409     #[ FindOnce :          WORD FindOnce(term,pattern)
00410
00411     Searches for a single match in term. The difference with multi
00412     lies mainly in the fact that here functions may occur.
00413     The functions have not been implemented yet. (10-Apr-1988)
00414     Wildcard powers are adjustable. The value closer to zero is taken.
00415     Positive and negative gives (o surprise) zero.
00416
00417 */
00418
00419 int FindOnce(PHEAD WORD *term, WORD *pattern)
00420 {
00421     GETBIDENTITY
00422     WORD *t, *m;
00423     WORD *tstop, *mstop;
00424     WORD *xstop, *ystop;
00425     WORD n, nt, *p, nq, mt, ch;
00426     WORD older[2*NORMSIZE], *q, newval1, newval2, newval3;
00427     AN.UsedOtherFind = 0;
00428     m = pattern;
00429     mstop = m + *m;
00430     m++;
00431     t = term;
00432     t += *term - 1;
00433     tstop = t - ABS(*t) + 1;
00434     t = term;
00435     t++;
00436     while ( t < tstop && *t > DOTPRODUCT ) t += t[1];
00437     while ( m < mstop && *m > DOTPRODUCT ) m += m[1];
00438     if ( m < mstop ) { do {
00439 /*
00440         #[ SYMBOLS :
00441 */
00442         if ( *m == SYMBOL ) {
00443             ystop = m + m[1];
00444             m += 2;
00445             n = 0;
00446             p = older;
00447             if ( t < tstop ) while ( *t != SYMBOL ) {
00448                 t += t[1];
00449                 if ( t >= tstop ) {
00450 TryZero:
00451                     do {
00452                         if ( *m >= 2*MAXPOWER ) return(0);
00453                         if ( m[1] >= 2*MAXPOWER ) nt = m[1];
00454                         else if ( m[1] <= -2*MAXPOWER ) nt = -m[1];
00455                         else return(0);
00456                         nt -= 2*MAXPOWER;
00457                         if ( ( ch = CheckWild(BHEAD nt,SYMTONUM,0,&newval3) ) != 0 ) {
00458                             if ( ch > 1 ) return(0);
00459                             if ( AN.oldtype != SYMTONUM ) return(0);
00460                             if ( *AN.MaskPointer == 2 ) return(0);
00461                         }
00462                         AddWild(BHEAD nt,SYMTONUM,0);
00463                         m += 2;
00464                     } while ( m < ystop );
00465                     goto EndLoop;

```

```

00466     }
00467 }
00468 else goto TryZero;
00469 xstop = t + t[1];
00470 t += 2;
00471 do {
00472     if ( *m == *t && t < xstop ) {
00473         nt = t[1];
00474         mt = m[1];
00475         if ( ( mt > 0 && mt <= nt ) ||
00476             ( mt < 0 && mt >= nt ) ) { m += 2; t += 2; }
00477     else if ( mt >= 2*MAXPOWER ) goto OnceL2;
00478     else if ( mt <= -2*MAXPOWER ) {
00479         nt = -nt;
00480         mt = -mt;
00481     OnceL2:
00482         mt -= 2*MAXPOWER;
00483         if ( ( ch = CheckWild(BHEAD mt, SYMTONUM, nt, &newval3) ) != 0 ) {
00484             if ( ch > 1 ) return(0);
00485             if ( AN.OLDTYPE != SYMTONUM ) return(0);
00486             if ( AN.OLDVALUE <= 0 ) {
00487                 if ( nt < AN.OLDVALUE ) nt = AN.OLDVALUE;
00488                 else {
00489                     if ( *AN.MaskPointer == 2 ) return(0);
00490                     if ( nt > 0 ) nt = 0;
00491                 }
00492             }
00493             if ( AN.OLDVALUE >= 0 ) {
00494                 if ( nt > AN.OLDVALUE ) nt = AN.OLDVALUE;
00495                 else {
00496                     if ( *AN.MaskPointer == 2 ) return(0);
00497                     if ( nt < 0 ) nt = 0;
00498                 }
00499             }
00500             AddWild(BHEAD mt, SYMTONUM, nt);
00501             m += 2;
00502             t += 2;
00503         }
00504     else {
00505         *p++ = *t++; *p++ = *t++; n += 2;
00506     }
00507 }
00508 else if ( *m >= 2*MAXPOWER ) {
00509     while ( t < xstop ) { *p++ = *t++; *p++ = *t++; n += 2; }
00510     nq = n;
00511     p = older;
00512     while ( nq > 0 ) {
00513         nt = p[1];
00514         if ( !CheckWild(BHEAD *m-2*MAXPOWER, SYMTOSYM, *p, &newval1) ) {
00515             mt = m[1];
00516             if ( ( mt > 0 && mt <= nt ) ||
00517                 ( mt < 0 && mt >= nt ) ) {
00518                 AddWild(BHEAD *m-2*MAXPOWER, SYMTOSYM, newval1);
00519                 break;
00520             }
00521         else if ( mt >= 2*MAXPOWER && mt != *m ) {
00522             mt -= 2*MAXPOWER;
00523             OnceL4a:
00524             if ( ( ch = CheckWild(BHEAD mt, SYMTONUM, nt, &newval3) ) != 0 ) {
00525                 if ( ch > 1 ) return(0);
00526                 if ( AN.OLDTYPE == SYMTONUM ) {
00527                     if ( AN.OLDVALUE >= 0 ) {
00528                         if ( nt > AN.OLDVALUE ) nt = AN.OLDVALUE;
00529                         else {
00530                             if ( *AN.MaskPointer == 2 ) return(0);
00531                             if ( nt < 0 ) nt = 0;
00532                         }
00533                     }
00534                     else {
00535                         if ( nt < AN.OLDVALUE ) nt = AN.OLDVALUE;
00536                         else {
00537                             if ( *AN.MaskPointer == 2 ) return(0);
00538                             if ( nt > 0 ) nt = 0;
00539                         }
00540                     }
00541                     AddWild(BHEAD mt, SYMTONUM, nt);
00542                     AddWild(BHEAD *m-2*MAXPOWER, SYMTOSYM, newval1);
00543                     break;
00544                 }
00545             }
00546         else {
00547             AddWild(BHEAD mt, SYMTONUM, nt);
00548             AddWild(BHEAD *m-2*MAXPOWER, SYMTOSYM, newval1);
00549             break;
00550         }
00551     }
00552     else if ( mt <= -2*MAXPOWER && mt != -( *m ) ) {
00553         nt = -nt;

```

```

00553             mt = -mt;
00554             goto OnceL4a;
00555         }
00556     }
00557     nq -= 2;
00558     p += 2;
00559 }
00560 if ( nq <= 0 ) return(0);
00561 nq -= 2;
00562 n -= 2;
00563 q = p + 2;
00564 while ( --nq >= 0 ) *p++ = *q++;
00565 m += 2;
00566 }
00567 else {
00568     if ( t >= xstop || *m < *t ) {
00569         if ( m[1] >= 2*MAXPOWER ) nt = m[1];
00570         else if ( m[1] <= -2*MAXPOWER ) nt = -m[1];
00571         else return(0);
00572         nt -= 2*MAXPOWER;
00573         if ( ( ch = CheckWild(BHEAD nt, SYMTONUM, 0, &newval3) ) != 0 ) {
00574             if ( ch > 1 ) return(0);
00575             if ( AN.oldtype != SYMTONUM ) return(0);
00576             if ( *AN.MaskPointer == 2 ) return(0);
00577         }
00578         AddWild(BHEAD nt, SYMTONUM, 0);
00579         m += 2;
00580     }
00581     else {
00582         *p++ = *t++; *p++ = *t++; n += 2;
00583     }
00584 }
00585 } while ( m < ystop );
00586 }
00587 /*
00588     #] SYMBOLS :
00589     #[ DOTPRODUCTS :
00590 */
00591 else if ( *m == DOTPRODUCT ) {
00592     ystop = m + m[1];
00593     m += 2;
00594     n = 0;
00595     p = older;
00596     if ( t < tstop ) {
00597         if ( *t < DOTPRODUCT ) goto OnceOp;
00598         while ( *t > DOTPRODUCT ) {
00599             t += t[1];
00600             if ( t >= tstop || *t < DOTPRODUCT ) {
00601 OnceOp:
00602                 do {
00603                     if ( *m >= (AM.OffsetVector+WILDOFFSET)
00604                         || m[1] >= (AM.OffsetVector+WILDOFFSET) ) return(0);
00605                     if ( m[2] >= 2*MAXPOWER ) {
00606                         nq = m[2] - 2*MAXPOWER;
00607                     }
00608                     else if ( m[2] <= -2*MAXPOWER ) {
00609                         nq = -m[2] - 2*MAXPOWER;
00610                     }
00611                     else return(0);
00612                     if ( CheckWild(BHEAD nq, SYMTONUM, (WORD)0, &newval3) ) {
00613                         if ( AN.oldtype != SYMTONUM ) return(0);
00614                         if ( *AN.MaskPointer == 2 ) return(0);
00615                     }
00616                     AddWild(BHEAD nq, SYMTONUM, (WORD)0);
00617                     m += 3;
00618                 } while ( m < ystop );
00619                 goto EndLoop;
00620             }
00621         }
00622     }
00623     else goto OnceOp;
00624     xstop = t + t[1];
00625     t += 2;
00626     do {
00627         if ( *m == *t && m[1] == t[1] && t < xstop ) {
00628             nt = t[2];
00629             mt = m[2];
00630 /*
00631             if ( ( nt > 0 && nt < mt ) ||
00632                 ( nt < 0 && nt > mt ) ) {
00633                 if ( mt <= -2*MAXPOWER ) {
00634                     mt = -mt;
00635                     nt = -nt;
00636                 }
00637                 else if ( mt < 2*MAXPOWER ) return(0);
00638                 mt -= 2*MAXPOWER;
00639                 if ( CheckWild(BHEAD mt, SYMTONUM, nt, &newval3) ) {

```

```

00640             if ( AN.oldtype != SYMTONUM ) return(0);
00641             if ( AN.oldvalue <= 0 ) {
00642                 if ( nt < AN.oldvalue ) nt = AN.oldvalue;
00643                 else {
00644                     if ( *AN.MaskPointer == 2 ) return(0);
00645                     if ( nt > 0 ) nt = 0;
00646                 }
00647             }
00648             if ( AN.oldvalue >= 0 ) {
00649                 if ( nt > AN.oldvalue ) nt = AN.oldvalue;
00650                 else {
00651                     if ( *AN.MaskPointer == 2 ) return(0);
00652                     if ( nt < 0 ) nt = 0;
00653                 }
00654             }
00655         }
00656         AddWild(BHEAD mt, SYMTONUM, nt);
00657         m += 3; t += 3;
00658     }
00659     else if ( ( nt > 0 && nt >= mt && mt > -2*MAXPOWER )
00660 || ( nt < 0 && nt <= mt && mt < 2*MAXPOWER ) ) {
00661         m += 3; t += 3;
00662     }
00663 */
00664     if ( ( mt > 0 && mt <= nt ) ||
00665         ( mt < 0 && mt >= nt ) ) { m += 3; t += 3; }
00666     else if ( mt >= 2*MAXPOWER ) goto OnceL7;
00667     else if ( mt <= -2*MAXPOWER ) {
00668         nt = -nt;
00669         mt = -mt;
00670     OnceL7:
00671         mt -= 2*MAXPOWER;
00672         if ( CheckWild(BHEAD mt, SYMTONUM, nt, &newval3) ) {
00673             if ( AN.oldtype != SYMTONUM ) return(0);
00674             if ( AN.oldvalue <= 0 ) {
00675                 if ( nt < AN.oldvalue ) nt = AN.oldvalue;
00676                 else {
00677                     if ( *AN.MaskPointer == 2 ) return(0);
00678                     if ( nt > 0 ) nt = 0;
00679                 }
00680             }
00681             if ( AN.oldvalue >= 0 ) {
00682                 if ( nt > AN.oldvalue ) nt = AN.oldvalue;
00683                 else {
00684                     if ( *AN.MaskPointer == 2 ) return(0);
00685                     if ( nt < 0 ) nt = 0;
00686                 }
00687             }
00688             AddWild(BHEAD mt, SYMTONUM, nt);
00689             m += 3;
00690             t += 3;
00691         }
00692     }
00693     else {
00694         *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
00695     }
00696 }
00697 else if ( *m >= (AM.OffsetVector+WILDOFFSET) ) {
00698     while ( t < xstop ) {
00699         *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
00700     }
00701     nq = n;
00702     p = older;
00703     while ( nq > 0 ) {
00704         if ( *m == m[1] ) {
00705             if ( *p != p[1] ) goto NextInDot;
00706         }
00707         if ( !CheckWild(BHEAD *m-WILDOFFSET, VECTOVEC, *p, &newval1) &&
00708             !CheckWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, p[1], &newval2) ) {
00709             nt = p[2];
00710             mt = m[2];
00711             if ( ( mt > 0 && nt >= mt ) ||
00712                 ( mt < 0 && nt <= mt ) ) {
00713                 AddWild(BHEAD *m-WILDOFFSET, VECTOVEC, newval1);
00714                 AddWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, newval2);
00715                 break;
00716             }
00717             if ( mt >= 2*MAXPOWER ) {
00718                 mt -= 2*MAXPOWER;
00719                 if ( CheckWild(BHEAD mt, SYMTONUM, nt, &newval3) ) {
00720                     if ( AN.oldtype == SYMTONUM ) {
00721                         if ( AN.oldvalue >= 0 ) {
00722                             if ( nt > AN.oldvalue ) nt = AN.oldvalue;
00723                             else {
00724                                 if ( *AN.MaskPointer == 2 ) return(0);
00725                                 if ( nt < 0 ) nt = 0;
00726                             }
00727                         }

```

```

00727         else {
00728             if ( nt < AN.oldvalue ) nt = AN.oldvalue;
00729             else {
00730                 if ( *AN.MaskPointer == 2 ) return(0);
00731                 if ( nt > 0 ) nt = 0;
00732             }
00733         }
00734         AddWild(BHEAD mt, SYMTONUM, nt);
00735         goto OnceL9;
00736     }
00737 }
00738     else {
00739         AddWild(BHEAD mt, SYMTONUM, nt);
00740         goto OnceL9;
00741     }
00742 }
00743     else if ( mt <= -2*MAXPOWER ) {
00744         mt = -mt;
00745         nt = -nt;
00746         goto OnceL9a;
00747     }
00748 }
00749 if ( !CheckWild(BHEAD *m-WILDOFFSET, VECTOVEC, p[1], &newval1) &&
00750     !CheckWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, *p, &newval2) ) {
00751     nt = p[2];
00752     mt = m[2];
00753     if ( ( mt > 0 && nt >= mt ) ||
00754         ( mt < 0 && nt <= mt ) ) {
00755         OnceL10: AddWild(BHEAD *m-WILDOFFSET, VECTOVEC, newval1);
00756         AddWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, newval2);
00757         break;
00758     }
00759     if ( mt >= 2*MAXPOWER ) {
00760         OnceL10a: mt -= 2*MAXPOWER;
00761         if ( CheckWild(BHEAD mt, SYMTONUM, nt, &newval3) ) {
00762             if ( AN.oldtype == SYMTONUM ) {
00763                 if ( AN.oldvalue >= 0 ) {
00764                     if ( nt > AN.oldvalue ) nt = AN.oldvalue;
00765                     else {
00766                         if ( *AN.MaskPointer == 2 ) return(0);
00767                         if ( nt < 0 ) nt = 0;
00768                     }
00769                 }
00770                 else {
00771                     if ( nt < AN.oldvalue ) nt = AN.oldvalue;
00772                     else {
00773                         if ( *AN.MaskPointer == 2 ) return(0);
00774                         if ( nt > 0 ) nt = 0;
00775                     }
00776                 }
00777                 AddWild(BHEAD mt, SYMTONUM, nt);
00778                 goto OnceL10;
00779             }
00780         }
00781         else {
00782             AddWild(BHEAD mt, SYMTONUM, nt);
00783             goto OnceL10;
00784         }
00785     }
00786     else if ( mt <= -2*MAXPOWER ) {
00787         mt = -mt;
00788         nt = -nt;
00789         goto OnceL10a;
00790     }
00791 }
00792 NextInDot:
00793     p += 3; nq -= 3;
00794 }
00795 if ( nq <= 0 ) return(0);
00796 else {
00797     q = p+3;
00798     nq -= 3;
00799     n -= 3;
00800     while ( --nq >= 0 ) *p++ = *q++;
00801 }
00802 m += 3;
00803 }
00804 else if ( m[1] >= (AM.OffsetVector+WILDOFFSET) ) {
00805     while ( *m >= *t && t < xstop ) {
00806         *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
00807     }
00808     nq = n;
00809     p = older;
00810     while ( nq > 0 ) {
00811         if ( *m == *p && !CheckWild(BHEAD m[1]-WILDOFFSET,
00812             VECTOVEC, p[1], &newval1) ) {
00813             nt = p[2];

```

```

00814         mt = m[2];
00815         if ( ( mt > 0 && nt >= mt ) ||
00816             ( mt < 0 && nt <= mt ) ) {
00817             AddWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, newval1);
00818             break;
00819         }
00820         else if ( mt >= 2*MAXPOWER ) {
00821             mt -= 2*MAXPOWER;
00822             OnceL7a:
00823             if ( CheckWild(BHEAD mt, SYMTONUM, nt, &newval3) ) {
00824                 if ( AN.oldtype == SYMTONUM ) {
00825                     if ( AN.oldvalue >= 0 ) {
00826                         if ( nt > AN.oldvalue ) nt = AN.oldvalue;
00827                         else {
00828                             if ( *AN.MaskPointer == 2 ) return(0);
00829                             if ( nt < 0 ) nt = 0;
00830                         }
00831                     }
00832                     else {
00833                         if ( nt < AN.oldvalue ) nt = AN.oldvalue;
00834                         else {
00835                             if ( *AN.MaskPointer == 2 ) return(0);
00836                             if ( nt > 0 ) nt = 0;
00837                         }
00838                     }
00839                     AddWild(BHEAD mt, SYMTONUM, nt);
00840                     AddWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, newval1);
00841                     break;
00842                 }
00843             }
00844             else {
00845                 AddWild(BHEAD mt, SYMTONUM, nt);
00846                 AddWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, newval1);
00847                 break;
00848             }
00849         }
00850         else if ( mt <= -2*MAXPOWER ) {
00851             mt = -mt;
00852             nt = -nt;
00853             goto OnceL7a;
00854         }
00855     }
00856     if ( *m == p[1] && !CheckWild(BHEAD m[1]-WILDOFFSET,
00857         VECTOVEC, *p, &newval1) ) {
00858         nt = p[2];
00859         mt = m[2];
00860         if ( ( mt > 0 && nt >= mt ) ||
00861             ( mt < 0 && nt <= mt ) ) {
00862             AddWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, newval1);
00863             break;
00864         }
00865         if ( mt >= 2*MAXPOWER ) {
00866             mt -= 2*MAXPOWER;
00867             OnceL8a:
00868             if ( CheckWild(BHEAD mt, SYMTONUM, nt, &newval3) ) {
00869                 if ( AN.oldtype == SYMTONUM ) {
00870                     if ( AN.oldvalue >= 0 ) {
00871                         if ( nt > AN.oldvalue ) nt = AN.oldvalue;
00872                         else {
00873                             if ( *AN.MaskPointer == 2 ) return(0);
00874                             if ( nt < 0 ) nt = 0;
00875                         }
00876                     }
00877                     else {
00878                         if ( nt < AN.oldvalue ) nt = AN.oldvalue;
00879                         else {
00880                             if ( *AN.MaskPointer == 2 ) return(0);
00881                             if ( nt > 0 ) nt = 0;
00882                         }
00883                     }
00884                     AddWild(BHEAD mt, SYMTONUM, nt);
00885                     AddWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, newval1);
00886                     break;
00887                 }
00888             }
00889             else {
00890                 AddWild(BHEAD mt, SYMTONUM, nt);
00891                 AddWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, newval1);
00892                 break;
00893             }
00894         }
00895         else if ( mt < -2*MAXPOWER ) {
00896             mt = -mt;
00897             nt = -nt;
00898             goto OnceL8a;
00899         }
00900     }
    }
    p += 3; nq -= 3;
}

```

```

00901         if ( nq <= 0 ) return(0);
00902         q = p+3;
00903         nq -= 3;
00904         n -= 3;
00905         while ( --nq >= 0 ) *p++ = *q++;
00906         m += 3;
00907     }
00908     else {
00909         if ( t >= xstop || *m < *t || ( *m == *t && m[1] < t[1] ) ) {
00910             if ( m[2] >= 2*MAXPOWER ) nt = m[2];
00911             else if ( m[2] <= -2*MAXPOWER ) nt = -m[2];
00912             else return(0);
00913             nt -= 2*MAXPOWER;
00914             if ( CheckWild(BHEAD nt, SYMTONUM, 0, &newval3) ) {
00915                 if ( AN.oldtype != SYMTONUM ) return(0);
00916                 if ( *AN.MaskPointer == 2 ) return(0);
00917             }
00918             AddWild(BHEAD nt, SYMTONUM, 0);
00919             m += 3;
00920         }
00921         else {
00922             *p++ = *t++; *p++ = *t++; *p++ = *t++; n += 3;
00923         }
00924     }
00925     } while ( m < ystop );
00926     t = xstop;
00927 }
00928 /*
00929     #] DOTPRODUCTS :
00930 */
00931     else {
00932         MLOCK(ErrorMessageLock);
00933         MesPrint("Error in pattern(2)");
00934         MUNLOCK(ErrorMessageLock);
00935         Terminate(-1);
00936     }
00937 EndLoop:;
00938     } while ( m < mstop ); }
00939     else {
00940         return(-1);
00941     }
00942     return(1);
00943 }
00944
00945 /*
00946     #] FindOnce :
00947     #[ FindMulti :          WORD FindMulti(term, pattern)
00948
00949     Note that multi cannot deal with wildcards. Those patterns revert
00950     to many which gives subsequent calls to once.
00951
00952 */
00953
00954 WORD FindMulti(PHEAD WORD *term, WORD *pattern)
00955 {
00956     GETBIDENTITY
00957     WORD *t, *m, *p;
00958     WORD *tstop, *mstop;
00959     WORD *xstop, *ystop;
00960     WORD mt, power, n, nq;
00961     WORD older[2*NORMSIZE], *q, newval1;
00962     AN.UsedOtherFind = 0;
00963     m = pattern;
00964     mstop = m + *m;
00965     m++;
00966     t = term;
00967     t += *term - 1;
00968     tstop = t - ABS(*t) + 1;
00969     t = term;
00970     t++;
00971     while ( t < tstop && *t > DOTPRODUCT ) t += t[1];
00972     while ( m < mstop && *m > DOTPRODUCT ) m += m[1];
00973     power = -1;
00974     if ( m < mstop ) { do {
00975         /*
00976             #] SYMBOLS :
00977         */
00978         if ( *m == SYMBOL ) {
00979             ystop = m + m[1];
00980             m += 2;
00981             if ( t >= tstop ) return(0);
00982             while ( *t != SYMBOL ) { t += t[1]; if ( t >= tstop ) return(0); }
00983             xstop = t + t[1];
00984             t += 2;
00985             p = older;
00986             n = 0;
00987             do {

```

```

00988         if ( *m >= 2*MAXPOWER ) {
00989             while ( t < xstop ) { *p++ = *t++; *p++ = *t++; n += 2; }
00990             nq = n;
00991             p = older;
00992             while ( nq > 0 ) {
00993                 if ( !CheckWild(BHEAD *m-2*MAXPOWER, SYMTOSYM, *p, &newvall) ) {
00994                     mt = p[1]/m[1];
00995                     if ( mt > 0 ) {
00996                         if ( power < 0 || mt < power ) power = mt;
00997                         AddWild(BHEAD *m-2*MAXPOWER, SYMTOSYM, newvall);
00998                         break;
00999                     }
01000                 }
01001                 nq -= 2;
01002                 p += 2;
01003             }
01004             if ( nq <= 0 ) return(0);
01005             nq -= 2;
01006             n -= 2;
01007             q = p + 2;
01008             while ( --nq >= 0 ) *p++ = *q++;
01009             m += 2;
01010         }
01011         else if ( t >= xstop ) return(0);
01012         else if ( *m == *t ) {
01013             if ( ( mt = t[1]/m[1] ) <= 0 ) return(0);
01014             if ( power < 0 || mt < power ) power = mt;
01015             m += 2;
01016             t += 2;
01017         }
01018         else if ( *m < *t ) return(0);
01019         else { *p++ = *t++; *p++ = *t++; n += 2; }
01020     } while ( m < ystop );
01021 }
01022 /*
01023     #] SYMBOLS :
01024     #[ DOTPRODUCTS :
01025 */
01026 else if ( *m == DOTPRODUCT ) {
01027     ystop = m + m[1];
01028     m += 2;
01029     if ( t >= tstop ) return(0);
01030     while ( *t != DOTPRODUCT ) { t += t[1]; if ( t >= tstop ) return(0); }
01031     xstop = t + t[1];
01032     t += 2;
01033     do {
01034         if ( t >= xstop ) return(0);
01035         if ( *t == *m ) {
01036             if ( t[1] == m[1] ) {
01037                 if ( ( mt = t[2]/m[2] ) <= 0 ) return(0);
01038                 if ( power < 0 || mt < power ) power = mt;
01039                 m += 3;
01040             }
01041             else if ( t[1] > m[1] ) return(0);
01042         }
01043         else if ( *t > *m ) return(0);
01044         t += 3;
01045     } while ( m < ystop );
01046     t = xstop;
01047 }
01048 /*
01049     #] DOTPRODUCTS :
01050 */
01051 else {
01052     MLOCK(ErrorMessageLock);
01053     MesPrint("Error in pattern(3)");
01054     MUNLOCK(ErrorMessageLock);
01055     Terminate(-1);
01056 }
01057 } while ( m < mstop ); }
01058 if ( power < 0 ) power = 0;
01059 return(power);
01060 }
01061
01062 /*
01063     #] FindMulti :
01064     #[ FindRest :          WORD FindRest(term,pattern)
01065
01066     This routine scans for anything but dotproducts and symbols.
01067
01068 */
01069
01070 int FindRest(PHEAD WORD *term, WORD *pattern)
01071 {
01072     GETBIDENTITY
01073     WORD *t, *m, *tt, wild, regular;
01074     WORD *tstop, *mstop;

```



```

01075     WORD *xstop, *ystop;
01076     WORD n, *p, nq;
01077     WORD older[NORMSIZE], *q, newval1, newval2;
01078     int i, ntw;
01079     AN.UsedOtherFind = 0;
01080     AN.findTerm = term; AN.findPattern = pattern;
01081     m = AN.WildValue;
01082     i = (m[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
01083     ntw = 0;
01084     while ( i > 0 ) {
01085         if ( m[0] == ARGTOARG ) ntw++;
01086         m += m[1];
01087         i--;
01088     }
01089     t = term;
01090     t += *term - 1;
01091     tstop = t - ABS(*t) + 1;
01092     t = term;
01093     t++; p = t;
01094     while ( t < tstop && *t > DOTPRODUCT ) t += t[1];
01095     tstop = t;
01096     t = p;
01097     m = pattern;
01098     mstop = m + *m;
01099     m++;
01100     p = m;
01101     while ( m < mstop && *m > DOTPRODUCT ) m += m[1];
01102     mstop = m;
01103     m = p;
01104     if ( m < mstop ) {
01105         do {
01106             /*
01107                 #[ FUNCTIONS :
01108             */
01109             if ( *m >= FUNCTION ) {
01110                 if ( *mstop > 5 && !MatchIsPossible(pattern,term) ) return(0);
01111                 ystop = m;
01112                 n = 0;
01113                 do {
01114                     m += m[1]; n++;
01115                 } while ( m < mstop && *m >= FUNCTION );
01116                 AT.WorkPointer += n;
01117                 while ( t < tstop && *t == SUBEXPRESSION ) t += t[1];
01118                 tt = xstop = t;
01119                 nq = 0;
01120                 while ( t < tstop && ( *t >= FUNCTION || *t == SUBEXPRESSION ) ) {
01121                     if ( *t != SUBEXPRESSION ) {
01122                         nq++;
01123                         if ( functions[*t-FUNCTION].commute ) tt = t + t[1];
01124                     }
01125                     t += t[1];
01126                 }
01127                 if ( nq < n ) return(0);
01128                 AN.terstart = term;
01129                 AN.terstop = t;
01130                 AN.terfirstcomm = tt;
01131                 AN.patstop = m;
01132                 AN.NumTotWildArgs = ntw;
01133                 if ( !ScanFunctions(BHEAD ystop,xstop,0) ) return(0);
01134             }
01135             /*
01136                 #] FUNCTIONS :
01137                 #[ VECTORS :
01138             */
01139             else if ( *m == VECTOR ) {
01140                 while ( t < tstop && *t != VECTOR ) t += t[1];
01141                 if ( t >= tstop ) return(0);
01142                 xstop = t + t[1];
01143                 ystop = m + m[1];
01144                 t += 2;
01145                 m += 2;
01146                 n = 0;
01147                 p = older;
01148                 do {
01149                     if ( *m == *t && m[1] == t[1] && t < xstop ) {
01150                         m += 2; t += 2;
01151                     }
01152                     else if ( *m >= (AM.OffsetVector+WILDOFFSET) ) {
01153                         if ( t < xstop ) {
01154                             p = older + n;
01155                             do { *p++ = *t++; n++; } while ( t < xstop );
01156                         }
01157                         p = older;
01158                         nq = n;
01159                         if ( ( m[1] < (AM.OffsetIndex+WILDOFFSET) )
01160                             || ( m[1] >= (AM.OffsetIndex+2*WILDOFFSET) ) ) {
01161                             while ( nq > 0 ) {

```

```

01162             if ( m[l] == p[l] ) {
01163                 if ( !CheckWild(BHEAD *m-WILDOFFSET, VECTOVEC, *p, &newval1) ) {
01164 RestL11:             AddWild(BHEAD *m-WILDOFFSET, VECTOVEC, newval1);
01165                     break;
01166                 }
01167             }
01168             p += 2;
01169             nq -= 2;
01170         }
01171     }
01172     else {          /* Double wildcard */
01173         while ( nq > 0 ) {
01174             if ( !CheckWild(BHEAD *m-WILDOFFSET, VECTOVEC, *p, &newval1) &&
01175                 !CheckWild(BHEAD m[l]-WILDOFFSET, INDTOIND, p[l], &newval2) ) {
01176                 AddWild(BHEAD m[l]-WILDOFFSET, INDTOIND, newval2);
01177                 goto RestL11;
01178             }
01179             p += 2;
01180             nq -= 2;
01181         }
01182     }
01183     if ( nq > 0 ) {
01184         nq -= 2; q = p + 2; n -= 2;
01185         while ( --nq >= 0 ) *p++ = *q++;
01186     }
01187     else return(0);
01188     m += 2;
01189 }
01190 else if ( ( *m <= *t )
01191 && ( m[l] >= (AM.OffsetIndex + WILDOFFSET) )
01192 && ( m[l] < (AM.OffsetIndex + 2*WILDOFFSET) ) ) {
01193     if ( *m == *t && t < xstop ) {
01194         p = older;
01195         p += n;
01196         *p++ = *t++;
01197         *p++ = *t++;
01198         n += 2;
01199     }
01200     p = older;
01201     nq = n;
01202     while ( nq > 0 ) {
01203         if ( *m == *p ) {
01204             if ( !CheckWild(BHEAD m[l]-WILDOFFSET, INDTOIND, p[l], &newval1) ) {
01205                 AddWild(BHEAD m[l]-WILDOFFSET, INDTOIND, newval1);
01206                 break;
01207             }
01208         }
01209         p += 2;
01210         nq -= 2;
01211     }
01212     if ( nq > 0 ) {
01213         nq -= 2; q = p + 2; n -= 2;
01214         while ( --nq >= 0 ) *p++ = *q++;
01215     }
01216     else return(0);
01217     m += 2;
01218 }
01219 else {
01220     if ( t >= xstop ) return(0);
01221     *p++ = *t++; *p++ = *t++; n += 2;
01222 }
01223 } while ( m < ystop );
01224 }
01225 /*
01226 #] VECTORS :
01227 #[ INDICES :
01228 */
01229 else if ( *m == INDEX ) {
01230 /*
01231     This needs only to say that there is a match, after matching
01232     a 'wildcard'. This has to be prepared in TestMatch. The C->rhs
01233     should provide the replacement inside the prototype!
01234     Next question: id,p=q/2+r/2
01235 */
01236     while ( *t != INDEX ) { t += t[l]; if ( t >= tstop ) return(0); }
01237     xstop = t + t[l];
01238     ystop = m + m[l];
01239     t += 2;
01240     m += 2;
01241     n = 0;
01242     p = older;
01243     do {
01244         if ( *m == *t && t < xstop && m < ystop ) {
01245             t++; m++;
01246         }
01247         else if ( ( *m >= (AM.OffsetIndex+WILDOFFSET) )
01248             && ( *m < (AM.OffsetIndex+2*WILDOFFSET) ) ) {

```

```

01249         while ( t < xstop ) {
01250             *p++ = *t++; n++;
01251         }
01252         if ( !n ) return(0);
01253         nq = n;
01254         q = older;
01255         do {
01256             if ( !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,*q,&newval1) ) {
01257                 AddWild(BHEAD *m-WILDOFFSET,INDTOIND,newval1);
01258                 break;
01259             }
01260             q++;
01261             nq--;
01262         } while ( nq > 0 );
01263         if ( nq <= 0 ) return (0);
01264         n--;
01265         nq--;
01266         p = q + 1;
01267         while ( nq > 0 ) { *q++ = *p++; nq--; }
01268         p--;
01269         m++;
01270     }
01271     else if ( ( *m >= (AM.OffsetVector+WILDOFFSET) )
01272     && ( *m < (AM.OffsetVector+2*WILDOFFSET) ) ) {
01273         while ( t < xstop ) {
01274             *p++ = *t++; n++;
01275         }
01276         if ( !n ) return(0);
01277         nq = n;
01278         q = older;
01279         do {
01280             if ( !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*q,&newval1) ) {
01281                 AddWild(BHEAD *m-WILDOFFSET,VECTOVEC,newval1);
01282                 break;
01283             }
01284             q++;
01285             nq--;
01286         } while ( nq > 0 );
01287         if ( nq <= 0 ) return (0);
01288         n--;
01289         nq--;
01290         p = q + 1;
01291         while ( nq > 0 ) { *q++ = *p++; nq--; }
01292         p--;
01293         m++;
01294     }
01295     else {
01296         if ( t >= xstop ) return(0);
01297         *p++ = *t++; n++;
01298     }
01299 } while ( m < ystop );
01300
01301 /*
01302     return(0);
01303 */
01304 }
01305 /*
01306     #] INDICES :
01307     #[ DELTAS :
01308 */
01309 else if ( *m == DELTA ) {
01310     while ( *t != DELTA ) { t += t[1]; if ( t >= tstop ) return(0); }
01311     xstop = t + t[1];
01312     ystop = m + m[1];
01313     t += 2;
01314     m += 2;
01315     n = 0;
01316     p = older;
01317     do {
01318         if ( *t == *m && t[1] == m[1] && t < xstop ) {
01319             m += 2;
01320             t += 2;
01321         }
01322         else if ( ( *m >= (AM.OffsetIndex+WILDOFFSET) )
01323         && ( *m < (AM.OffsetIndex+2*WILDOFFSET) )
01324         && ( m[1] >= (AM.OffsetIndex+WILDOFFSET) )
01325         && ( m[1] < (AM.OffsetIndex+2*WILDOFFSET) ) ) { /* Two dummies */
01326             while ( t < xstop ) {
01327                 *p++ = *t++; *p++ = *t++; n += 2;
01328             }
01329             if ( !n ) return(0);
01330             nq = n;
01331             q = older;
01332             do {
01333                 if ( !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,*q,&newval1) &&
01334                 !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,q[1],&newval2) ) {
01335                     AddWild(BHEAD *m-WILDOFFSET,INDTOIND,newval1);

```

```

01336             AddWild(BHEAD m[1]-WILDOFFSET,INDTOIND,newval2);
01337             break;
01338         }
01339         if ( !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,q[1],&newval1) &&
01340             !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,*q,&newval2) ) {
01341             AddWild(BHEAD *m-WILDOFFSET,INDTOIND,newval1);
01342             AddWild(BHEAD m[1]-WILDOFFSET,INDTOIND,newval2);
01343             break;
01344         }
01345         q += 2;
01346         nq -= 2;
01347     } while ( nq > 0 );
01348     if ( nq <= 0 ) return(0);
01349     n -= 2;
01350     nq -= 2;
01351     p = q + 2;
01352     while ( nq > 0 ) { *q++ = *p++; nq--; }
01353     p -= 2;
01354     m += 2;
01355 }
01356 else if ( ( m[1] >= (AM.OffsetIndex+WILDOFFSET) )
01357 && ( m[1] < (AM.OffsetIndex+2*WILDOFFSET) ) ) {
01358     wild = m[1]; regular = *m;
01359 OneWild:
01360     while ( ( regular == *t || regular == t[1] ) && t < xstop ) {
01361         *p++ = *t++; *p++ = *t++; n += 2;
01362     }
01363     if ( !n ) return(0);
01364     nq = n;
01365     q = older;
01366     do {
01367         if ( regular == *q && !CheckWild(BHEAD wild-WILDOFFSET,INDTOIND,q[1],&newval1)
01368         ) {
01369             AddWild(BHEAD wild-WILDOFFSET,INDTOIND,newval1);
01370             break;
01371         }
01372         if ( regular == q[1] && !CheckWild(BHEAD wild-WILDOFFSET,INDTOIND,*q,&newval1)
01373         ) {
01374             AddWild(BHEAD wild-WILDOFFSET,INDTOIND,newval1);
01375             break;
01376         }
01377         q += 2;
01378         nq -= 2;
01379     } while ( nq > 0 );
01380     if ( nq <= 0 ) return(0);
01381     n -= 2;
01382     nq -= 2;
01383     p = q + 2;
01384     while ( nq > 0 ) { *q++ = *p++; nq--; }
01385     p -= 2;
01386     m += 2;
01387 }
01388 else if ( ( *m >= (AM.OffsetIndex+WILDOFFSET) )
01389 && ( *m < (AM.OffsetIndex+2*WILDOFFSET) ) ) {
01390     wild = *m; regular = m[1];
01391     goto OneWild;
01392 }
01393 else {
01394     if ( t >= tstop || *m < *t || ( *m == *t && m[1] < t[1] ) )
01395         return(0);
01396     *p++ = *t++; *p++ = *t++; n += 2;
01397 } while ( m < ystop );
01398 }
01399 /*
01400 #] DELTAS :
01401 */
01402 else {
01403     MLOCK(ErrorMessageLock);
01404     MesPrint("Pattern not yet implemented");
01405     MUNLOCK(ErrorMessageLock);
01406     Terminate(-1);
01407 }
01408 } while ( m < mstop );
01409 return(1);
01410 }
01411 else return(-1);
01412 }
01413 /*
01414 #] FindRest :
01415 #] Patterns :
01416 */

```


4.40 flintinterface.cc

[Go to the documentation of this file.](#)

```

00001
00006 /* #[] License : */
00007 /*
00008 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009 *   When using this file you are requested to refer to the publication
00010 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011 *   This is considered a matter of courtesy as the development was paid
00012 *   for by FOM the Dutch physics granting agency and we would like to
00013 *   be able to track its scientific use to convince FOM of its value
00014 *   for the community.
00015 *
00016 *   This file is part of FORM.
00017 *
00018 *   FORM is free software: you can redistribute it and/or modify it under the
00019 *   terms of the GNU General Public License as published by the Free Software
00020 *   Foundation, either version 3 of the License, or (at your option) any later
00021 *   version.
00022 *
00023 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00026 *   details.
00027 *
00028 *   You should have received a copy of the GNU General Public License along
00029 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00030 */
00031 /* #] License : */
00032
00033 #ifdef HAVE_CONFIG_H
00034 #ifndef CONFIG_H_INCLUDED
00035 #define CONFIG_H_INCLUDED
00036 #include <config.h>
00037 #endif
00038 #endif
00039
00040 extern "C" {
00041 #include "form3.h"
00042 }
00043
00044 #include "flintinterface.h"
00045
00046 /*
00047 * #[] Types
00048 * Use fixed-size integer types (int32_t, uint32_t, int64_t, uint64_t) except when we refer
00049 * specifically to FORM term data, when we use WORD. This code has only been tested on 64bit
00050 * systems where WORDs are int32_t, and some functions (fmpz_get_form, fmpz_set_form) require this
00051 * to be the case. Enforce this:
00052 */
00053 static_assert(sizeof(WORD) == sizeof(int32_t), "flint interface expects 32bit WORD");
00054 static_assert(sizeof(WORD) == sizeof(uint32_t), "flint interface expects 32bit WORD");
00055 static_assert(BITSINWORD == 32, "flint interface expects 32bit WORD");
00056 static_assert(sizeof(ulong) == sizeof(uint64_t), "flint interface expects ulong is uint64_t");
00057 static_assert(sizeof(slong) == sizeof(int64_t), "flint interface expects slong is int64_t");
00058
00059 /*
00060 * Flint functions take arguments or return values which may be "slong" or "ulong" in its
00061 * documentation, and these are int64_t and uint64_t respectively.
00062 *
00063 * #] Types
00064 */
00065
00066 /*
00067 * FLINT's univariate poly has a dense representation. For sufficiently sparse polynomials it is
00068 * faster to use mpoly instead, which is sparse. For a density <= this threshold we switch, where
00069 * the density defined as is "number of terms" / "maximum degree".
00070 */
00071 #define UNIVARIATE_DENSITY_THR 0.02f
00072
00073 /*
00074 * #[] flint::cleanup :
00075 */
00076 void flint::cleanup(void) {
00077     flint_cleanup();
00078 }
00079 /*
00080 * #] flint::cleanup :
00081 * #[] flint::cleanup_master :
00082 */
00083 void flint::cleanup_master(void) {
00084     flint_cleanup_master();
00085 }
00086 */

```

```

00087     #] flint::cleanup_master :
00088
00089     #[ flint::divmod_mpoly :
00090     */
00091 WORD* flint::divmod_mpoly(PHEAD const WORD *a, const WORD *b, const bool return_rem,
00092     const WORD must_fit_term, const var_map_t &var_map) {
00093
00094     flint::mpoly_ctx ctx(var_map.size());
00095     flint::mpoly pa(ctx.d), pb(ctx.d), denpa(ctx.d), denpb(ctx.d);
00096
00097     flint::from_argument_mpoly(pa.d, denpa.d, a, false, var_map, ctx.d);
00098     flint::from_argument_mpoly(pb.d, denpb.d, b, false, var_map, ctx.d);
00099
00100     // The input won't have any symbols with negative powers, but there may be rational
00101     // coefficients. Verify this:
00102     if ( fmpz_mpoly_is_fmpz(denpa.d, ctx.d) != 1 ) {
00103         MLOCK(ErrorMessageLock);
00104         MesPrint("flint::divmod_mpoly: error: denpa is non-constant");
00105         MUNLOCK(ErrorMessageLock);
00106         Terminate(-1);
00107     }
00108     if ( fmpz_mpoly_is_fmpz(denpb.d, ctx.d) != 1 ) {
00109         MLOCK(ErrorMessageLock);
00110         MesPrint("flint::divmod_mpoly: error: denpb is non-constant");
00111         MUNLOCK(ErrorMessageLock);
00112         Terminate(-1);
00113     }
00114
00115     flint::fmpz scale;
00116     flint::mpoly div(ctx.d), rem(ctx.d);
00117
00118     fmpz_mpoly_quasidivrem(scale.d, div.d, rem.d, pa.d, pb.d, ctx.d);
00119
00120     // The quotient must be multiplied by the denominator of the divisor
00121     fmpz_mpoly_mul(div.d, div.d, denpb.d, ctx.d);
00122
00123     // The overall denominator of both div and rem is given by scale*denpa. This we will pass to
00124     // to_argument_mpoly's "denscale" argument which performs the division in the result. We have
00125     // already checked denpa is just an fmpz.
00126     fmpz_mul(scale.d, scale.d, fmpz_mpoly_term_coeff_ref(denpa.d, 0, ctx.d));
00127
00128     // Determine the size of the output by passing write = false.
00129     const bool with_arghead = false;
00130     bool write = false;
00131     const uint64_t prev_size = 0;
00132     const uint64_t out_size = return_rem ?
00133         (uint64_t)flint::to_argument_mpoly(BHEAD NULL, with_arghead, must_fit_term, write, prev_size,
00134         rem.d, var_map, ctx.d, scale.d)
00135         :
00136         (uint64_t)flint::to_argument_mpoly(BHEAD NULL, with_arghead, must_fit_term, write, prev_size,
00137         div.d, var_map, ctx.d, scale.d)
00138         ;
00139     WORD* res = (WORD *)Malloc1(sizeof(WORD)*out_size, "flint::divrem_mpoly");
00140
00141     // Write out the result
00142     write = true;
00143     if ( return_rem ) {
00144         (uint64_t)flint::to_argument_mpoly(BHEAD res, with_arghead, must_fit_term, write, prev_size,
00145         rem.d, var_map, ctx.d, scale.d);
00146     }
00147     else {
00148         (uint64_t)flint::to_argument_mpoly(BHEAD res, with_arghead, must_fit_term, write, prev_size,
00149         div.d, var_map, ctx.d, scale.d);
00150     }
00151     return res;
00152 }
00153
00154 /*
00155 #] flint::divmod_mpoly :
00156 #[ flint::divmod_poly :
00157 */
00158 WORD* flint::divmod_poly(PHEAD const WORD *a, const WORD *b, const bool return_rem,
00159     const WORD must_fit_term, const var_map_t &var_map) {
00160
00161     flint::poly pa, pb, denpa, denpb;
00162
00163     flint::from_argument_poly(pa.d, denpa.d, a, false);
00164     flint::from_argument_poly(pb.d, denpb.d, b, false);
00165
00166     // The input won't have any symbols with negative powers, but there may be rational
00167     // coefficients. Verify this:
00168     if ( fmpz_poly_length(denpa.d) != 1 ) {
00169         MLOCK(ErrorMessageLock);
00170         MesPrint("flint::divmod_poly: error: denpa is non-constant");
00171     }

```

```

00174     MUNLOCK(ErrorMessageLock);
00175     Terminate(-1);
00176 }
00177 if ( fmpz_poly_length(denpb.d) != 1 ) {
00178     MLOCK(ErrorMessageLock);
00179     MesPrint("flint::divmod_poly: error: denpb is non-constant");
00180     MUNLOCK(ErrorMessageLock);
00181     Terminate(-1);
00182 }
00183
00184 flint::fmpz scale;
00185 uint64_t scale_power = 0;
00186 flint::poly div, rem;
00187
00188 // Get the leading coefficient of pb. fmpz_poly_pseudo_divrem returns only the scaling power.
00189 fmpz_poly_get_coeff_fmpz(scale.d, pb.d, fmpz_poly_degree(pb.d));
00190
00191 fmpz_poly_pseudo_divrem(div.d, rem.d, (ulong*)&scale_power, pa.d, pb.d);
00192
00193 // The quotient must be multiplied by the denominator of the divisor
00194 fmpz_poly_mul(div.d, div.d, denpb.d);
00195
00196 // The overall denominator of both div and rem is given by scale^scale_power*denpa. This we will
00197 // pass to to_argument_poly's "denscale" argument which performs the division in the result. We
00198 // have already checked denpa is just an fmpz.
00199 fmpz_pow_ui(scale.d, scale.d, scale_power);
00200 fmpz_mul(scale.d, scale.d, fmpz_poly_get_coeff_ptr(denpa.d, 0));
00201
00202
00203 // Determine the size of the output by passing write = false.
00204 const bool with_arghead = false;
00205 bool write = false;
00206 const uint64_t prev_size = 0;
00207 const uint64_t out_size = return_rem ?
00208     (uint64_t)flint::to_argument_poly(BHEAD NULL, with_arghead, must_fit_term, write, prev_size,
00209         rem.d, var_map, scale.d)
00210     :
00211     (uint64_t)flint::to_argument_poly(BHEAD NULL, with_arghead, must_fit_term, write, prev_size,
00212         div.d, var_map, scale.d)
00213     ;
00214 WORD* res = (WORD *)Malloca(sizeof(WORD)*out_size, "flint::divrem_poly");
00215
00216
00217 // Write out the result
00218 write = true;
00219 if ( return_rem ) {
00220     (uint64_t)flint::to_argument_poly(BHEAD res, with_arghead, must_fit_term, write, prev_size,
00221         rem.d, var_map, scale.d);
00222 }
00223 else {
00224     (uint64_t)flint::to_argument_poly(BHEAD res, with_arghead, must_fit_term, write, prev_size,
00225         div.d, var_map, scale.d);
00226 }
00227
00228 return res;
00229 }
00230 /*
00231 #] flint::divmod_poly :
00232
00233 #[ flint::factorize_mpoly :
00234 */
00235 WORD* flint::factorize_mpoly(PHEAD const WORD *argin, WORD *argout, const bool with_arghead,
00236     const bool is_fun_arg, const var_map_t &var_map) {
00237
00238     flint::mpoly_ctx ctx(var_map.size());
00239     flint::mpoly arg(ctx.d, den(ctx.d), base(ctx.d));
00240
00241     flint::from_argument_mpoly(arg.d, den.d, argin, with_arghead, var_map, ctx.d);
00242     // The denominator must be 1:
00243     if ( fmpz_poly_is_one(den.d, ctx.d) != 1 ) {
00244         MLOCK(ErrorMessageLock);
00245         MesPrint("flint::factorize_mpoly error: den != 1");
00246         MUNLOCK(ErrorMessageLock);
00247         Terminate(-1);
00248     }
00249
00250
00251 // Now we can factor the mpoly:
00252 flint::mpoly_factor arg_fac(ctx.d);
00253 fmpz_mpoly_factor(arg_fac.d, arg.d, ctx.d);
00254 const int64_t num_factors = fmpz_mpoly_factor_length(arg_fac.d, ctx.d);
00255
00256 flint::fmpz overall_constant;
00257 fmpz_mpoly_factor_get_constant_fmpz(overall_constant.d, arg_fac.d, ctx.d);
00258 // FORM should always have taken the overall constant out in the content. Thus this overall
00259 // constant factor should be +- 1 here. Verify this:
00260 if ( ! ( fmpz_equal_si(overall_constant.d, 1) || fmpz_equal_si(overall_constant.d, -1) ) ) {

```



```

00261     MLOCK(ErrorMessageLock);
00262     MesPrint("flint::factorize_mpoly error: overall constant factor != +-1");
00263     MUNLOCK(ErrorMessageLock);
00264     Terminate(-1);
00265 }
00266
00267 // Construct the output. If argout is not NULL, we write the result there.
00268 // Otherwise, allocate memory.
00269 // The output is zero-terminated list of factors. If with_arghead, each has an arghead which
00270 // contains its size. Otherwise, the factors are zero separated.
00271
00272 // We only need to determine the size of the output if we are allocating memory, but we need to
00273 // loop through the factors to fix their signs anyway. Do both together in one loop:
00274
00275 // Initially 1, for the final trailing 0.
00276 uint64_t output_size = 1;
00277
00278 // For finding the highest symbol, in FORM's lexicographic ordering
00279 var_map_t var_map_inv;
00280 for ( auto x: var_map ) {
00281     var_map_inv[x.second] = x.first;
00282 }
00283
00284 // Store whether we should flip the factor sign in the ouput:
00285 vector<int32_t> base_sign(num_factors, 0);
00286
00287 for ( int64_t i = 0; i < num_factors; i++ ) {
00288     fmpz_mpoly_factor_get_base(base.d, arg_fac.d, i, ctx.d);
00289     const int64_t exponent = fmpz_mpoly_factor_get_exp_si(arg_fac.d, i, ctx.d);
00290
00291     // poly_factorize makes sure the highest power of the "highest symbol" (in FORM's
00292     // lexicographic ordering) has a positive coefficient. Check this, update overall_constant
00293     // of the factorization if necessary.
00294     // Store the sign per factor, so that we can flip the signs in the output without re-checking
00295     // the individual terms again.
00296     uint32_t max_var = 0; // FORM symbols start at 20, 0 is a good initial value.
00297     int32_t max_pow = -1;
00298     vector<int64_t> base_term_exponents(var_map.size(), 0);
00299
00300     for ( int64_t j = 0; j < fmpz_mpoly_length(base.d, ctx.d); j++ ) {
00301         fmpz_mpoly_get_term_exp_si((slong*)base_term_exponents.data(), base.d, (slong)j, ctx.d);
00302
00303         for ( size_t k = 0; k < var_map.size(); k++ ) {
00304             if ( base_term_exponents[k] > 0 && ( var_map_inv.at(k) > max_var ||
00305                 ( var_map_inv.at(k) == max_var && base_term_exponents[k] > max_pow ) ) ) {
00306
00307                 max_var = var_map_inv.at(k);
00308                 max_pow = base_term_exponents[k];
00309                 base_sign[i] = fmpz_sgn(fmpz_mpoly_term_coeff_ref(base.d, j, ctx.d));
00310             }
00311         }
00312     }
00313     // If this base's sign will be flipped an odd number of times, there is a contribution to
00314     // the overall sign of the whole factorization:
00315     if ( ( base_sign[i] == -1 ) && ( exponent % 2 == 1 ) ) {
00316         fmpz_neg(overall_constant.d, overall_constant.d);
00317     }
00318
00319     // Now determine the output size of the factor, if we are allocating the memory
00320     if ( argout == NULL ) {
00321         const bool write = false;
00322         for ( int64_t j = 0; j < exponent; j++ ) {
00323             output_size += (uint64_t)flint::to_argument_mpoly(BHEAD NULL, with_arghead,
00324                 is_fun_arg, write, 0, base.d, var_map, ctx.d);
00325         }
00326     }
00327 }
00328 if ( fmpz_sgn(overall_constant.d) == -1 && argout == NULL ) {
00329     // Add space for a fast-notation number or a normal-notation number and zero separator
00330     output_size += with_arghead ? 2 : 4+1;
00331 }
00332
00333 // Now make the allocation if necessary:
00334 if ( argout == NULL ) {
00335     argout = (WORD*)Malloc1(sizeof(WORD)*output_size, "flint::factorize_mpoly");
00336 }
00337
00338 // And now comes the actual output:
00339 WORD* old_argout = argout;
00340
00341 // If the overall sign is negative, first write a full-notation -1. It will be absorbed into the
00342 // overall factor in the content by the caller.
00343 if ( fmpz_sgn(overall_constant.d) == -1 ) {
00344     if ( with_arghead ) {
00345         // poly writes in fast notation in this case. Fast notation is expected by the caller, to
00346         // properly merge it with the overall factor of the content.
00347     }

```

```

00348         *argout++ = -SNUMBER;
00349         *argout++ = -1;
00350     }
00351     else {
00352         *argout++ = 4; // term size
00353         *argout++ = 1; // numerator
00354         *argout++ = 1; // denominator
00355         *argout++ = -3; // coeff size, negative number
00356         *argout++ = 0; // factor separator
00357     }
00358 }
00359
00360 for ( int64_t i = 0; i < num_factors; i++ ) {
00361     fmpz_mpoly_factor_get_base(base.d, arg_fac.d, i, ctx.d);
00362     const int64_t exponent = fmpz_mpoly_factor_get_exp_si(arg_fac.d, i, ctx.d);
00363
00364     if ( base_sign[i] == -1 ) {
00365         fmpz_mpoly_neg(base.d, base.d, ctx.d);
00366     }
00367
00368     const bool write = true;
00369     for ( int64_t j = 0; j < exponent; j++ ) {
00370         argout += flint::to_argument_mpoly(BHEAD argout, with_arghead, is_fun_arg, write,
00371             argout-old_argout, base.d, var_map, ctx.d);
00372     }
00373 }
00374 // Final trailing zero to denote the end of the factors.
00375 *argout++ = 0;
00376
00377 return old_argout;
00378 }
00379 /*
00380 #] flint::factorize_mpoly :
00381 #[ flint::factorize_poly :
00382 */
00383 WORD* flint::factorize_poly(PHEAD const WORD *argin, WORD *argout, const bool with_arghead,
00384     const bool is_fun_arg, const var_map_t &var_map) {
00385
00386     flint::poly arg, den;
00387
00388     flint::from_argument_poly(arg.d, den.d, argin, with_arghead);
00389     // The denominator must be 1:
00390     if ( fmpz_poly_is_one(den.d) != 1 ) {
00391         MLOCK(ErrorMessageLock);
00392         MesPrint("flint::factorize_poly error: den != 1");
00393         MUNLOCK(ErrorMessageLock);
00394         Terminate(-1);
00395     }
00396
00397     // Now we can factor the poly:
00398     flint::poly_factor arg_fac;
00399     fmpz_poly_factor(arg_fac.d, arg.d);
00400     // fmpz_poly_factor_t lacks some convenience functions which fmpz_mpoly_factor_t has.
00401     // I have worked out how to get the factors by looking at how fmpz_poly_factor_print works.
00402     const long num_factors = (arg_fac.d)->num;
00403
00404
00405
00406     // Construct the output. If argout is not NULL, we write the result there.
00407     // Otherwise, allocate memory.
00408     // The output is zero-terminated list of factors. If with_arghead, each has an arghead which
00409     // contains its size. Otherwise, the factors are zero separated.
00410     if ( argout == NULL ) {
00411         // First we need to determine the size of the output. This is the same procedure as the
00412         // loop below, but we don't write anything in to_argument_poly (write arg: false).
00413         // Initially 1, for the final trailing 0.
00414         uint64_t output_size = 1;
00415
00416         for ( long i = 0; i < num_factors; i++ ) {
00417             fmpz_poly_struct* base = (arg_fac.d)->p + i;
00418
00419             const long exponent = (arg_fac.d)->exp[i];
00420
00421             const bool write = false;
00422             for ( long j = 0; j < exponent; j++ ) {
00423                 output_size += (uint64_t)flint::to_argument_poly(BHEAD NULL, with_arghead,
00424                     is_fun_arg, write, 0, base, var_map);
00425             }
00426         }
00427
00428         argout = (WORD*)Malloca(sizeof(WORD)*output_size, "flint::factorize_poly");
00429     }
00430
00431     WORD* old_argout = argout;
00432
00433     for ( long i = 0; i < num_factors; i++ ) {
00434         fmpz_poly_struct* base = (arg_fac.d)->p + i;

```

```

00435
00436     const long exponent = (arg_fac.d)->exp[i];
00437
00438     const bool write = true;
00439     for ( long j = 0; j < exponent; j++ ) {
00440         argout += flint::to_argument_poly(BHEAD argout, with_arghead, is_fun_arg, write,
00441             argout-old_argout, base, var_map);
00442     }
00443 }
00444 *argout = 0;
00445
00446 return old_argout;
00447 }
00448 /*
00449 #] flint::factorize_poly :
00450
00451 #[ flint::form_sort :
00452 */
00453 // Sort terms using form's sorting routines. Uses a custom (faster) compare routine, since here
00454 // only symbols can appear.
00455 // This is a modified poly_sort from polywrap.cc.
00456 void flint::form_sort(PHEAD WORD *terms) {
00457     if ( terms[0] < 0 ) {
00458         // Fast notation, there is nothing to do
00459         return;
00460     }
00461
00462     const WORD oldsorttype = AR.SortType;
00463     AR.SortType = SORTHIGFIRST;
00464
00465     const WORD in_size = terms[0] - ARGHEAD;
00466     WORD out_size;
00467
00468     if ( NewSort(BHEAD0) ) {
00469         Terminate(-1);
00470     }
00471
00472     AR.CompareRoutine = (COMPAREDUMMY)(&CompareSymbols);
00473
00474     // Make sure the symbols are in the right order within the terms
00475     for ( WORD i = ARGHEAD; i < terms[0]; i += terms[i] ) {
00476         if ( SymbolNormalize(terms+i) < 0 || StoreTerm(BHEAD terms+i) ) {
00477             AR.SortType = oldsorttype;
00478             AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
00479             LowerSortLevel();
00480             Terminate(-1);
00481         }
00482     }
00483
00484     if ( ( out_size = EndSort(BHEAD terms+ARGHEAD, 1) ) < 0 ) {
00485         AR.SortType = oldsorttype;
00486         AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
00487         Terminate(-1);
00488     }
00489
00490     // Check the final size
00491     if ( in_size != out_size ) {
00492         MLOCK(ErrorMessageLock);
00493         MesPrint("flint::form_sort: error: unexpected sorted arg length change %d->%d", in_size,
00494             out_size);
00495         MUNLOCK(ErrorMessageLock);
00496         Terminate(-1);
00497     }
00498
00499     AR.SortType = oldsorttype;
00500     AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
00501     terms[1] = 0; // set dirty flag to zero
00502 }
00503 /*
00504 #] flint::form_sort :
00505
00506 #[ flint::from_argument_mpoly :
00507 */
00508 // Convert a FORM argument (or 0-terminated list of terms: with_arghead == false) to a
00509 // (multi-variate) fmpz_mpoly_t poly. The "denominator" is return in denpoly, and contains the
00510 // overall negative-power numeric and symbolic factor.
00511 uint64_t flint::from_argument_mpoly(fmpz_mpoly_t poly, fmpz_mpoly_t denpoly, const WORD *args,
00512     const bool with_arghead, const var_map_t &var_map, const fmpz_mpoly_ctx_t ctx) {
00513
00514     // Some callers re-use their poly, denpoly to avoid calling init/clear unnecessarily.
00515     // Make sure they are 0 to begin.
00516     fmpz_mpoly_set_si(poly, 0, ctx);
00517     fmpz_mpoly_set_si(denpoly, 0, ctx);
00518
00519     // First check for "fast notation" arguments:
00520     if ( *args == -SNUMBER ) {
00521         fmpz_mpoly_set_si(poly, *(args+1), ctx);

```

```

00522     fmpz_mpoly_set_si(denpoly, 1, ctx);
00523     return 2;
00524 }
00525
00526 if ( *args == -SYMBOL ) {
00527     // A "fast notation" SYMBOL has a power and coefficient of 1:
00528     vector<uint64_t> exponents(var_map.size(), 0);
00529     exponents[var_map.at(*(args+1))] = 1;
00530
00531     fmpz_mpoly_set_coeff_ui_ui(poly, (ulong)1, (ulong*)exponents.data(), ctx);
00532     fmpz_mpoly_set_si(denpoly, (ulong)1, ctx);
00533     return 2;
00534 }
00535
00536
00537 // Now we can iterate through the terms of the argument. If we have
00538 // an ARGHEAD, we already know where to terminate. Otherwise we'll have
00539 // to loop until the terminating 0.
00540 const WORD* arg_stop = with_arghead ? args+args[0] : (WORD*)UINT64_MAX;
00541 uint64_t arg_size = 0;
00542 if ( with_arghead ) {
00543     arg_size = args[0];
00544     args += ARGHEAD;
00545 }
00546
00547
00548 // Search for numerical or symbol denominators to create "denpoly".
00549 flint::fmpz den_coeff, tmp;
00550 fmpz_set_si(den_coeff.d, 1);
00551 vector<uint64_t> neg_exponents(var_map.size(), 0);
00552
00553 for ( const WORD* term = args; term < arg_stop; term += term[0] ) {
00554     const WORD* term_stop = term+term[0];
00555     const WORD coeff_size = (term_stop)[-1];
00556     const WORD* symbol_stop = term_stop - ABS(coeff_size);
00557     const WORD* t = term;
00558
00559     t++;
00560     if ( t == symbol_stop ) {
00561         // Just a number, no symbols
00562     }
00563     else {
00564         t++; // this entry is SYMBOL
00565         t++; // this entry just has the size of the symbol array, but we can use symbol_stop
00566
00567         for ( const WORD* s = t; s < symbol_stop; s += 2 ) {
00568             if ( *(s+1) < 0 ) {
00569                 neg_exponents[var_map.at(*s)] =
00570                     MaX(neg_exponents[var_map.at(*s)], (uint64_t)(-*(s+1))) );
00571             }
00572         }
00573     }
00574
00575     // Now check for a denominator in the coefficient:
00576     if ( *(symbol_stop+ABS(coeff_size/2)) != 1 ) {
00577         flint::fmpz_set_form(tmp.d, (UWORD*)(symbol_stop+ABS(coeff_size/2)), ABS(coeff_size/2));
00578         // Record the LCM of the coefficient denominators:
00579         fmpz_lcm(den_coeff.d, den_coeff.d, tmp.d);
00580     }
00581
00582     if ( !with_arghead && *term_stop == 0 ) {
00583         // + 1 for the terminating 0
00584         arg_size = term_stop - args + 1;
00585         break;
00586     }
00587 }
00588
00589 // Assemble denpoly.
00590 fmpz_mpoly_set_coeff_fmpz_ui(denpoly, den_coeff.d, (ulong*)neg_exponents.data(), ctx);
00591
00592
00593 // For the term coefficients
00594 flint::fmpz coeff;
00595
00596 for ( const WORD* term = args; term < arg_stop; term += term[0] ) {
00597     const WORD* term_stop = term+term[0];
00598     const WORD coeff_size = (term_stop)[-1];
00599     const WORD* symbol_stop = term_stop - ABS(coeff_size);
00600     const WORD* t = term;
00601
00602     vector<uint64_t> exponents(var_map.size(), 0);
00603
00604     t++; // skip over the total size entry
00605     if ( t == symbol_stop ) {
00606         // Just a number, no symbols
00607     }
00608     else {

```

```

00609         t++; // this entry is SYMBOL
00610         t++; // this entry just has the size of the symbol array, but we can use symbol_stop
00611         for ( const WORD* s = t; s < symbol_stop; s += 2 ) {
00612             exponents[var_map.at(*s)] = *(s+1);
00613         }
00614     }
00615     // Now read the coefficient
00616     flint::fmpz_set_form(coeff.d, (UWORD*)symbol_stop, coeff_size/2);
00617
00618     // Multiply by denominator LCM
00619     fmpz_mul(coeff.d, coeff.d, den_coeff.d);
00620
00621     // Shift by neg_exponents
00622     for ( size_t i = 0; i < var_map.size(); i++ ) {
00623         exponents[i] += neg_exponents[i];
00624     }
00625
00626     // Read the denominator if there is one, and divide it out of the coefficient
00627     if ( *(symbol_stop+ABS(coeff_size/2)) != 1 ) {
00628         flint::fmpz_set_form(tmp.d, (UWORD*)(symbol_stop+ABS(coeff_size/2)), ABS(coeff_size/2));
00629         // By construction, this is an exact division
00630         fmpz_divexact(coeff.d, coeff.d, tmp.d);
00631     }
00632
00633     // Push the term to the mpoly, remember to sort when finished! This is much faster than using
00634     // fmpz_mpoly_set_coeff_fmpz_ui when the terms arrive in the "wrong order".
00635     fmpz_mpoly_push_term_fmpz_ui(poly, coeff.d, (ulong*)exponents.data(), ctx);
00636
00637     if ( !with_arghead && *term_stop == 0 ) {
00638         break;
00639     }
00640 }
00641
00642 // And now sort the mpoly
00643 fmpz_mpoly_sort_terms(poly, ctx);
00644
00645 return arg_size;
00646 }
00647 /*
00648 #] flint::from_argument_mpoly :
00649 #[ flint::from_argument_poly :
00650 */
00651 // Convert a FORM argument (or 0-terminated list of terms: with_arghead == false) to a
00652 // (uni-variate) fmpz_poly_t poly. The "denominator" is return in denpoly, and contains the
00653 // overall negative-power numeric and symbolic factor.
00654 uint64_t flint::from_argument_poly(fmpz_poly_t poly, fmpz_poly_t denpoly, const WORD *args,
00655     const bool with_arghead) {
00656
00657     // Some callers re-use their poly, denpoly to avoid calling init/clear unnecessarily.
00658     // Make sure they are 0 to begin.
00659     fmpz_poly_set_si(poly, 0);
00660     fmpz_poly_set_si(denpoly, 0);
00661
00662     // First check for "fast notation" arguments:
00663     if ( *args == -SNUMBER ) {
00664         fmpz_poly_set_si(poly, *(args+1));
00665         fmpz_poly_set_si(denpoly, 1);
00666         return 2;
00667     }
00668
00669     if ( *args == -SYMBOL ) {
00670         // A "fast notation" SYMBOL has a power and coefficient of 1:
00671         fmpz_poly_set_coeff_si(poly, 1, 1);
00672         fmpz_poly_set_si(denpoly, 1);
00673         return 2;
00674     }
00675
00676     // Now we can iterate through the terms of the argument. If we have
00677     // an ARGHEAD, we already know where to terminate. Otherwise we'll have
00678     // to loop until the terminating 0.
00679     const WORD* arg_stop = with_arghead ? args+args[0] : (WORD*)UINT64_MAX;
00680     uint64_t arg_size = 0;
00681     if ( with_arghead ) {
00682         arg_size = args[0];
00683         args += ARGHEAD;
00684     }
00685
00686     // Search for numerical or symbol denominators to create "denpoly".
00687     flint::fmpz den_coeff, tmp;
00688     fmpz_set_si(den_coeff.d, 1);
00689     uint64_t neg_exponent = 0;
00690
00691     for ( const WORD* term = args; term < arg_stop; term += term[0] ) {
00692
00693         const WORD* term_stop = term+term[0];
00694         const WORD coeff_size = (term_stop)[-1];
00695         const WORD* symbol_stop = term_stop - ABS(coeff_size);

```

```

00696     const WORD* t = term;
00697
00698     t++; // skip over the total size entry
00699     if ( t == symbol_stop ) {
00700         // Just a number, no symbols
00701     }
00702     else {
00703         t++; // this entry is SYMBOL
00704         t++; // this entry is the size of the symbol array
00705         t++; // this is the first (and only) symbol code
00706         if ( *t < 0 ) {
00707             neg_exponent = MaX(neg_exponent, (uint64_t)(-(*t)) );
00708         }
00709     }
00710
00711     // Now check for a denominator in the coefficient:
00712     if ( *(symbol_stop+ABS(coeff_size/2)) != 1 ) {
00713         flint::fmpz_set_form(tmp.d, (UWORD*)(symbol_stop+ABS(coeff_size/2)), ABS(coeff_size/2));
00714         // Record the LCM of the coefficient denominators:
00715         fmpz_lcm(den_coeff.d, den_coeff.d, tmp.d);
00716     }
00717
00718     if ( *term_stop == 0 ) {
00719         // + 1 for the terminating 0
00720         arg_size = term_stop - args + 1;
00721         break;
00722     }
00723 }
00724 // Assemble denpoly.
00725 fmpz_poly_set_coeff_fmpz(denpoly, neg_exponent, den_coeff.d);
00726
00727 // For the term coefficients
00728 flint::fmpz coeff;
00729
00730 for ( const WORD* term = args; term < arg_stop; term += term[0] ) {
00731
00732     const WORD* term_stop = term+term[0];
00733     const WORD coeff_size = (term_stop)[-1];
00734     const WORD* symbol_stop = term_stop - ABS(coeff_size);
00735     const WORD* t = term;
00736
00737     uint64_t exponent = 0;
00738
00739     t++; // skip over the total size entry
00740     if ( t == symbol_stop ) {
00741         // Just a number, no symbols
00742     }
00743     else {
00744         t++; // this entry is SYMBOL
00745         t++; // this entry is the size of the symbol array
00746         t++; // this is the first (and only) symbol code
00747         exponent = *t++;
00748     }
00749
00750     // Now read the coefficient
00751     flint::fmpz_set_form(coeff.d, (UWORD*)symbol_stop, coeff_size/2);
00752
00753     // Multiply by denominator LCM
00754     fmpz_mul(coeff.d, coeff.d, den_coeff.d);
00755
00756     // Shift by neg_exponent
00757     exponent += neg_exponent;
00758
00759     // Read the denominator if there is one, and divide it out of the coefficient
00760     if ( *(symbol_stop+ABS(coeff_size/2)) != 1 ) {
00761         flint::fmpz_set_form(tmp.d, (UWORD*)(symbol_stop+ABS(coeff_size/2)), ABS(coeff_size/2));
00762         // By construction, this is an exact division
00763         fmpz_divexact(coeff.d, coeff.d, tmp.d);
00764     }
00765
00766     // Add the term to the poly
00767     fmpz_poly_set_coeff_fmpz(poly, exponent, coeff.d);
00768
00769     if ( *term_stop == 0 ) {
00770         break;
00771     }
00772 }
00773 }
00774
00775 return arg_size;
00776 }
00777 /*
00778 #] flint::from_argument_poly :
00779
00780 #[ flint::fmpz_get_form :
00781 */
00782 // Write FORM's long integer representation of an fmpz at a, and put the number of WORDs at na.

```

```

00783 // na carries the sign of the integer.
00784 WORD flint::fmpz_get_form(fmpz_t z, WORD *a) {
00785
00786     WORD na = 0;
00787     const int32_t sgn = fmpz_sgn(z);
00788     if ( sgn == -1 ) {
00789         fmpz_neg(z, z);
00790     }
00791     const int64_t nlimbs = fmpz_size(z);
00792
00793     // This works but is UB?
00794     //fmpz_get_ui_array(reinterpret_cast<uint64_t*>(a), nlimbs, z);
00795
00796     // Use fixed-size functions to get limb data where possible. These probably cover most real
00797     // cases.
00798     if ( nlimbs == 1 ) {
00799         const uint64_t limb = fmpz_get_ui(z);
00800         a[0] = (WORD)(limb & 0xFFFFFFFF);
00801         na++;
00802         a[1] = (WORD)(limb >> BITSINWORD);
00803         if ( a[1] != 0 ) {
00804             na++;
00805         }
00806     }
00807     else if ( nlimbs == 2 ) {
00808         uint64_t limb_hi = 0, limb_lo = 0;
00809         fmpz_get_uiui((ulong*)&limb_hi, (ulong*)&limb_lo, z);
00810         a[0] = (WORD)(limb_lo & 0xFFFFFFFF);
00811         na++;
00812         a[1] = (WORD)(limb_lo >> BITSINWORD);
00813         na++;
00814         a[2] = (WORD)(limb_hi & 0xFFFFFFFF);
00815         na++;
00816         a[3] = (WORD)(limb_hi >> BITSINWORD);
00817         if ( a[3] != 0 ) {
00818             na++;
00819         }
00820     }
00821     else {
00822         vector<uint64_t> limb_data(nlimbs, 0);
00823         fmpz_get_ui_array((ulong*)limb_data.data(), (ulong)nlimbs, z);
00824         for ( long i = 0; i < nlimbs; i++ ) {
00825             a[2*i] = (WORD)(limb_data[i] & 0xFFFFFFFF);
00826             na++;
00827             a[2*i+1] = (WORD)(limb_data[i] >> BITSINWORD);
00828             if ( a[2*i+1] != 0 || i < (nlimbs-1) ) {
00829                 // The final limb might fit in a single 32bit WORD. Only
00830                 // increment na if the final WORD is non zero.
00831                 na++;
00832             }
00833         }
00834     }
00835
00836     // And now put the sign in the number of limbs
00837     if ( sgn == -1 ) {
00838         na = -na;
00839     }
00840
00841     return na;
00842 }
00843 /*
00844 #] flint::fmpz_get_form :
00845 #[ flint::fmpz_set_form :
00846 */
00847 // Create an fmpz directly from FORM's long integer representation. fmpz uses 64bit unsigned limbs,
00848 // but FORM uses 32bit UWORDS on 64bit architectures so we can't use fmpz_set_ui_array directly.
00849 void flint::fmpz_set_form(fmpz_t z, UWORD *a, WORD na) {
00850
00851     if ( na == 0 ) {
00852         fmpz_zero(z);
00853         return;
00854     }
00855
00856     // Negative na represents a negative number
00857     int32_t sgn = 1;
00858     if ( na < 0 ) {
00859         sgn = -1;
00860         na = -na;
00861     }
00862
00863     // Remove padding. FORM stores numerators and denominators with equal numbers of limbs but we
00864     // don't need to do this within the fmpz. It is not necessary to do this really, the fmpz
00865     // doesn't add zero limbs unnecessarily, but we might be able to avoid creating the limb_data
00866     // array below.
00867     while ( a[na-1] == 0 ) {
00868         na--;
00869     }

```

```

00870
00871 // If the number fits in fixed-size fmpz_set functions, we don't need to use additional memory
00872 // to convert to uint64_t. These probably cover most real cases.
00873 if ( na == 1 ) {
00874     fmpz_set_ui(z, (uint64_t)a[0]);
00875 }
00876 else if ( na == 2 ) {
00877     fmpz_set_ui(z, (((uint64_t)a[1])«BITSINWORD) + (uint64_t)a[0]);
00878 }
00879 else if ( na == 3 ) {
00880     fmpz_set_uiui(z, (uint64_t)a[2], (((uint64_t)a[1])«BITSINWORD) + (uint64_t)a[0]);
00881 }
00882 else if ( na == 4 ) {
00883     fmpz_set_uiui(z, (((uint64_t)a[3])«BITSINWORD) + (uint64_t)a[2],
00884         (((uint64_t)a[1])«BITSINWORD) + (uint64_t)a[0]);
00885 }
00886 else {
00887     const int32_t nlimbs = (na+1)/2;
00888     vector<uint64_t> limb_data(nlimbs, 0);
00889     for ( int32_t i = 0; i < nlimbs; i++ ) {
00890         if ( 2*i+1 <= na-1 ) {
00891             limb_data[i] = (uint64_t)a[2*i] + (((uint64_t)a[2*i+1])«BITSINWORD);
00892         }
00893         else {
00894             limb_data[i] = (uint64_t)a[2*i];
00895         }
00896     }
00897     fmpz_set_ui_array(z, (ulong*)limb_data.data(), nlimbs);
00898 }
00899
00900 // Finally set the sign.
00901 if ( sgn == -1 ) {
00902     fmpz_neg(z, z);
00903 }
00904
00905 return;
00906 }
00907 /*
00908 #] flint::fmpz_set_form :
00909
00910 #[ flint::gcd_mpoly :
00911 */
00912 // Return a pointer to a buffer containing the GCD of the 0-terminated term lists at a and b.
00913 // If must_fit_term, this should be a TermMalloc buffer. Otherwise Malloc1 the buffer.
00914 // For multi-variate cases.
00915 WORD* flint::gcd_mpoly(PHEAD const WORD *a, const WORD *b, const WORD must_fit_term,
00916     const var_map_t &var_map) {
00917
00918     flint::mpoly_ctx ctx(var_map.size());
00919     flint::mpoly pa(ctx.d), pb(ctx.d), denpa(ctx.d), denpb(ctx.d), gcd(ctx.d);
00920
00921     flint::from_argument_mpoly(pa.d, denpa.d, a, false, var_map, ctx.d);
00922     flint::from_argument_mpoly(pb.d, denpb.d, b, false, var_map, ctx.d);
00923
00924     // denpa, denpb must be 1:
00925     if ( fmpz_mpoly_is_one(denpa.d, ctx.d) != 1 ) {
00926         MLOCK(ErrorMessageLock);
00927         MesPrint("flint::gcd_mpoly: error: denpa != 1");
00928         MUNLOCK(ErrorMessageLock);
00929         Terminate(-1);
00930     }
00931     if ( fmpz_mpoly_is_one(denpb.d, ctx.d) != 1 ) {
00932         MLOCK(ErrorMessageLock);
00933         MesPrint("flint::gcd_mpoly: error: denpb != 1");
00934         MUNLOCK(ErrorMessageLock);
00935         Terminate(-1);
00936     }
00937
00938     // poly returns pa if pa == pb, regardless of the lcoeff sign
00939     if ( fmpz_mpoly_equal(pa.d, pb.d, ctx.d) ) {
00940         fmpz_mpoly_set(gcd.d, pa.d, ctx.d);
00941     }
00942     else {
00943         // We need some gymnastics to have the same sign conventions as the poly class. It takes the
00944         // integer or univar content out of a,b, with the convention that the content sign matches
00945         // the lcoeff sign. Since FORM has already taken out the content, we are left with +-1. In
00946         // Flint, the content always has a positive sign so here we should find +1. Check this:
00947         flint::mpoly tmp(ctx.d);
00948         fmpz_mpoly_term_content(tmp.d, pa.d, ctx.d);
00949         if ( fmpz_mpoly_is_one(tmp.d, ctx.d) != 1 ) {
00950             MLOCK(ErrorMessageLock);
00951             MesPrint("flint::gcd_mpoly: error: content of 1st arg != 1");
00952             MUNLOCK(ErrorMessageLock);
00953             Terminate(-1);
00954         }
00955         fmpz_mpoly_term_content(tmp.d, pb.d, ctx.d);
00956         if ( fmpz_mpoly_is_one(tmp.d, ctx.d) != 1 ) {

```



```

00957         MLOCK(ErrorMessageLock);
00958         MesPrint("flint::gcd_mpoly: error: content of 2nd arg != 1");
00959         MUNLOCK(ErrorMessageLock);
00960         Terminate(-1);
00961     }
00962
00963     // The poly class now divides the content out of a,b so that they have a positive lcoeff.
00964     // Then it multiplies the final gcd (which is given a positive lcoeff also) by
00965     // gcd(cont a, cont b). There it has gcd(1,1) = gcd(-1,1) = gcd(1,-1) = 1, and
00966     // gcd(-1,-1) = -1 (because of the pa==pb early return). So: if both input polys have a
00967     // negative lcoeff, we will flip the sign in the final result.
00968     bool flip_sign = 0;
00969     if ( ( fmpz_sgn(fmpz_mpoly_term_coeff_ref(pa.d, 0, ctx.d)) == -1 ) &&
00970         ( fmpz_sgn(fmpz_mpoly_term_coeff_ref(pb.d, 0, ctx.d)) == -1 ) ) {
00971         flip_sign = 1;
00972     }
00973
00974     fmpz_mpoly_gcd(gcd.d, pa.d, pb.d, ctx.d);
00975     if ( flip_sign ) {
00976         fmpz_mpoly_neg(gcd.d, gcd.d, ctx.d);
00977     }
00978 }
00979
00980 // This is freed by the caller
00981 WORD *res;
00982 if ( must_fit_term ) {
00983     res = TermMalloc("flint::gcd_mpoly");
00984 }
00985 else {
00986     // Determine the size of the GCD by passing write = false.
00987     const bool with_arghead = false;
00988     const bool write = false;
00989     const uint64_t prev_size = 0;
00990     const uint64_t gcd_size = (uint64_t)flint::to_argument_mpoly(BHEAD NULL,
00991         with_arghead, must_fit_term, write, prev_size, gcd.d, var_map, ctx.d);
00992
00993     res = (WORD *)Malloca(sizeof(WORD)*gcd_size, "flint::gcd_mpoly");
00994 }
00995
00996 const bool with_arghead = false;
00997 const bool write = true;
00998 const uint64_t prev_size = 0;
00999 flint::to_argument_mpoly(BHEAD res, with_arghead, must_fit_term, write, prev_size, gcd.d,
01000     var_map, ctx.d);
01001
01002 return res;
01003 }
01004 /*
01005 #] flint::gcd_mpoly :
01006 #[ flint::gcd_poly :
01007 */
01008 // Return a pointer to a buffer containing the GCD of the 0-terminated term lists at a and b.
01009 // If must_fit_term, this should be a TermMalloc buffer. Otherwise Malloca the buffer.
01010 // For uni-variate cases.
01011 WORD* flint::gcd_poly(PHEAD const WORD *a, const WORD *b, const WORD must_fit_term,
01012     const var_map_t &var_map) {
01013
01014     flint::poly pa, pb, denpa, denpb, gcd;
01015
01016     flint::from_argument_poly(pa.d, denpa.d, a, false);
01017     flint::from_argument_poly(pb.d, denpb.d, b, false);
01018
01019     // denpa, denpb must be 1:
01020     if ( fmpz_poly_is_one(denpa.d) != 1 ) {
01021         MLOCK(ErrorMessageLock);
01022         MesPrint("flint::gcd_poly: error: denpa != 1");
01023         MUNLOCK(ErrorMessageLock);
01024         Terminate(-1);
01025     }
01026     if ( fmpz_poly_is_one(denpb.d) != 1 ) {
01027         MLOCK(ErrorMessageLock);
01028         MesPrint("flint::gcd_poly: error: denpb != 1");
01029         MUNLOCK(ErrorMessageLock);
01030         Terminate(-1);
01031     }
01032
01033     // poly returns pa if pa == pb, regardless of the lcoeff sign
01034     if ( fmpz_poly_equal(pa.d, pb.d) ) {
01035         fmpz_poly_set(gcd.d, pa.d);
01036     }
01037     else {
01038         // Here, we don't have to make any sign flips like the mpoly case, because poly's
01039         // integer_gcd(1,1) = integer_gcd(-1,1) = integer_gcd(1,-1) = integer_gcd(-1,-1) = +1.
01040         // Still, verify that the content is 1:
01041         flint::fmpz tmp;
01042         fmpz_poly_content(tmp.d, pa.d);
01043         if ( fmpz_is_one(tmp.d) != 1 ) {

```

```

01044         MLOCK(ErrorMessageLock);
01045         MesPrint("flint::gcd_poly: error: content of 1st arg != 1");
01046         MUNLOCK(ErrorMessageLock);
01047         Terminate(-1);
01048     }
01049     fmpz_poly_content(tmp.d, pb.d);
01050     if ( fmpz_is_one(tmp.d) != 1 ) {
01051         MLOCK(ErrorMessageLock);
01052         MesPrint("flint::gcd_poly: error: content of 2nd arg != 1");
01053         MUNLOCK(ErrorMessageLock);
01054         Terminate(-1);
01055     }
01056     fmpz_poly_gcd(gcd.d, pa.d, pb.d);
01057 }
01058
01059 // This is freed by the caller
01060 WORD *res;
01061 if ( must_fit_term ) {
01062     res = TermMalloc("flint::gcd_poly");
01063 }
01064 else {
01065     // Determine the size of the GCD by passing write = false.
01066     const bool with_arghead = false;
01067     const bool write = false;
01068     const uint64_t prev_size = 0;
01069     const uint64_t gcd_size = (uint64_t)flint::to_argument_poly(BHEAD NULL,
01070         with_arghead, must_fit_term, write, prev_size, gcd.d, var_map);
01071     res = (WORD *)Malloca(sizeof(WORD)*gcd_size, "flint::gcd_poly");
01072 }
01073
01074 const bool with_arghead = false;
01075 const uint64_t prev_size = 0;
01076 const bool write = true;
01077 flint::to_argument_poly(BHEAD res, with_arghead, must_fit_term, write, prev_size, gcd.d,
01078     var_map);
01079
01080 return res;
01081 }
01082 /*
01083 #] flint::gcd_poly :
01084
01085 #[ flint::get_variables :
01086
01087 */
01088 // Get the list of symbols which appear in the vector of expressions. These are polyratfun
01089 // numerators, denominators or expressions from calls to gcd_ etc. Return this list as a map
01090 // between indices and symbol codes.
01091 // TODO FACTORSYMBOL last?
01092 flint::var_map_t flint::get_variables(const vector<WORD*> &es, const bool with_arghead,
01093     const bool sort_vars) {
01094     int32_t num_vars = 0;
01095     // We count the total number of terms to determine "density".
01096     uint32_t num_terms = 0;
01097     // To be used if we sort by highest degree, as the poly code does.
01098     vector<int32_t> degrees;
01099     var_map_t var_map;
01100
01101     // extract all variables
01102     for ( size_t ei = 0; ei < es.size(); ei++ ) {
01103         WORD *e = es[ei];
01104
01105         // fast notation
01106         if ( *e == -SNUMBER ) {
01107             num_terms++;
01108         }
01109         else if ( *e == -SYMBOL ) {
01110             num_terms++;
01111             if ( !var_map.count(e[1]) ) {
01112                 var_map[e[1]] = num_vars++;
01113                 degrees.push_back(1);
01114             }
01115         }
01116     }
01117
01118     // JD: Here we need to check for non-symbol/number terms in fast notation.
01119     else if ( *e < 0 ) {
01120         MLOCK(ErrorMessageLock);
01121         MesPrint("ERROR: polynomials and polyratfuns must contain symbols only");
01122         MUNLOCK(ErrorMessageLock);
01123         Terminate(1);
01124     }
01125     else {
01126         for ( WORD i = with_arghead ? ARGHEAD:0; with_arghead ? i < e[0]:e[i] != 0; i += e[i] ) {
01127             num_terms++;
01128             if ( i+1 < i+e[i]-ABS(e[i+e[i]-1]) && e[i+1] != SYMBOL ) {
01129                 MLOCK(ErrorMessageLock);
01130                 MesPrint("ERROR: polynomials and polyratfuns must contain symbols only");
01131             }
01132         }
01133     }
01134 }

```

```

01131             MUNLOCK(ErrorMessageLock);
01132             Terminate(1);
01133         }
01134
01135         for ( WORD j = i+3; j<i+e[i]-ABS(e[i+e[i]-1]); j += 2 ) {
01136             if ( !var_map.count(e[j]) ) {
01137                 var_map[e[j]] = num_vars++;
01138                 degrees.push_back(e[j+1]);
01139             }
01140             else {
01141                 degrees[var_map[e[j]]] = MaX(degrees[var_map[e[j]]], e[j+1]);
01142             }
01143         }
01144     }
01145 }
01146
01147 if ( sort_vars ) {
01148     // bubble sort variables in decreasing order of degree
01149     // (this seems better for factorization)
01150     for ( size_t i = 0; i < var_map.size(); i++ ) {
01151         for ( size_t j = 0; j+1 < var_map.size(); j++ ) {
01152             if ( degrees[j] < degrees[j+1] ) {
01153                 swap(degrees[j], degrees[j+1]);
01154
01155                 // Find the map keys associated with the values we want to swap
01156                 uint32_t j0 = 0;
01157                 uint32_t j1 = 0;
01158                 for ( auto x: var_map ) {
01159                     if ( x.second == j ) {
01160                         j0 = x.first;
01161                     }
01162                     else if ( x.second == j+1 ) {
01163                         j1 = x.first;
01164                     }
01165                 }
01166                 swap(var_map.at(j0), var_map.at(j1));
01167             }
01168         }
01169     }
01170 }
01171
01172 // Otherwise, sort lexicographically in FORM's ordering
01173 else {
01174     for ( size_t i = 0; i < var_map.size(); i++ ) {
01175         for ( size_t j = 0; j+1 < var_map.size(); j++ ) {
01176             uint32_t j0 = 0;
01177             uint32_t j1 = 0;
01178             for ( auto x: var_map ) {
01179                 if ( x.second == j ) {
01180                     j0 = x.first;
01181                 }
01182                 else if ( x.second == j+1 ) {
01183                     j1 = x.first;
01184                 }
01185             }
01186             if ( j0 > j1 ) {
01187                 swap(var_map.at(j0), var_map.at(j1));
01188             }
01189         }
01190     }
01191 }
01192
01193 if ( var_map.size() == 1 ) {
01194     // In the univariate case, if the polynomials are sufficiently sparse force the use of the
01195     // multivariate routines, which use a sparse representation, by adding a dummy map entry.
01196     if ( (float)num_terms <= UNIVARIATE_DENSITY_THR * (float)degrees[0] ) {
01197         // -1 will never be a symbol code. Built-in symbols from 0 to 19, and 20 is the first
01198         // user symbol.
01199         var_map[-1] = num_vars;
01200     }
01201 }
01202
01203 return var_map;
01204 }
01205 /*
01206 #] flint::get_variables :
01207
01208 #[ flint::inverse_poly :
01209 */
01210 WORD* flint::inverse_poly(PHEAD const WORD *a, const WORD *b, const var_map_t &var_map) {
01211
01212     flint::poly pa, pb, denpa, denpb;
01213
01214     flint::from_argument_poly(pa.d, denpa.d, a, false);
01215     flint::from_argument_poly(pb.d, denpb.d, b, false);
01216
01217     // fmpz_poly_xgcd is undefined if the content of pa and pb are not 1. Take the content out.

```

```

01218 // fmpz_poly_content gives the non-negative content and fmpz_poly_primitive normalizes to a
01219 // non-negative lcoeff, so we need to add the sign to the content if the polys have a negative
01220 // lcoeff. We don't need to keep the content of pb, it is a numerical multiple of the modulus.
01221 flint::fmpz content_a, resultant;
01222 flint::poly inverse, tmp;
01223 fmpz_poly_content(content_a.d, pa.d);
01224 if ( fmpz_sgn(fmpz_poly_lead(pa.d)) == -1 ) {
01225     fmpz_neg(content_a.d, content_a.d);
01226 }
01227 fmpz_poly_primitive_part(pa.d, pa.d);
01228 fmpz_poly_primitive_part(pb.d, pb.d);
01229
01230 // Special cases:
01231 // Possibly strange that we give 1 for inverse_(x1,1) but here we take MMA's convention.
01232 if ( fmpz_poly_is_one(pa.d) && fmpz_poly_is_one(pb.d) ) {
01233     fmpz_poly_one(inverse.d);
01234     fmpz_one(resultant.d);
01235 }
01236 else if ( fmpz_poly_is_one(pb.d) ) {
01237     fmpz_poly_zero(inverse.d);
01238     fmpz_one(resultant.d);
01239 }
01240 else {
01241     // Now use the extended Euclidean algorithm to find inverse, resultant, tmp of the Bezout
01242     // identity: inverse*pa + tmp*pb = resultant. Then inverse/resultant is the multiplicative
01243     // inverse of pa mod pb. We'll divide by resultant in the denscale argument of
    to_argument_poly.
01244     fmpz_poly_xgcd(resultant.d, inverse.d, tmp.d, pa.d, pb.d);
01245
01246     // If the resultant is zero, the inverse does not exist:
01247     if ( fmpz_is_zero(resultant.d) ) {
01248         MLOCK(ErrorMessageLock);
01249         MesPrint("flint::inverse_poly error: inverse does not exist");
01250         MUNLOCK(ErrorMessageLock);
01251         Terminate(-1);
01252     }
01253 }
01254
01255 // Multiply inverse by denpa. denpb is a numerical multiple of the modulus, so doesn't matter.
01256 // We also need to divide by content_a, which we do in the denscale argument of to_argument_poly
01257 // by multiplying resultant by content_a here.
01258 fmpz_poly_mul(inverse.d, inverse.d, denpa.d);
01259 fmpz_mul(resultant.d, resultant.d, content_a.d);
01260
01261 WORD* res;
01262 // First determine the result size, and malloc. The result should have no arghead. Here we use
01263 // the "scale" argument of to_argument_poly since resultant might not be 1.
01264 const bool with_arghead = false;
01265 bool write = false;
01266 const bool must_fit_term = false;
01267 const uint64_t prev_size = 0;
01268 const uint64_t res_size = (uint64_t)flint::to_argument_poly(BHEAD NULL,
01269     with_arghead, must_fit_term, write, prev_size, inverse.d, var_map, resultant.d);
01270 res = (WORD*)Malloc1(sizeof(WORD)*res_size, "flint::inverse_poly");
01271
01272 write = true;
01273 flint::to_argument_poly(BHEAD res, with_arghead, must_fit_term, write, prev_size, inverse.d,
01274     var_map, resultant.d);
01275
01276 return res;
01277 }
01278 /*
01279 #] flint::inverse_poly :
01280
01281 #[ flint::mul_mpoly :
01282 */
01283 // Return a pointer to a buffer containing the product of the 0-terminated term lists at a and b.
01284 // For multi-variate cases.
01285 WORD* flint::mul_mpoly(PHEAD const WORD *a, const WORD *b, const var_map_t &var_map) {
01286
01287     flint::mpoly_ctx ctx(var_map.size());
01288     flint::mpoly pa(ctx.d), pb(ctx.d), denpa(ctx.d), denpb(ctx.d);
01289
01290     flint::from_argument_mpoly(pa.d, denpa.d, a, false, var_map, ctx.d);
01291     flint::from_argument_mpoly(pb.d, denpb.d, b, false, var_map, ctx.d);
01292
01293     // denpa, denpb must be integers. Negative symbol powers have been converted to extra symbols.
01294     if ( fmpz_mpoly_is_fmpz(denpa.d, ctx.d) != 1 ) {
01295         MLOCK(ErrorMessageLock);
01296         MesPrint("flint::mul_mpoly: error: denpa is non-constant");
01297         MUNLOCK(ErrorMessageLock);
01298         Terminate(-1);
01299     }
01300
01301     if ( fmpz_mpoly_is_fmpz(denpb.d, ctx.d) != 1 ) {
01302         MLOCK(ErrorMessageLock);
01303         MesPrint("flint::mul_mpoly: error: denpb is non-constant");
01304     }

```

```

01304         MUNLOCK(ErrorMessageLock);
01305         Terminate(-1);
01306     }
01307
01308     // Multiply numerators, store result in pa
01309     fmpz_mpoly_mul(pa.d, pa.d, pb.d, ctx.d);
01310     // Multiply denominators, store result in denpa, and convert to an fmpz:
01311     fmpz_mpoly_mul(denpa.d, denpa.d, denpb.d, ctx.d);
01312     flint::fmpz den;
01313     fmpz_mpoly_get_fmpz(den.d, denpa.d, ctx.d);
01314
01315     WORD* res;
01316     // First determine the result size, and malloc. The result should have no arghead. Here we use
01317     // the "scale" argument of to_argument_mpoly since den might not be 1.
01318     const bool with_arghead = false;
01319     bool write = false;
01320     const bool must_fit_term = false;
01321     const uint64_t prev_size = 0;
01322     const uint64_t mul_size = (uint64_t)flint::to_argument_mpoly(BHEAD NULL,
01323         with_arghead, must_fit_term, write, prev_size, pa.d, var_map, ctx.d, den.d);
01324     res = (WORD*)Malloca(sizeof(WORD)*mul_size, "flint::mul_mpoly");
01325
01326     write = true;
01327     flint::to_argument_mpoly(BHEAD res, with_arghead, must_fit_term, write, prev_size, pa.d,
01328         var_map, ctx.d, den.d);
01329
01330     return res;
01331 }
01332 /*
01333 #] flint::mul_mpoly :
01334 #[ flint::mul_poly :
01335 */
01336 // Return a pointer to a buffer containing the product of the 0-terminated term lists at a and b.
01337 // For uni-variate cases.
01338 WORD* flint::mul_poly(PHEAD const WORD *a, const WORD *b, const var_map_t &var_map) {
01339
01340     flint::poly pa, pb, denpa, denpb;
01341
01342     flint::from_argument_poly(pa.d, denpa.d, a, false);
01343     flint::from_argument_poly(pb.d, denpb.d, b, false);
01344
01345     // denpa, denpb must be integers. Negative symbol powers have been converted to extra symbols.
01346     if ( fmpz_poly_degree(denpa.d) != 0 ) {
01347         MLOCK(ErrorMessageLock);
01348         MesPrint("flint::mul_poly: error: denpa is non-constant");
01349         MUNLOCK(ErrorMessageLock);
01350         Terminate(-1);
01351     }
01352     if ( fmpz_poly_degree(denpb.d) != 0 ) {
01353         MLOCK(ErrorMessageLock);
01354         MesPrint("flint::mul_poly: error: denpb is non-constant");
01355         MUNLOCK(ErrorMessageLock);
01356         Terminate(-1);
01357     }
01358
01359     // Multiply numerators, store result in pa
01360     fmpz_poly_mul(pa.d, pa.d, pb.d);
01361     // Multiply denominators, store result in denpa, and convert to an fmpz:
01362     fmpz_poly_mul(denpa.d, denpa.d, denpb.d);
01363     flint::fmpz den;
01364     fmpz_poly_get_coeff_fmpz(den.d, denpa.d, 0);
01365
01366     WORD* res;
01367     // First determine the result size, and malloc. The result should have no arghead. Here we use
01368     // the "scale" argument of to_argument_poly since den might not be 1.
01369     const bool with_arghead = false;
01370     bool write = false;
01371     const bool must_fit_term = false;
01372     const uint64_t prev_size = 0;
01373     const uint64_t mul_size = (uint64_t)flint::to_argument_poly(BHEAD NULL,
01374         with_arghead, must_fit_term, write, prev_size, pa.d, var_map, den.d);
01375     res = (WORD*)Malloca(sizeof(WORD)*mul_size, "flint::mul_poly");
01376
01377     write = true;
01378     flint::to_argument_poly(BHEAD res, with_arghead, must_fit_term, write, prev_size, pa.d,
01379         var_map, den.d);
01380
01381     return res;
01382 }
01383 /*
01384 #] flint::mul_poly :
01385 #[ flint::ratfun_add_mpoly :
01386 */
01387 // Add the multi-variate FORM rational polynomials at t1 and t2. The result is written at out.
01388 void flint::ratfun_add_mpoly(PHEAD const WORD *t1, const WORD *t2, WORD *out,
01389     const var_map_t &var_map) {

```

```

01391
01392     flint::mpoly_ctx ctx(var_map.size());
01393     flint::mpoly gcd(ctx.d), num1(ctx.d), den1(ctx.d), num2(ctx.d), den2(ctx.d);
01394
01395     flint::ratfun_read_mpoly(t1, num1.d, den1.d, var_map, ctx.d);
01396     flint::ratfun_read_mpoly(t2, num2.d, den2.d, var_map, ctx.d);
01397
01398     if ( fmpz_mpoly_cmp(den1.d, den2.d, ctx.d) != 0 ) {
01399         fmpz_mpoly_gcd_cofactors(gcd.d, den1.d, den2.d, den1.d, den2.d, ctx.d);
01400
01401         fmpz_mpoly_mul(num1.d, num1.d, den2.d, ctx.d);
01402         fmpz_mpoly_mul(num2.d, num2.d, den1.d, ctx.d);
01403
01404         fmpz_mpoly_add(num1.d, num1.d, num2.d, ctx.d);
01405         fmpz_mpoly_mul(den1.d, den1.d, den2.d, ctx.d);
01406         fmpz_mpoly_mul(den1.d, den1.d, gcd.d, ctx.d);
01407     }
01408     else {
01409         fmpz_mpoly_add(num1.d, num1.d, num2.d, ctx.d);
01410     }
01411
01412     // Finally divide out any common factors between the resulting num1, den1:
01413     fmpz_mpoly_gcd_cofactors(gcd.d, num1.d, den1.d, num1.d, den1.d, ctx.d);
01414
01415     flint::util::fix_sign_fmpz_mpoly_ratfun(num1.d, den1.d, ctx.d);
01416
01417     // Result in FORM notation:
01418     *out++ = AR.PolyFun;
01419     WORD* args_size = out++;
01420     WORD* args_flag = out++;
01421     *args_flag = 0; // clean prf
01422     FILLFUN3(out); // Remainder of funhead, if it is larger than 3
01423
01424     const bool with_arghead = true;
01425     const bool must_fit_term = true;
01426     const bool write = true;
01427     // prev_size + 4, to account for final term size and coeff of "1/1"
01428     out += flint::to_argument_mpoly(BHEAD out, with_arghead, must_fit_term, write, out-args_size+4,
01429         num1.d, var_map, ctx.d);
01430     out += flint::to_argument_mpoly(BHEAD out, with_arghead, must_fit_term, write, out-args_size+4,
01431         den1.d, var_map, ctx.d);
01432
01433     *args_size = out - args_size + 1; // The +1 is to include the function ID
01434     AT.WorkPointer = out;
01435 }
01436 /*
01437 #] flint::ratfun_add_mpoly :
01438 #[ flint::ratfun_add_poly :
01439 */
01440 // Add the uni-variate FORM rational polynomials at t1 and t2. The result is written at out.
01441 void flint::ratfun_add_poly(PHEAD const WORD *t1, const WORD *t2, WORD *out,
01442     const var_map_t &var_map) {
01443
01444     flint::poly gcd, num1, den1, num2, den2;
01445
01446     flint::ratfun_read_poly(t1, num1.d, den1.d);
01447     flint::ratfun_read_poly(t2, num2.d, den2.d);
01448
01449     if ( fmpz_poly_equal(den1.d, den2.d) == 0 ) {
01450         flint::util::simplify_fmpz_poly(den1.d, den2.d, gcd.d);
01451
01452         fmpz_poly_mul(num1.d, num1.d, den2.d);
01453         fmpz_poly_mul(num2.d, num2.d, den1.d);
01454
01455         fmpz_poly_add(num1.d, num1.d, num2.d);
01456         fmpz_poly_mul(den1.d, den1.d, den2.d);
01457         fmpz_poly_mul(den1.d, den1.d, gcd.d);
01458     }
01459     else {
01460         fmpz_poly_add(num1.d, num1.d, num2.d);
01461     }
01462
01463     // Finally divide out any common factors between the resulting num1, den1:
01464     flint::util::simplify_fmpz_poly(num1.d, den1.d, gcd.d);
01465
01466     flint::util::fix_sign_fmpz_poly_ratfun(num1.d, den1.d);
01467
01468     // Result in FORM notation:
01469     *out++ = AR.PolyFun;
01470     WORD* args_size = out++;
01471     WORD* args_flag = out++;
01472     *args_flag = 0; // clean prf
01473     FILLFUN3(out); // Remainder of funhead, if it is larger than 3
01474
01475     const bool with_arghead = true;
01476     const bool must_fit_term = true;
01477     const bool write = true;

```

```

01478 // prev_size + 4, to account for final term size and coeff of "1/1"
01479 out += flint::to_argument_poly(BHEAD out, with_arghead, must_fit_term, write, out-args_size+4,
01480 num1.d, var_map);
01481 out += flint::to_argument_poly(BHEAD out, with_arghead, must_fit_term, write, out-args_size+4,
01482 den1.d, var_map);
01483
01484 *args_size = out - args_size + 1; // The +1 is to include the function ID
01485 AT.WorkPointer = out;
01486 }
01487 /*
01488 #] flint::ratfun_add_poly :
01489
01490 #[ flint::ratfun_normalize_mpoly :
01491 */
01492 // Multiply and simplify occurrences of the multi-variate FORM rational polynomials found in term.
01493 // The final term is written in place, with the rational polynomial at the end.
01494 void flint::ratfun_normalize_mpoly(PHEAD WORD *term, const var_map_t &var_map) {
01495
01496 // The length of the coefficient
01497 const WORD ncoeff = (term + *term)[-1];
01498 // The end of the term data, before the coefficient:
01499 const WORD *tstop = term + *term - ABS(ncoeff);
01500
01501 flint::mpoly_ctx ctx(var_map.size());
01502 flint::mpoly num1(ctx.d), den1(ctx.d), num2(ctx.d), den2(ctx.d), gcd(ctx.d);
01503
01504 // Start with "trivial" polynomials, and multiply in the num and den of the prf which appear.
01505 flint::fmpz tmpNum, tmpDen;
01506 flint::fmpz_set_form(tmpNum.d, (UWORD*)tstop, ncoeff/2);
01507 flint::fmpz_set_form(tmpDen.d, (UWORD*)tstop+ABS(ncoeff/2), ABS(ncoeff/2));
01508 fmpz_mpoly_set_fmpz(num1.d, tmpNum.d, ctx.d);
01509 fmpz_mpoly_set_fmpz(den1.d, tmpDen.d, ctx.d);
01510
01511 // Loop over the occurrences of PolyFun in the term, and multiply in to num1, den1.
01512 // s tracks where we are writing the non-PolyFun term data. The final PolyFun will
01513 // go at the end.
01514 WORD* term_size = term;
01515 WORD* s = term + 1;
01516 for ( WORD *t = term + 1; t < tstop; ) {
01517     if ( *t == AR.PolyFun ) {
01518         flint::ratfun_read_mpoly(t, num2.d, den2.d, var_map, ctx.d);
01519
01520         // get gcd of num1,den2 and num2,den1 and then assemble
01521         fmpz_mpoly_gcd_cofactors(gcd.d, num1.d, den2.d, num1.d, den2.d, ctx.d);
01522         fmpz_mpoly_gcd_cofactors(gcd.d, num2.d, den1.d, num2.d, den1.d, ctx.d);
01523
01524         fmpz_mpoly_mul(num1.d, num1.d, num2.d, ctx.d);
01525         fmpz_mpoly_mul(den1.d, den1.d, den2.d, ctx.d);
01526
01527         t += t[1];
01528     }
01529
01530     else {
01531         // Not a PolyFun, just copy or skip over
01532         WORD i = t[1];
01533         if ( s != t ) { NCOPY(s,t,i); }
01534         else { t += i; s += i; }
01535     }
01536 }
01537
01538 flint::util::fix_sign_fmpz_mpoly_ratfun(num1.d, den1.d, ctx.d);
01539
01540 // Result in FORM notation:
01541 WORD* out = s;
01542 *out++ = AR.PolyFun;
01543 WORD* args_size = out++;
01544 WORD* args_flag = out++;
01545 *args_flag &= ~MUSTCLEANPRF;
01546
01547 const bool with_arghead = true;
01548 const bool must_fit_term = true;
01549 const bool write = true;
01550 out += flint::to_argument_mpoly(BHEAD out, with_arghead, must_fit_term, write, out-term_size,
01551 num1.d, var_map, ctx.d);
01552 out += flint::to_argument_mpoly(BHEAD out, with_arghead, must_fit_term, write, out-term_size,
01553 den1.d, var_map, ctx.d);
01554
01555 *args_size = out - args_size + 1; // The +1 is to include the function ID
01556
01557 // +3 for the coefficient of 1/1, which is added after the check
01558 if ( sizeof(WORD)*(out-term_size+3) > (size_t)AM.MaxTer ) {
01559     MLOCK(ErrorMessageLock);
01560     MesPrint("flint::ratfun_normalize: output exceeds MaxTermSize");
01561     MUNLOCK(ErrorMessageLock);
01562     Terminate(-1);
01563 }
01564

```

```

01565     *out++ = 1;
01566     *out++ = 1;
01567     *out++ = 3; // the term's coefficient is now 1/1
01568
01569     *term_size = out - term_size;
01570 }
01571 /*
01572 #] flint::ratfun_normalize_mpoly :
01573 #[ flint::ratfun_normalize_poly :
01574 */
01575 // Multiply and simplify occurrences of the uni-variate FORM rational polynomials found in term.
01576 // The final term is written in place, with the rational polynomial at the end.
01577 void flint::ratfun_normalize_poly(PHEAD WORD *term, const var_map_t &var_map) {
01578
01579     // The length of the coefficient
01580     const WORD ncoeff = (term + *term)[-1];
01581     // The end of the term data, before the coefficient:
01582     const WORD *tstop = term + *term - ABS(ncoeff);
01583
01584     flint::poly num1, den1, num2, den2, gcd;
01585
01586     // Start with "trivial" polynomials, and multiply in the num and den of the prf which appear.
01587     flint::fmpz tmpNum, tmpDen;
01588     flint::fmpz_set_form(tmpNum.d, (UWORD*)tstop, ncoeff/2);
01589     flint::fmpz_set_form(tmpDen.d, (UWORD*)tstop+ABS(ncoeff/2), ABS(ncoeff/2));
01590     fmpz_poly_set_fmpz(num1.d, tmpNum.d);
01591     fmpz_poly_set_fmpz(den1.d, tmpDen.d);
01592
01593     // Loop over the occurrences of PolyFun in the term, and multiply in to num1, den1.
01594     // s tracks where we are writing the non-PolyFun term data. The final PolyFun will
01595     // go at the end.
01596     WORD* term_size = term;
01597     WORD* s = term + 1;
01598     for ( WORD *t = term + 1; t < tstop; ) {
01599         if ( *t == AR.PolyFun ) {
01600             flint::ratfun_read_poly(t, num2.d, den2.d);
01601
01602             // get gcd of num1,den2 and num2,den1 and then assemble
01603             flint::util::simplify_fmpz_poly(num1.d, den2.d, gcd.d);
01604             flint::util::simplify_fmpz_poly(num2.d, den1.d, gcd.d);
01605
01606             fmpz_poly_mul(num1.d, num1.d, num2.d);
01607             fmpz_poly_mul(den1.d, den1.d, den2.d);
01608
01609             t += t[1];
01610         }
01611
01612         else {
01613             // Not a PolyFun, just copy or skip over
01614             WORD i = t[1];
01615             if ( s != t ) { NCOPY(s,t,i); }
01616             else { t += i; s += i; }
01617         }
01618     }
01619
01620     flint::util::fix_sign_fmpz_poly_ratfun(num1.d, den1.d);
01621
01622     // Result in FORM notation:
01623     WORD* out = s;
01624     *out++ = AR.PolyFun;
01625     WORD* args_size = out++;
01626     WORD* args_flag = out++;
01627     *args_flag &= ~MUSTCLEANPRF;
01628
01629     const bool with_arghead = true;
01630     const bool must_fit_term = true;
01631     const bool write = true;
01632     out += flint::to_argument_poly(BHEAD out, with_arghead, must_fit_term, write, out-term_size,
01633     num1.d, var_map);
01634     out += flint::to_argument_poly(BHEAD out, with_arghead, must_fit_term, write, out-term_size,
01635     den1.d, var_map);
01636
01637     *args_size = out - args_size + 1; // The +1 is to include the function ID
01638
01639     // +3 for the coefficient of 1/1, which is added after the check
01640     if ( sizeof(WORD)*(out-term_size+3) > (size_t)AM.MaxTer ) {
01641         MLOCK(ErrorMessageLock);
01642         MesPrint("flint::ratfun_normalize: output exceeds MaxTermSize");
01643         MUNLOCK(ErrorMessageLock);
01644         Terminate(-1);
01645     }
01646
01647     *out++ = 1;
01648     *out++ = 1;
01649     *out++ = 3; // the term's coefficient is now 1/1
01650
01651     *term_size = out - term_size;

```



```

01652 }
01653 /*
01654 #] flint::ratfun_normalize_poly :
01655
01656 #[ flint::ratfun_read_mpoly :
01657 */
01658 // Read the multi-variate FORM rational polynomial at a and create fmpz_mpoly_t numerator and
01659 // denominator.
01660 void flint::ratfun_read_mpoly(const WORD *a, fmpz_mpoly_t num, fmpz_mpoly_t den,
01661     const var_map_t &var_map, fmpz_mpoly_ctx_t ctx) {
01662
01663     // The end of the arguments:
01664     const WORD* arg_stop = a+a[1];
01665
01666     const bool must_normalize = (a[2] & MUSTCLEANPRF) != 0;
01667
01668     a += FUNHEAD;
01669     if ( a >= arg_stop ) {
01670         MLOCK(ErrorMessageLock);
01671         MesPrint("ERROR: PolyRatFun cannot have zero arguments");
01672         MUNLOCK(ErrorMessageLock);
01673         Terminate(-1);
01674     }
01675
01676     // Polys to collect the "den of the num" and "den of the den".
01677     // Input can arrive like this when enabling the PolyRatFun or moving things into it.
01678     flint::mpoly den_num(ctx), den_den(ctx);
01679
01680     // Read the numerator
01681     flint::from_argument_mpoly(num, den_num.d, a, true, var_map, ctx);
01682     NEXTARG(a);
01683
01684     if ( a < arg_stop ) {
01685         // Read the denominator
01686         flint::from_argument_mpoly(den, den_den.d, a, true, var_map, ctx);
01687         NEXTARG(a);
01688     }
01689     else {
01690         // The denominator is 1
01691         MLOCK(ErrorMessageLock);
01692         MesPrint("implement this");
01693         MUNLOCK(ErrorMessageLock);
01694         Terminate(-1);
01695     }
01696     if ( a < arg_stop ) {
01697         MLOCK(ErrorMessageLock);
01698         MesPrint("ERROR: PolyRatFun cannot have more than two arguments");
01699         MUNLOCK(ErrorMessageLock);
01700         Terminate(-1);
01701     }
01702
01703     // Multiply the num by den_den and den by den_num:
01704     fmpz_mpoly_mul(num, num, den_den.d, ctx);
01705     fmpz_mpoly_mul(den, den, den_num.d, ctx);
01706
01707     if ( must_normalize ) {
01708         flint::mpoly gcd(ctx);
01709         fmpz_mpoly_gcd_cofactors(gcd.d, num, den, num, den, ctx);
01710     }
01711 }
01712 /*
01713 #] flint::ratfun_read_mpoly :
01714 #[ flint::ratfun_read_poly :
01715 */
01716 // Read the uni-variate FORM rational polynomial at a and create fmpz_mpoly_t numerator and
01717 // denominator.
01718 void flint::ratfun_read_poly(const WORD *a, fmpz_poly_t num, fmpz_poly_t den) {
01719
01720     // The end of the arguments:
01721     const WORD* arg_stop = a+a[1];
01722
01723     const bool must_normalize = (a[2] & MUSTCLEANPRF) != 0;
01724
01725     a += FUNHEAD;
01726     if ( a >= arg_stop ) {
01727         MLOCK(ErrorMessageLock);
01728         MesPrint("ERROR: PolyRatFun cannot have zero arguments");
01729         MUNLOCK(ErrorMessageLock);
01730         Terminate(-1);
01731     }
01732
01733     // Polys to collect the "den of the num" and "den of the den".
01734     // Input can arrive like this when enabling the PolyRatFun or moving things into it.
01735     flint::poly den_num, den_den;
01736
01737     // Read the numerator
01738     flint::from_argument_poly(num, den_num.d, a, true);

```

```

01739     NEXTARG(a);
01740
01741     if ( a < arg_stop ) {
01742         // Read the denominator
01743         flint::from_argument_poly(den, den_den.d, a, true);
01744         NEXTARG(a);
01745     }
01746     else {
01747         // The denominator is 1
01748         MLOCK(ErrorMessageLock);
01749         MesPrint("implement this");
01750         MUNLOCK(ErrorMessageLock);
01751         Terminate(-1);
01752     }
01753     if ( a < arg_stop ) {
01754         MLOCK(ErrorMessageLock);
01755         MesPrint("ERROR: PolyRatFun cannot have more than two arguments");
01756         MUNLOCK(ErrorMessageLock);
01757         Terminate(-1);
01758     }
01759
01760     // Multiply the num by den_den and den by den_num:
01761     fmpz_poly_mul(num, num, den_den.d);
01762     fmpz_poly_mul(den, den, den_num.d);
01763
01764     if ( must_normalize ) {
01765         flint::poly gcd;
01766         flint::util::simplify_fmpz_poly(num, den, gcd.d);
01767     }
01768 }
01769 /*
01770 #] flint::ratfun_read_poly :
01771 #[ flint::to_argument_mpoly :
01772 */
01773 // Convert a fmpz_mpoly_t to a FORM argument (or 0-terminated list of terms: with_arghead==false).
01774 // If the caller is building an output term, prev_size contains the size of the term so far, to
01775 // check that the output fits in AM.MaxTer if must_fit_term.
01776 // All coefficients will be divided by denscale (which might just be 1).
01777 // If write is false, we never write to out but only track the total would-be size. This lets this
01778 // function be repurposed as a "size of FORM notation" function without duplicating the code.
01779 #define IFW(x) { if ( write ) {x;} }
01780 uint64_t flint::to_argument_mpoly(PHEAD WORD *out, const bool with_arghead,
01781     const bool must_fit_term, const bool write, const uint64_t prev_size, const fmpz_mpoly_t poly,
01782     const var_map_t &var_map, const fmpz_mpoly_ctx_t ctx, const fmpz_t denscale) {
01783
01784     // out is modified later, keep the pointer at entry
01785     const WORD* out_entry = out;
01786
01787     // Track the total size written. We could do this with pointer differences, but if
01788     // write == false we don't write to or move out to be able to find the size that way.
01789     uint64_t ws = 0;
01790
01791     // Check there is at least space for ARGHEAD WORDs (the arghead or fast-notation number/symbol)
01792     if ( write && must_fit_term && (sizeof(WORD)*(prev_size + ARGHEAD) > (size_t)AM.MaxTer) ) {
01793         MLOCK(ErrorMessageLock);
01794         MesPrint("flint::to_argument_mpoly: output exceeds MaxTermSize");
01795         MUNLOCK(ErrorMessageLock);
01796         Terminate(-1);
01797     }
01798
01799     // Create the inverse of var_map, so we don't have to search it for each symbol written
01800     var_map_t var_map_inv;
01801     for ( auto x: var_map ) {
01802         var_map_inv[x.second] = x.first;
01803     }
01804
01805     vector<int64_t> exponents(var_map.size());
01806     const int64_t n_terms = fmpz_mpoly_length(poly, ctx);
01807
01808     if ( n_terms == 0 ) {
01809         if ( with_arghead ) {
01810             IFW(*out++ = -SNUMBER); ws++;
01811             IFW(*out++ = 0); ws++;
01812             return ws;
01813         }
01814         else {
01815             IFW(*out++ = 0); ws++;
01816             return ws;
01817         }
01818     }
01819
01820     // For dividing out denscale
01821     flint::fmpz coeff, den, gcd;
01822
01823     // The mpoly might be constant or a single symbol with coeff 1. Use fast notation if possible.
01824     if ( with_arghead && n_terms == 1 ) {
01825

```

```

01826     if ( fmpz_mpoly_is_fmpz(poly, ctx) ) {
01827         // The mpoly is constant. Use fast notation if the number is an integer and small enough:
01828
01829         fmpz_mpoly_get_term_coeff_fmpz(coeff.d, poly, 0, ctx);
01830         fmpz_set(den.d, denscale);
01831         flint::util::simplify_fmpz(coeff.d, den.d, gcd.d);
01832
01833         if ( fmpz_is_one(den.d) && fmpz_fits_si(coeff.d) ) {
01834             const int64_t fast_coeff = fmpz_get_si(coeff.d);
01835             // While ">=", could work here, FORM does not use fast notation for INT_MIN
01836             if ( fast_coeff > INT32_MIN && fast_coeff <= INT32_MAX ) {
01837                 IFW(*out++ = -SNUMBER); ws++;
01838                 IFW(*out++ = (WORD)fast_coeff); ws++;
01839                 return ws;
01840             }
01841         }
01842     }
01843
01844     else {
01845         fmpz_mpoly_get_term_coeff_fmpz(coeff.d, poly, 0, ctx);
01846         fmpz_set(den.d, denscale);
01847         flint::util::simplify_fmpz(coeff.d, den.d, gcd.d);
01848
01849         if ( fmpz_is_one(coeff.d) && fmpz_is_one(den.d) ) {
01850             // The coefficient is one. Now check the symbol powers:
01851
01852             fmpz_mpoly_get_term_exp_si((slong*)exponents.data(), poly, 0, ctx);
01853             int64_t use_fast = 0;
01854             uint32_t fast_symbol = 0;
01855
01856             for ( size_t i = 0; i < var_map.size(); i++ ) {
01857                 if ( exponents[i] == 1 ) fast_symbol = var_map_inv[i];
01858                 use_fast += exponents[i];
01859             }
01860
01861             // use_fast has collected the total degree. If it is 1, then fast_symbol holds the
code
01862             if ( use_fast == 1 ) {
01863                 IFW(*out++ = -SYMBOL); ws++;
01864                 IFW(*out++ = fast_symbol); ws++;
01865                 return ws;
01866             }
01867         }
01868     }
01869 }
01870
01871 WORD *tmp_coeff = (WORD *)NumberMalloc("flint::to_argument_mpoly");
01872 WORD *tmp_den = (WORD *)NumberMalloc("flint::to_argument_mpoly");
01873
01874 WORD* arg_size = 0;
01875 WORD* arg_flag = 0;
01876 if ( with_arghead ) {
01877     IFW(arg_size = out++); ws++; // total arg size
01878     IFW(arg_flag = out++); ws++;
01879     IFW(*arg_flag = 0); // clean argument
01880 }
01881
01882 for ( int64_t i = 0; i < n_terms; i++ ) {
01883
01884     fmpz_mpoly_get_term_exp_si((slong*)exponents.data(), poly, i, ctx);
01885
01886     fmpz_mpoly_get_term_coeff_fmpz(coeff.d, poly, i, ctx);
01887     fmpz_set(den.d, denscale);
01888     flint::util::simplify_fmpz(coeff.d, den.d, gcd.d);
01889
01890     uint32_t num_symbols = 0;
01891     for ( size_t j = 0; j < var_map.size(); j++ ) {
01892         if ( exponents[j] != 0 ) { num_symbols += 1; }
01893     }
01894
01895     // Convert the coefficient, write in temporary space
01896     const WORD num_size = flint::fmpz_get_form(coeff.d, tmp_coeff);
01897     const WORD den_size = flint::fmpz_get_form(den.d, tmp_den);
01898     const WORD coeff_size = [num_size, den_size] () -> WORD {
01899         WORD size = ABS(num_size) > ABS(den_size) ? ABS(num_size) : ABS(den_size);
01900         return size * SGN(num_size) * SGN(den_size);
01901     }();
01902
01903     // Now we have the number of symbols and the coeff size, we can determine the output size.
01904     // Check it fits if necessary: term size, num,den of "coeff_size", +1 for total coeff size
01905     uint64_t current_size = prev_size + ws + 1 + 2*ABS(coeff_size) + 1;
01906     if ( num_symbols ) {
01907         // symbols header, code,power of each symbol:
01908         current_size += 2 + 2*num_symbols;
01909     }
01910     if ( write && must_fit_term && (sizeof(WORD)*current_size > (size_t)AM.MaxTer) ) {

```

```

01912         MLOCK(ErrorMessageLock);
01913         MesPrint("flint::to_argument_mpoly: output exceeds MaxTermSize");
01914         MUNLOCK(ErrorMessageLock);
01915         Terminate(-1);
01916     }
01917
01918     WORD* term_size = 0;
01919     IFW(term_size = out++); ws++;
01920     if ( num_symbols ) {
01921         IFW(*out++ = SYMBOL); ws++;
01922         WORD* symbol_size = 0;
01923         IFW(symbol_size = out++); ws++;
01924         IFW(*symbol_size = 2);
01925
01926         for ( size_t j = 0; j < var_map.size(); j++ ) {
01927             if ( exponents[j] != 0 ) {
01928                 IFW(*out++ = var_map_inv[j]); ws++;
01929                 IFW(*out++ = exponents[j]); ws++;
01930                 IFW(*symbol_size += 2);
01931             }
01932         }
01933     }
01934
01935     // Copy numerator
01936     for ( WORD j = 0; j < ABS(num_size); j++ ) {
01937         IFW(*out++ = tmp_coeff[j]); ws++;
01938     }
01939     for ( WORD j = ABS(num_size); j < ABS(coeff_size); j++ ) {
01940         IFW(*out++ = 0); ws++;
01941     }
01942     // Copy denominator
01943     for ( WORD j = 0; j < ABS(den_size); j++ ) {
01944         IFW(*out++ = tmp_den[j]); ws++;
01945     }
01946     for ( WORD j = ABS(den_size); j < ABS(coeff_size); j++ ) {
01947         IFW(*out++ = 0); ws++;
01948     }
01949
01950     IFW(*out = 2*ABS(coeff_size) + 1); // the size of the coefficient
01951     IFW(if ( coeff_size < 0 ) { *out = -(*out); });
01952     IFW(out++); ws++;
01953
01954     IFW(*term_size = out - term_size);
01955 }
01956
01957 if ( with_arghead ) {
01958     IFW(*arg_size = out - arg_size);
01959     if ( write ) {
01960         // Sort into form highfirst ordering
01961         flint::form_sort(BHEAD (WORD*) (out_entry));
01962     }
01963 }
01964 else {
01965     // with no arghead, we write a terminating zero
01966     IFW(*out++ = 0); ws++;
01967 }
01968
01969 NumberFree(tmp_coeff, "flint::to_argument_mpoly");
01970 NumberFree(tmp_den, "flint::to_argument_mpoly");
01971
01972 return ws;
01973 }
01974
01975 // If no denscale argument is supplied, just set it to 1 and call the usual function
01976 uint64_t flint::to_argument_mpoly(PHEAD WORD *out, const bool with_arghead,
01977 const bool must_fit_term, const bool write, const uint64_t prev_size, const fmpz_mpoly_t poly,
01978 const var_map_t &var_map, const fmpz_mpoly_ctx_t ctx) {
01979
01980     flint::fmpz tmp;
01981     fmpz_set_ui(tmp.d, 1);
01982
01983     uint64_t ret = flint::to_argument_mpoly(BHEAD out, with_arghead, must_fit_term, write,
01984 prev_size, poly, var_map, ctx, tmp.d);
01985
01986     return ret;
01987 }
01988 /*
01989 #] flint::to_argument_mpoly :
01990 #[ flint::to_argument_poly :
01991 */
01992 // Convert a fmpz_poly_t to a FORM argument (or 0-terminated list of terms: with_arghead==false).
01993 // If the caller is building an output term, prev_size contains the size of the term so far, to
01994 // check that the output fits in AM.MaxTer if must_fit_term.
01995 // All coefficients will be divided by denscale (which might just be 1).
01996 // If write is false, we never write to out but only track the total would-be size. This lets this
01997 // function be repurposed as a "size of FORM notation" function without duplicating the code.
01998 uint64_t flint::to_argument_poly(PHEAD WORD *out, const bool with_arghead,

```

```

01999     const bool must_fit_term, const bool write, const uint64_t prev_size, const fmpz_poly_t poly,
02000     const var_map_t &var_map, const fmpz_t denscale) {
02001
02002     // Track the total size written. We could do this with pointer differences, but if
02003     // write == false we don't write to or move out to be able to find the size that way.
02004     uint64_t ws = 0;
02005
02006     // Check there is at least space for ARGHEAD WORDs (the arghead or fast-notation number/symbol)
02007     if ( write && must_fit_term && (sizeof(WORD)*(prev_size + ARGHEAD) > (size_t)AM.MaxTer) ) {
02008         MLOCK(ErrorMessageLock);
02009         MesPrint("flint::to_argument_poly: output exceeds MaxTermSize");
02010         MUNLOCK(ErrorMessageLock);
02011         Terminate(-1);
02012     }
02013
02014     // Create the inverse of var_map, so we don't have to search it for each symbol written
02015     var_map_t var_map_inv;
02016     for ( auto x: var_map ) {
02017         var_map_inv[x.second] = x.first;
02018     }
02019
02020     const int64_t n_terms = fmpz_poly_length(poly);
02021
02022     // The poly is zero
02023     if ( n_terms == 0 ) {
02024         if ( with_arghead ) {
02025             IFW(*out++ = -SNUMBER); ws++;
02026             IFW(*out++ = 0); ws++;
02027             return ws;
02028         }
02029         else {
02030             IFW(*out++ = 0); ws++;
02031             return ws;
02032         }
02033     }
02034
02035     // For dividing out denscale
02036     flint::fmpz coeff, den, gcd;
02037
02038     // The poly is constant, use fast notation if the coefficient is integer and small enough
02039     if ( with_arghead && n_terms == 1 ) {
02040
02041         fmpz_poly_get_coeff_fmpz(coeff.d, poly, 0);
02042         fmpz_set(den.d, denscale);
02043         flint::util::simplify_fmpz(coeff.d, den.d, gcd.d);
02044
02045         if ( fmpz_is_one(den.d) && fmpz_fits_si(coeff.d) ) {
02046             const long fast_coeff = fmpz_get_si(coeff.d);
02047             // While ">=", could work here, FORM does not use fast notation for INT_MIN
02048             if ( fast_coeff > INT_MIN && fast_coeff <= INT_MAX ) {
02049                 IFW(*out++ = -SNUMBER); ws++;
02050                 IFW(*out++ = (WORD)fast_coeff); ws++;
02051                 return ws;
02052             }
02053         }
02054     }
02055
02056     // The poly might be a single symbol with coeff 1, use fast notation if so.
02057     if ( with_arghead && n_terms == 2 ) {
02058         if ( fmpz_is_zero(fmpz_poly_get_coeff_ptr(poly, 0)) ) {
02059             // The constant term is zero
02060
02061             fmpz_poly_get_coeff_fmpz(coeff.d, poly, 1);
02062             fmpz_set(den.d, denscale);
02063             flint::util::simplify_fmpz(coeff.d, den.d, gcd.d);
02064
02065             if ( fmpz_is_one(coeff.d) && fmpz_is_one(den.d) ) {
02066                 // Single symbol with coeff 1. Use fast notation:
02067                 IFW(*out++ = -SYMBOL); ws++;
02068                 IFW(*out++ = var_map_inv[0]); ws++;
02069                 return ws;
02070             }
02071         }
02072     }
02073
02074     WORD *tmp_coeff = (WORD *)NumberMalloc("flint::to_argument_poly");
02075     WORD *tmp_den = (WORD *)NumberMalloc("flint::to_argument_poly");
02076
02077     WORD* arg_size = 0;
02078     WORD* arg_flag = 0;
02079     if ( with_arghead ) {
02080         IFW(arg_size = out++); ws++; // total arg size
02081         IFW(arg_flag = out++); ws++;
02082         IFW(*arg_flag = 0); // clean argument
02083     }
02084
02085     // In reverse, since we want a "highfirst" output

```

```

02086     for ( int64_t i = n_terms-1; i >= 0; i-- ) {
02087
02088         // fmpz_poly is dense, there might be many zero coefficients:
02089         if ( !fmpz_is_zero(fmpz_poly_get_coeff_ptr(poly, i)) ) {
02090
02091             fmpz_poly_get_coeff_fmpz(coeff.d, poly, i);
02092             fmpz_set(den.d, denscale);
02093             flint::util::simplify_fmpz(coeff.d, den.d, gcd.d);
02094
02095             // Convert the coefficient, write in temporary space
02096             const WORD num_size = flint::fmpz_get_form(coeff.d, tmp_coeff);
02097             const WORD den_size = flint::fmpz_get_form(den.d, tmp_den);
02098             const WORD coeff_size = [num_size, den_size] () -> WORD {
02099                 WORD size = ABS(num_size) > ABS(den_size) ? ABS(num_size) : ABS(den_size);
02100                 return size * SGN(num_size) * SGN(den_size);
02101             };
02102
02103             // Now we have the coeff size, we can determine the output size
02104             // Check it fits if necessary: symbol code, power, num, den of "coeff_size",
02105             // +1 for total coeff size
02106             uint64_t current_size = prev_size + ws + 1 + 2*ABS(coeff_size) + 1;
02107             if ( i > 0 ) {
02108                 // and also symbols header, code, power of the symbol
02109                 current_size += 4;
02110             }
02111             if ( write && must_fit_term && (sizeof(WORD)*current_size > (size_t)AM.MaxTer) ) {
02112                 MLOCK(ErrorMessageLock);
02113                 MesPrint("flint::to_argument_poly: output exceeds MaxTermSize");
02114                 MUNLOCK(ErrorMessageLock);
02115                 Terminate(-1);
02116             }
02117
02118             WORD* term_size = 0;
02119             IFW(term_size = out++); ws++;
02120
02121             if ( i > 0 ) {
02122                 IFW(*out++ = SYMBOL); ws++;
02123                 IFW(*out++ = 4); ws++; // The symbol array size, it is univariate
02124                 IFW(*out++ = var_map_inv[0]); ws++;
02125                 IFW(*out++ = i); ws++;
02126             }
02127
02128             // Copy numerator
02129             for ( WORD j = 0; j < ABS(num_size); j++ ) {
02130                 IFW(*out++ = tmp_coeff[j]); ws++;
02131             }
02132             for ( WORD j = ABS(num_size); j < ABS(coeff_size); j++ ) {
02133                 IFW(*out++ = 0); ws++;
02134             }
02135             // Copy denominator
02136             for ( WORD j = 0; j < ABS(den_size); j++ ) {
02137                 IFW(*out++ = tmp_den[j]); ws++;
02138             }
02139             for ( WORD j = ABS(den_size); j < ABS(coeff_size); j++ ) {
02140                 IFW(*out++ = 0); ws++;
02141             }
02142
02143             IFW(*out = 2*ABS(coeff_size) + 1); // the size of the coefficient
02144             IFW(if ( coeff_size < 0 ) { *out = -(*out); });
02145             IFW(out++); ws++;
02146
02147             IFW(*term_size = out - term_size);
02148         }
02149     }
02150
02151     if ( with_arghead ) {
02152         IFW(*arg_size = out - arg_size);
02153     }
02154     else {
02155         // with no arghead, we write a terminating zero
02156         IFW(*out++ = 0); ws++;
02157     }
02158
02159     NumberFree(tmp_coeff, "flint::to_argument_poly");
02160     NumberFree(tmp_den, "flint::to_argument_poly");
02161
02162     return ws;
02163 }
02164
02165 // If no denscale argument is supplied, just set it to 1 and call the usual function
02166 uint64_t flint::to_argument_poly(PHEAD WORD *out, const bool with_arghead,
02167     const bool must_fit_term, const bool write, const uint64_t prev_size, const fmpz_poly_t poly,
02168     const var_map_t &var_map) {
02169     flint::fmpz tmp;
02170     fmpz_set_ui(tmp.d, 1);

```

```

02173
02174     uint64_t ret = flint::to_argument_poly(BHEAD out, with_arghead, must_fit_term, write, prev_size,
02175         poly, var_map, tmp.d);
02176
02177     return ret;
02178 }
02179 /*
02180 #] flint::to_argument_poly :
02181 */
02182
02183 // Utility functions
02184 /*
02185 #[ flint::util::simplify_fmpz :
02186 */
02187 // Divide the GCD out of num and den
02188 inline void flint::util::simplify_fmpz(fmpz_t num, fmpz_t den, fmpz_t gcd) {
02189     fmpz_gcd(gcd, num, den);
02190     if ( !fmpz_is_one(gcd) ) {
02191         fmpz_divexact(num, num, gcd);
02192         fmpz_divexact(den, den, gcd);
02193     }
02194 }
02195 /*
02196 #] flint::util::simplify_fmpz :
02197 #[ flint::util::simplify_fmpz_poly :
02198 */
02199 // Divide the GCD out of num and den
02200 inline void flint::util::simplify_fmpz_poly(fmpz_poly_t num, fmpz_poly_t den, fmpz_poly_t gcd) {
02201     fmpz_poly_gcd(gcd, num, den);
02202     if ( !fmpz_poly_is_one(gcd) ) {
02203 #if __FLINT_RELEASE >= 30100
02204         // This should be faster than fmpz_poly_div, see https://github.com/flintlib/flint/pull/1766
02205         fmpz_poly_divexact(num, num, gcd);
02206         fmpz_poly_divexact(den, den, gcd);
02207     #else
02208         fmpz_poly_div(num, num, gcd);
02209         fmpz_poly_div(den, den, gcd);
02210     #endif
02211     }
02212 }
02213 /*
02214 #] flint::util::simplify_fmpz_poly :
02215 #[ flint::util::fix_sign_fmpz_mpoly_ratfun :
02216 */
02217 inline void flint::util::fix_sign_fmpz_mpoly_ratfun(fmpz_mpoly_t num, fmpz_mpoly_t den,
02218     const fmpz_mpoly_ctx_t ctx) {
02219     // Fix sign to align with poly: the leading denominator term should have a positive coeff
02220     if ( fmpz_sgn(fmpz_mpoly_term_coeff_ref(den, 0, ctx)) == -1 ) {
02221         fmpz_mpoly_neg(num, num, ctx);
02222         fmpz_mpoly_neg(den, den, ctx);
02223     }
02224 }
02225 /*
02226 #] flint::util::fix_sign_fmpz_mpoly_ratfun :
02227 #[ flint::util::fix_sign_fmpz_poly_ratfun :
02228 */
02229 inline void flint::util::fix_sign_fmpz_poly_ratfun(fmpz_poly_t num, fmpz_poly_t den) {
02230     // Fix sign to align with poly: the leading denominator term should have a positive coeff
02231     if ( fmpz_sgn(fmpz_poly_get_coeff_ptr(den, fmpz_poly_degree(den))) == -1 ) {
02232         fmpz_poly_neg(num, num);
02233         fmpz_poly_neg(den, den);
02234     }
02235 }
02236 /*
02237 #] flint::util::fix_sign_fmpz_poly_ratfun :
02238 */

```

4.41 flintinterface.h File Reference

```

#include "form3.h"
#include <flint/flint.h>
#include <flint/fmpz.h>
#include <flint/fmpz_mpoly.h>
#include <flint/fmpz_mpoly_factor.h>
#include <flint/fmpz_poly.h>
#include <flint/fmpz_poly_factor.h>
#include <cassert>

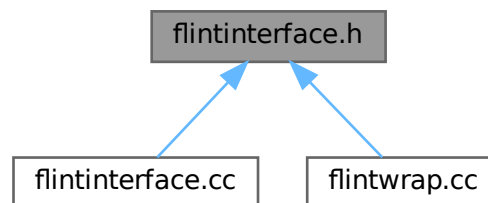
```

```
#include <stdint>
#include <iostream>
#include <map>
#include <vector>
```

Include dependency graph for flintinterface.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class [flint::fmpz](#)
- class [flint::poly](#)
- class [flint::poly_factor](#)
- class [flint::mpoly](#)
- class [flint::mpoly_factor](#)
- class [flint::mpoly_ctx](#)

Typedefs

- typedef `std::map< uint32_t, uint32_t >` [flint::var_map_t](#)

Functions

- void [flint::cleanup](#) (void)
- void [flint::cleanup_master](#) (void)
- WORD * [flint::divmod_mpoly](#) (PHEAD const WORD *, const WORD *, const bool, const WORD, const var_map_t &)
- WORD * [flint::divmod_poly](#) (PHEAD const WORD *, const WORD *, const bool, const WORD, const var_map_t &)
- WORD * [flint::factorize_mpoly](#) (PHEAD const WORD *, WORD *, const bool, const bool, const var_map_t &)

- `WORD * flint::factorize_poly` (PHEAD const WORD *, WORD *, const bool, const bool, const var_map_t &)
- `void flint::form_sort` (PHEAD WORD *)
- `uint64_t flint::from_argument_mpoly` (fmpz_mpoly_t, fmpz_mpoly_t, const WORD *, const bool, const var_map_t &, const fmpz_mpoly_ctx_t)
- `uint64_t flint::from_argument_poly` (fmpz_poly_t, fmpz_poly_t, const WORD *, const bool)
- `WORD flint::fmpz_get_form` (fmpz_t, WORD *)
- `void flint::fmpz_set_form` (fmpz_t, UWORD *, WORD)
- `WORD * flint::gcd_mpoly` (PHEAD const WORD *, const WORD *, const WORD, const var_map_t &)
- `WORD * flint::gcd_poly` (PHEAD const WORD *, const WORD *, const WORD, const var_map_t &)
- `var_map_t flint::get_variables` (const vector< WORD * > &, const bool, const bool)
- `WORD * flint::inverse_poly` (PHEAD const WORD *, const WORD *, const var_map_t &)
- `WORD * flint::mul_mpoly` (PHEAD const WORD *, const WORD *, const var_map_t &)
- `WORD * flint::mul_poly` (PHEAD const WORD *, const WORD *, const var_map_t &)
- `void flint::ratfun_add_mpoly` (PHEAD const WORD *, const WORD *, WORD *, const var_map_t &)
- `void flint::ratfun_add_poly` (PHEAD const WORD *, const WORD *, WORD *, const var_map_t &)
- `void flint::ratfun_normalize_mpoly` (PHEAD WORD *, const var_map_t &)
- `void flint::ratfun_normalize_poly` (PHEAD WORD *, const var_map_t &)
- `void flint::ratfun_read_mpoly` (const WORD *, fmpz_mpoly_t, fmpz_mpoly_t, const var_map_t &, fmpz_mpoly_ctx_t)
- `void flint::ratfun_read_poly` (const WORD *, fmpz_poly_t, fmpz_poly_t)
- `uint64_t flint::to_argument_mpoly` (PHEAD WORD *, const bool, const bool, const bool, const uint64_t, const fmpz_mpoly_t, const var_map_t &, const fmpz_mpoly_ctx_t)
- `uint64_t flint::to_argument_mpoly` (PHEAD WORD *, const bool, const bool, const bool, const uint64_t, const fmpz_mpoly_t, const var_map_t &, const fmpz_mpoly_ctx_t, const fmpz_t)
- `uint64_t flint::to_argument_poly` (PHEAD WORD *, const bool, const bool, const bool, const uint64_t, const fmpz_poly_t, const var_map_t &)
- `uint64_t flint::to_argument_poly` (PHEAD WORD *, const bool, const bool, const bool, const uint64_t, const fmpz_poly_t, const var_map_t &, const fmpz_t)
- `void flint::util::simplify_fmpz` (fmpz_t, fmpz_t, fmpz_t)
- `void flint::util::simplify_fmpz_poly` (fmpz_poly_t, fmpz_poly_t, fmpz_poly_t)
- `void flint::util::fix_sign_fmpz_mpoly_ratfun` (fmpz_mpoly_t, fmpz_mpoly_t, const fmpz_mpoly_ctx_t)
- `void flint::util::fix_sign_fmpz_poly_ratfun` (fmpz_poly_t, fmpz_poly_t)

4.41.1 Detailed Description

Prototypes for functions in [flintinterface.cc](#)

Definition in file [flintinterface.h](#).

4.41.2 Typedef Documentation

4.41.2.1 var_map_t

```
typedef std::map<uint32_t,uint32_t> flint::var_map_t
```

Definition at line 76 of file [flintinterface.h](#).

4.41.3 Function Documentation

4.41.3.1 cleanup()

```
void flint::cleanup (
    void )
```

Definition at line 76 of file [flintinterface.cc](#).

4.41.3.2 cleanup_master()

```
void flint::cleanup_master (
    void )
```

Definition at line 83 of file [flintinterface.cc](#).

4.41.3.3 divmod_mpoly()

```
WORD * flint::divmod_mpoly (
    PHEAD const WORD * a,
    const WORD * b,
    const bool return_rem,
    const WORD must_fit_term,
    const var_map_t & var_map )
```

Definition at line 91 of file [flintinterface.cc](#).

4.41.3.4 divmod_poly()

```
WORD * flint::divmod_poly (
    PHEAD const WORD * a,
    const WORD * b,
    const bool return_rem,
    const WORD must_fit_term,
    const var_map_t & var_map )
```

Definition at line 161 of file [flintinterface.cc](#).

4.41.3.5 factorize_mpoly()

```
WORD * flint::factorize_mpoly (
    PHEAD const WORD * argin,
    WORD * argout,
    const bool with_arghead,
    const bool is_fun_arg,
    const var_map_t & var_map )
```

Definition at line 235 of file [flintinterface.cc](#).

4.41.3.6 factorize_poly()

```
WORD * flint::factorize_poly (
    PHEAD const WORD * argin,
    WORD * argout,
    const bool with_arghead,
    const bool is_fun_arg,
    const var_map_t & var_map )
```

Definition at line 383 of file [flintinterface.cc](#).

4.41.3.7 form_sort()

```
void flint::form_sort (
    PHEAD WORD * terms )
```

Definition at line 456 of file [flintinterface.cc](#).

4.41.3.8 from_argument_mpoly()

```
uint64_t flint::from_argument_mpoly (
    fmpz_mpoly_t poly,
    fmpz_mpoly_t denpoly,
    const WORD * args,
    const bool with_arghead,
    const var_map_t & var_map,
    const fmpz_mpoly_ctx_t ctx )
```

Definition at line 511 of file [flintinterface.cc](#).

4.41.3.9 from_argument_poly()

```
uint64_t flint::from_argument_poly (
    fmpz_poly_t poly,
    fmpz_poly_t denpoly,
    const WORD * args,
    const bool with_arghead )
```

Definition at line 654 of file [flintinterface.cc](#).

4.41.3.10 fmpz_get_form()

```
WORD flint::fmpz_get_form (
    fmpz_t z,
    WORD * a )
```

Definition at line 784 of file [flintinterface.cc](#).

4.41.3.11 fmpz_set_form()

```
void flint::fmpz_set_form (
    fmpz_t z,
    UWORD * a,
    WORD na )
```

Definition at line 849 of file [flintinterface.cc](#).

4.41.3.12 gcd_mpoly()

```
WORD * flint::gcd_mpoly (
    PHEAD const WORD * a,
    const WORD * b,
    const WORD must_fit_term,
    const var_map_t & var_map )
```

Definition at line 915 of file [flintinterface.cc](#).

4.41.3.13 gcd_poly()

```
WORD * flint::gcd_poly (
    PHEAD const WORD * a,
    const WORD * b,
    const WORD must_fit_term,
    const var_map_t & var_map )
```

Definition at line 1011 of file [flintinterface.cc](#).

4.41.3.14 get_variables()

```
flint::var_map_t flint::get_variables (
    const vector< WORD * > & es,
    const bool with_arghead,
    const bool sort_vars )
```

Definition at line 1093 of file [flintinterface.cc](#).

4.41.3.15 inverse_poly()

```
WORD * flint::inverse_poly (
    PHEAD const WORD * a,
    const WORD * b,
    const var_map_t & var_map )
```

Definition at line 1210 of file [flintinterface.cc](#).

4.41.3.16 mul_mpoly()

```
WORD * flint::mul_mpoly (
    PHEAD const WORD * a,
    const WORD * b,
    const var_map_t & var_map )
```

Definition at line [1286](#) of file [flintinterface.cc](#).

4.41.3.17 mul_poly()

```
WORD * flint::mul_poly (
    PHEAD const WORD * a,
    const WORD * b,
    const var_map_t & var_map )
```

Definition at line [1338](#) of file [flintinterface.cc](#).

4.41.3.18 ratfun_add_mpoly()

```
void flint::ratfun_add_mpoly (
    PHEAD const WORD * t1,
    const WORD * t2,
    WORD * out,
    const var_map_t & var_map )
```

Definition at line [1389](#) of file [flintinterface.cc](#).

4.41.3.19 ratfun_add_poly()

```
void flint::ratfun_add_poly (
    PHEAD const WORD * t1,
    const WORD * t2,
    WORD * out,
    const var_map_t & var_map )
```

Definition at line [1441](#) of file [flintinterface.cc](#).

4.41.3.20 ratfun_normalize_mpoly()

```
void flint::ratfun_normalize_mpoly (
    PHEAD WORD * term,
    const var_map_t & var_map )
```

Definition at line [1494](#) of file [flintinterface.cc](#).

4.41.3.21 `ratfun_normalize_poly()`

```
void flint::ratfun_normalize_poly (
    PHEAD WORD * term,
    const var_map_t & var_map )
```

Definition at line 1577 of file [flintinterface.cc](#).

4.41.3.22 `ratfun_read_mpoly()`

```
void flint::ratfun_read_mpoly (
    const WORD * a,
    fmpz_mpoly_t num,
    fmpz_mpoly_t den,
    const var_map_t & var_map,
    fmpz_mpoly_ctx_t ctx )
```

Definition at line 1660 of file [flintinterface.cc](#).

4.41.3.23 `ratfun_read_poly()`

```
void flint::ratfun_read_poly (
    const WORD * a,
    fmpz_poly_t num,
    fmpz_poly_t den )
```

Definition at line 1718 of file [flintinterface.cc](#).

4.41.3.24 `to_argument_mpoly()` [1/2]

```
uint64_t flint::to_argument_mpoly (
    PHEAD WORD * out,
    const bool with_arghead,
    const bool must_fit_term,
    const bool write,
    const uint64_t prev_size,
    const fmpz_mpoly_t poly,
    const var_map_t & var_map,
    const fmpz_mpoly_ctx_t ctx )
```

Definition at line 1976 of file [flintinterface.cc](#).

4.41.3.25 `to_argument_mpoly()` [2/2]

```
uint64_t flint::to_argument_mpoly (
    PHEAD WORD * out,
    const bool with_arghead,
    const bool must_fit_term,
    const bool write,
    const uint64_t prev_size,
    const fmpz_mpoly_t poly,
    const var_map_t & var_map,
    const fmpz_mpoly_ctx_t ctx,
    const fmpz_t denscale )
```

Definition at line 1780 of file [flintinterface.cc](#).

4.41.3.26 to_argument_poly() [1/2]

```
uint64_t flint::to_argument_poly (
    PHEAD WORD * out,
    const bool with_arghead,
    const bool must_fit_term,
    const bool write,
    const uint64_t prev_size,
    const fmpz_poly_t poly,
    const var_map_t & var_map )
```

Definition at line 2167 of file [flintinterface.cc](#).

4.41.3.27 to_argument_poly() [2/2]

```
uint64_t flint::to_argument_poly (
    PHEAD WORD * out,
    const bool with_arghead,
    const bool must_fit_term,
    const bool write,
    const uint64_t prev_size,
    const fmpz_poly_t poly,
    const var_map_t & var_map,
    const fmpz_t denscale )
```

Definition at line 1998 of file [flintinterface.cc](#).

4.41.3.28 simplify_fmpz()

```
void flint::util::simplify_fmpz (
    fmpz_t num,
    fmpz_t den,
    fmpz_t gcd ) [inline]
```

Definition at line 2188 of file [flintinterface.cc](#).

4.41.3.29 simplify_fmpz_poly()

```
void flint::util::simplify_fmpz_poly (
    fmpz_poly_t num,
    fmpz_poly_t den,
    fmpz_poly_t gcd ) [inline]
```

Definition at line 2200 of file [flintinterface.cc](#).

4.41.3.30 fix_sign_fmpz_mpoly_ratfun()

```
void flint::util::fix_sign_fmpz_mpoly_ratfun (
    fmpz_mpoly_t num,
    fmpz_mpoly_t den,
    const fmpz_mpoly_ctx_t ctx ) [inline]
```

Definition at line 2217 of file [flintinterface.cc](#).

4.41.3.31 fix_sign_fmpz_poly_ratfun()

```
void flint::util::fix_sign_fmpz_poly_ratfun (
    fmpz_poly_t num,
    fmpz_poly_t den ) [inline]
```

Definition at line 2229 of file [flintinterface.cc](#).

4.42 flintinterface.h

[Go to the documentation of this file.](#)

```
00001 #pragma once
00006 /* #[] License : */
00007 /*
00008  * Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  * When using this file you are requested to refer to the publication
00010  * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  * This is considered a matter of courtesy as the development was paid
00012  * for by FOM the Dutch physics granting agency and we would like to
00013  * be able to track its scientific use to convince FOM of its value
00014  * for the community.
00015  *
00016  * This file is part of FORM.
00017  *
00018  * FORM is free software: you can redistribute it and/or modify it under the
00019  * terms of the GNU General Public License as published by the Free Software
00020  * Foundation, either version 3 of the License, or (at your option) any later
00021  * version.
00022  *
00023  * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00026  * details.
00027  *
00028  * You should have received a copy of the GNU General Public License along
00029  * with FORM. If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #[] License : */
00032
00033 extern "C" {
00034 #include "form3.h"
00035 }
00036
00037 #if defined(WINDOWS)
00038 // flint.h defines WORD(xx), which conflicts with the one defined in form3.h.
00039 #undef WORD
00040 #endif
00041
00042 #include <flint/flint.h>
00043 #if __FLINT_RELEASE >= 30000
00044 #include <flint/gr.h>
00045 #endif
00046 #include <flint/fmpz.h>
00047 #include <flint/fmpz_mpoly.h>
00048 #include <flint/fmpz_poly_factor.h>
00049 #include <flint/fmpz_poly.h>
00050 #include <flint/fmpz_poly_factor.h>
00051
00052 #if defined(WINDOWS)
00053 // Redefine WORD here to match form3.h.
00054 #undef WORD
00055 #define WORD FORM_WORD
00056 #endif
00057
00058 #include <cassert>
00059 #include <cstdint>
00060 #include <iostream>
00061 #include <map>
00062 #include <vector>
00063
00064
00065 // The bits of std that are needed:
00066 using std::cout;
00067 using std::endl;
00068 using std::map;
00069 using std::swap;
00070 using std::string;
```



```

00071 using std::vector;
00072
00073
00074 namespace flint {
00075
00076     typedef std::map<uint32_t,uint32_t> var_map_t;
00077
00078     // Small wrappers around the flint structs to enable RAII init and clear. "d" represents the
00079     // data, and this member will be passed to flint functions.
00080     class fmpz {
00081     public:
00082         fmpz_t d;
00083         fmpz() { fmpz_init(d); }
00084         ~fmpz() { fmpz_clear(d); }
00085         void print(const string& text) { cout << text; fmpz_print(d); cout << endl; }
00086     };
00087     class poly {
00088     public:
00089         fmpz_poly_t d;
00090         poly() { fmpz_poly_init(d); }
00091         ~poly() { fmpz_poly_clear(d); }
00092         void print(const string& text) {
00093             cout << text; fmpz_poly_print_pretty(d, "x"); cout << endl;
00094         }
00095     };
00096     class poly_factor {
00097     public:
00098         fmpz_poly_factor_t d;
00099         poly_factor() { fmpz_poly_factor_init(d); }
00100         ~poly_factor() { fmpz_poly_factor_clear(d); }
00101     };
00102     class mpoly {
00103     private:
00104         fmpz_mpoly_ctx_struct *ctx; // We need to keep a copy of the context pointer for clearing.
00105     public:
00106         fmpz_mpoly_t d;
00107         explicit mpoly(fmpz_mpoly_ctx_struct *ctx_in) : ctx(ctx_in) { fmpz_mpoly_init(d, ctx); }
00108         ~mpoly() { fmpz_mpoly_clear(d, ctx); }
00109         void print(const string& text) {
00110             cout << text;
00111             fmpz_mpoly_print_pretty(d, 0, ctx);
00112             cout << endl;
00113         }
00114     };
00115     class mpoly_factor {
00116     private:
00117         fmpz_mpoly_ctx_struct *ctx; // We need to keep a copy of the context pointer for clearing.
00118     public:
00119         fmpz_mpoly_factor_t d;
00120         explicit mpoly_factor(fmpz_mpoly_ctx_struct *ctx_in) : ctx(ctx_in) {
00121             fmpz_mpoly_factor_init(d, ctx);
00122         }
00123         ~mpoly_factor() { fmpz_mpoly_factor_clear(d, ctx); }
00124     };
00125     class mpoly_ctx {
00126     public:
00127         fmpz_mpoly_ctx_t d;
00128         explicit mpoly_ctx(int64_t nvars) { fmpz_mpoly_ctx_init(d, nvars, ORD_LEX); }
00129         ~mpoly_ctx() { fmpz_mpoly_ctx_clear(d); }
00130     };
00131
00132     void cleanup(void);
00133     void cleanup_master(void);
00134
00135     WORD* divmod_mpoly(PHEAD const WORD *, const WORD *, const bool, const WORD, const var_map_t &);
00136     WORD* divmod_poly(PHEAD const WORD *, const WORD *, const bool, const WORD, const var_map_t &);
00137
00138     WORD* factorize_mpoly(PHEAD const WORD *, WORD *, const bool, const bool, const var_map_t &);
00139     WORD* factorize_poly(PHEAD const WORD *, WORD *, const bool, const bool, const var_map_t &);
00140
00141     void form_sort(PHEAD WORD *);
00142
00143     uint64_t from_argument_mpoly(fmpz_mpoly_t, fmpz_mpoly_t, const WORD *, const bool,
00144         const var_map_t &, const fmpz_mpoly_ctx_t);
00145     uint64_t from_argument_poly(fmpz_poly_t, fmpz_poly_t, const WORD *, const bool);
00146
00147     WORD fmpz_get_form(fmpz_t, WORD *);
00148     void fmpz_set_form(fmpz_t, UWORD *, WORD);
00149
00150     WORD* gcd_mpoly(PHEAD const WORD *, const WORD *, const WORD, const var_map_t &);
00151     WORD* gcd_poly(PHEAD const WORD *, const WORD *, const WORD, const var_map_t &);
00152
00153     var_map_t get_variables(const vector <WORD *> &, const bool, const bool);
00154
00155     WORD* inverse_poly(PHEAD const WORD *, const WORD *, const var_map_t &);
00156
00157     WORD* mul_mpoly(PHEAD const WORD *, const WORD *, const var_map_t &);

```

```

00158     WORD* mul_poly(PHEAD const WORD *, const WORD *, const var_map_t &);
00159
00160     void ratfun_add_mpoly(PHEAD const WORD *, const WORD *, WORD *, const var_map_t &);
00161     void ratfun_add_poly(PHEAD const WORD *, const WORD *, WORD *, const var_map_t &);
00162
00163     void ratfun_normalize_mpoly(PHEAD WORD *, const var_map_t &);
00164     void ratfun_normalize_poly(PHEAD WORD *, const var_map_t &);
00165
00166     void ratfun_read_mpoly(const WORD *, fmpz_mpoly_t, fmpz_mpoly_t, const var_map_t &,
00167         fmpz_mpoly_ctx_t);
00168     void ratfun_read_poly(const WORD *, fmpz_poly_t, fmpz_poly_t);
00169
00170     uint64_t to_argument_mpoly(PHEAD WORD *, const bool, const bool, const bool, const uint64_t,
00171         const fmpz_mpoly_t, const var_map_t &, const fmpz_mpoly_ctx_t);
00172     uint64_t to_argument_mpoly(PHEAD WORD *, const bool, const bool, const bool, const uint64_t,
00173         const fmpz_mpoly_t, const var_map_t &, const fmpz_mpoly_ctx_t, const fmpz_t);
00174     uint64_t to_argument_poly(PHEAD WORD *, const bool, const bool, const bool, const uint64_t,
00175         const fmpz_poly_t, const var_map_t &);
00176     uint64_t to_argument_poly(PHEAD WORD *, const bool, const bool, const bool, const uint64_t,
00177         const fmpz_poly_t, const var_map_t &, const fmpz_t);
00178
00179     namespace util {
00180
00181         void simplify_fmpz(fmpz_t, fmpz_t, fmpz_t);
00182         void simplify_fmpz_poly(fmpz_poly_t, fmpz_poly_t, fmpz_poly_t);
00183
00184         void fix_sign_fmpz_mpoly_ratfun(fmpz_mpoly_t, fmpz_mpoly_t, const fmpz_mpoly_ctx_t);
00185         void fix_sign_fmpz_poly_ratfun(fmpz_poly_t, fmpz_poly_t);
00186
00187     }
00188 }
00189
00190 }

```

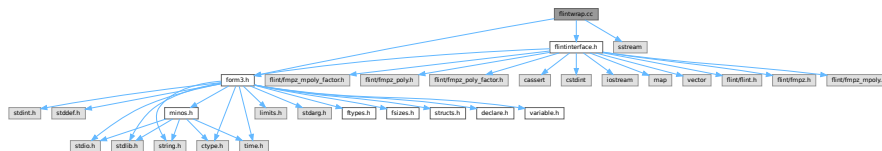
4.43 flintwrap.cc File Reference

```

#include "form3.h"
#include <sstream>
#include "flintinterface.h"

```

Include dependency graph for flintwrap.cc:



Functions

- void [flint_final_cleanup_thread](#) (void)
- void [flint_final_cleanup_master](#) (void)
- WORD * [flint_div](#) (PHEAD WORD *a, WORD *b, const WORD must_fit_term)
- int [flint_factorize_argument](#) (PHEAD WORD *argin, WORD *argout)
- WORD * [flint_factorize_dollar](#) (PHEAD WORD *argin)
- WORD * [flint_gcd](#) (PHEAD WORD *a, WORD *b, const WORD must_fit_term)
- WORD * [flint_inverse](#) (PHEAD WORD *a, WORD *b)
- WORD * [flint_mul](#) (PHEAD WORD *a, WORD *b)
- WORD * [flint_ratfun_add](#) (PHEAD WORD *t1, WORD *t2)
- int [flint_ratfun_normalize](#) (PHEAD WORD *term)
- WORD * [flint_rem](#) (PHEAD WORD *a, WORD *b, const WORD must_fit_term)
- void [flint_check_version](#) (void)

4.43.1 Detailed Description

Contains functions to call FLINT interface from the rest of FORM.

Definition in file [flintwrap.cc](#).

4.43.2 Function Documentation

4.43.2.1 flint_final_cleanup_thread()

```
void flint_final_cleanup_thread (
    void )
```

Definition at line [43](#) of file [flintwrap.cc](#).

4.43.2.2 flint_final_cleanup_master()

```
void flint_final_cleanup_master (
    void )
```

Definition at line [50](#) of file [flintwrap.cc](#).

4.43.2.3 flint_div()

```
WORD * flint_div (
    PHEAD WORD * a,
    WORD * b,
    const WORD must_fit_term )
```

Definition at line [57](#) of file [flintwrap.cc](#).

4.43.2.4 flint_factorize_argument()

```
int flint_factorize_argument (
    PHEAD WORD * argin,
    WORD *argout )
```

Definition at line [79](#) of file [flintwrap.cc](#).

4.43.2.5 flint_factorize_dollar()

```
WORD * flint_factorize_dollar (
    PHEAD WORD * argin )
```

Definition at line [100](#) of file [flintwrap.cc](#).

4.43.2.6 flint_gcd()

```
WORD * flint_gcd (
    PHEAD WORD * a,
    WORD * b,
    const WORD must_fit_term )
```

Definition at line 119 of file [flintwrap.cc](#).

4.43.2.7 flint_inverse()

```
WORD * flint_inverse (
    PHEAD WORD * a,
    WORD * b )
```

Definition at line 140 of file [flintwrap.cc](#).

4.43.2.8 flint_mul()

```
WORD * flint_mul (
    PHEAD WORD * a,
    WORD * b )
```

Definition at line 161 of file [flintwrap.cc](#).

4.43.2.9 flint_ratfun_add()

```
WORD * flint_ratfun_add (
    PHEAD WORD * t1,
    WORD * t2 )
```

Definition at line 180 of file [flintwrap.cc](#).

4.43.2.10 flint_ratfun_normalize()

```
int flint_ratfun_normalize (
    PHEAD WORD * term )
```

Definition at line 219 of file [flintwrap.cc](#).

4.43.2.11 flint_rem()

```
WORD * flint_rem (
    PHEAD WORD * a,
    WORD * b,
    const WORD must_fit_term )
```

Definition at line 292 of file [flintwrap.cc](#).

4.43.2.12 flint_check_version()

```
void flint_check_version (
    void )
```

Checks the FLINT library version at runtime.

This function should be called at startup. The program will terminate if a known buggy version of FLINT is detected.

Definition at line 321 of file [flintwrap.cc](#).

4.44 flintwrap.cc

[Go to the documentation of this file.](#)

```
00001
00005 /* #[] License : */
00006 /*
00007 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00008 * When using this file you are requested to refer to the publication
00009 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010 * This is considered a matter of courtesy as the development was paid
00011 * for by FOM the Dutch physics granting agency and we would like to
00012 * be able to track its scientific use to convince FOM of its value
00013 * for the community.
00014 *
00015 * This file is part of FORM.
00016 *
00017 * FORM is free software: you can redistribute it and/or modify it under the
00018 * terms of the GNU General Public License as published by the Free Software
00019 * Foundation, either version 3 of the License, or (at your option) any later
00020 * version.
00021 *
00022 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00025 * details.
00026 *
00027 * You should have received a copy of the GNU General Public License along
00028 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00029 */
00030 /* #[] License : */
00031
00032 extern "C" {
00033 #include "form3.h"
00034 }
00035
00036 #include <sstream>
00037 #include "flintinterface.h"
00038
00039
00040 /*
00041 #[] flint_final_cleanup_thread :
00042 */
00043 void flint_final_cleanup_thread(void) {
00044     flint::cleanup();
00045 }
00046 /*
00047 #[] flint_final_cleanup_thread :
00048 #[] flint_final_cleanup_master :
00049 */
00050 void flint_final_cleanup_master(void) {
00051     flint::cleanup_master();
00052 }
00053 /*
00054 #[] flint_final_cleanup_master :
00055 #[] flint_div :
00056 */
00057 WORD* flint_div(PHEAD WORD *a, WORD *b, const WORD must_fit_term) {
00058     // Extract expressions
00059     vector<WORD *> e;
00060     e.reserve(2);
00061     e.push_back(a);
00062     e.push_back(b);
00063     const bool with_arghead = false;
00064     const bool sort_vars = false;
00065     const flint::var_map_t var_map = flint::get_variables(e, with_arghead, sort_vars);
```

```

00066
00067     const bool return_rem = false;
00068     if ( var_map.size() > 1 ) {
00069         return flint::divmod_mpoly(BHEAD a, b, return_rem, must_fit_term, var_map);
00070     }
00071     else {
00072         return flint::divmod_poly(BHEAD a, b, return_rem, must_fit_term, var_map);
00073     }
00074 }
00075 /*
00076     #] flint_div :
00077     #[ flint_factorize_argument :
00078 */
00079 int flint_factorize_argument(PHEAD WORD *argin, WORD *argout) {
00080
00081     const bool with_arghead = true;
00082     const bool sort_vars = true;
00083     const flint::var_map_t var_map = flint::get_variables(vector<WORD*>(1,argin), with_arghead,
00084         sort_vars);
00085
00086     const bool is_fun_arg = true;
00087     if ( var_map.size() > 1 ) {
00088         flint::factorize_mpoly(BHEAD argin, argout, with_arghead, is_fun_arg, var_map);
00089     }
00090     else {
00091         flint::factorize_poly(BHEAD argin, argout, with_arghead, is_fun_arg, var_map);
00092     }
00093
00094     return 0;
00095 }
00096 /*
00097     #] flint_factorize_argument :
00098     #[ flint_factorize_dollar :
00099 */
00100 WORD* flint_factorize_dollar(PHEAD WORD *argin) {
00101
00102     const bool with_arghead = false;
00103     const bool sort_vars = true;
00104     const flint::var_map_t var_map = flint::get_variables(vector<WORD*>(1,argin), with_arghead,
00105         sort_vars);
00106
00107     const bool is_fun_arg = false;
00108     if ( var_map.size() > 1 ) {
00109         return flint::factorize_mpoly(BHEAD argin, NULL, with_arghead, is_fun_arg, var_map);
00110     }
00111     else {
00112         return flint::factorize_poly(BHEAD argin, NULL, with_arghead, is_fun_arg, var_map);
00113     }
00114 }
00115 /*
00116     #] flint_factorize_dollar :
00117     #[ flint_gcd :
00118 */
00119 WORD* flint_gcd(PHEAD WORD *a, WORD *b, const WORD must_fit_term) {
00120     // Extract expressions
00121     vector<WORD *> e;
00122     e.reserve(2);
00123     e.push_back(a);
00124     e.push_back(b);
00125     const bool with_arghead = false;
00126     const bool sort_vars = true;
00127     const flint::var_map_t var_map = flint::get_variables(e, with_arghead, sort_vars);
00128
00129     if ( var_map.size() > 1 ) {
00130         return flint::gcd_mpoly(BHEAD a, b, must_fit_term, var_map);
00131     }
00132     else {
00133         return flint::gcd_poly(BHEAD a, b, must_fit_term, var_map);
00134     }
00135 }
00136 /*
00137     #] flint_gcd :
00138     #[ flint_inverse :
00139 */
00140 WORD* flint_inverse(PHEAD WORD *a, WORD *b) {
00141     // Extract expressions
00142     vector<WORD *> e;
00143     e.reserve(2);
00144     e.push_back(a);
00145     e.push_back(b);
00146     const flint::var_map_t var_map = flint::get_variables(e, false, false);
00147
00148     if ( var_map.size() > 1 ) {
00149         MLOCK(ErrorMessageLock);
00150         MesPrint("flint_inverse: error: only univariate polynomials are supported.");
00151         MUNLOCK(ErrorMessageLock);
00152         Terminate(-1);

```

```

00153     }
00154
00155     return flint::inverse_poly(BHEAD a, b, var_map);
00156 }
00157 /*
00158 #] flint_inverse :
00159 #[ flint_mul :
00160 */
00161 WORD* flint_mul(PHEAD WORD *a, WORD *b) {
00162     // Extract expressions
00163     vector<WORD *> e;
00164     e.reserve(2);
00165     e.push_back(a);
00166     e.push_back(b);
00167     const flint::var_map_t var_map = flint::get_variables(e, false, false);
00168
00169     if ( var_map.size() > 1 ) {
00170         return flint::mul_mpoly(BHEAD a, b, var_map);
00171     }
00172     else {
00173         return flint::mul_poly(BHEAD a, b, var_map);
00174     }
00175 }
00176 /*
00177 #] flint_mul :
00178 #[ flint_ratfun_add :
00179 */
00180 WORD* flint_ratfun_add(PHEAD WORD *t1, WORD *t2) {
00181
00182     if ( AR.PolyFunExp == 1 ) {
00183         MLOCK(ErrorMessageLock);
00184         MesPrint("flint_ratfun_add: PolyFunExp unimplemented.");
00185         MUNLOCK(ErrorMessageLock);
00186         Terminate(-1);
00187     }
00188
00189     WORD *oldworkpointer = AT.WorkPointer;
00190
00191     // Extract expressions: the num and den of both prf
00192     vector<WORD *> e;
00193     e.reserve(4);
00194     for (WORD *t=t1+FUNHEAD; t<t1+t1[1];) {
00195         e.push_back(t);
00196         NEXTARG(t);
00197     }
00198     for (WORD *t=t2+FUNHEAD; t<t2+t2[1];) {
00199         e.push_back(t);
00200         NEXTARG(t);
00201     }
00202     const bool with_arghead = true;
00203     const bool sort_vars = true;
00204     const flint::var_map_t var_map = flint::get_variables(e, with_arghead, sort_vars);
00205
00206     if ( var_map.size() > 1 ) {
00207         flint::ratfun_add_mpoly(BHEAD t1, t2, oldworkpointer, var_map);
00208     }
00209     else {
00210         flint::ratfun_add_poly(BHEAD t1, t2, oldworkpointer, var_map);
00211     }
00212
00213     return oldworkpointer;
00214 }
00215 /*
00216 #] flint_ratfun_add :
00217 #[ flint_ratfun_normalize :
00218 */
00219 int flint_ratfun_normalize(PHEAD WORD *term) {
00220
00221     // The length of the coefficient
00222     const WORD ncoeff = (term + *term)[-1];
00223     // The end of the term data, before the coefficient:
00224     const WORD *tstop = term + *term - ABS(ncoeff);
00225
00226     // Search the term for multiple PolyFun or one dirty one.
00227     unsigned num_polyratfun = 0;
00228     for (WORD *t = term+1; t < tstop; t += t[1]) {
00229         if (*t == AR.PolyFun) {
00230             // Found one!
00231             num_polyratfun++;
00232             if ((t[2] & MUSTCLEANPRF) != 0) {
00233                 // Dirty, increment again to force normalisation for single PolyFun
00234                 num_polyratfun++;
00235             }
00236             if (num_polyratfun > 1) {
00237                 // We're not counting all occurrences, just determining if there
00238                 // is something to be done.
00239                 break;

```

```

00240     }
00241     }
00242 }
00243 if (num_polyratfun <= 1) {
00244     // There is nothing to do, return early
00245     return 0;
00246 }
00247
00248 // When there are polyratfun with only one argument: rename them temporarily to TMPPOLYFUN.
00249 for (WORD *t = term+1; t < tstop; t += t[1]) {
00250     if (*t == AR.PolyFun && (t[1] == FUNHEAD+t[FUNHEAD] || t[1] == FUNHEAD+2 ) ) {
00251         *t = TMPPOLYFUN;
00252     }
00253 }
00254
00255 // Extract all variables in the polyfuns
00256 // Collect pointers to each relevant argument
00257 vector<WORD *> e;
00258 e.reserve(4); // There could be more than 4 args in principle, but 4 is probably common.
00259 for (WORD *t=term+1; t<tstop; t+=t[1]) {
00260     if (*t == AR.PolyFun) {
00261         for (WORD *t2 = t+FUNHEAD; t2<t+t[1];) {
00262             e.push_back(t2);
00263             NEXTARG(t2);
00264         }
00265     }
00266 }
00267 }
00268 const bool with_arghead = true;
00269 const bool sort_vars = true;
00270 const flint::var_map_t var_map = flint::get_variables(e, with_arghead, sort_vars);
00271
00272 if ( var_map.size() > 1 ) {
00273     flint::ratfun_normalize_mpoly(BHEAD term, var_map);
00274 }
00275 else {
00276     flint::ratfun_normalize_poly(BHEAD term, var_map);
00277 }
00278
00279 // Undo renaming of single-argument PolyFun
00280 const WORD *new_tstop = term + *term - ABS((term + *term)[-1]);
00281 for (WORD *t=term+1; t<new_tstop; t+=t[1]) {
00282     if (*t == TMPPOLYFUN ) *t = AR.PolyFun;
00283 }
00284
00285 return 0;
00286 }
00287 }
00288 /*
00289 #] flint_ratfun_normalize :
00290 #[ flint_rem :
00291 */
00292 WORD* flint_rem(PHEAD WORD *a, WORD *b, const WORD must_fit_term) {
00293     // Extract expressions
00294     vector<WORD *> e;
00295     e.reserve(2);
00296     e.push_back(a);
00297     e.push_back(b);
00298     const bool with_arghead = false;
00299     const bool sort_vars = false;
00300     const flint::var_map_t var_map = flint::get_variables(e, with_arghead, sort_vars);
00301
00302     const bool return_rem = true;
00303     if ( var_map.size() > 1 ) {
00304         return flint::divmod_mpoly(BHEAD a, b, return_rem, must_fit_term, var_map);
00305     }
00306     else {
00307         return flint::divmod_poly(BHEAD a, b, return_rem, must_fit_term, var_map);
00308     }
00309 }
00310 /*
00311 #] flint_rem :
00312 #[ flint_check_version :
00313 */
00314
00321 void flint_check_version(void) {
00322     bool ok = true;
00323     std::stringstream ss(flint_version);
00324     int major, minor, patch;
00325     char dot1, dot2;
00326     if ( ss » major » dot1 » minor » dot2 » patch ) {
00327         if ( dot1 != '.' || dot2 != '.' || major < 0 || minor < 0 || patch < 0 ) {
00328             ok = false;
00329         }
00330         else if ( major * 10000 + minor * 100 + patch < 30200 ) {
00331             // flint < 3.2.0: https://github.com/form-dev/form/issues/679
00332             ok = false;

```



```

00333     }
00334 }
00335 else {
00336     ok = false;
00337 }
00338 if ( !ok ) {
00339     MesPrint("Bad FLINT version detected at runtime: %s", flint_version);
00340     Terminate(-2);
00341 }
00342 }
00343
00344 /*
00345  #] flint_check_version :
00346  */

```

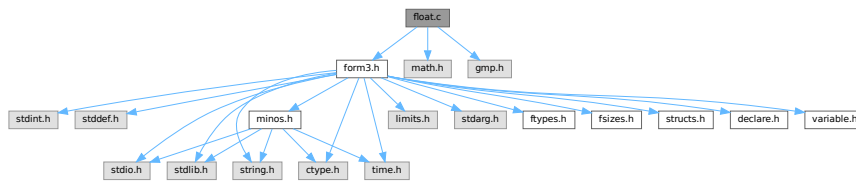
4.45 float.c File Reference

```

#include "form3.h"
#include <math.h>
#include <gmp.h>

```

Include dependency graph for float.c:



Macros

- `#define WITHCUTOFF`
- `#define GMPSPREAD (GMP_LIMB_BITS/BITSINWORD)`

Functions

- void `Form_mpf_init` (mpf_t t)
- void `Form_mpf_clear` (mpf_t t)
- void `Form_mpf_set_prec_raw` (mpf_t t, ULONG newprec)
- void `FormtoZ` (mpz_t z, UWORD *a, WORD na)
- void `ZtoForm` (UWORD *a, WORD *na, mpz_t z)
- long `FloatToInteger` (UWORD *out, mpf_t floatin, long *bitsused)
- void `IntegerToFloat` (mpf_t result, UWORD *formlong, int longsize)
- int `FloatToRat` (UWORD *ratout, WORD *nratout, mpf_t floatin)
- int `AddFloats` (PHEAD WORD *fun3, WORD *fun1, WORD *fun2)
- int `MulFloats` (PHEAD WORD *fun3, WORD *fun1, WORD *fun2)
- int `DivFloats` (PHEAD WORD *fun3, WORD *fun1, WORD *fun2)
- int `AddRatToFloat` (PHEAD WORD *outfun, WORD *infun, UWORD *formrat, WORD nrat)
- int `MulRatToFloat` (PHEAD WORD *outfun, WORD *infun, UWORD *formrat, WORD nrat)
- void `SimpleDelta` (mpf_t sum, int m)
- void `SimpleDeltaC` (mpf_t sum, int m)
- void `SingleTable` (mpf_t *tabl, int N, int m, int pow)
- void `DoubleTable` (mpf_t *tabout, mpf_t *tabin, int N, int m, int pow)
- void `EndTable` (mpf_t sum, mpf_t *tabin, int N, int m, int pow)

- void [deltaMZV](#) (mpf_t, WORD *, int)
- void [deltaEuler](#) (mpf_t, WORD *, int)
- void [deltaEulerC](#) (mpf_t, WORD *, int)
- void [CalculateMZVhalf](#) (mpf_t, WORD *, int)
- void [CalculateMZV](#) (mpf_t, WORD *, int)
- void [CalculateEuler](#) (mpf_t, WORD *, int)
- int [ExpandMZV](#) (WORD *term, WORD level)
- int [ExpandEuler](#) (WORD *term, WORD level)
- int [PackFloat](#) (WORD *, mpf_t)
- int [UnpackFloat](#) (mpf_t, WORD *)
- void [RatToFloat](#) (mpf_t result, UWORD *format, int ratsize)
- void [Form_mpf_empty](#) (mpf_t t)
- int [TestFloat](#) (WORD *fun)
- WORD [FloatFunToRat](#) (PHEAD UWORD *ratout, WORD *in)
- void [ZeroTable](#) (mpf_t *tab, int N)
- SBYTE * [ReadFloat](#) (SBYTE *s)
- UBYTE * [CheckFloat](#) (UBYTE *ss, int *spec)
- int [SetFloatPrecision](#) (WORD prec)
- int [PrintFloat](#) (WORD *fun, int numdigits)
- void [SetupMZVTables](#) (void)
- void [SetupMPFTables](#) (void)
- void [ClearMZVTables](#) (void)
- int [CoToFloat](#) (UBYTE *s)
- int [CoToRat](#) (UBYTE *s)
- int [CoStrictRounding](#) (UBYTE *s)
- int [CoChop](#) (UBYTE *s)
- int [ToFloat](#) (PHEAD WORD *term, WORD level)
- int [ToRat](#) (PHEAD WORD *term, WORD level)
- int [StrictRounding](#) (PHEAD WORD *term, WORD level, WORD prec, WORD base)
- int [Chop](#) (PHEAD WORD *term, WORD level)
- int [AddWithFloat](#) (PHEAD WORD **ps1, WORD **ps2)
- int [MergeWithFloat](#) (PHEAD WORD **interm1, WORD **interm2)
- int [EvaluateEuler](#) (PHEAD WORD *term, WORD level, WORD par)
- int [CoEvaluate](#) (UBYTE *s)

4.45.1 Detailed Description

This file contains numerical routines that deal with floating point numbers. We use the GMP for arbitrary precision floating point arithmetic. The functions of the type sin, cos, ln, sqrt etc are handled by the mpfr library. The reason that for MZV's and the general notation we use the GMP (mpf_) is because the contents of the structs have been frozen and can be used for storage in a Form function float_. With mpfr_ this is not safely possible. All mpfr_ related code is in the file [evaluate.c](#).

Definition in file [float.c](#).

4.45.2 Macro Definition Documentation

4.45.2.1 WITHCUTOFF

```
#define WITHCUTOFF
```

Definition at line 45 of file [float.c](#).

4.45.2.2 GMPSPREAD

```
#define GMPSPREAD (GMP_LIMB_BITS/BITSINWORD)
```

Definition at line 47 of file [float.c](#).

4.45.3 Function Documentation

4.45.3.1 Form_mpf_init()

```
void Form_mpf_init (  
    mpf_t t )
```

Definition at line 177 of file [float.c](#).

4.45.3.2 Form_mpf_clear()

```
void Form_mpf_clear (  
    mpf_t t )
```

Definition at line 201 of file [float.c](#).

4.45.3.3 Form_mpf_set_prec_raw()

```
void Form_mpf_set_prec_raw (  
    mpf_t t,  
    ULONG newprec )
```

Definition at line 227 of file [float.c](#).

4.45.3.4 FormtoZ()

```
void FormtoZ (  
    mpz_t z,  
    UWORD * a,  
    WORD na )
```

Definition at line 515 of file [float.c](#).

4.45.3.5 ZtoForm()

```
void ZtoForm (  
    UWORD * a,  
    WORD * na,  
    mpz_t z )
```

Definition at line 559 of file [float.c](#).

4.45.3.6 FloatToInteger()

```
long FloatToInteger (
    UWORD * out,
    mpf_t floatin,
    long * bitsused )
```

Definition at line 585 of file [float.c](#).

4.45.3.7 IntegerToFloat()

```
void IntegerToFloat (
    mpf_t result,
    UWORD * formlong,
    int longsize )
```

Definition at line 608 of file [float.c](#).

4.45.3.8 FloatToRat()

```
int FloatToRat (
    UWORD * ratout,
    WORD * nratout,
    mpf_t floatin )
```

Definition at line 672 of file [float.c](#).

4.45.3.9 AddFloats()

```
int AddFloats (
    PHEAD WORD * fun3,
    WORD * fun1,
    WORD * fun2 )
```

Definition at line 990 of file [float.c](#).

4.45.3.10 MulFloats()

```
int MulFloats (
    PHEAD WORD * fun3,
    WORD * fun1,
    WORD * fun2 )
```

Definition at line 1006 of file [float.c](#).

4.45.3.11 DivFloats()

```
int DivFloats (
    PHEAD WORD * fun3,
    WORD * fun1,
    WORD * fun2 )
```

Definition at line 1022 of file [float.c](#).

4.45.3.12 AddRatToFloat()

```
int AddRatToFloat (
    PHEAD WORD * outfun,
    WORD * infun,
    UWORD * formrat,
    WORD nrat )
```

Definition at line 1040 of file [float.c](#).

4.45.3.13 MulRatToFloat()

```
int MulRatToFloat (
    PHEAD WORD * outfun,
    WORD * infun,
    UWORD * formrat,
    WORD nrat )
```

Definition at line 1059 of file [float.c](#).

4.45.3.14 SimpleDelta()

```
void SimpleDelta (
    mpf_t sum,
    int m )
```

Definition at line 1871 of file [float.c](#).

4.45.3.15 SimpleDeltaC()

```
void SimpleDeltaC (
    mpf_t sum,
    int m )
```

Definition at line 1930 of file [float.c](#).

4.45.3.16 SingleTable()

```
void SingleTable (
    mpf_t * tabl,
    int N,
    int m,
    int pow )
```

Definition at line 1989 of file [float.c](#).

4.45.3.17 DoubleTable()

```
void DoubleTable (
    mpf_t * tabout,
    mpf_t * tabin,
    int N,
    int m,
    int pow )
```

Definition at line 2043 of file [float.c](#).

4.45.3.18 EndTable()

```
void EndTable (
    mpf_t sum,
    mpf_t * tabin,
    int N,
    int m,
    int pow )
```

Definition at line 2096 of file [float.c](#).

4.45.3.19 deltaMZV()

```
void deltaMZV (
    mpf_t result,
    WORD * indexes,
    int depth )
```

Definition at line 2143 of file [float.c](#).

4.45.3.20 deltaEuler()

```
void deltaEuler (
    mpf_t result,
    WORD * indexes,
    int depth )
```

Definition at line 2210 of file [float.c](#).

4.45.3.21 deltaEulerC()

```
void deltaEulerC (
    mpf_t result,
    WORD * indexes,
    int depth )
```

Definition at line 2266 of file [float.c](#).

4.45.3.22 CalculateMZVhalf()

```
void CalculateMZVhalf (
    mpf_t result,
    WORD * indexes,
    int depth )
```

Definition at line [2342](#) of file [float.c](#).

4.45.3.23 CalculateMZV()

```
void CalculateMZV (
    mpf_t result,
    WORD * indexes,
    int depth )
```

Definition at line [2367](#) of file [float.c](#).

4.45.3.24 CalculateEuler()

```
void CalculateEuler (
    mpf_t result,
    WORD * Zindexes,
    int depth )
```

Definition at line [2438](#) of file [float.c](#).

4.45.3.25 ExpandMZV()

```
int ExpandMZV (
    WORD * term,
    WORD level )
```

Definition at line [2520](#) of file [float.c](#).

4.45.3.26 ExpandEuler()

```
int ExpandEuler (
    WORD * term,
    WORD level )
```

Definition at line [2532](#) of file [float.c](#).

4.45.3.27 PackFloat()

```
int PackFloat (
    WORD * fun,
    mpf_t infloat )
```

Definition at line [271](#) of file [float.c](#).

4.45.3.28 UnpackFloat()

```
int UnpackFloat (
    mpf_t outfloat,
    WORD * fun )
```

Definition at line [366](#) of file [float.c](#).

4.45.3.29 RatToFloat()

```
void RatToFloat (
    mpf_t result,
    UWORD * format,
    int ratsize )
```

Definition at line [624](#) of file [float.c](#).

4.45.3.30 Form_mpf_empty()

```
void Form_mpf_empty (
    mpf_t t )
```

Definition at line [214](#) of file [float.c](#).

4.45.3.31 TestFloat()

```
int TestFloat (
    WORD * fun )
```

Definition at line [456](#) of file [float.c](#).

4.45.3.32 FloatFunToRat()

```
WORD FloatFunToRat (
    PHEAD UWORD * ratout,
    WORD * in )
```

Definition at line [646](#) of file [float.c](#).

4.45.3.33 ZeroTable()

```
void ZeroTable (
    mpf_t * tab,
    int N )
```

Definition at line [808](#) of file [float.c](#).

4.45.3.34 ReadFloat()

```
SBYTE * ReadFloat (
    SBYTE * s )
```

Definition at line 826 of file [float.c](#).

4.45.3.35 CheckFloat()

```
UBYTE * CheckFloat (
    UBYTE * ss,
    int * spec )
```

Definition at line 852 of file [float.c](#).

4.45.3.36 SetFloatPrecision()

```
int SetFloatPrecision (
    WORD prec )
```

Definition at line 905 of file [float.c](#).

4.45.3.37 PrintFloat()

```
int PrintFloat (
    WORD * fun,
    int numdigits )
```

Definition at line 934 of file [float.c](#).

4.45.3.38 SetupMZVTables()

```
void SetupMZVTables (
    void )
```

Definition at line 1076 of file [float.c](#).

4.45.3.39 SetupMPFTables()

```
void SetupMPFTables (
    void )
```

Definition at line 1140 of file [float.c](#).

4.45.3.40 ClearMZVTables()

```
void ClearMZVTables (
    void )
```

Definition at line 1174 of file [float.c](#).

4.45.3.41 CoToFloat()

```
int CoToFloat (
    UBYTE * s )
```

Definition at line 1251 of file [float.c](#).

4.45.3.42 CoToRat()

```
int CoToRat (
    UBYTE * s )
```

Definition at line 1273 of file [float.c](#).

4.45.3.43 CoStrictRounding()

```
int CoStrictRounding (
    UBYTE * s )
```

Definition at line 1300 of file [float.c](#).

4.45.3.44 CoChop()

```
int CoChop (
    UBYTE * s )
```

Definition at line 1347 of file [float.c](#).

4.45.3.45 ToFloat()

```
int ToFloat (
    PHEAD WORD * term,
    WORD level )
```

Definition at line 1443 of file [float.c](#).

4.45.3.46 ToRat()

```
int ToRat (
    PHEAD WORD * term,
    WORD level )
```

Definition at line 1475 of file [float.c](#).

4.45.3.47 StrictRounding()

```
int StrictRounding (
    PHEAD WORD * term,
    WORD level,
    WORD prec,
    WORD base )
```

Definition at line 1511 of file [float.c](#).

4.45.3.48 Chop()

```
int Chop (
    PHEAD WORD * term,
    WORD level )
```

Definition at line 1573 of file [float.c](#).

4.45.3.49 AddWithFloat()

```
int AddWithFloat (
    PHEAD WORD ** ps1,
    WORD ** ps2 )
```

Definition at line 1642 of file [float.c](#).

4.45.3.50 MergeWithFloat()

```
int MergeWithFloat (
    PHEAD WORD ** interm1,
    WORD ** interm2 )
```

Definition at line 1760 of file [float.c](#).

4.45.3.51 EvaluateEuler()

```
int EvaluateEuler (
    PHEAD WORD * term,
    WORD level,
    WORD par )
```

Definition at line 2560 of file [float.c](#).

4.45.3.52 CoEvaluate()

```
int CoEvaluate (
    UBYTE * s )
```

Definition at line 2680 of file [float.c](#).

4.46 float.c

[Go to the documentation of this file.](#)

```

00001
00011 /* #[] License : */
00012 /*
00013 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00014 *   When using this file you are requested to refer to the publication
00015 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00016 *   This is considered a matter of courtesy as the development was paid
00017 *   for by FOM the Dutch physics granting agency and we would like to
00018 *   be able to track its scientific use to convince FOM of its value
00019 *   for the community.
00020 *
00021 *   This file is part of FORM.
00022 *
00023 *   FORM is free software: you can redistribute it and/or modify it under the
00024 *   terms of the GNU General Public License as published by the Free Software
00025 *   Foundation, either version 3 of the License, or (at your option) any later
00026 *   version.
00027 *
00028 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00029 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00030 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00031 *   details.
00032 *
00033 *   You should have received a copy of the GNU General Public License along
00034 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00035 */
00036 /* #[] License : */
00037 /*
00038 *   #[] Includes : float.c
00039 */
00040
00041 #include "form3.h"
00042 #include <math.h>
00043 #include <gmp.h>
00044
00045 #define WITHCUTOFF
00046
00047 #define GMPSPREAD (GMP_LIMB_BITS/BITSINWORD)
00048
00049
00050 void Form_mpf_init(mpf_t t);
00051 void Form_mpf_clear(mpf_t t);
00052 void Form_mpf_set_prec_raw(mpf_t t, ULONG newprec);
00053 void FormtoZ(mpz_t z, UWORD *a, WORD na);
00054 void ZtoForm(UWORD *a, WORD *na, mpz_t z);
00055 long FloatToInteger(UWORD *out, mpf_t floatin, long *bitsused);
00056 void IntegerToFloat(mpf_t result, UWORD *formlong, int longsize);
00057 int FloatToRat(UWORD *ratout, WORD *nratout, mpf_t floatin);
00058 int AddFloats(PHEAD WORD *fun3, WORD *fun1, WORD *fun2);
00059 int MulFloats(PHEAD WORD *fun3, WORD *fun1, WORD *fun2);
00060 int DivFloats(PHEAD WORD *fun3, WORD *fun1, WORD *fun2);
00061 int AddRatToFloat(PHEAD WORD *outfun, WORD *infun, UWORD *formrat, WORD nrat);
00062 int MulRatToFloat(PHEAD WORD *outfun, WORD *infun, UWORD *formrat, WORD nrat);
00063 void SimpleDelta(mpf_t sum, int m);
00064 void SimpleDeltaC(mpf_t sum, int m);
00065 void SingleTable(mpf_t *tabl, int N, int m, int pow);
00066 void DoubleTable(mpf_t *tabout, mpf_t *tabin, int N, int m, int pow);
00067 void EndTable(mpf_t sum, mpf_t *tabin, int N, int m, int pow);
00068 void deltaMZV(mpf_t, WORD *, int);
00069 void deltaEuler(mpf_t, WORD *, int);
00070 void deltaEulerC(mpf_t, WORD *, int);
00071 void CalculateMZVhalf(mpf_t, WORD *, int);
00072 void CalculateMZV(mpf_t, WORD *, int);
00073 void CalculateEuler(mpf_t, WORD *, int);
00074 int ExpandMZV(WORD *term, WORD level);
00075 int ExpandEuler(WORD *term, WORD level);
00076 int PackFloat(WORD *, mpf_t);
00077 int UnpackFloat(mpf_t, WORD *);
00078 void RatToFloat(mpf_t result, UWORD *formrat, int ratsize);
00079
00080 /*
00081 *   #[] Includes :
00082 *   #[] Some GMP float info :
00083 *
00084 *   From gmp.h:
00085 *
00086 *   typedef struct
00087 *   {
00088 *       int _mp_prec;           Max precision, in number of `mp_limb_t's.
00089 *                               Set by mpf_init and modified by
00090 *                               mpf_set_prec. The area pointed to by the
00091 *                               _mp_d field contains `prec' + 1 limbs.

```

```

00092     int _mp_size;      abs(_mp_size) is the number of limbs the
00093                        last field points to.  If _mp_size is
00094                        negative this is a negative number.
00095     mp_exp_t _mp_exp; Exponent, in the base of 'mp_limb_t'.
00096     mp_limb_t *_mp_d; Pointer to the limbs.
00097 } __mpf_struct;
00098
00099 typedef __mpf_struct mpf_t[1];
00100
00101 Currently:
00102     sizeof(int) = 4
00103     sizeof(mpf_t) = 24
00104     sizeof(mp_limb_t) = 8,
00105     sizeof(mp_exp_t) = 8,
00106     sizeof a pointer is 8.
00107 If any of this changes in the future we would have to change the
00108 PackFloat and UnpackFloat routines.
00109
00110 Our float_ function is packed with the 4 arguments:
00111     -SNUMBER _mp_prec
00112     -SNUMBER _mp_size
00113     exponent which can be -SNUMBER or a regular term
00114     the limbs as n/1 in regular term format, or just an -SNUMBER.
00115 During normalization the sign is taken away from the second argument
00116 and put as the sign of the complete term. This is easier for the
00117 WriteInnerTerm routine in sch.c
00118
00119 Wildcarding of functions excludes matches with float_
00120 Float_ always ends up inside the bracket, just like poly(rat)fun.
00121
00122 From mpfr.h
00123
00124 The formats here are different. This has, among others, to do with
00125 the rounding. The result is that we need different aux variables
00126 when working with mpfr and we need a different conversion routine
00127 to float. It also means that we need to treat the mvz_ etc functions
00128 completely separately. For now we try to develop that in the file
00129 Evaluate.c. At a later stage we may merge the two files.
00130 Originally we had the sqrt in float.c but it seems better to put
00131 it in Evaluate.c
00132
00133 #] Some GMP float info :
00134 #[ Low Level :
00135
00136     In the low level routines we interact directly with the content
00137     of the GMP structs. This can be done safely, because their
00138     layout is documented. We pay particular attention to the init
00139     and clear routines, because they involve malloc/free calls.
00140
00141     #[ Explanations :
00142
00143 We need a number of facilities inside Form to deal with floating point
00144 numbers, taking into account that they are only meant for sporadic use.
00145 1: We need a special function float_ who's arguments contain a limb
00146    representation of the mpf_t of the gmp.
00147 2: Some functions may return a floating point number. This is then in
00148    the form of an occurrence of the function float_.
00149 3: We need a print routine to print this float_.
00150 4: We need routines for conversions:
00151     a: (long) int to float.
00152     b: rat to float.
00153     c: float to rat (if possible)
00154     d: float to integer (with rounding toward zero).
00155 5: We need calculational operations for add, sub, mul, div, exp.
00156 6: We need service routines to pack and unpack the function float_.
00157 7: The coefficient should be pulled inside float_.
00158 8: Float_ cannot occur inside PolyRatFun.
00159 9: We need to be able to treat float_ as the coefficient during sorting.
00160 10: For now, we need the functions mvzf_ and eulerf_ for the evaluation
00161     of mvz's and euler sums.
00162 11: We need a setting for the default precision.
00163 12: We need a special flag for the dirty field of arguments to avoid
00164     the normalization of the argument with the limbs.
00165     Alternatively, the float_ should be not a regular function, but
00166     get a status < FUNCTION. Just like SNUMBER, LNUMBER etc.
00167
00168 Note that we can keep this thread-safe. The only routine that is not
00169 thread-safe is the print routine, but that is in accordance with all
00170 other print routines in Form. When called from MesPrint it needs to
00171 be protected by a lock, as is done standard inside Form.
00172
00173 #] Explanations :
00174 #[ Form_mpf_init :
00175 */
00176
00177 void Form_mpf_init(mpf_t t)
00178 {

```

```

00179     mp_limb_t *d;
00180     int i, prec;
00181     if ( t->_mp_d ) { M_free(t->_mp_d,"Form_mpf_init"); t->_mp_d = 0; }
00182     prec = (AC.DefaultPrecision + 8*sizeof(mp_limb_t)-1)/(8*sizeof(mp_limb_t)) + 1;
00183     t->_mp_prec = prec;
00184     t->_mp_size = 0;
00185     t->_mp_exp = 0;
00186     d = (mp_limb_t *)Malloc1(prec*sizeof(mp_limb_t),"Form_mpf_init");
00187     if ( d == 0 ) {
00188         MesPrint("Fatal error in Malloc1 call in Form_mpf_init. Trying to allocate %ld bytes.",
00189                 prec*sizeof(mp_limb_t));
00190         Terminate(-1);
00191     }
00192     t->_mp_d = d;
00193     for ( i = 0; i < prec; i++ ) d[i] = 0;
00194 }
00195
00196 /*
00197     #] Form_mpf_init :
00198     #[ Form_mpf_clear :
00199 */
00200
00201 void Form_mpf_clear(mpf_t t)
00202 {
00203     if ( t->_mp_d ) { M_free(t->_mp_d,"Form_mpf_init"); t->_mp_d = 0; }
00204     t->_mp_prec = 0;
00205     t->_mp_size = 0;
00206     t->_mp_exp = 0;
00207 }
00208
00209 /*
00210     #] Form_mpf_clear :
00211     #[ Form_mpf_empty :
00212 */
00213
00214 void Form_mpf_empty(mpf_t t)
00215 {
00216     int i;
00217     for ( i = 0; i < t->_mp_prec; i++ ) t->_mp_d[i] = 0;
00218     t->_mp_size = 0;
00219     t->_mp_exp = 0;
00220 }
00221
00222 /*
00223     #] Form_mpf_empty :
00224     #[ Form_mpf_set_prec_raw :
00225 */
00226
00227 void Form_mpf_set_prec_raw(mpf_t t,ULONG newprec)
00228 {
00229     ULONG newpr = (newprec + 8*sizeof(mp_limb_t)-1)/(8*sizeof(mp_limb_t)) + 1;
00230     if ( newpr <= (ULONG)(t->_mp_prec) ) {
00231         int i, used = ABS(t->_mp_size) ;
00232         if ( newpr > (ULONG)used ) {
00233             for ( i = used; i < (int)newpr; i++ ) t->_mp_d[i] = 0;
00234         }
00235         if ( t->_mp_size < 0 ) newpr = -newpr;
00236         t->_mp_size = newpr;
00237     }
00238     else {
00239         /*
00240             We can choose here between "forget about it" or making a new malloc.
00241             If the program is well designed, we should never get here though.
00242             For now we choose the "forget it" option.
00243         */
00244         MesPrint("Trying too big a precision %ld in Form_mpf_set_prec_raw.",newprec);
00245         MesPrint("Old maximal precision is %ld.",
00246                 (size_t)(t->_mp_prec-1)*sizeof(mp_limb_t)*8);
00247         Terminate(-1);
00248     }
00249 }
00250
00251 /*
00252     #] Form_mpf_set_prec_raw :
00253     #[ PackFloat :
00254
00255     Puts an object of type mpf_t inside a function float_
00256     float_(prec, nlimbs, exp, limbs);
00257     Complication: the limbs and the exponent are long's. That means
00258     two times the size of a Form (U)WORD.
00259     To save space we could split the limbs in two because Form stores
00260     coefficients as rationals with equal space for numerator and denominator.
00261     Hence for each limb we could put the most significant UWORD in the
00262     numerator and the least significant limb in the denominator.
00263     This gives a problem with the compilation and subsequent potential
00264     normalization of the 'fraction'. Hence for now we take the wasteful
00265     approach. At a later stage we can try to veto normalization of FLOATFUN.

```

```

00266     Caution: gmp/fmp is little endian and Form is big endian.
00267
00268     Zero is represented by zero limbs (and zero exponent).
00269 */
00270
00271 int PackFloat(WORD *fun, mpf_t infloat)
00272 {
00273     WORD *t, nlimbs, num, nnum;
00274     mp_limb_t *d = infloat->_mp_d; /* Pointer to the limbs. */
00275     int i;
00276     long e = infloat->_mp_exp;
00277     t = fun;
00278     *t++ = FLOATFUN;
00279     t++;
00280     FILLFUN(t);
00281     *t++ = -SNUMBER;
00282     *t++ = infloat->_mp_prec;
00283     *t++ = -SNUMBER;
00284     *t++ = infloat->_mp_size;
00285 /*
00286     Now the exponent which is a signed long
00287 */
00288     if ( e < 0 ) {
00289         e = -e;
00290         if ( ( e > (BITSINWORD-1) ) == 0 ) {
00291             *t++ = -SNUMBER; *t++ = -e;
00292         }
00293         else {
00294             *t++ = ARGHEAD+6; *t++ = 0; FILLARG(t);
00295             *t++ = 6;
00296             *t++ = (UWORD)e;
00297             *t++ = (UWORD)(e >> BITSINWORD);
00298             *t++ = 1; *t++ = 0; *t++ = -5;
00299         }
00300     }
00301     else {
00302         if ( ( e > (BITSINWORD-1) ) == 0 ) {
00303             *t++ = -SNUMBER; *t++ = e;
00304         }
00305         else {
00306             *t++ = ARGHEAD+6; *t++ = 0; FILLARG(t);
00307             *t++ = 6;
00308             *t++ = (UWORD)e;
00309             *t++ = (UWORD)(e >> BITSINWORD);
00310             *t++ = 1; *t++ = 0; *t++ = 5;
00311         }
00312     }
00313 /*
00314     and now the limbs. Note that the number of active limbs could be less
00315     than prec+1 in which case we copy the smaller number.
00316 */
00317     nlimbs = infloat->_mp_size < 0 ? -infloat->_mp_size: infloat->_mp_size;
00318     if ( nlimbs == 0 ) {
00319         *t++ = -SNUMBER;
00320         *t++ = 0;
00321     }
00322     else if ( nlimbs == 1 && (ULONG)(*d) < ((ULONG)1)«(BITSINWORD-1) ) {
00323         *t++ = -SNUMBER;
00324         *t++ = (UWORD)(*d);
00325     }
00326     else {
00327         if ( d[nlimbs-1] < ((ULONG)1)«BITSINWORD ) {
00328             num = 2*nlimbs-1; /* number of UWORDS needed */
00329         }
00330         else {
00331             num = 2*nlimbs; /* number of UWORDS needed */
00332         }
00333         nnum = num;
00334         *t++ = 2*num+2+ARGHEAD;
00335         *t++ = 0;
00336         FILLARG(t);
00337         *t++ = 2*num+2;
00338         while ( num > 1 ) {
00339             *t++ = (UWORD)(*d); *t++ = (UWORD)((ULONG)(*d)»BITSINWORD); d++;
00340             num -= 2;
00341         }
00342         if ( num == 1 ) { *t++ = (UWORD)(*d); }
00343         *t++ = 1;
00344         for ( i = 1; i < nnum; i++ ) *t++ = 0;
00345         *t++ = 2*nnum+1; /* the sign is hidden in infloat->_mp_size */
00346     }
00347     fun[1] = t-fun;
00348     return(fun[1]);
00349 }
00350
00351 /*
00352     #] PackFloat :

```

```

00353     #[ UnpackFloat :
00354
00355     Takes the arguments of a function float_ and puts it inside an mpf_t.
00356     For notation see commentary for PutInFloat.
00357
00358     We should make a new allocation for the limbs if the precision exceeds
00359     the present precision of outfloat, or when outfloat has not been
00360     initialized yet.
00361
00362     The return value is -1 if the contents of the FLOATFUN cannot be converted.
00363     Otherwise the return value is the number of limbs.
00364 */
00365
00366 int UnpackFloat(mpf_t outfloat, WORD *fun)
00367 {
00368     UWORD *t;
00369     WORD *f;
00370     int num, i;
00371     mp_limb_t *d;
00372 /*
00373     Very first step: check whether we can use float_:
00374 */
00375     GETIDENTITY
00376     if ( AT.aux_ == 0 ) {
00377         MLOCK(ErrorMessageLock);
00378         MesPrint("Illegal attempt at using a float_ function without proper startup.");
00379         MesPrint("Please use %#StartFloat <options> first.");
00380         MUNLOCK(ErrorMessageLock);
00381         Terminate(-1);
00382     }
00383 /*
00384     Check first the number and types of the arguments
00385     There should be 4. -SNUMBER, -SNUMBER, -SNUMBER or a long number.
00386     + the limbs, either -SNUMBER or Long number in the form of a Form rational.
00387 */
00388     if ( TestFloat(fun) == 0 ) goto Incorrect;
00389     f = fun + FUNHEAD + 2;
00390     if ( ABS(f[1]) > f[-1]+1 ) goto Incorrect;
00391     if ( f[1] > outfloat->_mp_prec+1
00392         || outfloat->_mp_d == 0 ) {
00393         /*
00394             We need to make a new allocation.
00395             Unfortunately we cannot use Malloc1 because that is not
00396             recognised by gmp and hence we cannot clear with mpf_clear.
00397             Maybe we can do something about it by making our own
00398             mpf_init and mpf_clear?
00399         */
00400         if ( outfloat->_mp_d != 0 ) free(outfloat->_mp_d);
00401         outfloat->_mp_d = (mp_limb_t *)malloc((f[1]+1)*sizeof(mp_limb_t));
00402     }
00403     num = f[1];
00404     outfloat->_mp_size = num;
00405     if ( num < 0 ) { num = -num; }
00406     f += 2;
00407     if ( *f == -SNUMBER ) {
00408         outfloat->_mp_exp = (mp_exp_t)(f[1]);
00409         f += 2;
00410     }
00411     else {
00412         f += ARGHEAD+6;
00413         if ( f[-1] == -5 ) {
00414             outfloat->_mp_exp =
00415                 -(mp_exp_t)((((ULONG)(f[-4]))«BITSINWORD)+(UWORD)f[-5]);
00416         }
00417         else if ( f[-1] == 5 ) {
00418             outfloat->_mp_exp =
00419                 (mp_exp_t)((((ULONG)(f[-4]))«BITSINWORD)+(UWORD)f[-5]);
00420         }
00421     }
00422 /*
00423     And now the limbs if needed
00424 */
00425     d = outfloat->_mp_d;
00426     if ( outfloat->_mp_size == 0 ) {
00427         for ( i = 0; i < outfloat->_mp_prec; i++ ) *d++ = 0;
00428         return(0);
00429     }
00430     else if ( *f == -SNUMBER ) {
00431         *d++ = (mp_limb_t)f[1];
00432         for ( i = 0; i < outfloat->_mp_prec; i++ ) *d++ = 0;
00433         return(0);
00434     }
00435     num = (*f-ARGHEAD-2)/2; /* 2*number of limbs in the argument */
00436     t = (UWORD *) (f+ARGHEAD+1);
00437     while ( num > 1 ) {
00438         *d++ = (mp_limb_t)((((ULONG)(t[1]))«BITSINWORD)+t[0]);
00439         t += 2; num -= 2;

```



```

00440     }
00441     if ( num == 1 ) *d++ = (mp_limb_t)((UWORD)(*t));
00442     for ( i = d-outfloat->_mp_d; i <= outfloat->_mp_prec; i++ ) *d++ = 0;
00443     return(0);
00444 Incorrect:
00445     return(-1);
00446 }
00447
00448 /*
00449     #] UnpackFloat :
00450     #[ TestFloat :
00451
00452     Tells whether the function (with its arguments) is a legal float_.
00453     We assume, as in the whole float library that one limb is 2 WORDs.
00454 */
00455
00456 int TestFloat(WORD *fun)
00457 {
00458     WORD *fstop, *f, num, nnum, i;
00459     fstop = fun+fun[1];
00460     f = fun + FUNHEAD;
00461     num = 0;
00462     /*
00463     Count arguments
00464     */
00465     while ( f < fstop ) { num++; NEXTARG(f); }
00466     if ( num != 4 ) return(0);
00467     f = fun + FUNHEAD;
00468     if ( *f != -SNUMBER ) return(0);
00469     if ( f[1] < 0 ) return(0);
00470     f += 2;
00471     if ( *f != -SNUMBER ) return(0);
00472     num = ABS(f[1]); /* number of limbs */
00473     f += 2;
00474     if ( *f == -SNUMBER ) { f += 2; }
00475     else {
00476         if ( *f != ARGHEAD+6 ) return(0);
00477         if ( ABS(f[ARGHEAD+5]) != 5 ) return(0);
00478         if ( f[ARGHEAD+3] != 1 ) return(0);
00479         if ( f[ARGHEAD+4] != 0 ) return(0);
00480         f += *f;
00481     }
00482     if ( num == 0 ) return(1);
00483     if ( *f == -SNUMBER ) { if ( num != 1 ) return(0); }
00484     else {
00485         nnum = (ABS(f[*f-1])-1)/2;
00486         if ( (*f != 4*num+2+ARGHEAD) && (*f != 4*num+ARGHEAD) ) return(0);
00487         if ( ( nnum != 2*num ) && ( nnum != 2*num-1 ) ) return(0);
00488         f += ARGHEAD+nnum+1;
00489         if ( f[0] != 1 ) return(0);
00490         for ( i = 1; i < nnum; i++ ) {
00491             if ( f[i] != 0 ) return(0);
00492         }
00493     }
00494     return(1);
00495 }
00496
00497 /*
00498     #] TestFloat :
00499     #[ FormtoZ :
00500
00501     Converts a Form long integer to a GMP long integer.
00502
00503     typedef struct
00504     {
00505         int _mp_alloc;   Number of *limbs* allocated and pointed
00506                         to by the _mp_d field.
00507         int _mp_size;    abs(_mp_size) is the number of limbs the
00508                         last field points to. If _mp_size is
00509                         negative this is a negative number.
00510         mp_limb_t *_mp_d; Pointer to the limbs.
00511     } __mpz_struct;
00512     typedef __mpz_struct mpz_t[1];
00513 */
00514
00515 void FormtoZ(mpz_t z,UWORD *a,WORD na)
00516 {
00517     int nlimbs, sgn = 1;
00518     mp_limb_t *d;
00519     UWORD *b = a;
00520     WORD nb = na;
00521
00522     if ( nb == 0 ) {
00523         z->_mp_size = 0;
00524         z->_mp_d[0] = 0;
00525         return;
00526     }

```

```

00527     if ( nb < 0 ) { sgn = -1; nb = -nb; }
00528     nlimbs = (nb+1)/2;
00529     if ( mpz_cmp_si(z,0L) <= 1 ) {
00530         mpz_set_ui(z,10000000);
00531     }
00532     if ( z->_mp_alloc <= nlimbs ) {
00533         mpz_t zz;
00534         mpz_init(zz);
00535         while ( z->_mp_alloc <= nlimbs ) {
00536             mpz_mul(zz,z,z);
00537             mpz_set(z,zz);
00538         }
00539         mpz_clear(zz);
00540     }
00541     z->_mp_size = sgn*nlimbs;
00542     d = z->_mp_d;
00543     while ( nb > 1 ) {
00544         *d++ = (((mp_limb_t) (b[1])) << BITSINWORD) + (mp_limb_t) (b[0]);
00545         b += 2; nb -= 2;
00546     }
00547     if ( nb == 1 ) { *d = (mp_limb_t) (*b); }
00548 }
00549
00550 /*
00551     #] FormtoZ :
00552     #[ ZtoForm :
00553
00554     Converts a GMP long integer to a Form long integer.
00555     Mainly an exercise of going from little endian ULONGs.
00556     to big endian UWORDS
00557 */
00558
00559 void ZtoForm(UWORD *a,WORD *na,mpz_t z)
00560 {
00561     mp_limb_t *d = z->_mp_d;
00562     int nlimbs = ABS(z->_mp_size), i;
00563     UWORD j;
00564     if ( nlimbs == 0 ) { *na = 0; return; }
00565     *na = 0;
00566     for ( i = 0; i < nlimbs-1; i++ ) {
00567         *a++ = (UWORD) (*d);
00568         *a++ = (UWORD) ((*d++) << BITSINWORD);
00569         *na += 2;
00570     }
00571     *na += 1;
00572     *a++ = (UWORD) (*d);
00573     j = (UWORD) ((*d) << BITSINWORD);
00574     if ( j != 0 ) { *a = j; *na += 1; }
00575     if ( z->_mp_size < 0 ) *na = -*na;
00576 }
00577
00578 /*
00579     #] ZtoForm :
00580     #[ FloatToInteger :
00581
00582     Converts a GMP float to a long Form integer.
00583 */
00584
00585 long FloatToInteger(UWORD *out, mpf_t floatin, long *bitsused)
00586 {
00587     mpz_t z;
00588     WORD nout, nx;
00589     UWORD x;
00590     mpz_init(z);
00591     mpz_set_f(z,floatin);
00592     ZtoForm(out,&nout,z);
00593     mpz_clear(z);
00594     x = out[0]; nx = 0;
00595     while ( x ) { nx++; x >>= 1; }
00596     *bitsused = (nout-1)*BITSINWORD + nx;
00597     return(nout);
00598 }
00599
00600 /*
00601     #] FloatToInteger :
00602     #[ IntegerToFloat :
00603
00604     Converts a Form long integer to a gmp float of default size.
00605     We assume that sizeof(unsigned long int) = 2*sizeof(UWORD).
00606 */
00607
00608 void IntegerToFloat(mpf_t result, UWORD *formlong, int longsize)
00609 {
00610     mpz_t z;
00611     mpz_init(z);
00612     FormtoZ(z,formlong,longsize);
00613     mpf_set_z(result,z);

```

```

00614     mpz_clear(z);
00615 }
00616
00617 /*
00618     #] IntegerToFloat :
00619     #[ RatToFloat :
00620
00621     Converts a Form rational to a gmp float of default size.
00622 */
00623
00624 void RatToFloat(mpf_t result, UWORD *formrat, int ratsize)
00625 {
00626     GETIDENTITY
00627     int nnum, nden;
00628     UWORD *num, *den;
00629     int sgn = 0;
00630     if ( ratsize < 0 ) { ratsize = -ratsize; sgn = 1; }
00631     nnum = nden = (ratsize-1)/2;
00632     num = formrat; den = formrat+nnum;
00633     while ( num[nnum-1] == 0 ) { nnum--; }
00634     while ( den[nden-1] == 0 ) { nden--; }
00635     IntegerToFloat(aux2,num,nnum);
00636     IntegerToFloat(aux3,den,nden);
00637     mpf_div(result,aux2,aux3);
00638     if ( sgn > 0 ) mpf_neg(result,result);
00639 }
00640
00641 /*
00642     #] RatToFloat :
00643     #[ FloatFunToRat :
00644 */
00645
00646 WORD FloatFunToRat(PHEAD UWORD *ratout,WORD *in)
00647 {
00648     WORD oldin = in[0], nratout;
00649     in[0] = FLOATFUN;
00650     UnpackFloat(aux4,in);
00651     FloatToRat(ratout,&nratout,aux4);
00652     in[0] = oldin;
00653     return(nratout);
00654 }
00655
00656 /*
00657     #] FloatFunToRat :
00658     #[ FloatToRat :
00659
00660     Converts a gmp float to a Form rational.
00661     The algorithm we use is 'repeated fractions' of the style
00662         n0+1/(n1+1/(n2+1/(n3+....)))
00663     The moment we get an n that is very large we should have the proper fraction
00664     Main problem: what if the sequence keeps going?
00665         (there are always rounding errors)
00666     We need a cutoff with an error condition.
00667     Basically the product n0*n1*n2*... is an underlimit on the number of
00668     digits in the answer. We can put a limit on this.
00669     A return value of zero means that everything went fine.
00670 */
00671
00672 int FloatToRat(UWORD *ratout, WORD *nratout, mpf_t floatin)
00673 {
00674     GETIDENTITY
00675     WORD *out = AT.WorkPointer;
00676     WORD s; /* the sign. */
00677     UWORD *a, *b, *c, *d, *mc;
00678     WORD na, nb, nc, nd, i;
00679     int nout;
00680     LONG oldpWorkPointer = AT.pWorkPointer;
00681     long bitsused = 0, totalbitsused = 0, totalbits = AC.DefaultPrecision;
00682     int retval = 0, startnul;
00683     WantAddPointers(AC.DefaultPrecision); /* Horrible overkill */
00684     AT.pWorkSpace[AT.pWorkPointer++] = out;
00685     a = NumberMalloc("FloatToRat");
00686     b = NumberMalloc("FloatToRat");
00687
00688     s = mpf_sgn(floatin);
00689     if ( s < 0 ) mpf_neg(floatin,floatin);
00690
00691     Form_mpf_empty(aux1);
00692     Form_mpf_empty(aux2);
00693     Form_mpf_empty(aux3);
00694
00695     mpf_trunc(aux1,floatin); /* This should now be an integer */
00696     mpf_sub(aux2,floatin,aux1); /* and this >= 0 */
00697     if ( mpf_sgn(aux1) == 0 ) { *out++ = 0; startnul = 1; }
00698     else {
00699         nout = FloatToInteger((UWORD *)out,aux1,&totalbitsused);
00700         out += nout;

```

```

00701     startnul = 0;
00702 }
00703 AT.pWorkSpace[AT.pWorkPointer++] = out;
00704 if ( mpf_sgn(aux2) ) {
00705     for(;;) {
00706         mpf_ui_div(aux3,1,aux2);
00707         mpf_trunc(aux1,aux3);
00708         mpf_sub(aux2,aux3,aux1);
00709         if ( mpf_sgn(aux1) == 0 ) { *out++ = 0; }
00710         else {
00711             nout = FloatToInteger((UWORD *)out,aux1,&bitsused);
00712             out += nout;
00713             totalbitsused += bitsused;
00714         }
00715         if ( bitsused > (totalbits-totalbitsused)/2 ) { break; }
00716         if ( mpf_sgn(aux2) == 0 ) {
00717             /*if ( startnul == 1 )*/ AT.pWorkSpace[AT.pWorkPointer++] = out;
00718             break;
00719         }
00720         AT.pWorkSpace[AT.pWorkPointer++] = out;
00721     }
00722 /*
00723     At this point we have the function with the repeated fraction.
00724     Now we should work out the fraction. Form code would be:
00725     repeat id dum_(?a,x1?,x2?) = dum_(?a,x1+1/x2);
00726     id dum_(x?) = x;
00727     We have to work backwards. This is why we made a list of pointers
00728     in AT.pWorkSpace
00729
00730     Now we need the long integers a and b.
00731     Note that by construction there should never be a GCD!
00732 */
00733     out = (WORD *) (AT.pWorkSpace[--AT.pWorkPointer]);
00734     na = 1; a[0] = 1;
00735     c = (UWORD *) (AT.pWorkSpace[--AT.pWorkPointer]);
00736     nc = nb = ((UWORD *)out)-c;
00737     if ( nc > 10 ) {
00738         mc = c;
00739         c = (UWORD *) (AT.pWorkSpace[--AT.pWorkPointer]);
00740         nc = nb = ((UWORD *)mc)-c;
00741     }
00742     for ( i = 0; i < nb; i++ ) b[i] = c[i];
00743     mc = c = NumberMalloc("FloatToRat");
00744     while ( AT.pWorkPointer > oldpWorkPointer ) {
00745         d = (UWORD *) (AT.pWorkSpace[--AT.pWorkPointer]);
00746         nd = (UWORD *) (AT.pWorkSpace[AT.pWorkPointer+1])-d; /* This is the x1 in the formula */
00747         if ( nd == 1 && *d == 0 ) break;
00748         c = a; a = b; b = c; nc = na; na = nb; nb = nc; /* 1/x2 */
00749         MulLong(d,nd,a,na,mc,&nc);
00750         AddLong(mc,nc,b,nb,b,&nb);
00751     }
00752     NumberFree(mc,"FloatToRat");
00753     if ( startnul == 0 ) {
00754         c = a; a = b; b = c; nc = na; na = nb; nb = nc;
00755     }
00756 }
00757 else {
00758     c = (UWORD *) (AT.pWorkSpace[oldpWorkPointer]);
00759     na = (UWORD *)out - c;
00760     for ( i = 0; i < na; i++ ) a[i] = c[i];
00761     b[0] = 1; nb = 1;
00762 }
00763 /*
00764     Now we have the fraction in a/b. Create the rational and add the sign.
00765 */
00766     d = ratout;
00767     if ( na == nb ) {
00768         *nratout = (2*na+1)*s;
00769         for ( i = 0; i < na; i++ ) *d++ = a[i];
00770         for ( i = 0; i < nb; i++ ) *d++ = b[i];
00771     }
00772     else if ( na < nb ) {
00773         *nratout = (2*nb+1)*s;
00774         for ( i = 0; i < na; i++ ) *d++ = a[i];
00775         for ( i < nb; i++ ) *d++ = 0;
00776         for ( i = 0; i < nb; i++ ) *d++ = b[i];
00777     }
00778     else {
00779         *nratout = (2*na+1)*s;
00780         for ( i = 0; i < na; i++ ) *d++ = a[i];
00781         for ( i = 0; i < nb; i++ ) *d++ = b[i];
00782         for ( i < na; i++ ) *d++ = 0;
00783     }
00784     *d = (UWORD) (*nratout);
00785     NumberFree(b,"FloatToRat");
00786     NumberFree(a,"FloatToRat");
00787     AT.pWorkPointer = oldpWorkPointer;

```

```

00788 /*
00789 Just for check we go back to float
00790 */
00791 if ( s < 0 ) mpf_neg(floatin,floatin);
00792 /*
00793 {
00794     WORD n = *ratout;
00795     RatToFloat(aux1, ratout, n);
00796     mpf_sub(aux2, floatin, aux1);
00797     gmp_printf("Diff is %.*Fe\n", 40, aux2);
00798 }
00799 */
00800 return(retval);
00801 }
00802
00803 /*
00804 #] FloatToRat :
00805 #[ ZeroTable :
00806 */
00807 void ZeroTable(mpf_t *tab, int N)
00808 {
00809     int i, j;
00810     for ( i = 0; i < N; i++ ) {
00811         for ( j = 0; j < tab[i]->_mp_prec; j++ ) tab[i]->_mp_d[j] = 0;
00812     }
00813 }
00814 }
00815
00816 /*
00817 #] ZeroTable :
00818 #[ ReadFloat :
00819
00820 Used by the compiler. It reads a floating point number and
00821 outputs it at AT.WorkPointer as if it were a float_ function
00822 with its arguments.
00823 The call enters with s[-1] == TFLOAT.
00824 */
00825
00826 SBYTE *ReadFloat(SBYTE *s)
00827 {
00828     GETIDENTITY
00829     SBYTE *ss, c;
00830     ss = s;
00831     while ( *ss > 0 ) ss++;
00832     c = *ss; *ss = 0;
00833     gmp_sscanf((char *)s, "%Ff", aux1);
00834     if ( AT.WorkPointer > AT.WorkTop ) {
00835         MLOCK(ErrorMessageLock);
00836         MesWork();
00837         MUNLOCK(ErrorMessageLock);
00838         Terminate(-1);
00839     }
00840     PackFloat(AT.WorkPointer, aux1);
00841     *ss = c;
00842     return(ss);
00843 }
00844
00845 /*
00846 #] ReadFloat :
00847 #[ CheckFloat :
00848
00849 For the tokenizer. Tests whether the input string can be a float.
00850 */
00851
00852 UBYTE *CheckFloat(UBYTE *ss, int *spec)
00853 {
00854     GETIDENTITY
00855     UBYTE *s = ss;
00856     int gotdot = 0;
00857     /* A single dot is not a valid float */
00858     if ( *s == '.' && FG.cTable[s[-1]] != 1 && FG.cTable[s[1]] != 1 ) return(ss);
00859     while ( FG.cTable[s[-1]] == 1 ) s--;
00860     /* This cannot be a valid float */
00861     if ( *s != '.' && FG.cTable[*s] != 1 ) return(ss);
00862     *spec = 0;
00863     while ( FG.cTable[*s] == 1 ) s++;
00864     if ( *s == '.' ) {
00865         gotdot = 1;
00866         s++;
00867         while ( FG.cTable[*s] == 1 ) s++;
00868     }
00869     /*
00870 Now we have had the mantissa part, which may be zero.
00871 Check for an exponent.
00872 */
00873     if ( *s == 'e' || *s == 'E' ) {
00874         s++;

```

```

00875         if ( *s == '-' || *s == '+' ) s++;
00876         if ( FG.cTable[*s] == 1 ) {
00877             s++;
00878             while ( FG.cTable[*s] == 1 ) s++;
00879         }
00880         else { return(ss); }
00881     }
00882     else if ( gotdot == 0 ) return(ss); /* No radix point and no exponent */
00883     if ( AT.aux_ == 0 ) { /* no floating point system */
00884         *spec = -1;
00885         return(s);
00886     }
00887     return(s);
00888 }
00889
00890 /*
00891     #] CheckFloat :
00892     #] Low Level :
00893     #[ Float Routines :
00894         #[ SetFloatPrecision :
00895
00896         Sets the default precision (in bits) of the floats and allocate
00897         buffer space for an output string.
00898         The buffer is used by PrintFloat (decimal output) and Strictrounding
00899         (binary or decimal output), so it must accommodate the larger space
00900         requirement:
00901         exponent: up to 12 chars.
00902         mantissa: prec + a little bit extra
00903 */
00904
00905 int SetFloatPrecision(WORD prec)
00906 {
00907     if ( prec <= 0 ) {
00908         MesPrint("&Illegal value for number of bits for floating point constants: %d",prec);
00909         return(-1);
00910     }
00911     else {
00912         AC.DefaultPrecision = prec;
00913         if ( AO.floatspace != 0 ) M_free(AO.floatspace,"floatspace");
00914         AO.floatsize = (prec+20)*sizeof(char);
00915         AO.floatspace = (UBYTE *)Malloc1(AO.floatsize,"floatspace");
00916         mpf_set_default_prec(prec);
00917         return(0);
00918     }
00919 }
00920
00921 /*
00922     #] SetFloatPrecision :
00923     #[ PrintFloat :
00924
00925     Print the float_ function with its arguments as a floating point
00926     number with numdigits digits.
00927     If however we use rawfloat as a format option it prints it as a
00928     regular Form function and it should never come here because the
00929     print routines in sch.c should intercept it.
00930     The buffer AO.floatspace is allocated when the precision of the
00931     floats is set in the routine SetFloatPrecision.
00932 */
00933
00934 int PrintFloat(WORD *fun,int numdigits)
00935 {
00936     GETIDENTITY
00937     UBYTE *s1, *s2;
00938     int n = 0;
00939     int prec = (AC.DefaultPrecision-AC.MaxWeight-1)*log10(2.0);
00940     if ( numdigits > prec || numdigits == 0 ) {
00941         numdigits = prec;
00942     }
00943     /*
00944     GMP's gmp_snprintf always prints a non-zero number before the decimal point, so we
00945     ask for one digit less.
00946 */
00947     if ( UnpackFloat(aux4,fun) == 0 )
00948         n = gmp_snprintf((char *) (AO.floatspace),AO.floatsize,"%.*Fe",numdigits-1,aux4);
00949     if ( n > 0 ) {
00950         int n1, n2 = n;
00951         s1 = AO.floatspace+n;
00952         while ( s1 > AO.floatspace && s1[-1] != 'e'
00953             && s1[-1] != 'E' && s1[-1] != '.' ) { s1--; n2--; }
00954         if ( s1 > AO.floatspace && s1[-1] != '.' ) {
00955             s1--; n2--;
00956             s2 = s1; n1 = n2;
00957             while ( s1[-1] == '0' ) { s1--; n1--; }
00958             if ( s1[-1] == '.' ) { s1++; n1++; }
00959             n -= (n2-n1);
00960             while ( n1 < n ) { *s1++ = *s2++; n1++; }
00961             *s1 = 0;

```

```

00962     }
00963     if ( AC.OutputMode == FORTRANMODE ) {
00964         s1 = AO.floatspace+n;
00965         while ( s1 > AO.floatspace && *s1 != 'e' && *s1 != 'E' ) {
00966             s1--;
00967         }
00968         if ( ( AO.DoubleFlag & 2 ) == 2 ) { *s1 = 'Q'; } /* Quadruple precision fortran */
00969         else if ( ( AO.DoubleFlag & 1 ) == 1 ) { *s1 = 'D'; } /* Double precision fortran */
00970         else { *s1 = 'E'; } /* Single precision fortran */
00971     }
00972     if ( AC.OutputMode == MATHEMATICAMODE ) {
00973         s1 = AO.floatspace+n;
00974         s2 = s1+2; *s2-- = 0;
00975         while ( s1 > AO.floatspace && *s1 != 'e' && *s1 != 'E' ) {
00976             *s2-- = *s1--;
00977         }
00978         *s2-- = '^'; *s2 = '*'; /* Replace 'e' by '^*' */
00979         n++;
00980     }
00981 }
00982 return(n);
00983 }
00984
00985 /*
00986     #] PrintFloat :
00987     #[ AddFloats :
00988 */
00989
00990 int AddFloats(PHEAD WORD *fun3, WORD *fun1, WORD *fun2)
00991 {
00992     int retval = 0;
00993     if ( UnpackFloat(aux1,fun1) == 0 && UnpackFloat(aux2,fun2) == 0 ) {
00994         mpf_add(aux1,aux1,aux2);
00995         PackFloat(fun3,aux1);
00996     }
00997     else { retval = -1; }
00998     return(retval);
00999 }
01000
01001 /*
01002     #] AddFloats :
01003     #[ MulFloats :
01004 */
01005
01006 int MulFloats(PHEAD WORD *fun3, WORD *fun1, WORD *fun2)
01007 {
01008     int retval = 0;
01009     if ( UnpackFloat(aux1,fun1) == 0 && UnpackFloat(aux2,fun2) == 0 ) {
01010         mpf_mul(aux1,aux1,aux2);
01011         PackFloat(fun3,aux1);
01012     }
01013     else { retval = -1; }
01014     return(retval);
01015 }
01016
01017 /*
01018     #] MulFloats :
01019     #[ DivFloats :
01020 */
01021
01022 int DivFloats(PHEAD WORD *fun3, WORD *fun1, WORD *fun2)
01023 {
01024     int retval = 0;
01025     if ( UnpackFloat(aux1,fun1) == 0 && UnpackFloat(aux2,fun2) == 0 ) {
01026         mpf_div(aux1,aux1,aux2);
01027         PackFloat(fun3,aux1);
01028     }
01029     else { retval = -1; }
01030     return(retval);
01031 }
01032
01033 /*
01034     #] DivFloats :
01035     #[ AddRatToFloat :
01036
01037     Note: this can be optimized, because often the rat is rather simple.
01038 */
01039
01040 int AddRatToFloat(PHEAD WORD *outfun, WORD *infun, UWORD *formrat, WORD nrat)
01041 {
01042     int retval = 0;
01043     if ( UnpackFloat(aux2,infun) == 0 ) {
01044         RatToFloat(aux1,formrat,nrat);
01045         mpf_add(aux2,aux2,aux1);
01046         PackFloat(outfun,aux2);
01047     }
01048     else retval = -1;

```

```

01049     return(retval);
01050 }
01051
01052 /*
01053     #] AddRatToFloat :
01054     #[ MulRatToFloat :
01055
01056     Note: this can be optimized, because often the rat is rather simple.
01057 */
01058
01059 int MulRatToFloat(PHEAD WORD *outfun, WORD *infun, UWORD *formrat, WORD nrat)
01060 {
01061     int retval = 0;
01062     if ( UnpackFloat(aux2,infun) == 0 ) {
01063         RatToFloat(aux1,formrat,nrat);
01064         mpf_mul(aux2,aux2,aux1);
01065         PackFloat(outfun,aux2);
01066     }
01067     else retval = -1;
01068     return(retval);
01069 }
01070
01071 /*
01072     #] MulRatToFloat :
01073     #[ SetupMZVTables :
01074 */
01075
01076 void SetupMZVTables(void)
01077 {
01078     /*
01079     Sets up a table of N+1 mpf_t floats with variable precision.
01080     It assumes that each next element needs one bit less.
01081     Initializes all of them.
01082     We take some extra space, to have one limb overflow everywhere.
01083     Because the depth of the MZV's can get close to the weight
01084     and each deeper sum goes one higher, we make the tables size a bit bigger.
01085     This may not be needed if we fiddle with the sum boundaries.
01086     */
01087     #ifdef WITHPTHREADS
01088         int i, Nw, id, totnum;
01089         size_t N;
01090         mpf_t *a;
01091         Nw = AC.DefaultPrecision;
01092         N = (size_t)Nw;
01093         totnum = AM.totalnumberofthreads;
01094         for ( id = 0; id < totnum; id++ ) {
01095             AB[id]->T.mpf_tab1 = (void *)Malloc1((N+2)*sizeof(mpf_t),"mpftab1");
01096             a = (mpf_t *)AB[id]->T.mpf_tab1;
01097             for ( i = 0; i <= Nw; i++ ) {
01098                 /*
01099                 As explained in the comment above, we could make this variable precision
01100                 using mpf_init2.
01101                 */
01102                 mpf_init(a[i]);
01103             }
01104             AB[id]->T.mpf_tab2 = (void *)Malloc1((N+2)*sizeof(mpf_t),"mpftab2");
01105             a = (mpf_t *)AB[id]->T.mpf_tab2;
01106             for ( i = 0; i <= Nw; i++ ) {
01107                 mpf_init(a[i]);
01108             }
01109         }
01110     #else
01111         int i, Nw;
01112         size_t N;
01113         Nw = AC.DefaultPrecision;
01114         N = (size_t)Nw;
01115         AT.mpf_tab1 = (void *)Malloc1((N+2)*sizeof(mpf_t),"mpftab1");
01116         for ( i = 0; i <= Nw; i++ ) {
01117             /*
01118             As explained in the comment above, we could make this variable precision
01119             using mpf_init2.
01120             */
01121             mpf_init(mpftab1[i]);
01122         }
01123         AT.mpf_tab2 = (void *)Malloc1((N+2)*sizeof(mpf_t),"mpftab2");
01124         for ( i = 0; i <= Nw; i++ ) {
01125             mpf_init(mpftab2[i]);
01126         }
01127     #endif
01128     AS.delta_1 = (void *)Malloc1(sizeof(mpf_t),"delta1");
01129     mpf_init(mpfdelta1);
01130     SimpleDelta(mpfdelta1,1); /* this can speed up things. delta1 = ln(2) */
01131 }
01132
01133 /*
01134     #] SetupMZVTables :
01135     #[ SetupMPFTables :

```



```

01136
01137     Allocates the aux variables
01138 */
01139
01140 void SetupMPFTables(void)
01141 {
01142     #ifdef WITHPTHREADS
01143         int id, totnum;
01144         mpf_t *a;
01145     #ifdef WITHSORTBOTS
01146         totnum = Max(2*AM.totalnumberofthreads-3,AM.totalnumberofthreads);
01147     #endif
01148     for ( id = 0; id < totnum; id++ ) {
01149         /*
01150             We work here with a[0] etc because the aux1 etc contain B which
01151             in the current routine would be AB[0] only
01152         */
01153         AB[id]->T.aux_ = (void *)Malloc1(sizeof(mpf_t)*8,"AB[id]->T.aux_");
01154         a = (mpf_t *)AB[id]->T.aux_;
01155         mpf_inits(a[0],a[1],a[2],a[3],a[4],a[5],a[6],a[7],(mpf_ptr)0);
01156         if ( AB[id]->T.indi1 ) M_free(AB[id]->T.indi1,"indi1");
01157         AB[id]->T.indi1 = (WORD *)Malloc1(sizeof(WORD)*AC.MaxWeight*2,"indi1");
01158         AB[id]->T.indi2 = AB[id]->T.indi1 + AC.MaxWeight;
01159     }
01160     #else
01161         AT.aux_ = (void *)Malloc1(sizeof(mpf_t)*8,"AT.aux_");
01162         mpf_inits(aux1,aux2,aux3,aux4,aux5,auxjm,auxjjm,auxsum,(mpf_ptr)0);
01163         if ( AT.indi1 ) M_free(AT.indi1,"indi1");
01164         AT.indi1 = (WORD *)Malloc1(sizeof(WORD)*AC.MaxWeight*2,"indi1");
01165         AT.indi2 = AT.indi1 + AC.MaxWeight;
01166     #endif
01167 }
01168
01169 /*
01170     #] SetupMPFTables :
01171     #[ ClearMZVTables :
01172 */
01173
01174 void ClearMZVTables(void)
01175 {
01176     #ifdef WITHPTHREADS
01177         int i, id, totnum;
01178         mpf_t *a;
01179         totnum = AM.totalnumberofthreads;
01180         for ( id = 0; id < totnum; id++ ) {
01181             if ( AB[id]->T.mpf_tab1 ) {
01182                 /*
01183                     We work here with a[0] etc because the aux1, mpftab1 etc contain B
01184                     which in the current routine would be AB[0] only
01185                 */
01186                 a = (mpf_t *)AB[id]->T.mpf_tab1;
01187                 for ( i = 0; i <= AC.DefaultPrecision; i++ ) {
01188                     mpf_clear(a[i]);
01189                 }
01190                 M_free(AB[id]->T.mpf_tab1,"mpftab1");
01191                 AB[id]->T.mpf_tab1 = 0;
01192             }
01193             if ( AB[id]->T.mpf_tab2 ) {
01194                 a = (mpf_t *)AB[id]->T.mpf_tab2;
01195                 for ( i = 0; i <= AC.DefaultPrecision; i++ ) {
01196                     mpf_clear(a[i]);
01197                 }
01198                 M_free(AB[id]->T.mpf_tab2,"mpftab2");
01199                 AB[id]->T.mpf_tab2 = 0;
01200             }
01201         }
01202     #ifdef WITHSORTBOTS
01203         totnum = Max(2*AM.totalnumberofthreads-3,AM.totalnumberofthreads);
01204     #endif
01205     for ( id = 0; id < totnum; id++ ) {
01206         if ( AB[id]->T.aux_ ) {
01207             a = (mpf_t *)AB[id]->T.aux_;
01208             mpf_clears(a[0],a[1],a[2],a[3],a[4],a[5],a[6],a[7],(mpf_ptr)0);
01209             M_free(AB[id]->T.aux_,"AB[id]->T.aux_");
01210             AB[id]->T.aux_ = 0;
01211         }
01212         if ( AB[id]->T.indi1 ) { M_free(AB[id]->T.indi1,"indi1"); AB[id]->T.indi1 = 0; }
01213     }
01214     #else
01215         int i;
01216         if ( AT.mpf_tab1 ) {
01217             for ( i = 0; i <= AC.DefaultPrecision; i++ ) {
01218                 mpf_clear(mpftab1[i]);
01219             }
01220             M_free(AT.mpf_tab1,"mpftab1");
01221             AT.mpf_tab1 = 0;
01222         }

```

```

01223     if ( AT.mpf_tab2 ) {
01224         for ( i = 0; i <= AC.DefaultPrecision; i++ ) {
01225             mpf_clear(mpftab2[i]);
01226         }
01227         M_free(AT.mpf_tab2, "mpftab2");
01228         AT.mpf_tab2 = 0;
01229     }
01230     if ( AT.aux_ ) {
01231         mpf_clears(aux1,aux2,aux3,aux4,aux5,auxjm,auxjjm,auxsum, (mpf_ptr)0);
01232         M_free(AT.aux_, "AT.aux_");
01233         AT.aux_ = 0;
01234     }
01235     if ( AT.indil ) { M_free(AT.indil, "indil"); AT.indil = 0; }
01236 #endif
01237     if ( AO.floatspace ) { M_free(AO.floatspace, "floatspace"); AO.floatspace = 0;
01238         AO.floatsize = 0; }
01239     if ( AS.delta_1 ) {
01240         mpf_clear(mpfdelta1);
01241         M_free(AS.delta_1, "delta1");
01242         AS.delta_1 = 0;
01243     }
01244 }
01245
01246 /*
01247     #] ClearMZVTables :
01248     #[ CoToFloat :
01249 */
01250
01251 int CoToFloat (UBYTE *s)
01252 {
01253     GETIDENTITY
01254     if ( AT.aux_ == 0 ) {
01255         MesPrint("&Illegal attempt to convert to float_ without activating floating point numbers.");
01256         MesPrint("&Forgotten %#startfloat instruction?");
01257         return(1);
01258     }
01259     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
01260     if ( *s ) {
01261         MesPrint("&Illegal argument(s) in ToFloat statement: '%s'",s);
01262         return(1);
01263     }
01264     Add2Com(TYPETOFLOAT);
01265     return(0);
01266 }
01267
01268 /*
01269     #] CoToFloat :
01270     #[ CoToRat :
01271 */
01272
01273 int CoToRat (UBYTE *s)
01274 {
01275     GETIDENTITY
01276     if ( AT.aux_ == 0 ) {
01277         MesPrint("&Illegal attempt to convert from float_ without activating floating point
01278 numbers.");
01279         MesPrint("&Forgotten %#startfloat instruction?");
01280         return(1);
01281     }
01282     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
01283     if ( *s ) {
01284         MesPrint("&Illegal argument(s) in ToRat statement: '%s'",s);
01285         return(1);
01286     }
01287     Add2Com(TYPETORAT);
01288     return(0);
01289 }
01290
01291 /*
01292     #] CoToRat :
01293     #[ CoStrictRounding :
01294
01295     Syntax: StrictRounding [precision][base]
01296     - precision: number of digits to round to (optional)
01297     - base: 'd' for decimal (base 10) or 'b' for binary (base 2)
01298
01299     If no arguments are provided, uses default precision with binary base.
01300 */
01301 int CoStrictRounding (UBYTE *s)
01302 {
01303     GETIDENTITY
01304     WORD x;
01305     int base;
01306     if ( AT.aux_ == 0 ) {
01307         MesPrint("&Illegal attempt for strict rounding without activating floating point numbers.");
01308         MesPrint("&Forgotten %#startfloat instruction?");
01309         return(1);

```

```

01309     }
01310     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
01311     if ( *s == 0 ) {
01312         /* No subkey, which means round to default precision */
01313         x = AC.DefaultPrecision - AC.MaxWeight - 1;
01314         base = 2;
01315     }
01316     else if ( FG.cTable[*s] == 1 ) { /* number */
01317         ParseNumber(x,s)
01318         if ( tolower(*s) == 'd' ) { base = 10; s++; } /* decimal base */
01319         else if ( tolower(*s) == 'b' ) { base = 2; s++; } /* binary base */
01320         else goto IllPar; /* invalid base specification */
01321     }
01322     else {
01323         goto IllPar;
01324     }
01325     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
01326
01327     /* Check for invalid arguments */
01328     if ( *s ) {
01329 IllPar:
01330         MesPrint("%Illegal argument(s) in StrictRounding statement: '%s'",s);
01331         return(1);
01332     }
01333     Add4Com(TYPESTRICTROUNDING,x,base);
01334     return(0);
01335 }
01336 /*
01337     #] CoStrictRounding :
01338     #[ CoChop :
01339
01340     LHS notation of the chop statement:
01341         TYPECHOP, length, FLOATFUN, ...
01342     where FLOATFUN, ... represents the threshold of the chop statement in
01343     the notation of a float_ function with its arguments.
01344 */
01345
01346 int CoChop(UBYTE *s)
01347 {
01348     GETIDENTITY
01349     UBYTE *ss, c;
01350     WORD *w, *OldWork;
01351     int spec, pow = 1;
01352     unsigned long x;
01353     if ( AT.aux_ == 0 ) {
01354         MesPrint("%Illegal attempt to chop a float_ without activating floating point numbers.");
01355         MesPrint("%Forgotten %#startfloat instruction?");
01356         return(1);
01357     }
01358     if ( *s == 0 ) {
01359         MesPrint("%Chop needs a number (float, rational or power) as an argument.");
01360         return(1);
01361     }
01362     /* Create TYPECHOP header */
01363     w = OldWork = AT.WorkPointer;
01364     *w++ = TYPECHOP;
01365     w++;
01366     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
01367
01368     /*
01369     The argument of chop can be
01370     1: a floating-point number
01371     2: an integer, rational number or power
01372     */
01373     if ( FG.cTable[*s] == 1 || *s == '.' ) {
01374         /* 1: Attempt to parse as floating-point number */
01375         ss = CheckFloat(s, &spec);
01376         if ( ss > s ) {
01377             /* CheckFloat found a valid float */
01378             AT.WorkPointer = w;
01379             /*
01380             Reads the floating point number and outputs it at AT.WorkPointer as if it were a
01381             float_
01382             function with its arguments.
01383             */
01384             ReadFloat((SBYTE *)s);
01385             s = ss;
01386             w += w[1];
01387         }
01388     }
01389     else {
01390         /* 2: CheckFloat didn't find a float, we now try for rationals and powers */
01391         /* Parse the integer part (numerator for rationals) */
01392         if ( FG.cTable[*s] == 1 ) {
01393             ParseNumber(x,s)
01394             mpf_set_ui(aux1,x);

```

```

01395     }
01396     while ( *s == ' ' || *s == '\t' ) s++;
01397     /* Check for rational number or power */
01398     if ( *s == '/' || *s == '^' ) {
01399         c = *s; s++;
01400         while ( *s == ' ' || *s == '\t' ) s++;
01401         if ( *s == '-' ) { s++; pow = -1; } /* negative power */
01402         /* Parse the denominator or power */
01403         if ( FG.cTable[*s] == 1 ) {
01404             ParseNumber(x,s)
01405             if ( c == '/' ) { /* rational */
01406                 if ( x == 0 ) {
01407                     MesPrint("&Division by zero in chop statement.");
01408                     return(1);
01409                 }
01410                 /* Perform the division */
01411                 mpf_div_ui(aux1, aux1,x);
01412             }
01413             else { /* Power */
01414                 mpf_pow_ui(aux1,aux1,x);
01415                 if ( pow == -1 ) {
01416                     mpf_ui_div(aux1,(unsigned long) 1, aux1);
01417                 }
01418             }
01419         }
01420     }
01421     /* Put aux1 in the notation of a float_ function */
01422     PackFloat(w, aux1);
01423     w += w[1];
01424 }
01425 }
01426 if ( *s ) {
01427     MesPrint("&Illegal argument(s) in Chop statement: '%s'.",s);
01428     return(1);
01429 }
01430 AT.WorkPointer = OldWork;
01431 AT.WorkPointer[1] = w - AT.WorkPointer; /* Set total length */
01432 AddNtoL(AT.WorkPointer[1],AT.WorkPointer); /* Add the LHS to the compiler buffer */
01433 return(0);
01434 }
01435
01436 /*
01437 #] CoChop :
01438 #[ ToFloat :
01439
01440     Converts the coefficient to floating point if it is still a rat.
01441 */
01442
01443 int ToFloat(PHEAD WORD *term, WORD level)
01444 {
01445     WORD *t, *tstop, nsize, nsign = 3;
01446     t = term+*term;
01447     nsize = ABS(t[-1]);
01448     tstop = t - nsize;
01449     if ( t[-1] < 0 ) { nsign = -nsign; }
01450     if ( nsize == 3 && t[-2] == 1 && t[-3] == 1 ) { /* Could be float */
01451         t = term + 1;
01452         while ( t < tstop ) {
01453             if ( ( *t == FLOATFUN ) && ( t+t[1] == tstop ) ) {
01454                 return(Generator(BHEAD term,level));
01455             }
01456             t += t[1];
01457         }
01458     }
01459     RatToFloat(aux4, (UWORD *)tstop,nsize);
01460     PackFloat(tstop,aux4);
01461     tstop += tstop[1];
01462     *tstop++ = 1; *tstop++ = 1; *tstop++ = nsign;
01463     *term = tstop - term;
01464     AT.WorkPointer = tstop;
01465     return(Generator(BHEAD term,level));
01466 }
01467
01468 /*
01469 #] ToFloat :
01470 #[ ToRat :
01471
01472     Tries to convert the coefficient to rational if it is still a float.
01473 */
01474
01475 int ToRat(PHEAD WORD *term, WORD level)
01476 {
01477     WORD *tstop, *t, nsize, nsign, ncoef;
01478     /*
01479     1: find the float which should be at the end.
01480     */
01481     tstop = term + *term; nsize = ABS(tstop[-1]);

```

```

01482     nsign = tstop[-1] < 0 ? -1: 1; tstop -= nsize;
01483     t = term+1;
01484     while ( t < tstop ) {
01485         if ( *t == FLOATFUN && t + t[1] == tstop && TestFloat(t) &&
01486             nsize == 3 && tstop[0] == 1 && tstop[1] == 1 ) break;
01487         t += t[1];
01488     }
01489     if ( t < tstop ) {
01490 /*
01491         Now t points at the float_ function and everything is correct.
01492         The result can go straight over the float_ function.
01493 */
01494         UnpackFloat(aux4,t);
01495         if ( FloatToRat((UWORD *)t,&ncoef,aux4) == 0 ) {
01496             t += ABS(ncoef);
01497             t[-1] = ncoef*nsign;
01498             *term = t - term;
01499         }
01500     }
01501     return(Generator(BHEAD term,level));
01502 }
01503
01504 /*
01505     #] ToRat :
01506     #[ StrictRounding :
01507
01508     Rounds floating point numbers to a specified precision
01509     in a given base (decimal or binary).
01510 */
01511 int StrictRounding(PHEAD WORD *term, WORD level, WORD prec, WORD base) {
01512     WORD *t,*tstop;
01513     int sign,size,maxprec = AC.DefaultPrecision-AC.MaxWeight-1;
01514     /* maxprec is in bits */
01515     if ( base == 2 && prec > maxprec ) {
01516         prec = maxprec;
01517     }
01518     if ( base == 10 && prec > (int)(maxprec*log10(2.0)) ) {
01519         prec = maxprec*log10(2.0);
01520     }
01521     /* Find the float which should be at the end. */
01522     tstop = term + *term; size = ABS(tstop[-1]);
01523     sign = tstop[-1] < 0 ? -1: 1; tstop -= size;
01524     t = term+1;
01525     while ( t < tstop ) {
01526         if ( *t == FLOATFUN && t + t[1] == tstop && TestFloat(t) &&
01527             size == 3 && tstop[0] == 1 && tstop[1] == 1 ) {
01528             break;
01529         }
01530         t += t[1];
01531     }
01532     if ( t < tstop ) {
01533 /*
01534         Now t points at the float_ function and everything is correct.
01535         The result can go straight over the float_ function.
01536 */
01537         char *s;
01538         mp_exp_t exp;
01539         /* Extract the floating point value */
01540         UnpackFloat(aux4,t);
01541         /* Convert to string:
01542          - Format as MeN with M the mantissa and N the exponent
01543          - the generated string by mpf_get_str is the fraction/mantissa with
01544            an implicit radix point immediately to the left of the first digit.
01545          The applicable exponent is written in exp. */
01546         s = (char *)AO.floatspace;
01547         *s++ = '.';
01548         mpf_get_str(s,&exp, base, prec, aux4);
01549         while ( *s != 0 ) s++;
01550         *s++ = 'e';
01551         snprintf(s,AO.floatsize-(s-(char *)AO.floatspace),"%ld",exp);
01552         /* Negative base values are used to specify that the exponent is in decimal */
01553         mpf_set_str(aux4,(char *)AO.floatspace,-base);
01554         /* Pack the rounded floating point value back into the term */
01555         PackFloat(t,aux4);
01556         t+=t[1];
01557         *t++ = 1; *t++ = 1; *t++ = 3*sign;
01558         *term = t - term;
01559     }
01560     return(Generator(BHEAD term,level));
01561 }
01562 /*
01563     #] StrictRounding :
01564     #[ Chop :
01565
01566     Removes terms with a floating point number smaller than a given threshold.
01567
01568     Search for a FLOATFUN and compares its absolute value against the threshold

```

```

01569     specified in the chop statement. This threshold can be obtained from the
01570     LHS of the chop statement in the compiler buffer.
01571
01572  */
01573  int Chop(PHEAD WORD *term, WORD level)
01574  {
01575      WORD *tstop, *t, nsize, *threshold;
01576      CBUF *C = cbuf+AM.rbufnum;
01577      /* Find the float which should be at the end. */
01578      tstop = term + *term;
01579      nsize = ABS(tstop[-1]); tstop -= nsize;
01580      t = term+1;
01581      while ( t < tstop ) {
01582          if ( *t == FLOATFUN && t + t[1] == tstop && TestFloat(t) &&
01583              nsize == 3 && tstop[0] == 1 && tstop[1] == 1 ) break;
01584          t += t[1];
01585      }
01586      if ( t < tstop ) {
01587          /* Get threshold value from compiler buffer */
01588          threshold = C->lhs[level];
01589          threshold += 2; /* Skip TYPECHOP header */
01590          UnpackFloat(aux5, threshold);
01591
01592          /* Extract float and compute its absolute value */
01593          UnpackFloat(aux4, t);
01594          mpf_abs(aux4, aux4);
01595
01596          /* Remove if < threshold */
01597          if ( mpf_cmp(aux4, aux5) < 0 ) return(0);
01598      }
01599      return(Generator(BHEAD term, level));
01600  }
01601
01602  /*
01603      #] Chop :
01604      #] Float Routines :
01605      #[ Sorting :
01606
01607      We need a method to see whether two terms need addition that could
01608      involve floating point numbers.
01609      1: if PolyWise is active, we do not need anything, because
01610         Poly(Rat)Fun and floating point are mutually exclusive.
01611      2: This means that there should be only interference in AddCoef.
01612         and the AddRat parts in MergePatches, MasterMerge, SortBotMerge
01613         and PF_GetLoser.
01614      The compare routine(s) should recognise the float_ and give off
01615      a code (SortFloatMode) inside the AT struct:
01616      0: no float_
01617      1: float_ in term1 only
01618      2: float_ in term2 only
01619      3: float_ in both terms
01620      Be careful: we have several compare routines, because various codes
01621      use their own routines and we just set a variable with its address.
01622      Currently we have Compare1, CompareSymbols and CompareHSymbols.
01623      Only Compare1 should be active for this. The others should make sure
01624      that the proper variable is always zero.
01625      Remember: the sign of the coefficient is in the last word of the term.
01626
01627      We need two routines:
01628      1: AddWithFloat for SplitMerge which sorts by pointer.
01629      2: MergeWithFloat for MergePatches etc which keeps terms as much
01630         as possible in their location.
01631      The routines start out the same, because AT.SortFloatMode, set in
01632      Compare1, tells more or less what should be done.
01633      The difference is in where we leave the result.
01634
01635      In the future we may want to add a feature that makes the result
01636      zero if the sum comes within a certain distance of the numerical
01637      accuracy, like for instance 5% of the number of bits.
01638
01639      #[ AddWithFloat :
01640  */
01641
01642  int AddWithFloat(PHEAD WORD **ps1, WORD **ps2)
01643  {
01644      GETBIDENTITY
01645      SORTING *S = AT.SS;
01646      WORD *coef1, *coef2, size1, size2, *fun1, *fun2, *fun3;
01647      WORD *s1, *s2, sign3, j, jj, *t1, *t2, i;
01648      s1 = *ps1;
01649      s2 = *ps2;
01650      coef1 = s1+s1; size1 = coef1[-1]; coef1 -= ABS(coef1[-1]);
01651      coef2 = s2+s2; size2 = coef2[-1]; coef2 -= ABS(coef2[-1]);
01652      if ( AT.SortFloatMode == 3 ) {
01653          fun1 = s1+1; while ( fun1 < coef1 && fun1[0] != FLOATFUN ) fun1 += fun1[1];
01654          fun2 = s2+1; while ( fun2 < coef2 && fun2[0] != FLOATFUN ) fun2 += fun2[1];
01655          UnpackFloat(aux1, fun1);

```

```

01656         if ( size1 < 0 ) mpf_neg(aux1,aux1);
01657         UnpackFloat(aux2,fun2);
01658         if ( size2 < 0 ) mpf_neg(aux2,aux2);
01659     }
01660     else if ( AT.SortFloatMode == 1 ) {
01661         fun1 = s1+1; while ( fun1 < coef1 && fun1[0] != FLOATFUN ) fun1 += fun1[1];
01662         UnpackFloat(aux1,fun1);
01663         if ( size1 < 0 ) mpf_neg(aux1,aux1);
01664         fun2 = coef2;
01665         RatToFloat(aux2, (UWORD *)coef2, size2);
01666     }
01667     else if ( AT.SortFloatMode == 2 ) {
01668         fun2 = s2+1; while ( fun2 < coef2 && fun2[0] != FLOATFUN ) fun2 += fun2[1];
01669         fun1 = coef1;
01670         RatToFloat(aux1, (UWORD *)coef1, size1);
01671         UnpackFloat(aux2,fun2);
01672         if ( size2 < 0 ) mpf_neg(aux2,aux2);
01673     }
01674     else {
01675         MLOCK(ErrorMessageLock);
01676         MesPrint("Illegal value %d for AT.SortFloatMode in AddWithFloat.", AT.SortFloatMode);
01677         MUNLOCK(ErrorMessageLock);
01678         Terminate(-1);
01679         return(0);
01680     }
01681     mpf_add(aux3,aux1,aux2);
01682     sign3 = mpf_sgn(aux3);
01683     if ( sign3 < 0 ) mpf_neg(aux3,aux3);
01684     fun3 = TermMalloc("AddWithFloat");
01685     PackFloat(fun3,aux3);
01686     /*
01687     Now we have to calculate where the sumterm fits.
01688     If we are sloppy, we can be faster, but run the risk to need the
01689     extension space, even when it is not needed. At slightly lower speed
01690     (ie first creating the result in scribble space) we are better off.
01691     This is why we use TermMalloc.
01692
01693     The new term will have a rational coefficient of 1,1,+-.3.
01694     The size will be (fun1 or fun2 - term) + fun3 +3;
01695     */
01696     if ( AT.SortFloatMode == 3 ) {
01697         if ( fun1[1] == fun3[1] ) {
01698 Over1:
01699             i = fun3[1]; t1 = fun1; t2 = fun3; NCOPY(t1,t2,i);
01700             *t1++ = 1; *t1++ = 1; *t1++ = sign3 < 0 ? -3: 3;
01701             *s1 = t1-s1; goto Finished;
01702         }
01703         else if ( fun2[1] == fun3[1] ) {
01704 Over2:
01705             i = fun3[1]; t1 = fun2; t2 = fun3; NCOPY(t1,t2,i);
01706             *t1++ = 1; *t1++ = 1; *t1++ = sign3 < 0 ? -3: 3;
01707             *s2 = t1-s2; *ps1 = s2; goto Finished;
01708         }
01709         else if ( fun1[1] >= fun3[1] ) goto Over1;
01710         else if ( fun2[1] >= fun3[1] ) goto Over2;
01711     }
01712     else if ( AT.SortFloatMode == 1 ) {
01713         if ( fun1[1] >= fun3[1] ) goto Over1;
01714         else if ( fun3[1]+3 <= ABS(size2) ) goto Over2;
01715     }
01716     else if ( AT.SortFloatMode == 2 ) {
01717         if ( fun2[1] >= fun3[1] ) goto Over2;
01718         else if ( fun3[1]+3 <= ABS(size1) ) goto Over1;
01719     }
01720     /*
01721     Does not fit. Go to extension space.
01722     */
01723     jj = fun1-s1;
01724     j = jj+fun3[1]+3; /* space needed */
01725     if ( (S->sFill + j) >= S->sTop2 ) {
01726         GarbHand();
01727         s1 = *ps1; /* new position of the term after the garbage collection */
01728         fun1 = s1+jj;
01729     }
01730     t1 = S->sFill;
01731     for ( i = 0; i < jj; i++ ) *t1++ = s1[i];
01732     i = fun3[1]; s1 = fun3; NCOPY(t1,s1,i);
01733     *t1++ = 1; *t1++ = 1; *t1++ = sign3 < 0 ? -3: 3;
01734     *ps1 = S->sFill;
01735     **ps1 = t1-*ps1;
01736     S->sFill = t1;
01737 Finished:
01738     *ps2 = 0;
01739     TermFree(fun3,"AddWithFloat");
01740     AT.SortFloatMode = 0;
01741     if ( **ps1 > AM.MaxTer/( (LONG) (sizeof(WORD)) ) ) {
01742         MLOCK(ErrorMessageLock);

```

```

01743     MesPrint("Term too complex after addition in sort. MaxTermSize = %101",
01744             AM.MaxTer/sizeof(WORD));
01745     MUNLOCK(ErrorMessageLock);
01746     Terminate(-1);
01747 }
01748 return(1);
01749 }
01750
01751 /*
01752 #] AddWithFloat :
01753 #[ MergeWithFloat :
01754
01755 Note that we always park the result on top of term1.
01756 This makes life easier, because the original AddRat in MergePatches
01757 does this as well.
01758 */
01759
01760 int MergeWithFloat(PHEAD WORD **interm1, WORD **interm2)
01761 {
01762     GETBIDENTITY
01763     WORD *coef1, *coef2, size1, size2, *fun1, *fun2, *fun3, *tt;
01764     WORD sign3, jj, *t1, *t2, i, *term1 = *interm1, *term2 = *interm2;
01765     int retval = 0;
01766     coef1 = term1+*term1; size1 = coef1[-1]; coef1 -= ABS(size1);
01767     coef2 = term2+*term2; size2 = coef2[-1]; coef2 -= ABS(size2);
01768     if ( AT.SortFloatMode == 3 ) {
01769         fun1 = term1+1; while ( fun1 < coef1 && fun1[0] != FLOATFUN ) fun1 += fun1[1];
01770         fun2 = term2+1; while ( fun2 < coef2 && fun2[0] != FLOATFUN ) fun2 += fun2[1];
01771         UnpackFloat(aux1, fun1);
01772         if ( size1 < 0 ) mpf_neg(aux1, aux1);
01773         UnpackFloat(aux2, fun2);
01774         if ( size2 < 0 ) mpf_neg(aux2, aux2);
01775     }
01776     else if ( AT.SortFloatMode == 1 ) {
01777         fun1 = term1+1; while ( fun1 < coef1 && fun1[0] != FLOATFUN ) fun1 += fun1[1];
01778         UnpackFloat(aux1, fun1);
01779         if ( size1 < 0 ) mpf_neg(aux1, aux1);
01780         fun2 = coef2;
01781         RatToFloat(aux2, (UWORD *)coef2, size2);
01782     }
01783     else if ( AT.SortFloatMode == 2 ) {
01784         fun2 = term2+1; while ( fun2 < coef2 && fun2[0] != FLOATFUN ) fun2 += fun2[1];
01785         fun1 = coef1;
01786         RatToFloat(aux1, (UWORD *)coef1, size1);
01787         UnpackFloat(aux2, fun2);
01788         if ( size2 < 0 ) mpf_neg(aux2, aux2);
01789     }
01790     else {
01791         MLOCK(ErrorMessageLock);
01792         MesPrint("Illegal value %d for AT.SortFloatMode in MergeWithFloat.", AT.SortFloatMode);
01793         MUNLOCK(ErrorMessageLock);
01794         Terminate(-1);
01795         return(0);
01796     }
01797     mpf_add(aux3, aux1, aux2);
01798     sign3 = mpf_sgn(aux3);
01799 /*
01800 Now check whether we can park the result on top of one of the input terms.
01801 */
01802     if ( sign3 < 0 ) mpf_neg(aux3, aux3);
01803     fun3 = TermMalloc("MergeWithFloat");
01804     PackFloat(fun3, aux3);
01805     if ( AT.SortFloatMode == 3 ) {
01806         if ( fun1[1] + ABS(size1) == fun3[1] + 3 ) {
01807 OnTopOf1:   t1 = fun3; t2 = fun1;
01808             for ( i = 0; i < fun3[1]; i++ ) *t2++ = *t1++;
01809             *t2++ = 1; *t2++ = 1; *t2++ = sign3 < 0 ? -3: 3;
01810             retval = 1;
01811         }
01812         else if ( fun1[1] + ABS(size1) > fun3[1] + 3 ) {
01813 Shift1:     t2 = term1 + *term1; tt = t2;
01814             *--t2 = sign3 < 0 ? -3: 3; *--t2 = 1; *--t2 = 1;
01815             t1 = fun3 + fun3[1]; for ( i = 0; i < fun3[1]; i++ ) *--t2 = *--t1;
01816             t1 = fun1;
01817             while ( t1 > term1 ) *--t2 = *--t1;
01818             *t2 = tt-t2; term1 = t2;
01819             retval = 1;
01820         }
01821         else { /* Here we have to move term1 to the left to make room. */
01822             jj = fun3[1]-fun1[1]+3-ABS(size1); /* This is positive */
01823 Over1:     t2 = term1-jj; t1 = term1;
01824             while ( t1 < fun1 ) *t2++ = *t1++;
01825             term1 -= jj;
01826             *term1 += jj;
01827             for ( i = 0; i < fun3[1]; i++ ) *t2++ = fun3[i];
01828             *t2++ = 1; *t2++ = 1; *t2++ = sign3 < 0 ? -3: 3;
01829             retval = 1;

```



```

01830     }
01831 }
01832 else if ( AT.SortFloatMode == 1 ) {
01833     if ( fun1[1] + ABS(size1) == fun3[1] + 3 ) goto OnTopOf1;
01834     else if ( fun1[1] + ABS(size1) > fun3[1] + 3 ) goto Shift1;
01835     else {
01836         jj = fun3[1]-fun1[1]+3-ABS(size1); /* This is positive */
01837         goto Over1;
01838     }
01839 }
01840 else { /* Can only be 2, based on previous tests */
01841     if ( fun3[1] + 3 == ABS(size1) ) goto OnTopOf1;
01842     else if ( fun3[1] + 3 < ABS(size1) ) goto Shift1;
01843     else {
01844         jj = fun3[1]+3-ABS(size1); /* This is positive */
01845         goto Over1;
01846     }
01847 }
01848 *interm1 = term1;
01849 TermFree(fun3,"MergeWithFloat");
01850 AT.SortFloatMode = 0;
01851 return(retval);
01852 }
01853
01854 /*
01855     #] MergeWithFloat :
01856     #] Sorting :
01857     #[ MZV :
01858
01859     The functions here allow the arbitrary precision calculation
01860     of MZV's and Euler sums.
01861     They are called when the functions mzv_ and/or euler_ are used
01862     and the evaluate statement is applied.
01863     The output is in a float_ function.
01864     The expand statement tries to express the functions in terms of a basis.
01865     The bases are the 'standard basis for the euler sums and the
01866     pushdown bases from the datamine, unless otherwise specified.
01867
01868     #[ SimpleDelta :
01869 */
01870
01871 void SimpleDelta(mpf_t sum, int m)
01872 {
01873     long s = 1;
01874     long prec = AC.DefaultPrecision;
01875     unsigned long xprec = (unsigned long)prec, x;
01876     int j, jmax, n;
01877     mpf_t jm, jjm;
01878     mpf_init(jm); mpf_init(jjm);
01879     if ( m < 0 ) { s = -1; m = -m; }
01880
01881     /*
01882     We will loop until 1/2^j/j^m is smaller than the default precision.
01883     Just running to prec is however overkill, specially when m is large.
01884     We try to estimate a better value.
01885     jmax = prec - ((log2(prec)-1)*m).
01886     Hence we need the leading bit of prec.
01887     We are still overshooting a bit.
01888     */
01889     n = 0; x = xprec;
01890     while ( x ) { x >>= 1; n++; }
01891     /*
01892     We have now n = floor(log2(x))+1.
01893     */
01894     n--;
01895     jmax = (int)((int)xprec - (n-1)*m);
01896     /*
01897     For small prec and large m, the estimate can be wrong and even be negative,
01898     so we increase jmax until jmax + m*log2(jmax) > prec
01899     */
01900     if ( jmax < 0 ) jmax = 1;
01901     do {
01902         n = 0;
01903         x = (unsigned long)jmax;
01904         while (x) { x >>= 1; n++; }
01905         n--; // floor(log2(jmax))
01906         jmax++;
01907     } while ( jmax + m * n <= prec );
01908     mpf_set_ui(sum,0);
01909     for ( j = 1; j <= jmax; j++ ) {
01910 #ifdef WITHCUTOFF
01911         xprec--;
01912         mpf_set_prec_raw(jm,xprec);
01913         mpf_set_prec_raw(jjm,xprec);
01914 #endif
01915         mpf_set_ui(jm,1L);
01916         mpf_div_ui(jjm,jm,(unsigned long)j);

```

```

01917         mpf_pow_ui(jm, jjm, (unsigned long)m);
01918         mpf_div_2exp(jjm, jm, (unsigned long)j);
01919         if ( s == -1 && j%2 == 1 ) mpf_sub(sum, sum, jjm);
01920         else mpf_add(sum, sum, jjm);
01921     }
01922     mpf_clear(jjm); mpf_clear(jm);
01923 }
01924
01925 /*
01926     #] SimpleDelta :
01927     #[ SimpleDeltaC :
01928 */
01929
01930 void SimpleDeltaC(mpf_t sum, int m)
01931 {
01932     long s = 1;
01933     long prec = AC.DefaultPrecision;
01934     unsigned long xprec = (unsigned long)prec, x;
01935     int j, jmax, n;
01936     mpf_t jm, jjm;
01937     mpf_init(jm); mpf_init(jjm);
01938     if ( m < 0 ) { s = -1; m = -m; }
01939     /*
01940         We will loop until  $1/2^j/j^m$  is smaller than the default precision.
01941         Just running to prec is however overkill, specially when m is large.
01942         We try to estimate a better value.
01943          $jmax = prec - ((\log_2(prec)-1)*m)$ .
01944         Hence we need the leading bit of prec.
01945         We are still overshooting a bit.
01946     */
01947     n = 0; x = xprec;
01948     while ( x ) { x >= 1; n++; }
01949     /*
01950         We have now  $n = \text{floor}(\log_2(x))+1$ .
01951     */
01952     n--;
01953     jmax = (int)((int)xprec - (n-1)*m);
01954     /*
01955         For small prec and large m, the estimate can be wrong and even be negative,
01956         so we increase jmax until  $jmax + m*\log_2(jmax) > prec$ 
01957     */
01958     if ( jmax < 0 ) jmax = 1;
01959     do {
01960         n = 0;
01961         x = (unsigned long)jmax;
01962         while (x) { x >= 1; n++; }
01963         n--; // floor(log2(jmax))
01964         jmax++;
01965     } while ( jmax + m * n <= prec );
01966     if ( s < 0 ) jmax /= 2;
01967     mpf_set_si(sum, 0L);
01968     for ( j = 1; j <= jmax; j++ ) {
01969         #ifdef WITHCUTOFF
01970             xprec--;
01971             mpf_set_prec_raw(jm, xprec);
01972             mpf_set_prec_raw(jjm, xprec);
01973         #endif
01974         mpf_set_ui(jm, 1L);
01975         mpf_div_ui(jjm, jm, (unsigned long)j);
01976         mpf_pow_ui(jm, jjm, (unsigned long)m);
01977         if ( s == -1 ) mpf_div_2exp(jjm, jm, 2*(unsigned long)j);
01978         else mpf_div_2exp(jjm, jm, (unsigned long)j);
01979         mpf_add(sum, sum, jjm);
01980     }
01981     mpf_clear(jjm); mpf_clear(jm);
01982 }
01983
01984 /*
01985     #] SimpleDeltaC :
01986     #[ SingleTable :
01987 */
01988
01989 void SingleTable(mpf_t *tabl, int N, int m, int pow)
01990 {
01991     /*
01992         Creates a table T(1,...,N) with
01993          $T(i) = \text{sum\_}(j, i, N, [\text{sign\_}(j)]/2^j/j^m)$ 
01994         To make this table we have two options:
01995         1: run the sum backwards with the potential problem that the
01996            precision is difficult to manage.
01997         2: run the sum forwards. Take  $\text{sum\_}(j, 1, i-1, \dots)$  and later subtract.
01998         When doing Euler sums we may need also  $1/4^j$ 
01999         pow: 1 ->  $1/2^j$ 
02000             2 ->  $1/4^j$ 
02001     */
02002     GETIDENTITY
02003     long s = 1;

```

```

02004     int j;
02005 #ifdef WITHCUTOFF
02006     long prec = mpf_get_default_prec();
02007 #endif
02008     mpf_t jm, jjm;
02009     mpf_init(jm); mpf_init(jjm);
02010     if ( pow < 1 || pow > 2 ) {
02011         MLOCK(ErrorMessageLock);
02012         MesPrint("Wrong parameter pow in SingleTable: %d\n",pow);
02013         MUNLOCK(ErrorMessageLock);
02014         Terminate(-1);
02015     }
02016     if ( m < 0 ) { m = -m; s = -1; }
02017     mpf_set_si(auxsum,0L);
02018     for ( j = N; j > 0; j-- ) {
02019 #ifdef WITHCUTOFF
02020         mpf_set_prec_raw(jm,prec-j);
02021         mpf_set_prec_raw(jjm,prec-j);
02022 #endif
02023         mpf_set_ui(jm,1L);
02024         mpf_div_ui(jjm,jm,(unsigned long)j);
02025         mpf_pow_ui(jm,jjm,(unsigned long)m);
02026         mpf_div_2exp(jjm,jm,pow*(unsigned long)j);
02027         if ( pow == 2 ) mpf_add(auxsum,auxsum,jjm);
02028         else if ( s == -1 && j%2 == 1 ) mpf_sub(auxsum,auxsum,jjm);
02029         else mpf_add(auxsum,auxsum,jjm);
02030 /*
02031         And now copy auxsum to tablelement j
02032 */
02033         mpf_set(tabl[j],auxsum);
02034     }
02035     mpf_clear(jjm); mpf_clear(jm);
02036 }
02037
02038 /*
02039     #] SingleTable :
02040     #[ DoubleTable :
02041 */
02042
02043 void DoubleTable(mpf_t *tabout, mpf_t *tabin, int N, int m, int pow)
02044 {
02045     GETIDENTITY
02046     long s = 1;
02047 #ifdef WITHCUTOFF
02048     long prec = mpf_get_default_prec();
02049 #endif
02050     int j;
02051     mpf_t jm, jjm;
02052     mpf_init(jm); mpf_init(jjm);
02053     if ( pow < -1 || pow > 2 ) {
02054         MLOCK(ErrorMessageLock);
02055         MesPrint("Wrong parameter pow in DoubleTable: %d\n",pow);
02056         MUNLOCK(ErrorMessageLock);
02057         Terminate(-1);
02058     }
02059     if ( m < 0 ) { m = -m; s = -1; }
02060     mpf_set_ui(auxsum,0L);
02061     for ( j = N; j > 0; j-- ) {
02062 #ifdef WITHCUTOFF
02063         mpf_set_prec_raw(jm,prec-j);
02064         mpf_set_prec_raw(jjm,prec-j);
02065 #endif
02066         mpf_set_ui(jm,1L);
02067         mpf_div_ui(jjm,jm,(unsigned long)j);
02068         mpf_pow_ui(jm,jjm,(unsigned long)m);
02069         if ( pow == -1 ) {
02070             mpf_mul_2exp(jjm,jm,(unsigned long)j);
02071             mpf_mul(jm,jjm,tabin[j+1]);
02072         }
02073         else if ( pow > 0 ) {
02074             mpf_div_2exp(jjm,jm,pow*(unsigned long)j);
02075             mpf_mul(jm,jjm,tabin[j+1]);
02076         }
02077         else {
02078             mpf_mul(jm,jm,tabin[j+1]);
02079         }
02080         if ( pow == 2 ) mpf_add(auxsum,auxsum,jm);
02081         else if ( s == -1 && j%2 == 1 ) mpf_sub(auxsum,auxsum,jm);
02082         else mpf_add(auxsum,auxsum,jm);
02083 /*
02084         And now copy auxsum to tablelement j
02085 */
02086         mpf_set(tabout[j],auxsum);
02087     }
02088     mpf_clear(jjm); mpf_clear(jm);
02089 }
02090

```

```

02091 /*
02092     #] DoubleTable :
02093     #[ EndTable :
02094 */
02095
02096 void EndTable(mpf_t sum, mpf_t *tabin, int N, int m, int pow)
02097 {
02098     long s = 1;
02099     #ifdef WITHCUTOFF
02100     long prec = mpf_get_default_prec();
02101     #endif
02102     int j;
02103     mpf_t jm, jjm;
02104     mpf_init(jm); mpf_init(jjm);
02105     if ( pow < -1 || pow > 2 ) {
02106         MLOCK(ErrorMessageLock);
02107         MesPrint("Wrong parameter pow in EndTable: %d\n",pow);
02108         MUNLOCK(ErrorMessageLock);
02109         Terminate(-1);
02110     }
02111     if ( m < 0 ) { m = -m; s = -1; }
02112     mpf_set_si(sum,0L);
02113     for ( j = N; j > 0; j-- ) {
02114         #ifdef WITHCUTOFF
02115         mpf_set_prec_raw(jm,prec-j);
02116         mpf_set_prec_raw(jjm,prec-j);
02117         #endif
02118         mpf_set_ui(jm,1L);
02119         mpf_div_ui(jjm,jm,(unsigned long)j);
02120         mpf_pow_ui(jm,jjm,(unsigned long)m);
02121         if ( pow == -1 ) {
02122             mpf_mul_2exp(jjm,jm,(unsigned long)j);
02123             mpf_mul(jm,jjm,tabin[j+1]);
02124         }
02125         else if ( pow > 0 ) {
02126             mpf_div_2exp(jjm,jm,pow*(unsigned long)j);
02127             mpf_mul(jm,jjm,tabin[j+1]);
02128         }
02129         else {
02130             mpf_mul(jm,jm,tabin[j+1]);
02131         }
02132         if ( s == -1 && j%2 == 1 ) mpf_sub(sum,sum,jm);
02133         else mpf_add(sum,sum,jm);
02134     }
02135     mpf_clear(jjm); mpf_clear(jm);
02136 }
02137
02138 /*
02139     #] EndTable :
02140     #[ deltaMZV :
02141 */
02142
02143 void deltaMZV(mpf_t result, WORD *indexes, int depth)
02144 {
02145     GETIDENTITY
02146     /*
02147     Because all sums go roughly like 1/2^j we need about depth steps.
02148     */
02149     /* MesPrint("deltaMZV(%a)",depth,indexes); */
02150     if ( depth == 1 ) {
02151         if ( indexes[0] == 1 ) {
02152             mpf_set(result,mpfdelta1);
02153             return;
02154         }
02155         SimpleDelta(result,indexes[0]);
02156     }
02157     else if ( depth == 2 ) {
02158         if ( indexes[0] == 1 && indexes[1] == 1 ) {
02159             mpf_pow_ui(result,mpfdelta1,2);
02160             mpf_div_2exp(result,result,1);
02161         }
02162         else {
02163             SingleTable(mpf_t1,AC.DefaultPrecision-AC.MaxWeight+1,indexes[0],1);
02164             EndTable(result,mpf_t1,AC.DefaultPrecision-AC.MaxWeight,indexes[1],0);
02165         };
02166     }
02167     else if ( depth > 2 ) {
02168         mpf_t *mpftab3;
02169         int d;
02170         /*
02171         Check first whether this is a power of delta1 = ln(2)
02172         */
02173         for ( d = 0; d < depth; d++ ) {
02174             if ( indexes[d] != 1 ) break;
02175         }
02176         if ( d == depth ) { /* divide by fac_(depth) */
02177             mpf_pow_ui(result,mpfdelta1,depth);

```

```

02178         for ( d = 2; d <= depth; d++ ) {
02179             mpf_div_ui(result,result,(unsigned long)d);
02180         }
02181     }
02182     else {
02183         d = depth-1;
02184         SingleTable(mpftab1,AC.DefaultPrecision-AC.MaxWeight+d,*indexes,1);
02185         d--; indexes++;
02186         for(;;) {
02187             DoubleTable(mpftab2,mpftab1,AC.DefaultPrecision-AC.MaxWeight+d,*indexes,0);
02188             d--; indexes++;
02189             if ( d == 0 ) {
02190                 EndTable(result,mpftab2,AC.DefaultPrecision-AC.MaxWeight,*indexes,0);
02191                 break;
02192             }
02193             mpftab3 = (mpf_t *)AT.mpf_tab1; AT.mpf_tab1 = AT.mpf_tab2;
02194             AT.mpf_tab2 = (void *)mpftab3;
02195         }
02196     }
02197 }
02198 else if ( depth == 0 ) {
02199     mpf_set_ui(result,1L);
02200 }
02201 }
02202
02203 /*
02204     #] deltaMZV :
02205     #[ deltaEuler :
02206
02207     Regular Euler delta with - signs, but everywhere 1/2^j
02208 */
02209
02210 void deltaEuler(mpf_t result, WORD *indexes, int depth)
02211 {
02212     GETIDENTITY
02213     int m;
02214     #ifdef DEBUG
02215     int i;
02216     printf(" deltaEuler(");
02217     for ( i = 0; i < depth; i++ ) {
02218         if ( i != 0 ) printf(",");
02219         printf("%d",indexes[i]);
02220     }
02221     printf(") = ");
02222     #endif
02223     if ( depth == 1 ) {
02224         SimpleDelta(result,indexes[0]);
02225     }
02226     else if ( depth == 2 ) {
02227         SingleTable(mpftab1,AC.DefaultPrecision-AC.MaxWeight+1,indexes[0],1);
02228         m = indexes[1]; if ( indexes[0] < 0 ) m = -m;
02229         EndTable(result,mpftab1,AC.DefaultPrecision-AC.MaxWeight,m,0);
02230     }
02231     else if ( depth > 2 ) {
02232         int d;
02233         mpf_t *mpftab3;
02234         d = depth-1;
02235         SingleTable(mpftab1,AC.DefaultPrecision-AC.MaxWeight+d,*indexes,1);
02236         d--; indexes++;
02237         m = *indexes; if ( indexes[-1] < 0 ) m = -m;
02238         for(;;) {
02239             DoubleTable(mpftab2,mpftab1,AC.DefaultPrecision-AC.MaxWeight+d,m,0);
02240             d--; indexes++;
02241             m = *indexes; if ( indexes[-1] < 0 ) m = -m;
02242             if ( d == 0 ) {
02243                 EndTable(result,mpftab2,AC.DefaultPrecision-AC.MaxWeight,m,0);
02244                 break;
02245             }
02246             mpftab3 = (mpf_t *)AT.mpf_tab1; AT.mpf_tab1 = AT.mpf_tab2;
02247             AT.mpf_tab2 = (void *)mpftab3;
02248         }
02249     }
02250     else if ( depth == 0 ) {
02251         mpf_set_ui(result,1L);
02252     }
02253     #ifdef DEBUG
02254     gmp_printf("%.Fe\n",40,result);
02255     #endif
02256 }
02257
02258 /*
02259     #] deltaEuler :
02260     #[ deltaEulerC :
02261
02262     Conjugate Euler delta without - signs, but possibly 1/4^j
02263     When there is a - in the string we have 1/4.
02264 */

```

```

02265
02266 void deltaEulerC(mpf_t result, WORD *indexes, int depth)
02267 {
02268     GETIDENTITY
02269     int m;
02270 #ifdef DEBUG
02271     int i;
02272     printf(" deltaEulerC(");
02273     for ( i = 0; i < depth; i++ ) {
02274         if ( i != 0 ) printf(",");
02275         printf("%d",indexes[i]);
02276     }
02277     printf(") = ");
02278 #endif
02279     mpf_set_ui(result,0);
02280     if ( depth == 1 ) {
02281         if ( indexes[0] == 0 ) {
02282             MLOCK(ErrorMessageLock);
02283             MesPrint("Illegal index in depth=1 deltaEulerC: %d\n",indexes[0]);
02284             MUNLOCK(ErrorMessageLock);
02285             Terminate(-1);
02286         }
02287         if ( indexes[0] < 0 ) SimpleDeltaC(result,indexes[0]);
02288         else SimpleDelta(result,indexes[0]);
02289     }
02290     else if ( depth == 2 ) {
02291         int par;
02292         m = indexes[0];
02293         if ( m < 0 ) SingleTable(mpftab1,AC.DefaultPrecision-AC.MaxWeight+depth,-m,2);
02294         else SingleTable(mpftab1,AC.DefaultPrecision-AC.MaxWeight+depth, m,1);
02295         m = indexes[1];
02296         if ( m < 0 ) { m = -m; par = indexes[0] < 0 ? 0: 1; }
02297         else { par = indexes[0] < 0 ? -1: 0; }
02298         EndTable(result,mpftab1,AC.DefaultPrecision-AC.MaxWeight,m,par);
02299     }
02300     else if ( depth > 2 ) {
02301         int d, par;
02302         mpf_t *mpftab3;
02303         d = depth-1;
02304         m = indexes[0];
02305         ZeroTable(mpftab1,AC.DefaultPrecision+1);
02306         if ( m < 0 ) SingleTable(mpftab1,AC.DefaultPrecision-AC.MaxWeight+d+1,-m,2);
02307         else SingleTable(mpftab1,AC.DefaultPrecision-AC.MaxWeight+d+1, m,1);
02308         d--; indexes++; m = indexes[0];
02309         if ( m < 0 ) { m = -m; par = indexes[-1] < 0 ? 0: 1; }
02310         else { par = indexes[-1] < 0 ? -1: 0; }
02311         for(;;) {
02312             ZeroTable(mpftab2,AC.DefaultPrecision-AC.MaxWeight+d);
02313             DoubleTable(mpftab2,mpftab1,AC.DefaultPrecision-AC.MaxWeight+d,m,par);
02314             d--; indexes++; m = indexes[0];
02315             if ( m < 0 ) { m = -m; par = indexes[-1] < 0 ? 0: 1; }
02316             else { par = indexes[-1] < 0 ? -1: 0; }
02317             if ( d == 0 ) {
02318                 mpf_set_ui(result,0);
02319                 EndTable(result,mpftab2,AC.DefaultPrecision-AC.MaxWeight,m,par);
02320                 break;
02321             }
02322             mpftab3 = (mpf_t *)AT.mpf_tab1; AT.mpf_tab1 = AT.mpf_tab2;
02323             AT.mpf_tab2 = (void *)mpftab3;
02324         }
02325     }
02326     else if ( depth == 0 ) {
02327         mpf_set_ui(result,1L);
02328     }
02329 #ifdef DEBUG
02330     gmp_printf("%.Fe\n",40,result);
02331 #endif
02332 }
02333
02334 /*
02335     #] deltaEulerC :
02336     #[ CalculateMZVhalf :
02337
02338     This routine is mainly for support when large numbers of
02339     MZV's have to be calculated at the same time.
02340 */
02341
02342 void CalculateMZVhalf(mpf_t result, WORD *indexes, int depth)
02343 {
02344     int i;
02345     if ( depth < 0 ) {
02346         MLOCK(ErrorMessageLock);
02347         MesPrint("Illegal depth in CalculateMZVhalf: %d",depth);
02348         MUNLOCK(ErrorMessageLock);
02349         Terminate(-1);
02350     }
02351     for ( i = 0; i < depth; i++ ) {

```

```

02352         if ( indexes[i] <= 0 ) {
02353             MLOCK(ErrorMessageLock);
02354             MesPrint("Illegal index[%d] in CalculateMZVhalf: %d",i,indexes[i]);
02355             MUNLOCK(ErrorMessageLock);
02356             Terminate(-1);
02357         }
02358     }
02359     deltaMZV(result,indexes,depth);
02360 }
02361
02362 /*
02363     #] CalculateMZVhalf :
02364     #[ CalculateMZV :
02365 */
02366
02367 void CalculateMZV(mpf_t result, WORD *indexes, int depth)
02368 {
02369     GETIDENTITY
02370     int num1, num2 = 0, i, j = 0;
02371     if ( depth < 0 ) {
02372         MLOCK(ErrorMessageLock);
02373         MesPrint("Illegal depth in CalculateMZV: %d",depth);
02374         MUNLOCK(ErrorMessageLock);
02375         Terminate(-1);
02376     }
02377     if ( indexes[0] == 1 ) {
02378         MLOCK(ErrorMessageLock);
02379         MesPrint("Divergent MZV in CalculateMZV");
02380         MUNLOCK(ErrorMessageLock);
02381         Terminate(-1);
02382     }
02383     /* MesPrint("calculateMZV(%a)",depth,indexes); */
02384     for ( i = 0; i < depth; i++ ) {
02385         if ( indexes[i] <= 0 ) {
02386             MLOCK(ErrorMessageLock);
02387             MesPrint("Illegal index[%d] in CalculateMZV: %d",i,indexes[i]);
02388             MUNLOCK(ErrorMessageLock);
02389             Terminate(-1);
02390         }
02391         AT.indi1[i] = indexes[i];
02392     }
02393     num1 = depth; i = 0;
02394     deltaMZV(result,indexes,depth);
02395     for(;;) {
02396         if ( AT.indi1[0] > 1 ) {
02397             AT.indi1[0] -= 1;
02398             for ( j = num2; j > 0; j-- ) AT.indi2[j] = AT.indi2[j-1];
02399             AT.indi2[0] = 1; num2++;
02400         }
02401         else {
02402             AT.indi2[0] += 1;
02403             for ( j = 1; j < num1; j++ ) AT.indi1[j-1] = AT.indi1[j];
02404             num1--;
02405             if ( num1 == 0 ) break;
02406         }
02407         deltaMZV(aux1,AT.indi1,num1);
02408         deltaMZV(aux2,AT.indi2,num2);
02409         mpf_mul(aux3,aux1,aux2);
02410         mpf_add(result,result,aux3);
02411     }
02412     deltaMZV(aux3,AT.indi2,num2);
02413     mpf_add(result,result,aux3);
02414 }
02415
02416 /*
02417     #] CalculateMZV :
02418     #[ CalculateEuler :
02419
02420     There is a problem of notation here.
02421     Basically there are two notations.
02422     1:  $Z(m_1,m_2,m_3) = \sum_{j_3=1}^{\infty} \text{sign}_{-}(m_3) / j_3^{\text{abs}_{-}(m_3)} * \sum_{j_2=1}^{\infty} \sum_{j_1=1}^{\infty} \text{sign}_{-}(m_2) / j_2^{\text{abs}_{-}(m_2)} * \sum_{j_1=1}^{\infty} \text{sign}_{-}(m_1) / j_1^{\text{abs}_{-}(m_1)}$ 
02423
02424     See Broadhurst,1996
02425     2:  $L(m_1,m_2,m_3) = \sum_{j_1=1}^{\infty} \sum_{j_2=1}^{\infty} \sum_{j_3=1}^{\infty} \text{sign}_{-}(m_1) * \text{sign}_{-}(m_2) * \text{sign}_{-}(m_3) / j_3^{\text{abs}_{-}(m_3)} / (j_2+j_3)^{\text{abs}_{-}(m_2)} / (j_1+j_2+j_3)^{\text{abs}_{-}(m_1)}$ 
02426
02427     See Borwein, Bradley, Broadhurst and Lisonek, 1999
02428     The L corresponds with the H of the datamine up to  $\text{sign}_{-}(m_1*m_2*m_3)$ 
02429     The algorithm follows 2, but the more common notation is 1.
02430     Hence we start with a conversion.
02431 */
02432
02433 void CalculateEuler(mpf_t result, WORD *Zindexes, int depth)

```

```

02439 {
02440     GETIDENTITY
02441     int s1, num1, num2, i, j;
02442
02443     WORD *indexes = AT.WorkPointer;
02444     for ( i = 0; i < depth; i++ ) indexes[i] = Zindexes[i];
02445     for ( i = 0; i < depth-1; i++ ) {
02446         if ( Zindexes[i] < 0 ) {
02447             for ( j = i+1; j < depth; j++ ) indexes[j] = -indexes[j];
02448         }
02449     }
02450
02451     if ( depth < 0 ) {
02452         MLOCK(ErrorMessageLock);
02453         MesPrint("Illegal depth in CalculateEuler: %d\n",depth);
02454         MUNLOCK(ErrorMessageLock);
02455         Terminate(-1);
02456     }
02457     if ( indexes[0] == 1 ) {
02458         MLOCK(ErrorMessageLock);
02459         MesPrint("Divergent Euler sum in CalculateEuler\n");
02460         MUNLOCK(ErrorMessageLock);
02461         Terminate(-1);
02462     }
02463     for ( i = 0, j = 0; i < depth; i++ ) {
02464         if ( indexes[i] == 0 ) {
02465             MLOCK(ErrorMessageLock);
02466             MesPrint("Illegal index[%d] in CalculateEuler: %d\n",i,indexes[i]);
02467             MUNLOCK(ErrorMessageLock);
02468             Terminate(-1);
02469         }
02470         if ( indexes[i] < 0 ) j = 1;
02471         AT.indi1[i] = indexes[i];
02472     }
02473     if ( j == 0 ) {
02474         CalculateMZV(result,indexes,depth);
02475         return;
02476     }
02477     num1 = depth; AT.indi2[0] = 0; num2 = 0;
02478     deltaEuler(result,AT.indi1,depth);
02479     s1 = 0;
02480     for(;;) {
02481         s1++;
02482         if ( AT.indi1[0] > 1 ) {
02483             AT.indi1[0] -= 1;
02484             for ( j = num2; j > 0; j-- ) AT.indi2[j] = AT.indi2[j-1];
02485             AT.indi2[0] = 1; num2++;
02486         }
02487         else if ( AT.indi1[0] < -1 ) {
02488             AT.indi1[0] += 1;
02489             for ( j = num2; j > 0; j-- ) AT.indi2[j] = AT.indi2[j-1];
02490             AT.indi2[0] = 1; num2++;
02491         }
02492         else if ( AT.indi1[0] == -1 ) {
02493             for ( j = num2; j > 0; j-- ) AT.indi2[j] = AT.indi2[j-1];
02494             AT.indi2[0] = -1; num2++;
02495             for ( j = 1; j < num1; j++ ) AT.indi1[j-1] = AT.indi1[j];
02496             num1--;
02497         }
02498         else {
02499             AT.indi2[0] = AT.indi2[0] < 0 ? AT.indi2[0]-1 : AT.indi2[0]+1;
02500             for ( j = 1; j < num1; j++ ) AT.indi1[j-1] = AT.indi1[j];
02501             num1--;
02502         }
02503         if ( num1 == 0 ) break;
02504         deltaEuler(aux1,AT.indi1,num1);
02505         deltaEulerC(aux2,AT.indi2,num2);
02506         mpf_mul(aux3,aux1,aux2);
02507         if ( (depth+num1+num2+s1)%2 == 0 ) mpf_add(result,result,aux3);
02508         else mpf_sub(result,result,aux3);
02509     }
02510     deltaEulerC(aux3,AT.indi2,num2);
02511     if ( (depth+num2+s1)%2 == 0 ) mpf_add(result,result,aux3);
02512     else mpf_sub(result,result,aux3);
02513 }
02514
02515 /*
02516     #] CalculateEuler :
02517     #[ ExpandMZV :
02518 */
02519
02520 int ExpandMZV(WORD *term, WORD level)
02521 {
02522     DUMMYUSE(term);
02523     DUMMYUSE(level);
02524     return(0);
02525 }

```



```

02526
02527 /*
02528     #] ExpandMZV :
02529     #[ ExpandEuler :
02530 */
02531
02532 int ExpandEuler(WORD *term, WORD level)
02533 {
02534     DUMMYUSE(term);
02535     DUMMYUSE(level);
02536     return(0);
02537 }
02538
02539 /*
02540     #] ExpandEuler :
02541     #[ EvaluateEuler :
02542
02543     There are various steps here:
02544     1: locate an Euler function.
02545     2: evaluate it into a float.
02546     3: convert the coefficient to a float and multiply.
02547     4: Put the new term together.
02548     5: Restart to see whether there are more Euler functions.
02549     The compiler should check that there is no conflict between
02550     the evaluate command and a potential polyfun or polyratfun.
02551     Yet, when reading in expressions there can be a conflict.
02552     Hence the Normalize routine should check as well.
02553
02554     par == MZV: MZV
02555     par == EULER: Euler
02556     par == MZVHALF: MZV sums in 1/2 rather than 1. -> deltaMZV.
02557     par == ALLMZVFUNCTIONS: all of the above.
02558 */
02559
02560 int EvaluateEuler(PHEAD WORD *term, WORD level, WORD par)
02561 {
02562     WORD *t, *tstop, *tt, *tnext, sumweight, depth, first = 1, *newterm, i;
02563     WORD *oldworkpointer = AT.WorkPointer, nsize, *indexes;
02564     int retval;
02565     tstop = term + *term; tstop -= ABS(tstop[-1]);
02566     if ( AT.WorkPointer < term+*term ) AT.WorkPointer = term + *term;
02567 /*
02568     Step 1. We also need to verify that the argument we find is legal
02569 */
02570     t = term+1;
02571     while ( t < tstop ) {
02572         if ( ( *t == par ) || ( ( par == ALLMZVFUNCTIONS ) && (
02573             *t == MZV || *t == EULER || *t == MZVHALF ) ) ) {
02574             /* all arguments should be small integers */
02575             indexes = AT.WorkPointer;
02576             sumweight = 0; depth = 0;
02577             tnext = t + t[1]; tt = t + FUNHEAD;
02578             while ( tt < tnext ) {
02579                 if ( *tt != -SNUMBER ) goto nextfun;
02580                 if ( tt[1] == 0 ) goto nextfun;
02581                 indexes[depth] = tt[1];
02582                 sumweight += ABS(tt[1]); depth++;
02583                 tt += 2;
02584             }
02585             /* euler sum without arguments, i.e. mzv_(), euler_() or mzvhalf_() */
02586             if ( depth == 0 ) goto nextfun;
02587             if ( sumweight > AC.MaxWeight ) {
02588                 MLOCK(ErrorMessageLock);
02589                 MesPrint("Error: Weight of Euler/MZV sum greater than %d.",AC.MaxWeight);
02590                 MesPrint("Please increase the maximum weight in %#startfloat.");
02591                 MUNLOCK(ErrorMessageLock);
02592                 Terminate(-1);
02593             }
02594 /*
02595             Step 2: evaluate:
02596 */
02597             AT.WorkPointer += depth;
02598             if ( first ) {
02599                 if ( *t == MZV ) CalculateMZV(aux4,indexes,depth);
02600                 else if ( *t == EULER ) CalculateEuler(aux4,indexes,depth);
02601                 else if ( *t == MZVHALF ) CalculateMZVhalf(aux4,indexes,depth);
02602                 first = 0;
02603             }
02604             else {
02605                 if ( *t == MZV ) CalculateMZV(aux5,indexes,depth);
02606                 else if ( *t == EULER ) CalculateEuler(aux5,indexes,depth);
02607                 else if ( *t == MZVHALF ) CalculateMZVhalf(aux5,indexes,depth);
02608                 mpf_mul(aux4,aux4,aux5);
02609             }
02610             *t = 0;
02611         }
02612     nextfun:

```

```

02613         t += t[1];
02614     }
02615     if ( first == 1 ) return(Generator(BHEAD term,level));
02616 /*
02617     Step 3:
02618     Now the regular coefficient, if it is not 1/1.
02619     We have two cases: size +- 3, or bigger.
02620 */
02621     nsize = term[*term-1];
02622     if ( nsize < 0 ) {
02623         mpf_neg(aux4,aux4);
02624         nsize = -nsize;
02625     }
02626     if ( nsize == 3 ) {
02627         if ( tstop[0] != 1 ) {
02628             mpf_mul_ui(aux4,aux4, (ULONG) ((UWORD)tstop[0]));
02629         }
02630         if ( tstop[1] != 1 ) {
02631             mpf_div_ui(aux4,aux4, (ULONG) ((UWORD)tstop[1]));
02632         }
02633     }
02634     else {
02635         RatToFloat(aux5, (UWORD *)tstop,nsize);
02636         mpf_mul(aux4,aux4,aux5);
02637     }
02638 /*
02639     Now we have to locate possible other float_ functions.
02640 */
02641     t = term+1;
02642     while ( t < tstop ) {
02643         if ( *t == FLOATFUN ) {
02644             UnpackFloat(aux5,t);
02645             mpf_mul(aux4,aux4,aux5);
02646         }
02647         t += t[1];
02648     }
02649 /*
02650     Now we should compose the new term in the WorkSpace.
02651 */
02652     t = term+1;
02653     newterm = AT.WorkPointer;
02654     tt = newterm+1;
02655     while ( t < tstop ) {
02656         if ( *t == 0 || *t == FLOATFUN ) t += t[1];
02657         else {
02658             i = t[1]; NCOPY(tt,t,i);
02659         }
02660     }
02661     PackFloat(tt,aux4);
02662     tt += tt[1];
02663     *tt++ = 1; *tt++ = 1; *tt++ = 3;
02664     *newterm = tt-newterm;
02665     AT.WorkPointer = tt;
02666     retval = Generator(BHEAD newterm,level);
02667     AT.WorkPointer = oldworkpointer;
02668     return(retval);
02669 }
02670
02671 /*
02672     #] EvaluateEuler :
02673     #] MZV :
02674     #[ Functions :
02675     #[ CoEvaluate :
02676
02677     Current subkeys: mzv_, euler_ and sqrt_.
02678 */
02679
02680 int CoEvaluate(UBYTE *s)
02681 {
02682     GETIDENTITY
02683     UBYTE *subkey, c;
02684     WORD numfun, type;
02685     int error = 0;
02686     if ( AT.aux_ == 0 ) {
02687         MesPrint("&Illegal attempt to evaluate a function without activating floating point
numbers.");
02688         MesPrint("&Forgotten %%startfloat instruction?");
02689         return(1);
02690     }
02691     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
02692     if ( *s == 0 ) {
02693 /*
02694         No subkey, which means all functions.
02695 */
02696         Add3Com(TYPEEVALUATE,ALLFUNCTIONS);
02697 /*
02698         The MZV, EULER and MZVHALF are done separately

```

```

02699 */
02700     Add3Com(TYPEEVALUATE, ALLMZVFUNCTIONS);
02701     return(0);
02702 }
02703 while ( *s ) {
02704     subkey = s;
02705     while ( FG.cTable[*s] == 0 ) s++;
02706     if ( *s == '2' ) s++; /* cases li2_ and atan2_ */
02707     if ( *s == '-' ) s++;
02708     c = *s; *s = 0;
02709 /*
02710     We still need provisions for pi_ and possibly other constants.
02711 */
02712     if ( ( ( type = GetName(AC.varnames, subkey, &numfun, NOAUTO) ) != CFUNCTION )
02713         || ( functions[numfun].spec != 0 ) ) {
02714
02715         if ( type == CSYMBOL ) {
02716             Add4Com(TYPEEVALUATE, SYMBOL, numfun);
02717             break;
02718         }
02719 /*
02720         This cannot work.
02721 */
02722         MesPrint("%s should be a built in function that can be evaluated numerically.", s);
02723         return(1);
02724     }
02725     else {
02726         switch ( numfun+FUNCTION ) {
02727             case MZV:
02728             case EULER:
02729             case MZVHALF:
02730 /*
02731             The following functions are treated in evaluate.c
02732 */
02733             case SQRTFUNCTION:
02734             case LNFUNCTION:
02735             case EXPFUNCTION:
02736             case SINFUNCTION:
02737             case COSFUNCTION:
02738             case TANFUNCTION:
02739             case ASINFUNCTION:
02740             case ACOSFUNCTION:
02741             case ATANFUNCTION:
02742             case ATAN2FUNCTION:
02743             case SINHFUNCTION:
02744             case COSHFUNCTION:
02745             case TANHFUNCTION:
02746             case ASINHFUNCTION:
02747             case ACOSHFUNCTION:
02748             case ATANHFUNCTION:
02749             case LI2FUNCTION:
02750             case LINFUNCTION:
02751             case AGMFUNCTION:
02752             case GAMMAFUN:
02753 /*
02754             At a later stage we can add more functions from mpfr here
02755             mpfr_(number, arg(s))
02756 */
02757             Add3Com(TYPEEVALUATE, numfun+FUNCTION);
02758             break;
02759             default:
02760                 MesPrint("%s should be a built in function that can be evaluated numerically.", s);
02761                 error = 1;
02762                 break;
02763         }
02764     }
02765     *s = c;
02766     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
02767 }
02768 return(error);
02769 }
02770
02771 /*
02772     #] CoEvaluate :
02773     #] Functions :
02774 */

```

4.47 form3.h File Reference

```

#include <stdint.h>
#include <stddef.h>

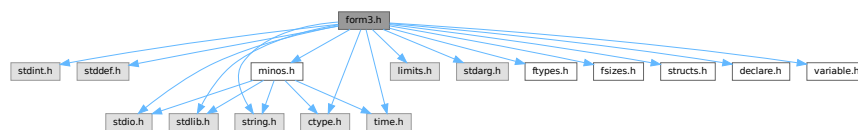
```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <limits.h>
#include <stdarg.h>
#include <time.h>
#include "ftypes.h"
#include "fsize.h"
#include "minos.h"
#include "structs.h"
#include "declare.h"
#include "variable.h"

```

Include dependency graph for form3.h:



Macros

- #define MAJORVERSION 5
- #define MINORVERSION 0
- #define PRODUCTIONDATE "8-nov-2022"
- #define BETAVERSION
- #define inline
- #define NDEBUG
- #define STATIC_ASSERT(condition) STATIC_ASSERT__1(condition, __LINE__)
- #define STATIC_ASSERT__1(X, L) STATIC_ASSERT__2(X, L)
- #define STATIC_ASSERT__2(X, L) STATIC_ASSERT__3(X, L)
- #define STATIC_ASSERT__3(X, L) typedef char static_assertion_failed_##L[(!!(X))*2-1]
- #define BITSINWORD 32
- #define BITSINLONG 64
- #define WORD_MIN_VALUE INT32_MIN
- #define WORD_MAX_VALUE INT32_MAX
- #define LONG_MIN_VALUE INT64_MIN
- #define LONG_MAX_VALUE INT64_MAX
- #define TOPBITONLY ((ULONG)1 << (BITSINWORD - 1)) /* 0x00008000UL */
- #define TOPLONGBITONLY ((ULONG)1 << (BITSINLONG - 1)) /* 0x80000000UL */
- #define SPECMASK ((UWORD)1 << (BITSINWORD - 1)) /* 0x8000U */
- #define WILDMASK ((UWORD)1 << (BITSINWORD - 2)) /* 0x4000U */
- #define WORDMASK ((ULONG)FULLMAX - 1) /* 0x0000FFFFUL */
- #define AWORDMASK (WORDMASK << BITSINWORD) /* 0xFFFF0000UL */
- #define FULLMAX ((LONG)1 << BITSINWORD) /* 0x00010000L */
- #define MAXPOSITIVE ((LONG)(TOPBITONLY - 1)) /* 0x00007FFFUL */
- #define MAXLONG ((LONG)(TOPLONGBITONLY - 1)) /* 0x7FFFFFFFL */
- #define MAXPOSITIVE2 (MAXPOSITIVE / 2) /* 0x00003FFFUL */
- #define MAXPOSITIVE4 (MAXPOSITIVE / 4) /* 0x00001FFFUL */
- #define form_alignof(type) offsetof(struct { char c_; type x_; }, x_)
- #define WITHSORTBOTS
- #define FILES FILE

- `#define Uopen(x, y) fopen(x,y)`
- `#define Uflush(x) fflush(x)`
- `#define Uclose(x) fclose(x)`
- `#define Uread(x, y, z, u) fread(x,y,z,u)`
- `#define Uwrite(x, y, z, u) fwrite(x,y,z,u)`
- `#define Usetbuf(x, y) setbuf(x,y)`
- `#define Useek(x, y, z) fseek(x,y,z)`
- `#define Utell(x) ftell(x)`
- `#define Ugetpos(x, y) fgetpos(x,y)`
- `#define Usetpos(x, y) fsetpos(x,y)`
- `#define Usync(x) fflush(x)`
- `#define Utruncate(x) _chsize(_fileno(x),0)`
- `#define Ustdout stdout`
- `#define MAX_OPEN_FILES FOPEN_MAX`
- `#define bzero(b, len) (memset((b), 0, (len)), (void)0)`
- `#define GetPID() ((LONG)GetCurrentProcessId())`

Typedefs

- `typedef int32_t WORD`
- `typedef int64_t LONG`
- `typedef uint32_t UWORD`
- `typedef uint64_t ULONG`
- `typedef signed char SBYTE`
- `typedef unsigned char UBYTE`
- `typedef unsigned int UINT`
- `typedef ULONG RLONG`
- `typedef int64_t MLONG`
- `typedef char BOOL`

Functions

- `STATIC_ASSERT (sizeof(WORD) *8==BITSINWORD)`
- `STATIC_ASSERT (sizeof(LONG) *8==BITSINLONG)`
- `STATIC_ASSERT (sizeof(LONG) >=sizeof(int *))`
- `STATIC_ASSERT (sizeof(int *) >=sizeof(int))`
- `STATIC_ASSERT (sizeof(int) >=sizeof(WORD))`
- `STATIC_ASSERT (sizeof(WORD) >=sizeof(char))`
- `STATIC_ASSERT (sizeof(char)==1)`
- `STATIC_ASSERT (sizeof(off_t) >=sizeof(LONG))`

4.47.1 Detailed Description

Contains critical defines for the compilation process Also contains the inclusion of all necessary header files. There are also some system dependencies concerning file functions.

Definition in file [form3.h](#).

4.47.2 Macro Definition Documentation

4.47.2.1 MAJORVERSION

```
#define MAJORVERSION 5
```

Definition at line 47 of file [form3.h](#).

4.47.2.2 MINORVERSION

```
#define MINORVERSION 0
```

Definition at line 48 of file [form3.h](#).

4.47.2.3 PRODUCTIONDATE

```
#define PRODUCTIONDATE "8-nov-2022"
```

Definition at line 53 of file [form3.h](#).

4.47.2.4 BETAVERSION

```
#define BETAVERSION
```

Definition at line 57 of file [form3.h](#).

4.47.2.5 inline

```
#define inline
```

Definition at line 120 of file [form3.h](#).

4.47.2.6 NDEBUG

```
#define NDEBUG
```

Definition at line 147 of file [form3.h](#).

4.47.2.7 STATIC_ASSERT

```
#define STATIC_ASSERT(  
    condition ) STATIC_ASSERT__1(condition, __LINE__)
```

Definition at line 158 of file [form3.h](#).

4.47.2.8 STATIC_ASSERT__1

```
#define STATIC_ASSERT__1(  
    X,  
    L ) STATIC_ASSERT__2 (X,L)
```

Definition at line 159 of file [form3.h](#).

4.47.2.9 STATIC_ASSERT__2

```
#define STATIC_ASSERT__2(  
    X,  
    L ) STATIC_ASSERT__3 (X,L)
```

Definition at line 160 of file [form3.h](#).

4.47.2.10 STATIC_ASSERT__3

```
#define STATIC_ASSERT__3(  
    X,  
    L ) typedef char static_assertion_failed_##L[ (!! (X)) *2-1]
```

Definition at line 161 of file [form3.h](#).

4.47.2.11 BITSINWORD

```
#define BITSINWORD 32
```

Definition at line 200 of file [form3.h](#).

4.47.2.12 BITSINLONG

```
#define BITSINLONG 64
```

Definition at line 201 of file [form3.h](#).

4.47.2.13 WORD_MIN_VALUE

```
#define WORD_MIN_VALUE INT32_MIN
```

Definition at line 202 of file [form3.h](#).

4.47.2.14 WORD_MAX_VALUE

```
#define WORD_MAX_VALUE INT32_MAX
```

Definition at line 203 of file [form3.h](#).

4.47.2.15 LONG_MIN_VALUE

```
#define LONG_MIN_VALUE INT64_MIN
```

Definition at line 204 of file [form3.h](#).

4.47.2.16 LONG_MAX_VALUE

```
#define LONG_MAX_VALUE INT64_MAX
```

Definition at line 205 of file [form3.h](#).

4.47.2.17 TOPBITONLY

```
#define TOPBITONLY ((ULONG)1 << (BITSINWORD - 1)) /* 0x00008000UL */
```

Definition at line 242 of file [form3.h](#).

4.47.2.18 TOPLONGBITONLY

```
#define TOPLONGBITONLY ((ULONG)1 << (BITSINLONG - 1)) /* 0x80000000UL */
```

Definition at line 243 of file [form3.h](#).

4.47.2.19 SPECMASK

```
#define SPECMASK ((UWORD)1 << (BITSINWORD - 1)) /* 0x8000U */
```

Definition at line 244 of file [form3.h](#).

4.47.2.20 WILDMASK

```
#define WILDMASK ((UWORD)1 << (BITSINWORD - 2)) /* 0x4000U */
```

Definition at line 245 of file [form3.h](#).

4.47.2.21 WORDMASK

```
#define WORDMASK ((ULONG)FULLMAX - 1) /* 0x0000FFFFUL */
```

Definition at line 246 of file [form3.h](#).

4.47.2.22 AWORDMASK

```
#define AWORDMASK (WORDMASK << BITSINWORD) /* 0xFFFF0000UL */
```

Definition at line 247 of file [form3.h](#).

4.47.2.23 FULLMAX

```
#define FULLMAX ((LONG)1 << BITSINWORD) /* 0x00010000L */
```

Definition at line 248 of file [form3.h](#).

4.47.2.24 MAXPOSITIVE

```
#define MAXPOSITIVE ((LONG)(TOPBITONLY - 1)) /* 0x00007FFFL */
```

Definition at line 249 of file [form3.h](#).

4.47.2.25 MAXLONG

```
#define MAXLONG ((LONG)(TOPLONGBITONLY - 1)) /* 0x7FFFFFFFL */
```

Definition at line 250 of file [form3.h](#).

4.47.2.26 MAXPOSITIVE2

```
#define MAXPOSITIVE2 (MAXPOSITIVE / 2) /* 0x00003FFFL */
```

Definition at line 251 of file [form3.h](#).

4.47.2.27 MAXPOSITIVE4

```
#define MAXPOSITIVE4 (MAXPOSITIVE / 4) /* 0x00001FFFL */
```

Definition at line 252 of file [form3.h](#).

4.47.2.28 form_alignof

```
#define form_alignof(  
    type ) offsetof(struct { char c_; type x_; }, x_)
```

Definition at line 268 of file [form3.h](#).

4.47.2.29 WITHSORTBOTS

```
#define WITHSORTBOTS
```

Definition at line 288 of file [form3.h](#).

4.47.2.30 FILES

```
#define FILES FILE
```

Definition at line 369 of file [form3.h](#).

4.47.2.31 Uopen

```
#define Uopen(  
    x,  
    y ) fopen(x,y)
```

Definition at line 370 of file [form3.h](#).

4.47.2.32 Uflush

```
#define Uflush(  
    x ) fflush(x)
```

Definition at line 371 of file [form3.h](#).

4.47.2.33 Uclose

```
#define Uclose(  
    x ) fclose(x)
```

Definition at line 372 of file [form3.h](#).

4.47.2.34 Uread

```
#define Uread(  
    x,  
    y,  
    z,  
    u ) fread(x,y,z,u)
```

Definition at line 373 of file [form3.h](#).

4.47.2.35 Uwrite

```
#define Uwrite(  
    x,  
    y,  
    z,  
    u ) fwrite(x,y,z,u)
```

Definition at line 374 of file [form3.h](#).

4.47.2.36 Usetbuf

```
#define Usetbuf(  
    x,  
    y ) setbuf(x,y)
```

Definition at line 375 of file [form3.h](#).

4.47.2.37 Useek

```
#define Useek(  
    x,  
    y,  
    z ) fseek(x,y,z)
```

Definition at line 376 of file [form3.h](#).

4.47.2.38 Utell

```
#define Utell(  
    x ) ftell(x)
```

Definition at line 377 of file [form3.h](#).

4.47.2.39 Ugetpos

```
#define Ugetpos(  
    x,  
    y ) fgetpos(x,y)
```

Definition at line 378 of file [form3.h](#).

4.47.2.40 Usetpos

```
#define Usetpos(  
    x,  
    y ) fsetpos(x,y)
```

Definition at line 379 of file [form3.h](#).

4.47.2.41 Usync

```
#define Usync(  
    x ) fflush(x)
```

Definition at line 380 of file [form3.h](#).

4.47.2.42 Utruncate

```
#define Utruncate(  
    x ) _chsize(_fileno(x),0)
```

Definition at line [381](#) of file [form3.h](#).

4.47.2.43 Ustdout

```
#define Ustdout stdout
```

Definition at line [382](#) of file [form3.h](#).

4.47.2.44 MAX_OPEN_FILES

```
#define MAX_OPEN_FILES FOPEN_MAX
```

Definition at line [383](#) of file [form3.h](#).

4.47.2.45 bzero

```
#define bzero(  
    b,  
    len ) (memset((b), 0, (len)), (void)0)
```

Definition at line [384](#) of file [form3.h](#).

4.47.2.46 GetPID

```
#define GetPID( ) ((LONG)GetCurrentProcessId())
```

Definition at line [385](#) of file [form3.h](#).

4.47.3 Typedef Documentation

4.47.3.1 WORD

```
typedef int32_t WORD
```

Definition at line [196](#) of file [form3.h](#).

4.47.3.2 LONG

```
typedef int64_t LONG
```

Definition at line [197](#) of file [form3.h](#).

4.47.3.3 UWORD

```
typedef uint32_t UWORD
```

Definition at line 198 of file [form3.h](#).

4.47.3.4 ULONG

```
typedef uint64_t ULONG
```

Definition at line 199 of file [form3.h](#).

4.47.3.5 SBYTE

```
typedef signed char SBYTE
```

Definition at line 231 of file [form3.h](#).

4.47.3.6 UBYTE

```
typedef unsigned char UBYTE
```

Definition at line 232 of file [form3.h](#).

4.47.3.7 UINT

```
typedef unsigned int UINT
```

Definition at line 233 of file [form3.h](#).

4.47.3.8 RLONG

```
typedef ULONG RLONG
```

Definition at line 234 of file [form3.h](#).

4.47.3.9 MLONG

```
typedef int64_t MLONG
```

Definition at line 235 of file [form3.h](#).

4.47.3.10 BOOL

```
typedef char BOOL
```

Definition at line 240 of file [form3.h](#).

4.48 form3.h

[Go to the documentation of this file.](#)

```

00001
00008 /* # [ License : */
00009 /*
00010 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00011 * When using this file you are requested to refer to the publication
00012 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013 * This is considered a matter of courtesy as the development was paid
00014 * for by FOM the Dutch physics granting agency and we would like to
00015 * be able to track its scientific use to convince FOM of its value
00016 * for the community.
00017 *
00018 * This file is part of FORM.
00019 *
00020 * FORM is free software: you can redistribute it and/or modify it under the
00021 * terms of the GNU General Public License as published by the Free Software
00022 * Foundation, either version 3 of the License, or (at your option) any later
00023 * version.
00024 *
00025 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00028 * details.
00029 *
00030 * You should have received a copy of the GNU General Public License along
00031 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00032 */
00033 /* # ] License : */
00034
00035 #ifndef __FORM3H__
00036 #define __FORM3H__
00037
00038 #ifdef HAVE_CONFIG_H
00039
00040 #ifndef CONFIG_H_INCLUDED
00041 #define CONFIG_H_INCLUDED
00042 #include <config.h>
00043 #endif
00044
00045 #else /* HAVE_CONFIG_H */
00046
00047 #define MAJORVERSION 5
00048 #define MINORVERSION 0
00049
00050 #ifdef __DATE__
00051 #define PRODUCTIONDATE __DATE__
00052 #else
00053 #define PRODUCTIONDATE "8-nov-2022"
00054 #endif
00055
00056 /*#undef BETAVERSION */
00057 #define BETAVERSION
00058
00059 #ifdef LINUX32
00060 #define UNIX
00061 #define LINUX
00062 #define _FILE_OFFSET_BITS 64
00063 #define WITHZLIB
00064 #define WITHGMP
00065 #define WITHPOSIXCLOCK
00066 #endif
00067
00068 #ifdef LINUX64
00069 #define UNIX
00070 #define LINUX
00071 #define WITHZLIB
00072 #define WITHGMP
00073 #define WITHPOSIXCLOCK
00074 #define WITHFLOAT
00075 #endif
00076
00077 #ifdef APPLE32
00078 #define UNIX
00079 #define _FILE_OFFSET_BITS 64
00080 #define WITHZLIB
00081 #endif
00082
00083 #ifdef APPLE64
00084 #define UNIX
00085 #define WITHZLIB
00086 #define WITHGMP
00087 #define WITHPOSIXCLOCK
00088 #define WITHFLOAT

```

```

00089 #define HAVE_UNORDERED_MAP
00090 #define HAVE_UNORDERED_SET
00091 #endif
00092
00093 #ifdef CYGWIN32
00094 #define UNIX
00095 #endif
00096
00097 #ifdef _MSC_VER
00098 #define WINDOWS
00099 #define _CRT_SECURE_NO_WARNINGS
00100 #endif
00101
00102 /*
00103  * We must not define WITHPOSIXCLOCK in compiling the sequential FORM or ParFORM.
00104  */
00105 #if !defined(WITHPTHREADS) && defined(WITHPOSIXCLOCK)
00106 #undef WITHPOSIXCLOCK
00107 #endif
00108
00109 #if !defined(__cplusplus) && !defined(inline)
00110 #if (defined(__STDC_VERSION__) && __STDC_VERSION__ >= 199901L) || (defined(__GNUC__) &&
00111     !defined(__STRICT_ANSI__))
00112 /* "inline" is available. */
00113 #elif defined(__GNUC__)
00114 /* GNU C compiler has "__inline__". */
00115 #define inline __inline__
00116 #elif defined(_MSC_VER)
00117 /* Microsoft C compiler has "__inline". */
00118 #define inline __inline
00119 #else
00120 /* Inline functions may be not supported. Define "inline" to be empty. */
00121 #define inline
00122 #endif
00123
00124 #endif /* HAVE_CONFIG_H */
00125
00126 /* Workaround for MSVC. */
00127 #if defined(_MSC_VER)
00128 /*
00129  * Old versions of MSVC didn't support C99 function `snprintf`, which is used
00130  * in poly.cc. On the other hand, macroizing `snprintf` gives a fatal error
00131  * with MSVC >= 2015.
00132  */
00133 #if _MSC_VER < 1900
00134 #define snprintf _snprintf
00135 #endif
00136 #endif
00137
00138 /*
00139  * Translate our dialect "DEBUGGING" to the standard "NDEBUG".
00140  */
00141 #ifdef DEBUGGING
00142 #ifdef NDEBUG
00143 #undef NDEBUG
00144 #endif
00145 #else
00146 #ifndef NDEBUG
00147 #define NDEBUG
00148 #endif
00149 #endif
00150
00151 /*
00152  * STATIC_ASSERT(condition) will fail to be compiled if the given
00153  * condition is false.
00154  */
00155 #define STATIC_ASSERT(condition)
00156 #define STATIC_ASSERT(condition) STATIC_ASSERT__1(condition, __LINE__)
00157 #define STATIC_ASSERT__1(X, L) STATIC_ASSERT__2(X, L)
00158 #define STATIC_ASSERT__2(X, L) STATIC_ASSERT__3(X, L)
00159 #define STATIC_ASSERT__3(X, L) \
00160     typedef char static_assertion_failed_##L[(!!(X))*2-1]
00161
00162 /*
00163  * UNIX or WINDOWS must be defined.
00164  */
00165 #if defined(UNIX)
00166 #define mBSD
00167 #define ANSI
00168 #elif defined(WINDOWS)
00169 #define ANSI
00170 #define WIN32_LEAN_AND_MEAN
00171 #include <windows.h>
00172 #include <io.h>

```

```

00175 #include <fcntl.h>
00176 /* Undefine/rename conflicted symbols. */
00177 #undef MAXLONG /* WinNT.h */
00178 #define WORD FORM_WORD /* WinDef.h */
00179 #define LONG FORM_LONG /* WinNT.h */
00180 #define ULONG FORM_ULONG /* WinDef.h */
00181 #define BOOL FORM_BOOL /* WinDef.h */
00182 #undef CreateFile /* WinBase.h */
00183 #undef CopyFile /* WinBase.h */
00184 #define OpenFile FORM_OpenFile /* WinBase.h */
00185 #define ReOpenFile FORM_ReOpenFile /* WinBase.h */
00186 #define ReadFile FORM_ReadFile /* WinBase.h */
00187 #define WriteFile FORM_WriteFile /* WinBase.h */
00188 #define DeleteObject FORM_DeleteObject /* WinGDI.h */
00189 #else
00190 #error UNIX or WINDOWS must be defined!
00191 #endif
00192
00193 #include <stdint.h>
00194
00195 #if UINTPTR_MAX == UINT64_MAX
00196     typedef int32_t WORD;
00197     typedef int64_t LONG;
00198     typedef uint32_t UWORD;
00199     typedef uint64_t ULONG;
00200     #define BITSINWORD 32
00201     #define BITSINLONG 64
00202     #define WORD_MIN_VALUE INT32_MIN
00203     #define WORD_MAX_VALUE INT32_MAX
00204     #define LONG_MIN_VALUE INT64_MIN
00205     #define LONG_MAX_VALUE INT64_MAX
00206 #elif UINTPTR_MAX == UINT32_MAX
00207     typedef int16_t WORD;
00208     typedef int32_t LONG;
00209     typedef uint16_t UWORD;
00210     typedef uint32_t ULONG;
00211     #define BITSINWORD 16
00212     #define BITSINLONG 32
00213     #define WORD_MIN_VALUE INT16_MIN
00214     #define WORD_MAX_VALUE INT16_MAX
00215     #define LONG_MIN_VALUE INT32_MIN
00216     #define LONG_MAX_VALUE INT32_MAX
00217 #else
00218     #error Can not detect if this is a 32-bit or 64-bit platform.
00219 #endif
00220
00221 STATIC_ASSERT(sizeof(WORD) * 8 == BITSINWORD);
00222 STATIC_ASSERT(sizeof(LONG) * 8 == BITSINLONG);
00223 STATIC_ASSERT(sizeof(WORD) * 2 == sizeof(LONG));
00224 STATIC_ASSERT(sizeof(LONG) >= sizeof(int *));
00225 STATIC_ASSERT(sizeof(LONG) >= sizeof(int *));
00226 STATIC_ASSERT(sizeof(int *) >= sizeof(int));
00227 STATIC_ASSERT(sizeof(int) >= sizeof(WORD));
00228 STATIC_ASSERT(sizeof(WORD) >= sizeof(char));
00229 STATIC_ASSERT(sizeof(char) == 1);
00230
00231 typedef signed char SBYTE;
00232 typedef unsigned char UBYTE;
00233 typedef unsigned int UINT;
00234 typedef ULONG RLONG; /* Used in reken.c. */
00235 typedef int64_t MLONG; /* See commentary in minos.h. */
00236 /*
00237  * NOTE: we don't use the standard _Bool (or C++ bool) because its size is
00238  * implementation-dependent.
00239  */
00240 typedef char BOOL;
00241
00242 /* E.g. in 32-bits */
00243 #define TOPBITONLY ((ULONG)1 << (BITSINWORD - 1)) /* 0x00008000UL */
00244 #define TOPLONGBITONLY ((ULONG)1 << (BITSINLONG - 1)) /* 0x80000000UL */
00245 #define SPECMASK ((UWORD)1 << (BITSINWORD - 1)) /* 0x8000U */
00246 #define WILDMASK ((UWORD)1 << (BITSINWORD - 2)) /* 0x4000U */
00247 #define WORDMASK ((ULONG)FULLMAX - 1) /* 0x0000FFFFUL */
00248 #define AWORDMASK (WORDMASK << BITSINWORD) /* 0xFFFF0000UL */
00249 #define FULLMAX ((LONG)1 << BITSINWORD) /* 0x00010000L */
00250 #define MAXPOSITIVE ((LONG)(TOPBITONLY - 1)) /* 0x00007FFF */
00251 #define MAXLONG ((LONG)(TOPLONGBITONLY - 1)) /* 0x7FFFFFFF */
00252 #define MAXPOSITIVE2 (MAXPOSITIVE / 2) /* 0x00003FFF */
00253 #define MAXPOSITIVE4 (MAXPOSITIVE / 4) /* 0x00001FFF */
00254 /*
00255  * form_alignof(type) returns the number of bytes used in the alignment of
00256  * the type.
00257  */
00258 #if !defined(form_alignof)
00259 #if defined(__GNUC__)
00260 /* GNU C compiler has "__alignof__". */
00261 #define form_alignof(type) __alignof__(type)

```



```

00262 #elif defined(_MSC_VER)
00263 /* Microsoft C compiler has "__alignof". */
00264 #define form_alignof(type) __alignof(type)
00265 #elif !defined(__cplusplus)
00266 /* Generic case in C. */
00267 #include <stddef.h>
00268 #define form_alignof(type) offsetof(struct { char c_; type x_; }, x_)
00269 #else
00270 /* Generic case in C++, at least works with a POD struct. */
00271 #include <cstdint>
00272 namespace alignof_impl_ {
00273 template<typename T> struct calc {
00274     struct X { char c_; T x_; };
00275     enum { value = offsetof(X, x_) };
00276 };
00277 }
00278 #define form_alignof(type) alignof_impl_::calc<type>::value
00279 #endif
00280 #endif
00281
00282
00283 /*
00284 #define WITHPCOUNTER
00285 #define DEBUGGINGLOCKS
00286 #define WITHSTATS
00287 */
00288 #define WITHSORTBOTS
00289
00290 #include <stdio.h>
00291 #include <stdlib.h>
00292 #include <string.h>
00293 #include <ctype.h>
00294 #include <limits.h>
00295 #include <stdarg.h>
00296 #include <time.h>
00297 #ifdef WINDOWS
00298 #include "fwin.h"
00299 #endif
00300 #ifdef UNIX
00301 #include <unistd.h>
00302 #include <fcntl.h>
00303 #include <sys/file.h>
00304 #include "unix.h"
00305 #endif
00306 #ifdef WITHZLIB
00307 #ifdef WITHZSTD
00308 #include <zstd_zlibwrapper.h>
00309 #else
00310 #include <zlib.h>
00311 #endif
00312 #endif
00313 #ifdef WITHPTHREADS
00314 #include <pthread.h>
00315 #endif
00316
00317 /*
00318 PARALLELCODE indicates code that is common for TFORM and ParFORM but
00319 should not be there for sequential FORM.
00320 */
00321 #if defined(WITHMPI) || defined(WITHPTHREADS)
00322 #define PARALLELCODE
00323 #endif
00324
00325 #include "ftypes.h"
00326 #include "fsizes.h"
00327 #include "minos.h"
00328 #include "structs.h"
00329 #include "declare.h"
00330 #include "variable.h"
00331
00332 STATIC_ASSERT(sizeof(off_t) >= sizeof(LONG));
00333
00334 /*
00335 * The interface to file routines for UNIX or non-UNIX (Windows).
00336 */
00337 #ifdef UNIX
00338
00339 #define UFILES
00340 typedef struct FileS {
00341     int descriptor;
00342 } FILES;
00343 extern FILES *Uopen(char *, char *);
00344 extern int Uclose(FILES *);
00345 extern size_t Uread(char *, size_t, size_t, FILES *);
00346 extern size_t Uwrite(char *, size_t, size_t, FILES *);
00347 extern int Useek(FILES *, off_t, int);
00348 extern off_t Utell(FILES *);

```

```

00349 extern void    Uflush(FILE *);
00350 extern int      Ugetpos(FILE *, fpos_t *);
00351 extern int      Usetpos(FILE *, fpos_t *);
00352 extern void      Usetbuf(FILE *, char *);
00353 #define Usync(f) fsync(f->descriptor)
00354 #define Utruncate(f) { \
00355     if ( ftruncate(f->descriptor, 0) ) { \
00356         MLOCK(ErrorMessageLock); \
00357         MesPrint("Utruncate failed"); \
00358         MUNLOCK(ErrorMessageLock); \
00359         /* Calling Terminate() here may cause an infinite loop due to CleanUpSort(). */ \
00360         /* Terminate(-1); */ \
00361     } \
00362 }
00363 extern FILE *Ustdout;
00364 #define MAX_OPEN_FILES getdtablesize()
00365 #define GetPID() ((LONG)getpid())
00366
00367 #else /* UNIX */
00368
00369 #define FILES FILE
00370 #define Uopen(x,y) fopen(x,y)
00371 #define Uflush(x) fflush(x)
00372 #define Uclose(x) fclose(x)
00373 #define Uread(x,y,z,u) fread(x,y,z,u)
00374 #define Uwrite(x,y,z,u) fwrite(x,y,z,u)
00375 #define Usetbuf(x,y) setbuf(x,y)
00376 #define Useek(x,y,z) fseek(x,y,z)
00377 #define Utell(x) ftell(x)
00378 #define Ugetpos(x,y) fgetpos(x,y)
00379 #define Usetpos(x,y) fsetpos(x,y)
00380 #define Usync(x) fflush(x)
00381 #define Utruncate(x) _chsize(_fileno(x),0)
00382 #define Ustdout stdout
00383 #define MAX_OPEN_FILES FOPEN_MAX
00384 #define bzero(b,len) (memset((b), 0, (len)), (void)0)
00385 #define GetPID() ((LONG)GetCurrentProcessId())
00386
00387 #endif /* UNIX */
00388
00389 /*
00390 * Some system may implement the POSIX "environ" variable as a macro, e.g.,
00391 * https://github.com/mingw-w64/mingw-w64/blob/v10.0.0/mingw-w64-headers/crt/stdlib.h#L704
00392 * which breaks the definition of DoShattering() in diagrams.c that uses
00393 * "environ" as a formal parameter. Because FORM doesn't use the POSIX "environ"
00394 * or its variant anyway, we can just undefine it.
00395 */
00396 #ifdef environ
00397 #undef environ
00398 #endif
00399
00400 #ifdef WITHMPI
00401 #include "parallel.h"
00402 #endif
00403
00404 #endif /* __FORM3H__ */

```

4.49 fsizes.h File Reference

This graph shows which files directly or indirectly include this file:



Macros

- #define [MAXPRENAMESIZE](#) 128
- #define [MAXENAME](#) 16
- #define [MAXSAVEFUNCTION](#) 16384
- #define [MAXPARLEVEL](#) 100
- #define [MAXNUMBERSIZE](#) 200

- #define [MAXREPEAT](#) 100
- #define [NORMSIZE](#) 1000
- #define [INITNODESIZE](#) 10
- #define [INITNAMESIZE](#) 100
- #define [NUMFIXED](#) 128
- #define [MAXNEST](#) 100
- #define [MAXMATCH](#) 30
- #define [MAXIF](#) 20
- #define [SIZEFACS](#) 640L
- #define [NUMFACS](#) 50
- #define [MAXLOOPS](#) 30
- #define [MAXLABELS](#) 20
- #define [COMMERCIALSIZE](#) 24
- #define [MAXFLAGS](#) 16
- #define [COMPRESSBUFFER](#) 90000
- #define [FORTRANCONTINUATIONLINES](#) 15
- #define [MAXLEVELS](#) 2000
- #define [MAXLHS](#) 400
- #define [MAXWILDC](#) 100
- #define [NUMTABLEENTRIES](#) 1000
- #define [COMPILERBUFFER](#) 20000
- #define [MAXPATCHES](#) 256
- #define [MAXFPATCHES](#) 256
- #define [SORTIOSIZE](#) 200000L
- #define [SSMALLBUFFER](#) 2560016L
- #define [SSMALLOVERFLOW](#) 3840032L
- #define [STERMSSMALL](#) 10000L
- #define [SLARGEBUFFER](#) 26880512L
- #define [SMAXPATCHES](#) 64
- #define [SMAXFPATCHES](#) 64
- #define [SSORTIOSIZE](#) 32768L
- #define [SPECTATORSIZE](#) 1048576L
- #define [MAXFLEVELS](#) 30
- #define [COMPINC](#) 2
- #define [MAXNUMSIZE](#) 10
- #define [MAXBRACKETBUFFERSIZE](#) 200000
- #define [SFHSIZE](#) 40
- #define [DEFAULTPROCESSBUCKETSIZE](#) 1000
- #define [SHMWINSIZE](#) 65536L
- #define [TABLEEXTENSION](#) 6
- #define [GZIPDEFAULT](#) 6
- #define [DEFAULTTTHREADS](#) 0
- #define [DEFAULTTTHREADBUCKETSIZE](#) 500
- #define [DEFAULTTTHREADLOADBALANCING](#) 1
- #define [THREADSCRATCHSIZE](#) 100000L
- #define [THREADSCRATCHOUTSIZE](#) 2500000L
- #define [NUMSTORECACHES](#) 4
- #define [SIZESTORECACHE](#) 32768
- #define [INDENTSPACE](#) 3
- #define [MULTIINDENTSPACE](#) 1
- #define [MAXMULTIBRACKETLEVELS](#) 25
- #define [FBUFFERSIZE](#) 1026
- #define [NPAIR1](#) 38
- #define [NPAIR2](#) 89
- #define [MAXLINELENGTH](#) 256

- `#define MINALLOC 32`
- `#define JUMPRATIO 4`
- `#define MAXPARTICLES 20`
- `#define MAXCOUPLINGS 20`
- `#define NUMOPTIONS 20`
- `#define MAXLEGS 20`

4.49.1 Detailed Description

The definition the default values for certain buffer sizes etc.

Definition in file [fsizes.h](#).

4.49.2 Macro Definition Documentation

4.49.2.1 MAXPRENAMESIZE

```
#define MAXPRENAMESIZE 128
```

Definition at line 36 of file [fsizes.h](#).

4.49.2.2 MAXENAME

```
#define MAXENAME 16
```

Definition at line 73 of file [fsizes.h](#).

4.49.2.3 MAXSAVEFUNCTION

```
#define MAXSAVEFUNCTION 16384
```

Definition at line 74 of file [fsizes.h](#).

4.49.2.4 MAXPARLEVEL

```
#define MAXPARLEVEL 100
```

Definition at line 76 of file [fsizes.h](#).

4.49.2.5 MAXNUMBERSIZE

```
#define MAXNUMBERSIZE 200
```

Definition at line 77 of file [fsizes.h](#).

4.49.2.6 MAXREPEAT

```
#define MAXREPEAT 100
```

Definition at line 79 of file [fsizes.h](#).

4.49.2.7 NORMSIZE

```
#define NORMSIZE 1000
```

Definition at line 80 of file [fsizes.h](#).

4.49.2.8 INITNODESIZE

```
#define INITNODESIZE 10
```

Definition at line 82 of file [fsizes.h](#).

4.49.2.9 INITNAME SIZE

```
#define INITNAME SIZE 100
```

Definition at line 83 of file [fsizes.h](#).

4.49.2.10 NUMFIXED

```
#define NUMFIXED 128
```

Definition at line 85 of file [fsizes.h](#).

4.49.2.11 MAXNEST

```
#define MAXNEST 100
```

Definition at line 86 of file [fsizes.h](#).

4.49.2.12 MAXMATCH

```
#define MAXMATCH 30
```

Definition at line 87 of file [fsizes.h](#).

4.49.2.13 MAXIF

```
#define MAXIF 20
```

Definition at line 88 of file [fsizes.h](#).

4.49.2.14 SIZEFACS

```
#define SIZEFACS 640L
```

Definition at line 89 of file [fsizes.h](#).

4.49.2.15 NUMFACS

```
#define NUMFACS 50
```

Definition at line 90 of file [fsizes.h](#).

4.49.2.16 MAXLOOPS

```
#define MAXLOOPS 30
```

Definition at line 91 of file [fsizes.h](#).

4.49.2.17 MAXLABELS

```
#define MAXLABELS 20
```

Definition at line 92 of file [fsizes.h](#).

4.49.2.18 COMMERCIALSIZE

```
#define COMMERCIALSIZE 24
```

Definition at line 93 of file [fsizes.h](#).

4.49.2.19 MAXFLAGS

```
#define MAXFLAGS 16
```

Definition at line 94 of file [fsizes.h](#).

4.49.2.20 COMPRESSBUFFER

```
#define COMPRESSBUFFER 90000
```

Definition at line 99 of file [fsizes.h](#).

4.49.2.21 FORTRANCONTINUATIONLINES

```
#define FORTRANCONTINUATIONLINES 15
```

Definition at line 100 of file [fsizes.h](#).

4.49.2.22 MAXLEVELS

```
#define MAXLEVELS 2000
```

Definition at line 101 of file [fsizes.h](#).

4.49.2.23 MAXLHS

```
#define MAXLHS 400
```

Definition at line 102 of file [fsizes.h](#).

4.49.2.24 MAXWILDC

```
#define MAXWILDC 100
```

Definition at line 103 of file [fsizes.h](#).

4.49.2.25 NUMTABLEENTRIES

```
#define NUMTABLEENTRIES 1000
```

Definition at line 104 of file [fsizes.h](#).

4.49.2.26 COMPILERBUFFER

```
#define COMPILERBUFFER 20000
```

Definition at line 105 of file [fsizes.h](#).

4.49.2.27 MAXPATCHES

```
#define MAXPATCHES 256
```

Definition at line 131 of file [fsizes.h](#).

4.49.2.28 MAXFPATCHES

```
#define MAXFPATCHES 256
```

Definition at line 132 of file [fsizes.h](#).

4.49.2.29 SORTIOSIZE

```
#define SORTIOSIZE 200000L
```

Definition at line 133 of file [fsizes.h](#).

4.49.2.30 SSMALLBUFFER

```
#define SSMALLBUFFER 2560016L
```

Definition at line 135 of file [fsizes.h](#).

4.49.2.31 SSMALLOVERFLOW

```
#define SSMALLOVERFLOW 3840032L
```

Definition at line 136 of file [fsizes.h](#).

4.49.2.32 STERMSSMALL

```
#define STERMSSMALL 10000L
```

Definition at line 137 of file [fsizes.h](#).

4.49.2.33 SLARGEBUFFER

```
#define SLARGEBUFFER 26880512L
```

Definition at line 138 of file [fsizes.h](#).

4.49.2.34 SMAXPATCHES

```
#define SMAXPATCHES 64
```

Definition at line 139 of file [fsizes.h](#).

4.49.2.35 SMAXFPATCHES

```
#define SMAXFPATCHES 64
```

Definition at line 140 of file [fsizes.h](#).

4.49.2.36 SSORTIOSIZE

```
#define SSORTIOSIZE 32768L
```

Definition at line 141 of file [fsizes.h](#).

4.49.2.37 SPECTATORSIZE

```
#define SPECTATORSIZE 1048576L
```

Definition at line 143 of file [fsizes.h](#).

4.49.2.38 MAXFLEVELS

```
#define MAXFLEVELS 30
```

Definition at line 145 of file [fsizes.h](#).

4.49.2.39 COMPINC

```
#define COMPINC 2
```

Definition at line 147 of file [fsizes.h](#).

4.49.2.40 MAXNUMSIZE

```
#define MAXNUMSIZE 10
```

Definition at line 149 of file [fsizes.h](#).

4.49.2.41 MAXBRACKETBUFFERSIZE

```
#define MAXBRACKETBUFFERSIZE 200000
```

Definition at line 151 of file [fsizes.h](#).

4.49.2.42 SFHSIZE

```
#define SFHSIZE 40
```

Definition at line 153 of file [fsizes.h](#).

4.49.2.43 DEFAULTPROCESSBUCKETSIZE

```
#define DEFAULTPROCESSBUCKETSIZE 1000
```

Definition at line 155 of file [fsizes.h](#).

4.49.2.44 SHMWINSIZE

```
#define SHMWINSIZE 65536L
```

Definition at line 156 of file [fsizes.h](#).

4.49.2.45 TABLEEXTENSION

```
#define TABLEEXTENSION 6
```

Definition at line 158 of file [fsizes.h](#).

4.49.2.46 GZIPDEFAULT

```
#define GZIPDEFAULT 6
```

Definition at line 160 of file [fsizes.h](#).

4.49.2.47 DEFAULTTHREADS

```
#define DEFAULTTHREADS 0
```

Definition at line 161 of file [fsizes.h](#).

4.49.2.48 DEFAULTTHREADBUCKETSIZE

```
#define DEFAULTTHREADBUCKETSIZE 500
```

Definition at line 162 of file [fsizes.h](#).

4.49.2.49 DEFAULTTHREADLOADBALANCING

```
#define DEFAULTTHREADLOADBALANCING 1
```

Definition at line 163 of file [fsizes.h](#).

4.49.2.50 THREADSCRATCHSIZE

```
#define THREADSCRATCHSIZE 100000L
```

Definition at line 164 of file [fsizes.h](#).

4.49.2.51 THREADSCRATCHOUTSIZE

```
#define THREADSCRATCHOUTSIZE 2500000L
```

Definition at line 165 of file [fsizes.h](#).

4.49.2.52 NUMSTORECACHES

```
#define NUMSTORECACHES 4
```

Definition at line 177 of file [fsizes.h](#).

4.49.2.53 SIZESTORECACHE

```
#define SIZESTORECACHE 32768
```

Definition at line 178 of file [fsizes.h](#).

4.49.2.54 INDENTSPACE

```
#define INDENTSPACE 3
```

Definition at line 180 of file [fsizes.h](#).

4.49.2.55 MULTIINDENTSPACE

```
#define MULTIINDENTSPACE 1
```

Definition at line 182 of file [fsizes.h](#).

4.49.2.56 MAXMULTIBRACKETLEVELS

```
#define MAXMULTIBRACKETLEVELS 25
```

Definition at line 183 of file [fsizes.h](#).

4.49.2.57 FBUFFERSIZE

```
#define FBUFFERSIZE 1026
```

Definition at line 185 of file [fsizes.h](#).

4.49.2.58 NPAIR1

```
#define NPAIR1 38
```

Definition at line 189 of file [fsizes.h](#).

4.49.2.59 NPAIR2

```
#define NPAIR2 89
```

Definition at line 190 of file [fsizes.h](#).

4.49.2.60 MAXLINELENGTH

```
#define MAXLINELENGTH 256
```

Definition at line 192 of file [fsizes.h](#).

4.49.2.61 MINALLOC

```
#define MINALLOC 32
```

Definition at line 193 of file [fsizes.h](#).

4.49.2.62 JUMPRATIO

```
#define JUMPRATIO 4
```

Definition at line 195 of file [fsizes.h](#).

4.49.2.63 MAXPARTICLES

```
#define MAXPARTICLES 20
```

Definition at line 199 of file [fsizes.h](#).

4.49.2.64 MAXCOUPLINGS

```
#define MAXCOUPLINGS 20
```

Definition at line 200 of file [fsizes.h](#).

4.49.2.65 NUMOPTIONS

```
#define NUMOPTIONS 20
```

Definition at line 201 of file [fsizes.h](#).

4.49.2.66 MAXLEGS

```
#define MAXLEGS 20
```

Definition at line 202 of file [fsizes.h](#).

4.50 fsizes.h

[Go to the documentation of this file.](#)

```
00001
00006 /* # [ License : */
00007 /*
00008 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00009 * When using this file you are requested to refer to the publication
00010 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011 * This is considered a matter of courtesy as the development was paid
00012 * for by FOM the Dutch physics granting agency and we would like to
00013 * be able to track its scientific use to convince FOM of its value
00014 * for the community.
00015 *
00016 * This file is part of FORM.
00017 *
00018 * FORM is free software: you can redistribute it and/or modify it under the
00019 * terms of the GNU General Public License as published by the Free Software
00020 * Foundation, either version 3 of the License, or (at your option) any later
00021 * version.
00022 *
00023 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00026 * details.
00027 *
```

```

00028 *   You should have received a copy of the GNU General Public License along
00029 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00030 */
00031 /* #] License : */
00032
00033 /*
00034     First the fixed variables
00035 */
00036 #define MAXPRENAMESIZE 128
00037 /*
00038     The following variables are default sizes.  They can be changed
00039     into values read from the setup file
00040
00041     Remark (21-dec-2008 JV): WILDOFFSET*3 should be larger than WILDMASK!!!!
00042     old value was WILDOFFSET 200000100
00043     be careful with old .sav files!!!
00044 */
00045 #if BITSINWORD == 32
00046     #define MAXPOWER 500000000
00047     #define MAXVARIABLES 200000050
00048     #define MAXDOLLARVARIABLES 1000000000L
00049     #define WILDOFFSET 400000100
00050     #define MAXINNAMETREE 2000000000
00051     #define MAXDUMMIES 100000000
00052     #define WORKBUFFER 40000000
00053     #define MAXTER 40000
00054     #define HALFMAX 0x10000
00055     #define MAXSUBEXPRESSIONS 0x1FFFFFFF
00056     #define MAXFILESTREAMSIZE 1024
00057 #elif BITSINWORD == 16
00058     #define MAXPOWER 10000
00059     #define MAXVARIABLES 8050
00060     #define MAXDOLLARVARIABLES 32000
00061     #define WILDOFFSET 6100
00062     #define MAXINNAMETREE 32768
00063     #define MAXDUMMIES 1000
00064     #define WORKBUFFER 10000000
00065     #define MAXTER 10000
00066     #define HALFMAX 0x100
00067     #define MAXSUBEXPRESSIONS 0x3FFF
00068     #define MAXFILESTREAMSIZE 1048576
00069 #else
00070     #error Only 64-bit and 32-bit platforms are supported.
00071 #endif
00072
00073 #define MAXENAME 16
00074 #define MAXSAVEFUNCTION 16384
00075
00076 #define MAXPARLEVEL 100
00077 #define MAXNUMBERSIZE 200
00078
00079 #define MAXREPEAT 100
00080 #define NORMSIZE 1000
00081
00082 #define INITNODESIZE 10
00083 #define INITNAMESIZE 100
00084
00085 #define NUMFIXED 128
00086 #define MAXNEST 100
00087 #define MAXMATCH 30
00088 #define MAXIF 20
00089 #define SIZEFACS 640L
00090 #define NUMFACS 50
00091 #define MAXLOOPS 30
00092 #define MAXLABELS 20
00093 #define COMMERCIALSIZE 24
00094 #define MAXFLAGS 16
00095 /*
00096     The next quantities should still be eliminated from the program
00097     This should be together with changes in setfile!
00098 */
00099 #define COMPRESSBUFFER 90000
00100 #define FORTRANCONTINUATIONLINES 15
00101 #define MAXLEVELS 2000
00102 #define MAXLHS 400
00103 #define MAXWILDC 100
00104 #define NUMTABLEENTRIES 1000
00105 #define COMPILERBUFFER 20000
00106
00107 #if BITSINWORD == 32
00108     #ifdef WITHPTHREADS
00109         #define SMALLBUFFER 300000000L
00110         #define SMALLOVERFLOW 600000000L
00111         #define TERMSSMALL 30000000L
00112         #define LARGEBUFFER 1500000000L
00113         #define SCRATCHSIZE 500000000L
00114     #else

```

```
00115         #define SMALLBUFFER 150000000L
00116         #define SMALLOVERFLOW 300000000L
00117         #define TERMSSMALL 2000000L
00118         #define LARGEBUFFER 800000000L
00119         #define SCRATCHSIZE 500000000L
00120     #endif
00121 #elif BITSINWORD == 16
00122     #define SMALLBUFFER 100000000L
00123     #define SMALLOVERFLOW 200000000L
00124     #define TERMSSMALL 1000000L
00125     #define LARGEBUFFER 500000000L
00126     #define SCRATCHSIZE 500000000L
00127 #else
00128     #error Only 64-bit and 32-bit platforms are supported.
00129 #endif
00130
00131 #define MAXPATCHES 256
00132 #define MAXFPATCHES 256
00133 #define SORTIOSIZE 200000L
00134
00135 #define SSMALLBUFFER 2560016L
00136 #define SSMALLOVERFLOW 3840032L
00137 #define STERMSSMALL 10000L
00138 #define SLARGEBUFFER 26880512L
00139 #define SMAXPATCHES 64
00140 #define SMAXFPATCHES 64
00141 #define SSORTIOSIZE 32768L
00142
00143 #define SPECTATORSIZE 1048576L
00144
00145 #define MAXFLEVELS 30
00146
00147 #define COMPINC 2
00148
00149 #define MAXNUMSIZE 10
00150
00151 #define MAXBRACKETBUFFERSIZE 200000
00152
00153 #define SFHSIZE 40
00154
00155 #define DEFAULTPROCESSBUCKETSIZE 1000
00156 #define SHMWINSIZE 65536L
00157
00158 #define TABLEEXTENSION 6
00159
00160 #define GZIPDEFAULT 6
00161 #define DEFAULTTHREADS 0
00162 #define DEFAULTTHREADBUCKETSIZE 500
00163 #define DEFAULTTHREADLOADBALANCING 1
00164 #define THREADSCRATCHSIZE 100000L
00165 #define THREADSCRATCHOUTSIZE 2500000L
00166
00167 #if BITSINWORD == 32
00168     #define MAXTABLECOMBUF 100000000000L
00169     #define MAXCOMBUFRHS 1000000000L
00170 #elif BITSINWORD == 16
00171     #define MAXTABLECOMBUF 1000000L
00172     #define MAXCOMBUFRHS 32500L
00173 #else
00174     #error Only 64-bit and 32-bit platforms are supported.
00175 #endif
00176
00177 #define NUMSTORECACHES 4
00178 #define SIZESTORECACHE 32768
00179
00180 #define INDENTSPACE 3
00181
00182 #define MULTIINDENTSPACE 1
00183 #define MAXMULTIBRACKETLEVELS 25
00184
00185 #define FBUFFERSIZE 1026
00186 /*
00187     For the random number generator (see commentary there)
00188 */
00189 #define NPAIR1 38
00190 #define NPAIR2 89
00191
00192 #define MAXLINELENGTH 256
00193 #define MINALLOC 32
00194
00195 #define JUMPRATIO 4
00196 /*
00197     Note: MAXCOUPLINGS should be at least MAXPARTICLES/2+1
00198 */
00199 #define MAXPARTICLES 20
00200 #define MAXCOUPLINGS 20
00201 #define NUMOPTIONS 20
```

```

00202 #define MAXLEGS 20
00203
00204 #ifdef WITHFLOAT
00205 #define MAXWEIGHT 0
00206 #define DEFAULTPRECISION 1000
00207 #endif

```

4.51 ftypes.h File Reference

This graph shows which files directly or indirectly include this file:



Macros

- #define [WITHOUTERROR](#) 0
- #define [WITHERROR](#) 1
- #define [FILESTREAM](#) 0
- #define [PREVARSTREAM](#) 1
- #define [PREREADSTREAM](#) 2
- #define [PIPESTREAM](#) 3
- #define [PRECALCSTREAM](#) 4
- #define [DOLLARSTREAM](#) 5
- #define [PREREADSTREAM2](#) 6
- #define [EXTERNALCHANNELSTREAM](#) 7
- #define [PREREADSTREAM3](#) 8
- #define [REVERSEFILESTREAM](#) 9
- #define [INPUTSTREAM](#) 10
- #define [ENDOFSTREAM](#) 0xFF
- #define [ENDOFINPUT](#) 0xFF
- #define [SUBROUTINEFILE](#) 0
- #define [PROCEDUREFILE](#) 1
- #define [HEADERFILE](#) 2
- #define [SETUPFILE](#) 3
- #define [TABLEBASEFILE](#) 4
- #define [FIRSTMODULE](#) -1
- #define [GLOBALMODULE](#) 0
- #define [SORTMODULE](#) 1
- #define [STOREMODULE](#) 2
- #define [CLEARMODULE](#) 3
- #define [ENDMODULE](#) 4
- #define [POLYFUN](#) 0
- #define [NOPARALLEL_DOLLAR](#) 0x0001
- #define [NOPARALLEL_RHS](#) 0x0002
- #define [NOPARALLEL_CONVPOLY](#) 0x0004
- #define [NOPARALLEL_SPECTATOR](#) 0x0008
- #define [NOPARALLEL_USER](#) 0x0010
- #define [NOPARALLEL_TBLDOLLAR](#) 0x0100
- #define [NOPARALLEL_NPROC](#) 0x0200
- #define [PARALLELFLAG](#) 0x0000
- #define [PRENOACTION](#) 0

- #define PRERAISEAFTER 1
- #define PRELOWERAFTER 2
- #define WITHSEMICOLON 0
- #define WITHOUTSEMICOLON 1
- #define MODULEINSTACK 8
- #define EXECUTINGIF 0
- #define LOOKINGFORELSE 1
- #define LOOKINGFORENDIF 2
- #define NEWSTATEMENT 1
- #define OLDSTATEMENT 0
- #define EXECUTINGPRESWITCH 0
- #define SEARCHINGPRECASE 1
- #define SEARCHINGPREENDSWITCH 2
- #define PREPROONLY 1
- #define DUMPTOCOMPILER 2
- #define DUMPOUTTERMS 4
- #define DUMPINTERMS 8
- #define DUMPTOSORT 16
- #define DUMPTOPARALLEL 32
- #define THREADSDEBUG 64
- #define ERROROUT 0
- #define INPUTOUT 1
- #define STATSOUT 2
- #define EXPRSOUT 3
- #define WRITEOUT 4
- #define EXTERNALCHANNELOUT 5
- #define NUMERICALLOOP 0
- #define LISTEDLOOP 1
- #define ONEEXPRESSION 2
- #define PRETYPENONE 0
- #define PRETYPEIF 1
- #define PRETYPEDO 2
- #define PRETYPEPROCEDURE 3
- #define PRETYPESWITCH 4
- #define PRETYPEINSIDE 5
- #define DECLARATION 1
- #define SPECIFICATION 2
- #define DEFINITION 3
- #define STATEMENT 4
- #define TOOOUTPUT 5
- #define ATENDOFMODULE 6
- #define MIXED2 8 /* unused */
- #define MIXED 9
- #define NOAUTO 0
- #define PARTEST 1 /* parentheses test */
- #define WITHAUTO 2 /* auto-declaration */
- #define ALLVARIABLES -1
- #define SYMBOLONLY 1
- #define INDEXONLY 2
- #define VECTORONLY 4
- #define FUNCTIONONLY 8
- #define SETONLY 16
- #define EXPRESSIONONLY 32

Defines: compiler types

Type of variable found by the compiler.

- #define CDELETE -1
- #define ANYTYPE -1
- #define CSYMBOL 0
- #define CINDEX 1
- #define CVECTOR 2
- #define CFUNCTION 3
- #define CSET 4
- #define CEXPRESSION 5
- #define CDOTPRODUCT 6
- #define CNUMBER 7
- #define CSUBEXP 8
- #define CDELTA 9
- #define CDOLLAR 10
- #define CDUBIOUS 11
- #define CRANGE 12
- #define CMODEL 13
- #define CVECTOR1 21
- #define CDOUBLEDOT 22
- #define TSYMBOL -1
- #define TINDEX -2
- #define TVECTOR -3
- #define TFUNCTION -4
- #define TSET -5
- #define TEXPRESSSION -6
- #define TDOTPRODUCT -7
- #define TNUMBER -8
- #define TSUBEXP -9
- #define TDELTA -10
- #define TDOLLAR -11
- #define TDUBIOUS -12
- #define LPARENTHESIS -13
- #define RPARENTHESIS -14
- #define TWILDCARD -15
- #define TWILDARG -16
- #define TDOT -17
- #define LBRACE -18
- #define RBRACE -19
- #define TCOMMA -20
- #define TFUNOPEN -21
- #define TFUNCLOSE -22
- #define TMULTIPLY -23
- #define TDIVIDE -24
- #define TPOWER -25
- #define TPLUS -26
- #define TMINUS -27
- #define TNOT -28
- #define TENDOFIT -29
- #define TSETOPEN -30
- #define TSETCLOSE -31
- #define TGENINDEX -32
- #define TCONJUGATE -33
- #define LRPARENTHESSES -34
- #define TNUMBER1 -35
- #define TPOWER1 -36
- #define TEMPTY -37
- #define TSETNUM -38
- #define TSGAMMA -39
- #define TSETDOL -40
- #define TFLOAT -41
- #define TYPEISFUN 0
- #define TYPEISSUB 1

- #define TYPEISMYSTERY -1
- #define LHSIDEX 2
- #define LHSIDE 1
- #define RHSIDE 0
- #define FORTRANMODE 1
- #define REDUCEMODE 2
- #define MAPLEMODE 3
- #define MATHEMATICAMODE 4
- #define CMODE 5
- #define VORTRANMODE 6
- #define PFORTRANMODE 7
- #define DOUBLEFORTRANMODE 33
- #define DOUBLEPRECISIONFLAG 32
- #define NODOUBLEMASK 31
- #define QUADRUPLEFORTRANMODE 65
- #define QUADRUPLEPRECISIONFLAG 64
- #define ALLINTEGERDOUBLE 128
- #define NOQUADMASK 63
- #define NORMALFORMAT 0
- #define NOSPACEFORMAT 1
- #define ISNOTFORTRAN90 0
- #define ISFORTRAN90 1
- #define ALSOREVERSE 1
- #define CHISHOLM 2
- #define NOTRICK 16
- #define SORTLOWFIRST 0
- #define SORTHIGHFIRST 1
- #define SORTPOWERFIRST 2
- #define SORTANTIPOWER 3
- #define STATSSPLITMERGE 0
- #define STATSMERGETOFILE 1
- #define STATSPOSTSORT 2
- #define NOCHECKLOGTYPE 0
- #define CHECKLOGTYPE 1
- #define NMIN4SHIFT 4
- #define SYMBOL 1
- #define DOTPRODUCT 2
- #define VECTOR 3
- #define INDEX 4
- #define EXPRESSION 5
- #define SUBEXPRESSION 6
- #define DOLLAREXPRESSION 7
- #define SETSET 8
- #define ARGWILD 9
- #define MINVECTOR 10
- #define SETEXP 11
- #define DOLLAREXPR2 12
- #define HAAKJE0 9
- #define FUNCTION 20
- #define TMPPOLYFUN 14
- #define ARGFIELD 15
- #define SNUMBER 16
- #define LNUMBER 17
- #define HAAKJE 18
- #define DELTA 19
- #define EXPONENT 20
- #define DENOMINATOR 21
- #define SETFUNCTION 22
- #define GAMMA 23
- #define GAMMAI 24
- #define GAMMAFIVE 25
- #define GAMMASIX 26
- #define GAMMASEVEN 27
- #define SUMF1 28

- `#define SUMF2` 29
- `#define DUMFUN` 30
- `#define REPLACEMENT` 31
- `#define REVERSEFUNCTION` 32
- `#define DISTRIBUTION` 33
- `#define DELTA3` 34
- `#define DUMMYFUN` 35
- `#define DUMMYTEN` 36
- `#define LEVICIVITA` 37
- `#define FACTORIAL` 38
- `#define INVERSEFACTORIAL` 39
- `#define BINOMIAL` 40
- `#define NUMARGSFUN` 41
- `#define SIGNFUN` 42
- `#define MODFUNCTION` 43
- `#define MOD2FUNCTION` 44
- `#define MINFUNCTION` 45
- `#define MAXFUNCTION` 46
- `#define ABSFUNCTION` 47
- `#define SIGFUNCTION` 48
- `#define INTFUNCTION` 49
- `#define THETA` 50
- `#define THETA2` 51
- `#define DELTA2` 52
- `#define DELTAP` 53
- `#define BERNOULLIFUNCTION` 54
- `#define COUNTFUNCTION` 55
- `#define MATCHFUNCTION` 56
- `#define PATTERNFUNCTION` 57
- `#define TERMFUNCTION` 58
- `#define CONJUGATION` 59
- `#define ROOTFUNCTION` 60
- `#define TABLEFUNCTION` 61
- `#define FIRSTBRACKET` 62
- `#define TERMSINEXPR` 63
- `#define NUMTERMSFUN` 64
- `#define GCDFUNCTION` 65
- `#define DIVFUNCTION` 66
- `#define REMFUNCTION` 67
- `#define MAXPOWEROF` 68
- `#define MINPOWEROF` 69
- `#define TABLESTUB` 70
- `#define FACTORIN` 71
- `#define TERMSINBRACKET` 72
- `#define WILDARGFUN` 73
- `#define SQRTFUNCTION` 74
- `#define LNFUNCTION` 75
- `#define SINFUNCTION` 76
- `#define COSFUNCTION` 77
- `#define TANFUNCTION` 78
- `#define ASINFUNCTION` 79
- `#define ACOSFUNCTION` 80
- `#define ATANFUNCTION` 81
- `#define ATAN2FUNCTION` 82
- `#define SINHFUNCTION` 83
- `#define COSHFUNCTION` 84
- `#define TANHFUNCTION` 85
- `#define ASINHFUNCTION` 86
- `#define ACOSHFUNCTION` 87
- `#define ATANHFUNCTION` 88
- `#define LI2FUNCTION` 89
- `#define LINFUNCTION` 90
- `#define EXTRASYMFUN` 91
- `#define RANDOMFUNCTION` 92

- #define [RANPERM](#) 93
- #define [NUMFACTORS](#) 94
- #define [FIRSTTERM](#) 95
- #define [CONTENTTERM](#) 96
- #define [PRIMENUMBER](#) 97
- #define [EXTEUCLIDEAN](#) 98
- #define [MAKERATIONAL](#) 99
- #define [INVERSEFUNCTION](#) 100
- #define [IDFUNCTION](#) 101
- #define [PUTFIRST](#) 102
- #define [PERMUTATIONS](#) 103
- #define [PARTITIONS](#) 104
- #define [MULFUNCTION](#) 105
- #define [MOEBIUS](#) 106
- #define [TOPOLOGIES](#) 107
- #define [DIAGRAMS](#) 108
- #define [TOPO](#) 109
- #define [NODEFUNCTION](#) 110
- #define [EDGE](#) 111
- #define [SIZEOFFUNCTION](#) 112
- #define [BLOCK](#) 113
- #define [ONEPI](#) 114
- #define [PHI](#) 115
- #define [FLOATFUN](#) 116
- #define [TOFLOAT](#) 117
- #define [TORAT](#) 118
- #define [MZV](#) 119
- #define [EULER](#) 120
- #define [MZVHALF](#) 121
- #define [AGMFUNCTION](#) 122
- #define [GAMMAFUN](#) 123
- #define [EXPFUNCTION](#) 124
- #define [HPLFUNCTION](#) 125
- #define [MPLFUNCTION](#) 126
- #define [MAXBUILTINFUNCTION](#) 126
- #define [FIRSTUSERFUNCTION](#) 150
- #define [ALLFUNCTIONS](#) -1
- #define [ALLMZVFUNCTIONS](#) -2
- #define [ISYMBOL](#) 0
- #define [PISYMBOL](#) 1
- #define [COEFFSYMBOL](#) 2
- #define [NUMERATORSYMBOL](#) 3
- #define [DENOMINATORSYMBOL](#) 4
- #define [WILDARGSYMBOL](#) 5
- #define [DIMENSIONSYMBOL](#) 6
- #define [FACTORSYMBOL](#) 7
- #define [SEPARATESYMBOL](#) 8
- #define [EESYMBOL](#) 9
- #define [EMSYMBOL](#) 10
- #define [BUILTINSYMBOLS](#) 11
- #define [FIRSTUSERSYMBOL](#) 20
- #define [BUILTININDICES](#) 1
- #define [BUILTINVECTORS](#) 1
- #define [BUILTINDOLLARS](#) 1
- #define [WILDARGVECTOR](#) 0
- #define [WILDARGINDEX](#) 0
- #define [TYPEEXPRESSION](#) 0
- #define [TYPEIDNEW](#) 1
- #define [TYPEIDOLD](#) 2
- #define [TYPEOPERATION](#) 3
- #define [TYPEREPEAT](#) 4
- #define [TYPEENDREPEAT](#) 5
- #define [TYPECOUNT](#) 20
- #define [TYPEMULT](#) 21

- #define [TYPEGOTO](#) 22
- #define [TYPEDISCARD](#) 23
- #define [TYPEIF](#) 24
- #define [TYPEELSE](#) 25
- #define [TYPEELIF](#) 26
- #define [TYPEENDIF](#) 27
- #define [TYPESUM](#) 28
- #define [TYPECHISHOLM](#) 29
- #define [TYPEREVERSE](#) 30
- #define [TYPEARG](#) 31
- #define [TYPENORM](#) 32
- #define [TYPENORM2](#) 33
- #define [TYPENORM3](#) 34
- #define [TYPEEXIT](#) 35
- #define [TYPESETEXIT](#) 36
- #define [TYPEPRINT](#) 37
- #define [TYPEFPRINT](#) 38
- #define [TYPEREDEFPRE](#) 39
- #define [TYPESPLITARG](#) 40
- #define [TYPESPLITARG2](#) 41
- #define [TYPEFACTARG](#) 42
- #define [TYPEFACTARG2](#) 43
- #define [TYPETRY](#) 44
- #define [TYPEASSIGN](#) 45
- #define [TYPERENUMBER](#) 46
- #define [TYPESUMFIX](#) 47
- #define [TYPEFINDLOOP](#) 48
- #define [TYPEUNRAVEL](#) 49
- #define [TYPEADJUSTBOUNDS](#) 50
- #define [TYPEINSIDE](#) 51
- #define [TYPETERM](#) 52
- #define [TYPESORT](#) 53
- #define [TYPEDETCURDUM](#) 54
- #define [TYPEINEXPRESSION](#) 55
- #define [TYPESPLITFIRSTARG](#) 56
- #define [TYPESPLITLASTARG](#) 57
- #define [TYPEMERGE](#) 58
- #define [TYPETESTUSE](#) 59
- #define [TYPEAPPLY](#) 60
- #define [TYPEAPPLYRESET](#) 61
- #define [TYPECHAININ](#) 62
- #define [TYPECHAINOUT](#) 63
- #define [TYPENORM4](#) 64
- #define [TYPEFACTOR](#) 65
- #define [TYPEARGIMPLODE](#) 66
- #define [TYPEARGEXPLODE](#) 67
- #define [TYPEDENOMINATORS](#) 68
- #define [TYPESTUFFLE](#) 69
- #define [TYPEDROPCOEFFICIENT](#) 70
- #define [TYPETRANSFORM](#) 71
- #define [TYPETOPOLYNOMIAL](#) 72
- #define [TYPEFROMPOLYNOMIAL](#) 73
- #define [TYPEDOLOOP](#) 74
- #define [TYPEENDDOLOOP](#) 75
- #define [TYPEDROPSYMBOLS](#) 76
- #define [TYPEPUTINSIDE](#) 77
- #define [TYPETOSPECTATOR](#) 78
- #define [TYPEARGTOEXTRASymbol](#) 79
- #define [TYPECANONICALIZE](#) 80
- #define [TYPESWITCH](#) 81
- #define [TYPEENDSWITCH](#) 82
- #define [TYPESETUSERFLAG](#) 83
- #define [TYPECLEARUSERFLAG](#) 84
- #define [TYPEALLLOOPS](#) 85

- #define TYPEALLPATHS 86
- #define TAKETRACE 1
- #define CONTRACT 2
- #define RATIO 3
- #define SYMMETRIZE 4
- #define TENVEC 5
- #define SUMNUM1 6
- #define SUMNUM2 7
- #define WILDDUMMY 0
- #define SYMTONUM 1
- #define SYMTOSYM 2
- #define SYMTOSUB 3
- #define VECTOMIN 4
- #define VECTOVEC 5
- #define VECTOSUB 6
- #define INDTOIND 7
- #define INDTOSUB 8
- #define FUNTOFUN 9
- #define ARGTOARG 10
- #define ARLTOARL 11
- #define EXPTOEXP 12
- #define FROMBRAC 13
- #define FROMSET 14
- #define SETTONUM 15
- #define WILDCARDS 16
- #define SETNUMBER 17
- #define LOADDOLLAR 18
- #define NUMTONUM 20
- #define NUMTOSYM 21
- #define NUMTOIND 22
- #define NUMTOSUB 23
- #define EATTENSOR 0x2000
- #define CLEANFLAG 0
- #define DIRTYFLAG 1
- #define DIRTYSYMFLAG 2
- #define MUSTCLEANPRF 4
- #define SUBTERMUSED1 8
- #define SUBTERMUSED2 16
- #define ALLDIRTY (DIRTYFLAG|DIRTYSYMFLAG)
- #define ARGHEAD 2
- #define FUNHEAD 3
- #define SUBEXPSIZE 5
- #define EXPRHEAD 5
- #define TYPEARGHEADSIZE 6
- #define SKIP 1
- #define DROP 2
- #define HIDE 3
- #define UNHIDE 4
- #define INTOHIDE 5
- #define LOCALEXPRESSSION 0
- #define SKIPLEXPRESSSION 1
- #define DROPLEXPRESSSION 2
- #define DROPPEDLEXPRESSSION 3
- #define GLOBALEXPRESSSION 4
- #define SKIPGEXPRESSSION 5
- #define DROPGEXPRESSSION 6
- #define STOREDLEXPRESSSION 8
- #define HIDDENLEXPRESSSION 9
- #define HIDDENGEXPRESSSION 13
- #define INCLEXPRESSSION 9
- #define HIDELEXPRESSSION 10
- #define HIDEGEXPRESSSION 14
- #define DROPHLEXPRESSSION 11
- #define DROPHGEXPRESSSION 15

- #define UNHIDELEXPRESSION 12
- #define UNHIDEGEXPRESSION 16
- #define INTOHIDELEXPRESSION 17
- #define INTOHIDEGEXPRESSION 18
- #define SPECTATOREXPRESSION 19
- #define DROPSPECTATOREXPRESSION 20
- #define SKIPUNHIDELEXPRESSION 21
- #define SKIPUNHIDEGEXPRESSION 22
- #define PRINTOFF 0
- #define PRINTON 1
- #define PRINTCONTENTS 2
- #define PRINTCONTENT 3
- #define PRINTLFILE 4
- #define PRINTONETERM 8
- #define PRINTONEFUNCTION 16
- #define PRINTALL 32
- #define REGULAREXPRESSION -1
- #define REDEFINEDEXPRESSION -2
- #define NEWLYDEFINEDEXPRESSION -3

Defines: function specs

Function specifications.

- #define VERTEXFUNCTION -1
- #define GENERALFUNCTION 0
- #define TENSORFUNCTION 2
- #define GAMMAFUNCTION 4
- #define POS_ 0 /* integer > 0 */
- #define POS0_ 1 /* integer >= 0 */
- #define NEG_ 2 /* integer < 0 */
- #define NEG0_ 3 /* integer <= 0 */
- #define EVEN_ 4 /* integer (even) */
- #define ODD_ 5 /* integer (odd) */
- #define Z_ 6 /* integer */
- #define SYMBOL_ 7 /* symbol only */
- #define FIXED_ 8 /* fixed index */
- #define INDEX_ 9 /* index only */
- #define Q_ 10 /* rational */
- #define DUMMYINDEX_ 11 /* dummy index only */
- #define VECTOR_ 12 /* vector only */
- #define GAMMA1 0
- #define GAMMA5 -1
- #define GAMMA6 -2
- #define GAMMA7 -3
- #define FUNNYVEC -4
- #define FUNNYWILD -5
- #define SUMMEDIND -6
- #define NOINDEX -7
- #define FUNNYDOLLAR -8
- #define EMPTYINDEX -9
- #define MINSPEC -10
- #define USEDFLAG 2
- #define DUMMYFLAG 1
- #define MAINSORT 0
- #define FUNCTIONSORT 1
- #define SUBSORT 2
- #define FLOATMODE 1
- #define RATIONALMODE 0
- #define NUMSPECSETS 10
- #define ISZERO 1
- #define ISUNMODIFIED 2
- #define ISCOMPRESSED 4
- #define ISINRHS 8

- #define ISFACTORIZED 16
- #define TOBEFACTORED 32
- #define TOBEUNFACTORED 64
- #define KEEPZERO 128
- #define VARTYPENONE 0
- #define VARTYPECOMPLEX 1
- #define VARTYPEIMAGINARY 2
- #define VARTYPEROOTOFUNITY 4
- #define VARTYPEMINUS 8
- #define CYCLESYMMETRIC 1
- #define RCYCLESYMMETRIC 2
- #define SYMMETRIC 3
- #define ANTISYMMETRIC 4
- #define REVERSEORDER 256
- #define SUBMULTI 1
- #define SUBONCE 2
- #define SUBONLY 3
- #define SUBMANY 4
- #define SUBVECTOR 5
- #define SUBSELECT 6
- #define SUBALL 7
- #define SUBMASK 15
- #define SUBDISORDER 16
- #define SUBAFTER 32
- #define SUBAFTERNOT 64
- #define IDHEAD 6
- #define DOLLARFLAG 1
- #define NORMALIZEFLAG 2
- #define GIDENT 1
- #define GFIVE 4
- #define GPLUS 3
- #define GMINUS 2
- #define LONGNUMBER 1
- #define MATCH 2
- #define COEFFI 3
- #define SUBEXPR 4
- #define MULTIPLEOF 5
- #define IFDOLLAR 6
- #define IFEXPRESSION 7
- #define IFDOLLAREXTRA 8
- #define IFISFACTORIZED 9
- #define IFOCCURS 10
- #define IFUSERFLAG 11
- #define IFFLOATNUMBER 12
- #define GREATER 0
- #define GREATEREQUAL 1
- #define LESS 2
- #define LESSEQUAL 3
- #define EQUAL 4
- #define NOTEQUAL 5
- #define ORCOND 6
- #define ANDCOND 7
- #define DUMMY 1
- #define SORT 1
- #define STORE 2
- #define END 3
- #define GLOBAL 4
- #define CLEAR 5
- #define VECTBIT 1
- #define DOTPBIT 2
- #define FUNBIT 4
- #define SETBIT 8
- #define EXTRAPARAMETER 0x4000
- #define GENCOMMUTE 0

- #define GENNONCOMMUTE 0x2000
- #define NAMENOTFOUND -9
- #define DOLUNDEFINED 0
- #define DOLNUMBER 1
- #define DOLARGUMENT 2
- #define DOLSUBTERM 3
- #define DOLTERMS 4
- #define DOLWILDARGS 5
- #define DOLINDEX 6
- #define DOLZERO 7
- #define FINDLOOP 0
- #define REPLACELOOP 1
- #define NOFUNPOWERS 0
- #define COMFUNPOWERS 1
- #define ALLFUNPOWERS 2
- #define PROPERORDERFLAG 0
- #define REGULAR 0
- #define FINISH 1
- #define POLYADD 1
- #define POLYSUB 2
- #define POLYMUL 3
- #define POLYDIV 4
- #define POLYREM 5
- #define POLYGCD 6
- #define POLYINTFAC 7
- #define POLYNORM 8
- #define MODNONE 0
- #define MODSUM 1
- #define MODMAX 2
- #define MODMIN 3
- #define MODLOCAL 4
- #define ELEMENTUSED 1
- #define ELEMENTLOADED 2
- #define POSNEG 0x1
- #define INVERSETABLE 0x2
- #define COEFFICIENTSONLY 0x4
- #define ALSOPOWERS 0x8
- #define ALSOFUNARGS 0x10
- #define ALSODOLLARS 0x20
- #define NOINVERSES 0x40
- #define POSITIVEONLY 0
- #define UNPACK 0x80
- #define NOUNPACK 0
- #define FROMFUNCTION 0x100
- #define VARNAMES 0
- #define AUTONAMES 1
- #define EXPRNAMES 2
- #define DOLLARNAMES 3
- #define DUMMYBUFFER 1
- #define ALLARGS 1
- #define NUMARG 2
- #define ARGRANGE 3
- #define MAKEARGS 4
- #define MAXRANGEINDICATOR 4
- #define REPLACEARG 5
- #define ENCODEARG 6
- #define DECODEARG 7
- #define IMplodeARG 8
- #define EXPLODEARG 9
- #define PERMUTEARG 10
- #define REVERSEARG 11
- #define CYCLEARG 12
- #define ISLYNDON 13
- #define ISLYNDONR 14

- #define TOLYNDON 15
- #define TOLYNDONR 16
- #define ADDARG 17
- #define MULTIPLYARG 18
- #define DROPARG 19
- #define SELECTARG 20
- #define DEDUPARG 21
- #define ZTOHARG 22
- #define HTOZARG 23
- #define BASECODE 1
- #define YESLYNDON 1
- #define NOLYNDON 2
- #define TOPOLYNOMIALFLAG 1
- #define FACTARGFLAG 2
- #define OLDFACTARG 1
- #define NEWFACTARG 0
- #define FROMMODULEOPTION 0
- #define FROMPOINTINSTRUCTION 1
- #define EXTRASYMBOL 0
- #define REGULARSYMBOL 1
- #define EXPRESSIONNUMBER 2
- #define O_NONE 0
- #define O_CSE 1
- #define O_CSEGREEDY 2
- #define O_GREEDY 3
- #define O_OCCURRENCE 0
- #define O_MCTS 1
- #define O_SIMULATED_ANNEALING 2
- #define O_FORWARD 0
- #define O_BACKWARD 1
- #define O_FORWARDORBACKWARD 2
- #define O_FORWARDANDBACKWARD 3
- #define OPTHEAD 3
- #define DOALL 1
- #define ONLYFUNCTIONS 2
- #define INUSE 1
- #define COULDCOMMUTE 2
- #define DOESNOTCOMMUTE 4
- #define DICT_NONUMBERS 0
- #define DICT_INTEGERONLY 1
- #define DICT_RATIONALONLY 2
- #define DICT_ALLNUMBERS 3
- #define DICT_NOVARIABLES 0
- #define DICT_DOVARIABLES 1
- #define DICT_NOSPECIALS 0
- #define DICT_DOSPECIALS 1
- #define DICT_NOFUNWITHARGS 0
- #define DICT_DOFUNWITHARGS 1
- #define DICT_NOTINDOLLARS 0
- #define DICT_INDOLLARS 1
- #define DICT_INTEGERNUMBER 1
- #define DICT_RATIONALNUMBER 2
- #define DICT_SYMBOL 3
- #define DICT_VECTOR 4
- #define DICT_INDEX 5
- #define DICT_FUNCTION 6
- #define DICT_FUNCTION_WITH_ARGUMENTS 7
- #define DICT_SPECIALCHARACTER 8
- #define DICT_RANGE 9
- #define READSPECTATORFLAG 3
- #define GLOBALSPECTATORFLAG 1
- #define ORDEREDSET 1
- #define DENSETABLE 1
- #define SPARSETABLE 0

- `#define TOPOLOGIESONLY 1`
- `#define WITHOUTNODES 2`
- `#define WITHEDGES 4`
- `#define WITHBLOCKS 8`
- `#define WITHONEPISETS 16`
- `#define WITHSYMMETRIZEI 32`
- `#define WITHSYMMETRIZEF 64`
- `#define CHECKEXTERN 128`
- `#define ONEPARTI 256`
- `#define ONEPARTR 512`
- `#define ONSHELL 1024`
- `#define OFFSHELL 2048`
- `#define NOSIGMA 4096`
- `#define SIGMA 8192`
- `#define NOSNAIL 16384`
- `#define SNAIL 32768`
- `#define NOTADPOLE 65536`
- `#define TADPOLE 131072`
- `#define SIMPLE 262144`
- `#define NOTSIMPLE 524288`
- `#define BIPART 1048576`
- `#define NONBIPART 2097152`
- `#define CYCLI 4194304`
- `#define CYCLR 8388608`
- `#define FLOOP 16777216`
- `#define NOTFLOOP 33554432`

Typedefs

- `typedef void(* PVFUNWP) (WORD *)`
- `typedef void(* PVFUNV) (void)`
- `typedef int(* CFUN) (void)`
- `typedef int(* TFUN) (UBYTE *)`
- `typedef int(* TFUN1) (UBYTE *, int)`

4.51.1 Detailed Description

Contains the definitions of many internal codes Rather than using numbers directly we do this by defines, making it much easier to change things. Changing things is sometimes also a good way of testing the code.

Definition in file [ftypes.h](#).

4.51.2 Macro Definition Documentation

4.51.2.1 WITHOUTERROR

```
#define WITHOUTERROR 0
```

The next macros were introduced when TFORM was programmed. In the case of workers, each worker may need some private data. These can in principle be accessed by some posix calls but that is unnecessarily slow. The passing of a pointer to the complete data struct with private data will be much faster. And anyway, there would have to be a macro that either makes the posix call (TFORM) or doesn't (FORM). The solution by having macro's that either pass the pointer (TFORM) or don't pass it (FORM) is seen as the best solution.

In the declarations and the calling of the functions we have to use the PHEAD or the BHEAD macro, respectively, if the pointer is to be passed. These macro's contain the comma as well. Hence we need special macro's if there are no other arguments. These are called PHEAD0 and BHEAD0.

Definition at line 51 of file [ftypes.h](#).

4.51.2.2 WITHERROR

```
#define WITHERROR 1
```

Definition at line 52 of file [ftypes.h](#).

4.51.2.3 FILESTREAM

```
#define FILESTREAM 0
```

Definition at line 58 of file [ftypes.h](#).

4.51.2.4 PREVARSTREAM

```
#define PREVARSTREAM 1
```

Definition at line 59 of file [ftypes.h](#).

4.51.2.5 PREREADSTREAM

```
#define PREREADSTREAM 2
```

Definition at line 60 of file [ftypes.h](#).

4.51.2.6 PIPESTREAM

```
#define PIPESTREAM 3
```

Definition at line 61 of file [ftypes.h](#).

4.51.2.7 PRECALCSTREAM

```
#define PRECALCSTREAM 4
```

Definition at line 62 of file [ftypes.h](#).

4.51.2.8 DOLLARSTREAM

```
#define DOLLARSTREAM 5
```

Definition at line 63 of file [ftypes.h](#).

4.51.2.9 PREREADSTREAM2

```
#define PREREADSTREAM2 6
```

Definition at line 64 of file [ftypes.h](#).

4.51.2.10 EXTERNALCHANNELSTREAM

```
#define EXTERNALCHANNELSTREAM 7
```

Definition at line 65 of file [ftypes.h](#).

4.51.2.11 PREREADSTREAM3

```
#define PREREADSTREAM3 8
```

Definition at line 66 of file [ftypes.h](#).

4.51.2.12 REVERSEFILESTREAM

```
#define REVERSEFILESTREAM 9
```

Definition at line 67 of file [ftypes.h](#).

4.51.2.13 INPUTSTREAM

```
#define INPUTSTREAM 10
```

Definition at line 68 of file [ftypes.h](#).

4.51.2.14 ENDOFSTREAM

```
#define ENDOFSTREAM 0xFF
```

Definition at line 70 of file [ftypes.h](#).

4.51.2.15 ENDOFINPUT

```
#define ENDOFINPUT 0xFF
```

Definition at line 71 of file [ftypes.h](#).

4.51.2.16 SUBROUTINEFILE

```
#define SUBROUTINEFILE 0
```

Definition at line 77 of file [ftypes.h](#).

4.51.2.17 PROCEDUREFILE

```
#define PROCEDUREFILE 1
```

Definition at line 78 of file [ftypes.h](#).

4.51.2.18 HEADERFILE

```
#define HEADERFILE 2
```

Definition at line 79 of file [ftypes.h](#).

4.51.2.19 SETUPFILE

```
#define SETUPFILE 3
```

Definition at line 80 of file [ftypes.h](#).

4.51.2.20 TABLEBASEFILE

```
#define TABLEBASEFILE 4
```

Definition at line 81 of file [ftypes.h](#).

4.51.2.21 FIRSTMODULE

```
#define FIRSTMODULE -1
```

Definition at line 87 of file [ftypes.h](#).

4.51.2.22 GLOBALMODULE

```
#define GLOBALMODULE 0
```

Definition at line 88 of file [ftypes.h](#).

4.51.2.23 SORTMODULE

```
#define SORTMODULE 1
```

Definition at line 89 of file [ftypes.h](#).

4.51.2.24 STOREMODULE

```
#define STOREMODULE 2
```

Definition at line 90 of file [ftypes.h](#).

4.51.2.25 CLEARMODULE

```
#define CLEARMODULE 3
```

Definition at line 91 of file [ftypes.h](#).

4.51.2.26 ENDMODULE

```
#define ENDMODULE 4
```

Definition at line 92 of file [ftypes.h](#).

4.51.2.27 POLYFUN

```
#define POLYFUN 0
```

Definition at line 94 of file [ftypes.h](#).

4.51.2.28 NOPARALLEL_DOLLAR

```
#define NOPARALLEL_DOLLAR 0x0001
```

Definition at line 96 of file [ftypes.h](#).

4.51.2.29 NOPARALLEL_RHS

```
#define NOPARALLEL_RHS 0x0002
```

Definition at line 97 of file [ftypes.h](#).

4.51.2.30 NOPARALLEL_CONVPOLY

```
#define NOPARALLEL_CONVPOLY 0x0004
```

Definition at line 98 of file [ftypes.h](#).

4.51.2.31 NOPARALLEL_SPECTATOR

```
#define NOPARALLEL_SPECTATOR 0x0008
```

Definition at line 99 of file [ftypes.h](#).

4.51.2.32 NOPARALLEL_USER

```
#define NOPARALLEL_USER 0x0010
```

Definition at line 100 of file [ftypes.h](#).

4.51.2.33 NOPARALLEL_TBLDOLLAR

```
#define NOPARALLEL_TBLDOLLAR 0x0100
```

Definition at line 101 of file [ftypes.h](#).

4.51.2.34 NOPARALLEL_NPROC

```
#define NOPARALLEL_NPROC 0x0200
```

Definition at line 102 of file [ftypes.h](#).

4.51.2.35 PARALLELFLAG

```
#define PARALLELFLAG 0x0000
```

Definition at line 103 of file [ftypes.h](#).

4.51.2.36 PRENOACTION

```
#define PRENOACTION 0
```

Definition at line 105 of file [ftypes.h](#).

4.51.2.37 PRERAISEAFTER

```
#define PRERAISEAFTER 1
```

Definition at line 106 of file [ftypes.h](#).

4.51.2.38 PRELOWERAFTER

```
#define PRELOWERAFTER 2
```

Definition at line 107 of file [ftypes.h](#).

4.51.2.39 WITHSEMICOLON

```
#define WITHSEMICOLON 0
```

Definition at line 114 of file [ftypes.h](#).

4.51.2.40 WITHOUTSEMICOLON

```
#define WITHOUTSEMICOLON 1
```

Definition at line 115 of file [ftypes.h](#).

4.51.2.41 MODULEINSTACK

```
#define MODULEINSTACK 8
```

Definition at line 116 of file [ftypes.h](#).

4.51.2.42 EXECUTINGIF

```
#define EXECUTINGIF 0
```

Definition at line 117 of file [ftypes.h](#).

4.51.2.43 LOOKINGFORELSE

```
#define LOOKINGFORELSE 1
```

Definition at line 118 of file [ftypes.h](#).

4.51.2.44 LOOKINGFORENDIF

```
#define LOOKINGFORENDIF 2
```

Definition at line 119 of file [ftypes.h](#).

4.51.2.45 NEWSTATEMENT

```
#define NEWSTATEMENT 1
```

Definition at line 120 of file [ftypes.h](#).

4.51.2.46 OLDSTATEMENT

```
#define OLDSTATEMENT 0
```

Definition at line 121 of file [ftypes.h](#).

4.51.2.47 EXECUTINGPRESWITCH

```
#define EXECUTINGPRESWITCH 0
```

Definition at line 123 of file [ftypes.h](#).

4.51.2.48 SEARCHINGPRECASE

```
#define SEARCHINGPRECASE 1
```

Definition at line 124 of file [ftypes.h](#).

4.51.2.49 SEARCHINGPREENDSWITCH

```
#define SEARCHINGPREENDSWITCH 2
```

Definition at line 125 of file [ftypes.h](#).

4.51.2.50 PREPROONLY

```
#define PREPROONLY 1
```

Definition at line 127 of file [ftypes.h](#).

4.51.2.51 DUMPTOCOMPILER

```
#define DUMPTOCOMPILER 2
```

Definition at line 128 of file [ftypes.h](#).

4.51.2.52 DUMPOUTTERMS

```
#define DUMPOUTTERMS 4
```

Definition at line 129 of file [ftypes.h](#).

4.51.2.53 DUMPINTERMS

```
#define DUMPINTERMS 8
```

Definition at line 130 of file [ftypes.h](#).

4.51.2.54 DUMPTOSORT

```
#define DUMPTOSORT 16
```

Definition at line 131 of file [ftypes.h](#).

4.51.2.55 DUMPTOPARALLEL

```
#define DUMPTOPARALLEL 32
```

Definition at line 132 of file [ftypes.h](#).

4.51.2.56 THREADSDEBUG

```
#define THREADSDEBUG 64
```

Definition at line 133 of file [ftypes.h](#).

4.51.2.57 ERROROUT

```
#define ERROROUT 0
```

Definition at line 135 of file [ftypes.h](#).

4.51.2.58 INPUTOUT

```
#define INPUTOUT 1
```

Definition at line 136 of file [ftypes.h](#).

4.51.2.59 STATSOUT

```
#define STATSOUT 2
```

Definition at line 137 of file [ftypes.h](#).

4.51.2.60 EXPRSOUT

```
#define EXPRSOUT 3
```

Definition at line 138 of file [ftypes.h](#).

4.51.2.61 WRITEOUT

```
#define WRITEOUT 4
```

Definition at line 139 of file [ftypes.h](#).

4.51.2.62 EXTERNALCHANNELOUT

```
#define EXTERNALCHANNELOUT 5
```

Definition at line 141 of file [ftypes.h](#).

4.51.2.63 NUMERICALLOOP

```
#define NUMERICALLOOP 0
```

Definition at line 143 of file [ftypes.h](#).

4.51.2.64 LISTEDLOOP

```
#define LISTEDLOOP 1
```

Definition at line 144 of file [ftypes.h](#).

4.51.2.65 ONEEXPRESSION

```
#define ONEEXPRESSION 2
```

Definition at line 145 of file [ftypes.h](#).

4.51.2.66 PRETYPENONE

```
#define PRETYPENONE 0
```

Definition at line 147 of file [ftypes.h](#).

4.51.2.67 PRETYPEIF

```
#define PRETYPEIF 1
```

Definition at line 148 of file [ftypes.h](#).

4.51.2.68 PRETYPEDO

```
#define PRETYPEDO 2
```

Definition at line 149 of file [ftypes.h](#).

4.51.2.69 PRETYPEPROCEDURE

```
#define PRETYPEPROCEDURE 3
```

Definition at line 150 of file [ftypes.h](#).

4.51.2.70 PRETYPESWITCH

```
#define PRETYPESWITCH 4
```

Definition at line 151 of file [ftypes.h](#).

4.51.2.71 PRETYPEINSIDE

```
#define PRETYPEINSIDE 5
```

Definition at line 152 of file [ftypes.h](#).

4.51.2.72 DECLARATION

```
#define DECLARATION 1
```

Definition at line 158 of file [ftypes.h](#).

4.51.2.73 SPECIFICATION

```
#define SPECIFICATION 2
```

Definition at line 159 of file [ftypes.h](#).

4.51.2.74 DEFINITION

```
#define DEFINITION 3
```

Definition at line 160 of file [ftypes.h](#).

4.51.2.75 STATEMENT

```
#define STATEMENT 4
```

Definition at line 161 of file [ftypes.h](#).

4.51.2.76 TOOOUTPUT

```
#define TOOOUTPUT 5
```

Definition at line 162 of file [ftypes.h](#).

4.51.2.77 ATENDOFMODULE

```
#define ATENDOFMODULE 6
```

Definition at line 163 of file [ftypes.h](#).

4.51.2.78 MIXED2

```
#define MIXED2 8 /* unused */
```

Definition at line 164 of file [ftypes.h](#).

4.51.2.79 MIXED

```
#define MIXED 9
```

Definition at line 165 of file [ftypes.h](#).

4.51.2.80 NOAUTO

```
#define NOAUTO 0
```

Definition at line 181 of file [ftypes.h](#).

4.51.2.81 PARTEST

```
#define PARTEST 1 /* parentheses test */
```

Definition at line 182 of file [ftypes.h](#).

4.51.2.82 WITHAUTO

```
#define WITHAUTO 2 /* auto-declaration */
```

Definition at line 183 of file [ftypes.h](#).

4.51.2.83 ALLVARIABLES

```
#define ALLVARIABLES -1
```

Definition at line 185 of file [ftypes.h](#).

4.51.2.84 SYMBOLONLY

```
#define SYMBOLONLY 1
```

Definition at line 186 of file [ftypes.h](#).

4.51.2.85 INDEXONLY

```
#define INDEXONLY 2
```

Definition at line 187 of file [ftypes.h](#).

4.51.2.86 VECTORONLY

```
#define VECTORONLY 4
```

Definition at line 188 of file [ftypes.h](#).

4.51.2.87 FUNCTIONONLY

```
#define FUNCTIONONLY 8
```

Definition at line 189 of file [ftypes.h](#).

4.51.2.88 SETONLY

```
#define SETONLY 16
```

Definition at line 190 of file [ftypes.h](#).

4.51.2.89 EXPRESSIONONLY

```
#define EXPRESSIONONLY 32
```

Definition at line 191 of file [ftypes.h](#).

4.51.2.90 CDELETE

```
#define CDELETE -1
```

Definition at line 202 of file [ftypes.h](#).

4.51.2.91 ANYTYPE

```
#define ANYTYPE -1
```

Definition at line 203 of file [ftypes.h](#).

4.51.2.92 CSYMBOL

```
#define CSYMBOL 0
```

Definition at line 204 of file [ftypes.h](#).

4.51.2.93 CINDEX

```
#define CINDEX 1
```

Definition at line 205 of file [ftypes.h](#).

4.51.2.94 CVECTOR

```
#define CVECTOR 2
```

Definition at line 206 of file [ftypes.h](#).

4.51.2.95 CFUNCTION

```
#define CFUNCTION 3
```

Definition at line 207 of file [ftypes.h](#).

4.51.2.96 CSET

```
#define CSET 4
```

Definition at line 208 of file [ftypes.h](#).

4.51.2.97 CEXPRESSION

```
#define CEXPRESSION 5
```

Definition at line 209 of file [ftypes.h](#).

4.51.2.98 CDOTPRODUCT

```
#define CDOTPRODUCT 6
```

Definition at line 210 of file [ftypes.h](#).

4.51.2.99 CNUMBER

```
#define CNUMBER 7
```

Definition at line 211 of file [ftypes.h](#).

4.51.2.100 CSUBEXP

```
#define CSUBEXP 8
```

Definition at line 212 of file [ftypes.h](#).

4.51.2.101 CDELTA

```
#define CDELTA 9
```

Definition at line 213 of file [ftypes.h](#).

4.51.2.102 CDOLLAR

```
#define CDOLLAR 10
```

Definition at line 214 of file [ftypes.h](#).

4.51.2.103 CDUBIOUS

```
#define CDUBIOUS 11
```

Definition at line 215 of file [ftypes.h](#).

4.51.2.104 CRANGE

```
#define CRANGE 12
```

Definition at line 216 of file [ftypes.h](#).

4.51.2.105 CMODEL

```
#define CMODEL 13
```

Definition at line 217 of file [ftypes.h](#).

4.51.2.106 CVECTOR1

```
#define CVECTOR1 21
```

Definition at line 218 of file [ftypes.h](#).

4.51.2.107 CDOUBLEDOT

```
#define CDOUBLEDOT 22
```

Definition at line 219 of file [ftypes.h](#).

4.51.2.108 TSYMBOL

```
#define TSYMBOL -1
```

Definition at line 227 of file [ftypes.h](#).

4.51.2.109 TINDEX

```
#define TINDEX -2
```

Definition at line 228 of file [ftypes.h](#).

4.51.2.110 TVECTOR

```
#define TVECTOR -3
```

Definition at line 229 of file [ftypes.h](#).

4.51.2.111 TFUNCTION

```
#define TFUNCTION -4
```

Definition at line 230 of file [ftypes.h](#).

4.51.2.112 TSET

```
#define TSET -5
```

Definition at line 231 of file [ftypes.h](#).

4.51.2.113 TEXPRESSSION

```
#define TEXPRESSSION -6
```

Definition at line 232 of file [ftypes.h](#).

4.51.2.114 TDOTPRODUCT

```
#define TDOTPRODUCT -7
```

Definition at line 233 of file [ftypes.h](#).

4.51.2.115 TNUMBER

```
#define TNUMBER -8
```

Definition at line 234 of file [ftypes.h](#).

4.51.2.116 TSUBEXP

```
#define TSUBEXP -9
```

Definition at line 235 of file [ftypes.h](#).

4.51.2.117 TDELTA

```
#define TDELTA -10
```

Definition at line 236 of file [ftypes.h](#).

4.51.2.118 TDOLLAR

```
#define TDOLLAR -11
```

Definition at line 237 of file [ftypes.h](#).

4.51.2.119 TDUBIOUS

```
#define TDUBIOUS -12
```

Definition at line 238 of file [ftypes.h](#).

4.51.2.120 LPARENTHESIS

```
#define LPARENTHESIS -13
```

Definition at line 239 of file [ftypes.h](#).

4.51.2.121 RPARENTHESIS

```
#define RPARENTHESIS -14
```

Definition at line 240 of file [ftypes.h](#).

4.51.2.122 TWILDCARD

```
#define TWILDCARD -15
```

Definition at line 241 of file [ftypes.h](#).

4.51.2.123 TWILDARG

```
#define TWILDARG -16
```

Definition at line 242 of file [ftypes.h](#).

4.51.2.124 TDOT

```
#define TDOT -17
```

Definition at line 243 of file [ftypes.h](#).

4.51.2.125 LBRACE

```
#define LBRACE -18
```

Definition at line 244 of file [ftypes.h](#).

4.51.2.126 RBRACE

```
#define RBRACE -19
```

Definition at line 245 of file [ftypes.h](#).

4.51.2.127 TCOMMA

```
#define TCOMMA -20
```

Definition at line 246 of file [ftypes.h](#).

4.51.2.128 TFUNOPEN

```
#define TFUNOPEN -21
```

Definition at line 247 of file [ftypes.h](#).

4.51.2.129 TFUNCLOSE

```
#define TFUNCLOSE -22
```

Definition at line 248 of file [ftypes.h](#).

4.51.2.130 TMULTIPLY

```
#define TMULTIPLY -23
```

Definition at line 249 of file [ftypes.h](#).

4.51.2.131 TDIVIDE

```
#define TDIVIDE -24
```

Definition at line 250 of file [ftypes.h](#).

4.51.2.132 TPOWER

```
#define TPOWER -25
```

Definition at line 251 of file [ftypes.h](#).

4.51.2.133 TPLUS

```
#define TPLUS -26
```

Definition at line 252 of file [ftypes.h](#).

4.51.2.134 TMINUS

```
#define TMINUS -27
```

Definition at line 253 of file [ftypes.h](#).

4.51.2.135 TNOT

```
#define TNOT -28
```

Definition at line 254 of file [ftypes.h](#).

4.51.2.136 TENDOFIT

```
#define TENDOFIT -29
```

Definition at line 255 of file [ftypes.h](#).

4.51.2.137 TSETOPEN

```
#define TSETOPEN -30
```

Definition at line 256 of file [ftypes.h](#).

4.51.2.138 TSETCLOSE

```
#define TSETCLOSE -31
```

Definition at line 257 of file [ftypes.h](#).

4.51.2.139 TGENINDEX

```
#define TGENINDEX -32
```

Definition at line 258 of file [ftypes.h](#).

4.51.2.140 TCONJUGATE

```
#define TCONJUGATE -33
```

Definition at line 259 of file [ftypes.h](#).

4.51.2.141 LRPARENTHESSES

```
#define LRPARENTHESSES -34
```

Definition at line 260 of file [ftypes.h](#).

4.51.2.142 TNUMBER1

```
#define TNUMBER1 -35
```

Definition at line 261 of file [ftypes.h](#).

4.51.2.143 TPOWER1

```
#define TPOWER1 -36
```

Definition at line 262 of file [ftypes.h](#).

4.51.2.144 TEMPTY

```
#define TEMPTY -37
```

Definition at line 263 of file [ftypes.h](#).

4.51.2.145 TSETNUM

```
#define TSETNUM -38
```

Definition at line 264 of file [ftypes.h](#).

4.51.2.146 TSGAMMA

```
#define TSGAMMA -39
```

Definition at line 265 of file [ftypes.h](#).

4.51.2.147 TSETDOL

```
#define TSETDOL -40
```

Definition at line 266 of file [ftypes.h](#).

4.51.2.148 TFLOAT

```
#define TFLOAT -41
```

Definition at line 267 of file [ftypes.h](#).

4.51.2.149 TYPEISFUN

```
#define TYPEISFUN 0
```

Definition at line 269 of file [ftypes.h](#).

4.51.2.150 TYPEISSUB

```
#define TYPEISSUB 1
```

Definition at line 270 of file [ftypes.h](#).

4.51.2.151 TYPEISMYSTERY

```
#define TYPEISMYSTERY -1
```

Definition at line 271 of file [ftypes.h](#).

4.51.2.152 LHSIDEX

```
#define LHSIDEX 2
```

Definition at line 273 of file [ftypes.h](#).

4.51.2.153 LHSIDE

```
#define LHSIDE 1
```

Definition at line 274 of file [ftypes.h](#).

4.51.2.154 RHSIDE

```
#define RHSIDE 0
```

Definition at line 275 of file [ftypes.h](#).

4.51.2.155 FORTRANMODE

```
#define FORTRANMODE 1
```

Definition at line 281 of file [ftypes.h](#).

4.51.2.156 REDUCEMODE

```
#define REDUCEMODE 2
```

Definition at line 282 of file [ftypes.h](#).

4.51.2.157 MAPLEMODE

```
#define MAPLEMODE 3
```

Definition at line 283 of file [ftypes.h](#).

4.51.2.158 MATHEMATICAMODE

```
#define MATHEMATICAMODE 4
```

Definition at line 284 of file [ftypes.h](#).

4.51.2.159 CMODE

```
#define CMODE 5
```

Definition at line 285 of file [ftypes.h](#).

4.51.2.160 VORTRANMODE

```
#define VORTRANMODE 6
```

Definition at line 286 of file [ftypes.h](#).

4.51.2.161 PFORTRANMODE

```
#define PFORTRANMODE 7
```

Definition at line 287 of file [ftypes.h](#).

4.51.2.162 DOUBLEFORTRANMODE

```
#define DOUBLEFORTRANMODE 33
```

Definition at line 288 of file [ftypes.h](#).

4.51.2.163 DOUBLEPRECISIONFLAG

```
#define DOUBLEPRECISIONFLAG 32
```

Definition at line 289 of file [ftypes.h](#).

4.51.2.164 NODOUBLEMASK

```
#define NODOUBLEMASK 31
```

Definition at line 290 of file [ftypes.h](#).

4.51.2.165 QUADRUPLEFORTRANMODE

```
#define QUADRUPLEFORTRANMODE 65
```

Definition at line 291 of file [ftypes.h](#).

4.51.2.166 QUADRUPLEPRECISIONFLAG

```
#define QUADRUPLEPRECISIONFLAG 64
```

Definition at line 292 of file [ftypes.h](#).

4.51.2.167 ALLINTEGERDOUBLE

```
#define ALLINTEGERDOUBLE 128
```

Definition at line 293 of file [ftypes.h](#).

4.51.2.168 NOQUADMASK

```
#define NOQUADMASK 63
```

Definition at line 294 of file [ftypes.h](#).

4.51.2.169 NORMALFORMAT

```
#define NORMALFORMAT 0
```

Definition at line 295 of file [ftypes.h](#).

4.51.2.170 NOSPACEFORMAT

```
#define NOSPACEFORMAT 1
```

Definition at line 296 of file [ftypes.h](#).

4.51.2.171 ISNOTFORTRAN90

```
#define ISNOTFORTRAN90 0
```

Definition at line 298 of file [ftypes.h](#).

4.51.2.172 ISFORTRAN90

```
#define ISFORTRAN90 1
```

Definition at line 299 of file [ftypes.h](#).

4.51.2.173 ALSOREVERSE

```
#define ALSOREVERSE 1
```

Definition at line 301 of file [ftypes.h](#).

4.51.2.174 CHISHOLM

```
#define CHISHOLM 2
```

Definition at line 302 of file [ftypes.h](#).

4.51.2.175 NOTRICK

```
#define NOTRICK 16
```

Definition at line 303 of file [ftypes.h](#).

4.51.2.176 SORTLOWFIRST

```
#define SORTLOWFIRST 0
```

Definition at line 305 of file [ftypes.h](#).

4.51.2.177 SORTHIGHFIRST

```
#define SORTHIGHFIRST 1
```

Definition at line 306 of file [ftypes.h](#).

4.51.2.178 SORTPOWERFIRST

```
#define SORTPOWERFIRST 2
```

Definition at line 307 of file [ftypes.h](#).

4.51.2.179 SORTANTIPOWER

```
#define SORTANTIPOWER 3
```

Definition at line 308 of file [ftypes.h](#).

4.51.2.180 STATSSPLITMERGE

```
#define STATSSPLITMERGE 0
```

Definition at line 314 of file [ftypes.h](#).

4.51.2.181 STATSMERGETOFILE

```
#define STATSMERGETOFILE 1
```

Definition at line 315 of file [ftypes.h](#).

4.51.2.182 STATSPOSTSORT

```
#define STATSPOSTSORT 2
```

Definition at line 316 of file [ftypes.h](#).

4.51.2.183 NOCHECKLOGTYPE

```
#define NOCHECKLOGTYPE 0
```

Definition at line 319 of file [ftypes.h](#).

4.51.2.184 CHECKLOGTYPE

```
#define CHECKLOGTYPE 1
```

Definition at line 320 of file [ftypes.h](#).

4.51.2.185 NMIN4SHIFT

```
#define NMIN4SHIFT 4
```

Definition at line 322 of file [ftypes.h](#).

4.51.2.186 SYMBOL

```
#define SYMBOL 1
```

Definition at line 336 of file [ftypes.h](#).

4.51.2.187 DOTPRODUCT

```
#define DOTPRODUCT 2
```

Definition at line 337 of file [ftypes.h](#).

4.51.2.188 VECTOR

```
#define VECTOR 3
```

Definition at line 338 of file [ftypes.h](#).

4.51.2.189 INDEX

```
#define INDEX 4
```

Definition at line 339 of file [ftypes.h](#).

4.51.2.190 EXPRESSION

```
#define EXPRESSION 5
```

Definition at line 340 of file [ftypes.h](#).

4.51.2.191 SUBEXPRESSION

```
#define SUBEXPRESSION 6
```

Definition at line 341 of file [ftypes.h](#).

4.51.2.192 DOLLAREXPRESSION

```
#define DOLLAREXPRESSION 7
```

Definition at line 342 of file [ftypes.h](#).

4.51.2.193 SETSET

```
#define SETSET 8
```

Definition at line 343 of file [ftypes.h](#).

4.51.2.194 ARGWILD

```
#define ARGWILD 9
```

Definition at line 344 of file [ftypes.h](#).

4.51.2.195 MINVECTOR

```
#define MINVECTOR 10
```

Definition at line 345 of file [ftypes.h](#).

4.51.2.196 SETEXP

```
#define SETEXP 11
```

Definition at line 346 of file [ftypes.h](#).

4.51.2.197 DOLLAREXP2

```
#define DOLLAREXP2 12
```

Definition at line 347 of file [ftypes.h](#).

4.51.2.198 HAAKJE0

```
#define HAAKJE0 9
```

Definition at line 348 of file [ftypes.h](#).

4.51.2.199 FUNCTION

```
#define FUNCTION 20
```

Definition at line 349 of file [ftypes.h](#).

4.51.2.200 TMPPOLYFUN

```
#define TMPPOLYFUN 14
```

Definition at line 351 of file [ftypes.h](#).

4.51.2.201 ARGFIELD

```
#define ARGFIELD 15
```

Definition at line 352 of file [ftypes.h](#).

4.51.2.202 SNUMBER

```
#define SNUMBER 16
```

Definition at line 353 of file [ftypes.h](#).

4.51.2.203 LNUMBER

```
#define LNUMBER 17
```

Definition at line 354 of file [ftypes.h](#).

4.51.2.204 HAAKJE

```
#define HAAKJE 18
```

Definition at line 355 of file [ftypes.h](#).

4.51.2.205 DELTA

```
#define DELTA 19
```

Definition at line 356 of file [ftypes.h](#).

4.51.2.206 EXPONENT

```
#define EXPONENT 20
```

Definition at line 357 of file [ftypes.h](#).

4.51.2.207 DENOMINATOR

```
#define DENOMINATOR 21
```

Definition at line 358 of file [ftypes.h](#).

4.51.2.208 SETFUNCTION

```
#define SETFUNCTION 22
```

Definition at line 359 of file [ftypes.h](#).

4.51.2.209 GAMMA

```
#define GAMMA 23
```

Definition at line 360 of file [ftypes.h](#).

4.51.2.210 GAMMAI

```
#define GAMMAI 24
```

Definition at line 361 of file [ftypes.h](#).

4.51.2.211 GAMMAFIVE

```
#define GAMMAFIVE 25
```

Definition at line 362 of file [ftypes.h](#).

4.51.2.212 GAMMASIX

```
#define GAMMASIX 26
```

Definition at line 363 of file [ftypes.h](#).

4.51.2.213 GAMMASEVEN

```
#define GAMMASEVEN 27
```

Definition at line 364 of file [ftypes.h](#).

4.51.2.214 SUMF1

```
#define SUMF1 28
```

Definition at line 365 of file [ftypes.h](#).

4.51.2.215 SUMF2

```
#define SUMF2 29
```

Definition at line 366 of file [ftypes.h](#).

4.51.2.216 DUMFUN

```
#define DUMFUN 30
```

Definition at line 367 of file [ftypes.h](#).

4.51.2.217 REPLACEMENT

```
#define REPLACEMENT 31
```

Definition at line 368 of file [ftypes.h](#).

4.51.2.218 REVERSEFUNCTION

```
#define REVERSEFUNCTION 32
```

Definition at line 369 of file [ftypes.h](#).

4.51.2.219 DISTRIBUTION

```
#define DISTRIBUTION 33
```

Definition at line 370 of file [ftypes.h](#).

4.51.2.220 DELTA3

```
#define DELTA3 34
```

Definition at line 371 of file [ftypes.h](#).

4.51.2.221 DUMMYFUN

```
#define DUMMYFUN 35
```

Definition at line 372 of file [ftypes.h](#).

4.51.2.222 DUMMYTEN

```
#define DUMMYTEN 36
```

Definition at line 373 of file [ftypes.h](#).

4.51.2.223 LEVICIVITA

```
#define LEVICIVITA 37
```

Definition at line 374 of file [ftypes.h](#).

4.51.2.224 FACTORIAL

```
#define FACTORIAL 38
```

Definition at line 375 of file [ftypes.h](#).

4.51.2.225 INVERSEFACTORIAL

```
#define INVERSEFACTORIAL 39
```

Definition at line 376 of file [ftypes.h](#).

4.51.2.226 BINOMIAL

```
#define BINOMIAL 40
```

Definition at line 377 of file [ftypes.h](#).

4.51.2.227 NUMARGSFUN

```
#define NUMARGSFUN 41
```

Definition at line 378 of file [ftypes.h](#).

4.51.2.228 SIGNFUN

```
#define SIGNFUN 42
```

Definition at line 379 of file [ftypes.h](#).

4.51.2.229 MODFUNCTION

```
#define MODFUNCTION 43
```

Definition at line 380 of file [ftypes.h](#).

4.51.2.230 MOD2FUNCTION

```
#define MOD2FUNCTION 44
```

Definition at line 381 of file [ftypes.h](#).

4.51.2.231 MINFUNCTION

```
#define MINFUNCTION 45
```

Definition at line 382 of file [ftypes.h](#).

4.51.2.232 MAXFUNCTION

```
#define MAXFUNCTION 46
```

Definition at line 383 of file [ftypes.h](#).

4.51.2.233 ABSFUNCTION

```
#define ABSFUNCTION 47
```

Definition at line 384 of file [ftypes.h](#).

4.51.2.234 SIGFUNCTION

```
#define SIGFUNCTION 48
```

Definition at line 385 of file [ftypes.h](#).

4.51.2.235 INTFUNCTION

```
#define INTFUNCTION 49
```

Definition at line 386 of file [ftypes.h](#).

4.51.2.236 THETA

```
#define THETA 50
```

Definition at line 387 of file [ftypes.h](#).

4.51.2.237 THETA2

```
#define THETA2 51
```

Definition at line 388 of file [ftypes.h](#).

4.51.2.238 DELTA2

```
#define DELTA2 52
```

Definition at line 389 of file [ftypes.h](#).

4.51.2.239 DELTAP

```
#define DELTAP 53
```

Definition at line 390 of file [ftypes.h](#).

4.51.2.240 BERNOULLIFUNCTION

```
#define BERNOULLIFUNCTION 54
```

Definition at line 391 of file [ftypes.h](#).

4.51.2.241 COUNTFUNCTION

```
#define COUNTFUNCTION 55
```

Definition at line 392 of file [ftypes.h](#).

4.51.2.242 MATCHFUNCTION

```
#define MATCHFUNCTION 56
```

Definition at line 393 of file [ftypes.h](#).

4.51.2.243 PATTERNFUNCTION

```
#define PATTERNFUNCTION 57
```

Definition at line 394 of file [ftypes.h](#).

4.51.2.244 TERMFUNCTION

```
#define TERMFUNCTION 58
```

Definition at line 395 of file [ftypes.h](#).

4.51.2.245 CONJUGATION

```
#define CONJUGATION 59
```

Definition at line 396 of file [ftypes.h](#).

4.51.2.246 ROOTFUNCTION

```
#define ROOTFUNCTION 60
```

Definition at line 397 of file [ftypes.h](#).

4.51.2.247 TABLEFUNCTION

```
#define TABLEFUNCTION 61
```

Definition at line 398 of file [ftypes.h](#).

4.51.2.248 FIRSTBRACKET

```
#define FIRSTBRACKET 62
```

Definition at line 399 of file [ftypes.h](#).

4.51.2.249 TERMSINEXPR

```
#define TERMSINEXPR 63
```

Definition at line 400 of file [ftypes.h](#).

4.51.2.250 NUMTERMSFUN

```
#define NUMTERMSFUN 64
```

Definition at line 401 of file [ftypes.h](#).

4.51.2.251 GCDFUNCTION

```
#define GCDFUNCTION 65
```

Definition at line 402 of file [ftypes.h](#).

4.51.2.252 DIVFUNCTION

```
#define DIVFUNCTION 66
```

Definition at line 403 of file [ftypes.h](#).

4.51.2.253 REMFUNCTION

```
#define REMFUNCTION 67
```

Definition at line 404 of file [ftypes.h](#).

4.51.2.254 MAXPOWEROF

```
#define MAXPOWEROF 68
```

Definition at line 405 of file [ftypes.h](#).

4.51.2.255 MINPOWEROF

```
#define MINPOWEROF 69
```

Definition at line 406 of file [ftypes.h](#).

4.51.2.256 TABLESTUB

```
#define TABLESTUB 70
```

Definition at line 407 of file [ftypes.h](#).

4.51.2.257 FACTORIN

```
#define FACTORIN 71
```

Definition at line 408 of file [ftypes.h](#).

4.51.2.258 TERMSINBRACKET

```
#define TERMSINBRACKET 72
```

Definition at line 409 of file [ftypes.h](#).

4.51.2.259 WILDARGFUN

```
#define WILDARGFUN 73
```

Definition at line 410 of file [ftypes.h](#).

4.51.2.260 SQRTFUNCTION

```
#define SQRTFUNCTION 74
```

Definition at line 418 of file [ftypes.h](#).

4.51.2.261 LNFUNCTION

```
#define LNFUNCTION 75
```

Definition at line 419 of file [ftypes.h](#).

4.51.2.262 SINFUNCTION

```
#define SINFUNCTION 76
```

Definition at line 420 of file [ftypes.h](#).

4.51.2.263 COSFUNCTION

```
#define COSFUNCTION 77
```

Definition at line 421 of file [ftypes.h](#).

4.51.2.264 TANFUNCTION

```
#define TANFUNCTION 78
```

Definition at line 422 of file [ftypes.h](#).

4.51.2.265 ASINFUNCTION

```
#define ASINFUNCTION 79
```

Definition at line 423 of file [ftypes.h](#).

4.51.2.266 ACOSFUNCTION

```
#define ACOSFUNCTION 80
```

Definition at line 424 of file [ftypes.h](#).

4.51.2.267 ATANFUNCTION

```
#define ATANFUNCTION 81
```

Definition at line 425 of file [ftypes.h](#).

4.51.2.268 ATAN2FUNCTION

```
#define ATAN2FUNCTION 82
```

Definition at line 426 of file [ftypes.h](#).

4.51.2.269 SINHFUNCTION

```
#define SINHFUNCTION 83
```

Definition at line 427 of file [ftypes.h](#).

4.51.2.270 COSHFUNCTION

```
#define COSHFUNCTION 84
```

Definition at line 428 of file [ftypes.h](#).

4.51.2.271 TANHFUNCTION

```
#define TANHFUNCTION 85
```

Definition at line 429 of file [ftypes.h](#).

4.51.2.272 ASINHFUNCTION

```
#define ASINHFUNCTION 86
```

Definition at line 430 of file [ftypes.h](#).

4.51.2.273 ACOSHFUNCTION

```
#define ACOSHFUNCTION 87
```

Definition at line 431 of file [ftypes.h](#).

4.51.2.274 ATANHFUNCTION

```
#define ATANHFUNCTION 88
```

Definition at line 432 of file [ftypes.h](#).

4.51.2.275 LI2FUNCTION

```
#define LI2FUNCTION 89
```

Definition at line 433 of file [ftypes.h](#).

4.51.2.276 LINFUNCTION

```
#define LINFUNCTION 90
```

Definition at line 434 of file [ftypes.h](#).

4.51.2.277 EXTRASYMFUN

```
#define EXTRASYMFUN 91
```

Definition at line 436 of file [ftypes.h](#).

4.51.2.278 RANDOMFUNCTION

```
#define RANDOMFUNCTION 92
```

Definition at line 437 of file [ftypes.h](#).

4.51.2.279 RANPERM

```
#define RANPERM 93
```

Definition at line 438 of file [ftypes.h](#).

4.51.2.280 NUMFACTORS

```
#define NUMFACTORS 94
```

Definition at line 439 of file [ftypes.h](#).

4.51.2.281 FIRSTTERM

```
#define FIRSTTERM 95
```

Definition at line 440 of file [ftypes.h](#).

4.51.2.282 CONTENTTERM

```
#define CONTENTTERM 96
```

Definition at line 441 of file [ftypes.h](#).

4.51.2.283 PRIMENUMBER

```
#define PRIMENUMBER 97
```

Definition at line 442 of file [ftypes.h](#).

4.51.2.284 EXTEUCLIDEAN

```
#define EXTEUCLIDEAN 98
```

Definition at line 443 of file [ftypes.h](#).

4.51.2.285 MAKERATIONAL

```
#define MAKERATIONAL 99
```

Definition at line 444 of file [ftypes.h](#).

4.51.2.286 INVERSEFUNCTION

```
#define INVERSEFUNCTION 100
```

Definition at line 445 of file [ftypes.h](#).

4.51.2.287 IDFUNCTION

```
#define IDFUNCTION 101
```

Definition at line 446 of file [ftypes.h](#).

4.51.2.288 PUTFIRST

```
#define PUTFIRST 102
```

Definition at line 447 of file [ftypes.h](#).

4.51.2.289 PERMUTATIONS

```
#define PERMUTATIONS 103
```

Definition at line 448 of file [ftypes.h](#).

4.51.2.290 PARTITIONS

```
#define PARTITIONS 104
```

Definition at line 449 of file [ftypes.h](#).

4.51.2.291 MULFUNCTION

```
#define MULFUNCTION 105
```

Definition at line 450 of file [ftypes.h](#).

4.51.2.292 MOEBIUS

```
#define MOEBIUS 106
```

Definition at line 451 of file [ftypes.h](#).

4.51.2.293 TOPOLOGIES

```
#define TOPOLOGIES 107
```

Definition at line 452 of file [ftypes.h](#).

4.51.2.294 DIAGRAMS

```
#define DIAGRAMS 108
```

Definition at line 453 of file [ftypes.h](#).

4.51.2.295 TOPO

```
#define TOPO 109
```

Definition at line 454 of file [ftypes.h](#).

4.51.2.296 NODEFUNCTION

```
#define NODEFUNCTION 110
```

Definition at line 455 of file [ftypes.h](#).

4.51.2.297 EDGE

```
#define EDGE 111
```

Definition at line 456 of file [ftypes.h](#).

4.51.2.298 SIZEOFFUNCTION

```
#define SIZEOFFUNCTION 112
```

Definition at line 457 of file [ftypes.h](#).

4.51.2.299 BLOCK

```
#define BLOCK 113
```

Definition at line 458 of file [ftypes.h](#).

4.51.2.300 ONEPI

```
#define ONEPI 114
```

Definition at line 459 of file [ftypes.h](#).

4.51.2.301 PHI

```
#define PHI 115
```

Definition at line 460 of file [ftypes.h](#).

4.51.2.302 FLOATFUN

```
#define FLOATFUN 116
```

Definition at line 461 of file [ftypes.h](#).

4.51.2.303 TOFLOAT

```
#define TOFLOAT 117
```

Definition at line 462 of file [ftypes.h](#).

4.51.2.304 TORAT

```
#define TORAT 118
```

Definition at line 463 of file [ftypes.h](#).

4.51.2.305 MZV

```
#define MZV 119
```

Definition at line 464 of file [ftypes.h](#).

4.51.2.306 EULER

```
#define EULER 120
```

Definition at line 465 of file [ftypes.h](#).

4.51.2.307 MZVHALF

```
#define MZVHALF 121
```

Definition at line 466 of file [ftypes.h](#).

4.51.2.308 AGMFUNCTION

```
#define AGMFUNCTION 122
```

Definition at line 467 of file [ftypes.h](#).

4.51.2.309 GAMMAFUN

```
#define GAMMAFUN 123
```

Definition at line 468 of file [ftypes.h](#).

4.51.2.310 EXPFUNCTION

```
#define EXPFUNCTION 124
```

Definition at line 469 of file [ftypes.h](#).

4.51.2.311 HPLFUNCTION

```
#define HPLFUNCTION 125
```

Definition at line 470 of file [ftypes.h](#).

4.51.2.312 MPLFUNCTION

```
#define MPLFUNCTION 126
```

Definition at line 471 of file [ftypes.h](#).

4.51.2.313 MAXBUILTINFUNCTION

```
#define MAXBUILTINFUNCTION 126
```

Definition at line 472 of file [ftypes.h](#).

4.51.2.314 FIRSTUSERFUNCTION

```
#define FIRSTUSERFUNCTION 150
```

Definition at line 474 of file [ftypes.h](#).

4.51.2.315 ALLFUNCTIONS

```
#define ALLFUNCTIONS -1
```

Definition at line 476 of file [ftypes.h](#).

4.51.2.316 ALLMZVFUNCTIONS

```
#define ALLMZVFUNCTIONS -2
```

Definition at line 477 of file [ftypes.h](#).

4.51.2.317 ISYMBOL

```
#define ISYMBOL 0
```

Definition at line 485 of file [ftypes.h](#).

4.51.2.318 PISYMBOL

```
#define PISYMBOL 1
```

Definition at line 486 of file [ftypes.h](#).

4.51.2.319 COEFFSYMBOL

```
#define COEFFSYMBOL 2
```

Definition at line 487 of file [ftypes.h](#).

4.51.2.320 NUMERATORSYMBOL

```
#define NUMERATORSYMBOL 3
```

Definition at line 488 of file [ftypes.h](#).

4.51.2.321 DENOMINATORSYMBOL

```
#define DENOMINATORSYMBOL 4
```

Definition at line 489 of file [ftypes.h](#).

4.51.2.322 WILDARGSYMBOL

```
#define WILDARGSYMBOL 5
```

Definition at line 490 of file [ftypes.h](#).

4.51.2.323 DIMENSIONSYPMBOL

```
#define DIMENSIONSYPMBOL 6
```

Definition at line 491 of file [ftypes.h](#).

4.51.2.324 FACTORSYMBOL

```
#define FACTORSYMBOL 7
```

Definition at line 492 of file [ftypes.h](#).

4.51.2.325 SEPARATESYMBOL

```
#define SEPARATESYMBOL 8
```

Definition at line 493 of file [ftypes.h](#).

4.51.2.326 EESYMBOL

```
#define EESYMBOL 9
```

Definition at line 494 of file [ftypes.h](#).

4.51.2.327 EMSYMBOL

```
#define EMSYMBOL 10
```

Definition at line 495 of file [ftypes.h](#).

4.51.2.328 BUILTINSYMBOLS

```
#define BUILTINSYMBOLS 11
```

Definition at line 497 of file [ftypes.h](#).

4.51.2.329 FIRSTUSERSYMBOL

```
#define FIRSTUSERSYMBOL 20
```

Definition at line 498 of file [ftypes.h](#).

4.51.2.330 BUILTININDICES

```
#define BUILTININDICES 1
```

Definition at line 500 of file [ftypes.h](#).

4.51.2.331 BUILTINVECTORS

```
#define BUILTINVECTORS 1
```

Definition at line 501 of file [ftypes.h](#).

4.51.2.332 BUILTINDOLLARS

```
#define BUILTINDOLLARS 1
```

Definition at line 502 of file [ftypes.h](#).

4.51.2.333 WILDARGVECTOR

```
#define WILDARGVECTOR 0
```

Definition at line 504 of file [ftypes.h](#).

4.51.2.334 WILDARGINDEX

```
#define WILDARGINDEX 0
```

Definition at line 505 of file [ftypes.h](#).

4.51.2.335 TYPEEXPRESSION

```
#define TYPEEXPRESSION 0
```

Definition at line 516 of file [ftypes.h](#).

4.51.2.336 TYPEIDNEW

```
#define TYPEIDNEW 1
```

Definition at line 517 of file [ftypes.h](#).

4.51.2.337 TYPEIDOLD

```
#define TYPEIDOLD 2
```

Definition at line 518 of file [ftypes.h](#).

4.51.2.338 TYPEOPERATION

```
#define TYPEOPERATION 3
```

Definition at line 519 of file [ftypes.h](#).

4.51.2.339 TYPEEREPEAT

```
#define TYPEEREPEAT 4
```

Definition at line 520 of file [ftypes.h](#).

4.51.2.340 TYPEENDREPEAT

```
#define TYPEENDREPEAT 5
```

Definition at line 521 of file [ftypes.h](#).

4.51.2.341 TYPECOUNT

```
#define TYPECOUNT 20
```

Definition at line 525 of file [ftypes.h](#).

4.51.2.342 TYPEMULT

```
#define TYPEMULT 21
```

Definition at line 526 of file [ftypes.h](#).

4.51.2.343 TYPEGOTO

```
#define TYPEGOTO 22
```

Definition at line 527 of file [ftypes.h](#).

4.51.2.344 TYPEDISCARD

```
#define TYPEDISCARD 23
```

Definition at line 528 of file [ftypes.h](#).

4.51.2.345 TYPEIF

```
#define TYPEIF 24
```

Definition at line 529 of file [ftypes.h](#).

4.51.2.346 TYPEELSE

```
#define TYPEELSE 25
```

Definition at line 530 of file [ftypes.h](#).

4.51.2.347 TYPEELIF

```
#define TYPEELIF 26
```

Definition at line 531 of file [ftypes.h](#).

4.51.2.348 TYPEENDIF

```
#define TYPEENDIF 27
```

Definition at line 532 of file [ftypes.h](#).

4.51.2.349 TYPESUM

```
#define TYPESUM 28
```

Definition at line 533 of file [ftypes.h](#).

4.51.2.350 TYPECHISHOLM

```
#define TYPECHISHOLM 29
```

Definition at line 534 of file [ftypes.h](#).

4.51.2.351 TPEREVERSE

```
#define TPEREVERSE 30
```

Definition at line 535 of file [ftypes.h](#).

4.51.2.352 TYPEARG

```
#define TYPEARG 31
```

Definition at line 536 of file [ftypes.h](#).

4.51.2.353 TYPENORM

```
#define TYPENORM 32
```

Definition at line 537 of file [ftypes.h](#).

4.51.2.354 TYPEENORM2

```
#define TYPEENORM2 33
```

Definition at line 538 of file [ftypes.h](#).

4.51.2.355 TYPEENORM3

```
#define TYPEENORM3 34
```

Definition at line 539 of file [ftypes.h](#).

4.51.2.356 TYPEEXIT

```
#define TYPEEXIT 35
```

Definition at line 540 of file [ftypes.h](#).

4.51.2.357 TYPESETEXIT

```
#define TYPESETEXIT 36
```

Definition at line 541 of file [ftypes.h](#).

4.51.2.358 TYPEPRINT

```
#define TYPEPRINT 37
```

Definition at line 542 of file [ftypes.h](#).

4.51.2.359 TYPEFPRINT

```
#define TYPEFPRINT 38
```

Definition at line 543 of file [ftypes.h](#).

4.51.2.360 TYPEREDEFPRE

```
#define TYPEREDEFPRE 39
```

Definition at line 544 of file [ftypes.h](#).

4.51.2.361 TYPESPLITARG

```
#define TYPESPLITARG 40
```

Definition at line 545 of file [ftypes.h](#).

4.51.2.362 TYPESPLITARG2

```
#define TYPESPLITARG2 41
```

Definition at line 546 of file [ftypes.h](#).

4.51.2.363 TYPEFACTARG

```
#define TYPEFACTARG 42
```

Definition at line 547 of file [ftypes.h](#).

4.51.2.364 TYPEFACTARG2

```
#define TYPEFACTARG2 43
```

Definition at line 548 of file [ftypes.h](#).

4.51.2.365 TYPETRY

```
#define TYPETRY 44
```

Definition at line 549 of file [ftypes.h](#).

4.51.2.366 TYPEASSIGN

```
#define TYPEASSIGN 45
```

Definition at line 550 of file [ftypes.h](#).

4.51.2.367 TYPERENUMBER

```
#define TYPERENUMBER 46
```

Definition at line 551 of file [ftypes.h](#).

4.51.2.368 TYPESUMFIX

```
#define TYPESUMFIX 47
```

Definition at line 552 of file [ftypes.h](#).

4.51.2.369 TYPEFINDLOOP

```
#define TYPEFINDLOOP 48
```

Definition at line 553 of file [ftypes.h](#).

4.51.2.370 TYPEUNRAVEL

```
#define TYPEUNRAVEL 49
```

Definition at line 554 of file [ftypes.h](#).

4.51.2.371 TYPEADJUSTBOUNDS

```
#define TYPEADJUSTBOUNDS 50
```

Definition at line 555 of file [ftypes.h](#).

4.51.2.372 TYPEINSIDE

```
#define TYPEINSIDE 51
```

Definition at line 556 of file [ftypes.h](#).

4.51.2.373 TYPETERM

```
#define TYPETERM 52
```

Definition at line 557 of file [ftypes.h](#).

4.51.2.374 TYPESORT

```
#define TYPESORT 53
```

Definition at line 558 of file [ftypes.h](#).

4.51.2.375 TYPEDETCURDUM

```
#define TYPEDETCURDUM 54
```

Definition at line 559 of file [ftypes.h](#).

4.51.2.376 TYPEINEXPRESSION

```
#define TYPEINEXPRESSION 55
```

Definition at line 560 of file [ftypes.h](#).

4.51.2.377 TYPESPLITFIRSTARG

```
#define TYPESPLITFIRSTARG 56
```

Definition at line 561 of file [ftypes.h](#).

4.51.2.378 TYPESPLITLASTARG

```
#define TYPESPLITLASTARG 57
```

Definition at line 562 of file [ftypes.h](#).

4.51.2.379 TYPEMERGE

```
#define TYPEMERGE 58
```

Definition at line 563 of file [ftypes.h](#).

4.51.2.380 TYPETESTUSE

```
#define TYPETESTUSE 59
```

Definition at line 564 of file [ftypes.h](#).

4.51.2.381 TYPEAPPLY

```
#define TYPEAPPLY 60
```

Definition at line 565 of file [ftypes.h](#).

4.51.2.382 TYPEAPPLYRESET

```
#define TYPEAPPLYRESET 61
```

Definition at line 566 of file [ftypes.h](#).

4.51.2.383 TYPECHAININ

```
#define TYPECHAININ 62
```

Definition at line 567 of file [ftypes.h](#).

4.51.2.384 TYPECHAINOUT

```
#define TYPECHAINOUT 63
```

Definition at line 568 of file [ftypes.h](#).

4.51.2.385 TYPENORM4

```
#define TYPENORM4 64
```

Definition at line 569 of file [ftypes.h](#).

4.51.2.386 TYPEFACTOR

```
#define TYPEFACTOR 65
```

Definition at line 570 of file [ftypes.h](#).

4.51.2.387 TYPEARGIMPLODE

```
#define TYPEARGIMPLODE 66
```

Definition at line 571 of file [ftypes.h](#).

4.51.2.388 TYPEARGEXPLODE

```
#define TYPEARGEXPLODE 67
```

Definition at line 572 of file [ftypes.h](#).

4.51.2.389 TYPEDENOMINATORS

```
#define TYPEDENOMINATORS 68
```

Definition at line 573 of file [ftypes.h](#).

4.51.2.390 TYPESTUFFLE

```
#define TYPESTUFFLE 69
```

Definition at line 574 of file [ftypes.h](#).

4.51.2.391 TYPEDROPCOEFFICIENT

```
#define TYPEDROPCOEFFICIENT 70
```

Definition at line 575 of file [ftypes.h](#).

4.51.2.392 TYPETRANSFORM

```
#define TYPETRANSFORM 71
```

Definition at line 576 of file [ftypes.h](#).

4.51.2.393 TYPETOPOLYNOMIAL

```
#define TYPETOPOLYNOMIAL 72
```

Definition at line 577 of file [ftypes.h](#).

4.51.2.394 TYPEFROMPOLYNOMIAL

```
#define TYPEFROMPOLYNOMIAL 73
```

Definition at line 578 of file [ftypes.h](#).

4.51.2.395 TYPEDOLOOP

```
#define TYPEDOLOOP 74
```

Definition at line 579 of file [ftypes.h](#).

4.51.2.396 TYPEENDDOLOOP

```
#define TYPEENDDOLOOP 75
```

Definition at line 580 of file [ftypes.h](#).

4.51.2.397 TYPEDROPSYMBOLS

```
#define TYPEDROPSYMBOLS 76
```

Definition at line 581 of file [ftypes.h](#).

4.51.2.398 TYPEPUTINSIDE

```
#define TYPEPUTINSIDE 77
```

Definition at line 582 of file [ftypes.h](#).

4.51.2.399 TYPETOSPECTATOR

```
#define TYPETOSPECTATOR 78
```

Definition at line 583 of file [ftypes.h](#).

4.51.2.400 TYPEARGTOEXTRASymbol

```
#define TYPEARGTOEXTRASymbol 79
```

Definition at line 584 of file [ftypes.h](#).

4.51.2.401 TYPECANONICALIZE

```
#define TYPECANONICALIZE 80
```

Definition at line 585 of file [ftypes.h](#).

4.51.2.402 TYPESWITCH

```
#define TYPESWITCH 81
```

Definition at line 586 of file [ftypes.h](#).

4.51.2.403 TYPEENDSWITCH

```
#define TYPEENDSWITCH 82
```

Definition at line 587 of file [ftypes.h](#).

4.51.2.404 TYPESETUSERFLAG

```
#define TYPESETUSERFLAG 83
```

Definition at line 588 of file [ftypes.h](#).

4.51.2.405 TYPECLEARUSERFLAG

```
#define TYPECLEARUSERFLAG 84
```

Definition at line 589 of file [ftypes.h](#).

4.51.2.406 TYPEALLLOOPS

```
#define TYPEALLLOOPS 85
```

Definition at line 590 of file [ftypes.h](#).

4.51.2.407 TYPEALLPATHS

```
#define TYPEALLPATHS 86
```

Definition at line 591 of file [ftypes.h](#).

4.51.2.408 TAKETRACE

```
#define TAKETRACE 1
```

Definition at line 605 of file [ftypes.h](#).

4.51.2.409 CONTRACT

```
#define CONTRACT 2
```

Definition at line 606 of file [ftypes.h](#).

4.51.2.410 RATIO

```
#define RATIO 3
```

Definition at line 607 of file [ftypes.h](#).

4.51.2.411 SYMMETRIZE

```
#define SYMMETRIZE 4
```

Definition at line 608 of file [ftypes.h](#).

4.51.2.412 TENVEC

```
#define TENVEC 5
```

Definition at line 609 of file [ftypes.h](#).

4.51.2.413 SUMNUM1

```
#define SUMNUM1 6
```

Definition at line 610 of file [ftypes.h](#).

4.51.2.414 SUMNUM2

```
#define SUMNUM2 7
```

Definition at line 611 of file [ftypes.h](#).

4.51.2.415 WILDDUMMY

```
#define WILDDUMMY 0
```

Definition at line 617 of file [ftypes.h](#).

4.51.2.416 SYMTONUM

```
#define SYMTONUM 1
```

Definition at line 618 of file [ftypes.h](#).

4.51.2.417 SYMTOSYM

```
#define SYMTOSYM 2
```

Definition at line 619 of file [ftypes.h](#).

4.51.2.418 SYMTOSUB

```
#define SYMTOSUB 3
```

Definition at line [620](#) of file [ftypes.h](#).

4.51.2.419 VECTOMIN

```
#define VECTOMIN 4
```

Definition at line [621](#) of file [ftypes.h](#).

4.51.2.420 VECTOVEC

```
#define VECTOVEC 5
```

Definition at line [622](#) of file [ftypes.h](#).

4.51.2.421 VECTOSUB

```
#define VECTOSUB 6
```

Definition at line [623](#) of file [ftypes.h](#).

4.51.2.422 INDTOIND

```
#define INDTOIND 7
```

Definition at line [624](#) of file [ftypes.h](#).

4.51.2.423 INDTOSUB

```
#define INDTOSUB 8
```

Definition at line [625](#) of file [ftypes.h](#).

4.51.2.424 FUNTOFUN

```
#define FUNTOFUN 9
```

Definition at line [626](#) of file [ftypes.h](#).

4.51.2.425 ARGTOARG

```
#define ARGTOARG 10
```

Definition at line [627](#) of file [ftypes.h](#).

4.51.2.426 ARLTOARL

```
#define ARLTOARL 11
```

Definition at line 628 of file [ftypes.h](#).

4.51.2.427 EXPTOEXP

```
#define EXPTOEXP 12
```

Definition at line 629 of file [ftypes.h](#).

4.51.2.428 FROMBRAC

```
#define FROMBRAC 13
```

Definition at line 630 of file [ftypes.h](#).

4.51.2.429 FROMSET

```
#define FROMSET 14
```

Definition at line 631 of file [ftypes.h](#).

4.51.2.430 SETTONUM

```
#define SETTONUM 15
```

Definition at line 632 of file [ftypes.h](#).

4.51.2.431 WILDCARDS

```
#define WILDCARDS 16
```

Definition at line 633 of file [ftypes.h](#).

4.51.2.432 SETNUMBER

```
#define SETNUMBER 17
```

Definition at line 634 of file [ftypes.h](#).

4.51.2.433 LOADDOLLAR

```
#define LOADDOLLAR 18
```

Definition at line 635 of file [ftypes.h](#).

4.51.2.434 NUMTONUM

```
#define NUMTONUM 20
```

Definition at line 639 of file [ftypes.h](#).

4.51.2.435 NUMTOSYM

```
#define NUMTOSYM 21
```

Definition at line 640 of file [ftypes.h](#).

4.51.2.436 NUMTOIND

```
#define NUMTOIND 22
```

Definition at line 641 of file [ftypes.h](#).

4.51.2.437 NUMTOSUB

```
#define NUMTOSUB 23
```

Definition at line 642 of file [ftypes.h](#).

4.51.2.438 EATTENSOR

```
#define EATTENSOR 0x2000
```

Definition at line 646 of file [ftypes.h](#).

4.51.2.439 CLEANFLAG

```
#define CLEANFLAG 0
```

Definition at line 652 of file [ftypes.h](#).

4.51.2.440 DIRTYFLAG

```
#define DIRTYFLAG 1
```

Definition at line 653 of file [ftypes.h](#).

4.51.2.441 DIRTSYMFLAG

```
#define DIRTSYMFLAG 2
```

Definition at line 654 of file [ftypes.h](#).

4.51.2.442 MUSTCLEANPRF

```
#define MUSTCLEANPRF 4
```

Definition at line 655 of file [ftypes.h](#).

4.51.2.443 SUBTERMUSED1

```
#define SUBTERMUSED1 8
```

Definition at line 656 of file [ftypes.h](#).

4.51.2.444 SUBTERMUSED2

```
#define SUBTERMUSED2 16
```

Definition at line 657 of file [ftypes.h](#).

4.51.2.445 ALLDIRTY

```
#define ALLDIRTY (DIRTYFLAG|DIRTSYMFLAG)
```

Definition at line 658 of file [ftypes.h](#).

4.51.2.446 ARGHEAD

```
#define ARGHEAD 2
```

Definition at line 660 of file [ftypes.h](#).

4.51.2.447 FUNHEAD

```
#define FUNHEAD 3
```

Definition at line 661 of file [ftypes.h](#).

4.51.2.448 SUBEXPSIZE

```
#define SUBEXPSIZE 5
```

Definition at line 662 of file [ftypes.h](#).

4.51.2.449 EXPRHEAD

```
#define EXPRHEAD 5
```

Definition at line 663 of file [ftypes.h](#).

4.51.2.450 TYPEARGHEADSIZE

```
#define TYPEARGHEADSIZE 6
```

Definition at line 664 of file [ftypes.h](#).

4.51.2.451 SKIP

```
#define SKIP 1
```

Definition at line 671 of file [ftypes.h](#).

4.51.2.452 DROP

```
#define DROP 2
```

Definition at line 672 of file [ftypes.h](#).

4.51.2.453 HIDE

```
#define HIDE 3
```

Definition at line 673 of file [ftypes.h](#).

4.51.2.454 UNHIDE

```
#define UNHIDE 4
```

Definition at line 674 of file [ftypes.h](#).

4.51.2.455 INTOHIDE

```
#define INTOHIDE 5
```

Definition at line 675 of file [ftypes.h](#).

4.51.2.456 LOCALEXPRESSION

```
#define LOCALEXPRESSION 0
```

Definition at line 681 of file [ftypes.h](#).

4.51.2.457 SKIPLEXPRESSION

```
#define SKIPLEXPRESSION 1
```

Definition at line 682 of file [ftypes.h](#).

4.51.2.458 DROPLEXPRESSION

```
#define DROPLEXPRESSION 2
```

Definition at line 683 of file [ftypes.h](#).

4.51.2.459 DROPPEDEXPRESSION

```
#define DROPPEDEXPRESSION 3
```

Definition at line 684 of file [ftypes.h](#).

4.51.2.460 GLOBALEXPRESSION

```
#define GLOBALEXPRESSION 4
```

Definition at line 685 of file [ftypes.h](#).

4.51.2.461 SKIPGEXPRESSION

```
#define SKIPGEXPRESSION 5
```

Definition at line 686 of file [ftypes.h](#).

4.51.2.462 DROPGEXPRESSION

```
#define DROPGEXPRESSION 6
```

Definition at line 687 of file [ftypes.h](#).

4.51.2.463 STOREDEXPRESSION

```
#define STOREDEXPRESSION 8
```

Definition at line 688 of file [ftypes.h](#).

4.51.2.464 HIDDENLEXPRESSION

```
#define HIDDENLEXPRESSION 9
```

Definition at line 689 of file [ftypes.h](#).

4.51.2.465 HIDDENGEXPRESSION

```
#define HIDDENGEXPRESSION 13
```

Definition at line 690 of file [ftypes.h](#).

4.51.2.466 INCEXPRESSION

```
#define INCEXPRESSION 9
```

Definition at line 691 of file [ftypes.h](#).

4.51.2.467 HIDELEXPRESSION

```
#define HIDELEXPRESSION 10
```

Definition at line 692 of file [ftypes.h](#).

4.51.2.468 HIDEGEXPRESSION

```
#define HIDEGEXPRESSION 14
```

Definition at line 693 of file [ftypes.h](#).

4.51.2.469 DROPHLEXPRESSION

```
#define DROPHLEXPRESSION 11
```

Definition at line 694 of file [ftypes.h](#).

4.51.2.470 DROPHGEXPRESSION

```
#define DROPHGEXPRESSION 15
```

Definition at line 695 of file [ftypes.h](#).

4.51.2.471 UNHIDELEXPRESSION

```
#define UNHIDELEXPRESSION 12
```

Definition at line 696 of file [ftypes.h](#).

4.51.2.472 UNHIDEGEXPRESSION

```
#define UNHIDEGEXPRESSION 16
```

Definition at line 697 of file [ftypes.h](#).

4.51.2.473 INTOHIDELEXPRESSION

```
#define INTOHIDELEXPRESSION 17
```

Definition at line 698 of file [ftypes.h](#).

4.51.2.474 INTOHIDEGEXPRESSION

```
#define INTOHIDEGEXPRESSION 18
```

Definition at line 699 of file [ftypes.h](#).

4.51.2.475 SPECTATOREXPRESSION

```
#define SPECTATOREXPRESSION 19
```

Definition at line 700 of file [ftypes.h](#).

4.51.2.476 DROPSPECTATOREXPRESSION

```
#define DROPSPECTATOREXPRESSION 20
```

Definition at line 701 of file [ftypes.h](#).

4.51.2.477 SKIPUNHIDELEXPRESSION

```
#define SKIPUNHIDELEXPRESSION 21
```

Definition at line 702 of file [ftypes.h](#).

4.51.2.478 SKIPUNHIDEGEXPRESSION

```
#define SKIPUNHIDEGEXPRESSION 22
```

Definition at line 703 of file [ftypes.h](#).

4.51.2.479 PRINTOFF

```
#define PRINTOFF 0
```

Definition at line 705 of file [ftypes.h](#).

4.51.2.480 PRINTON

```
#define PRINTON 1
```

Definition at line 706 of file [ftypes.h](#).

4.51.2.481 PRINTCONTENTS

```
#define PRINTCONTENTS 2
```

Definition at line 707 of file [ftypes.h](#).

4.51.2.482 PRINTCONTENT

```
#define PRINTCONTENT 3
```

Definition at line 708 of file [ftypes.h](#).

4.51.2.483 PRINTLFILE

```
#define PRINTLFILE 4
```

Definition at line 709 of file [ftypes.h](#).

4.51.2.484 PRINTONETERM

```
#define PRINTONETERM 8
```

Definition at line 710 of file [ftypes.h](#).

4.51.2.485 PRINTONEFUNCTION

```
#define PRINTONEFUNCTION 16
```

Definition at line 711 of file [ftypes.h](#).

4.51.2.486 PRINTALL

```
#define PRINTALL 32
```

Definition at line 712 of file [ftypes.h](#).

4.51.2.487 REGULAREXPRESSION

```
#define REGULAREXPRESSION -1
```

Definition at line 718 of file [ftypes.h](#).

4.51.2.488 REDEFINEDEXPRESSION

```
#define REDEFINEDEXPRESSION -2
```

Definition at line 719 of file [ftypes.h](#).

4.51.2.489 NEWLYDEFINEDEXPRESSION

```
#define NEWLYDEFINEDEXPRESSION -3
```

Definition at line 720 of file [ftypes.h](#).

4.51.2.490 VERTEXFUNCTION

```
#define VERTEXFUNCTION -1
```

Definition at line 729 of file [ftypes.h](#).

4.51.2.491 GENERALFUNCTION

```
#define GENERALFUNCTION 0
```

Definition at line 730 of file [ftypes.h](#).

4.51.2.492 TENSORFUNCTION

```
#define TENSORFUNCTION 2
```

Definition at line 731 of file [ftypes.h](#).

4.51.2.493 GAMMAFUNCTION

```
#define GAMMAFUNCTION 4
```

Definition at line 732 of file [ftypes.h](#).

4.51.2.494 POS_

```
#define POS_ 0 /* integer > 0 */
```

Definition at line 739 of file [ftypes.h](#).

4.51.2.495 POS0_

```
#define POS0_ 1 /* integer >= 0 */
```

Definition at line 740 of file [ftypes.h](#).

4.51.2.496 NEG_

```
#define NEG_ 2 /* integer < 0 */
```

Definition at line 741 of file [ftypes.h](#).

4.51.2.497 NEG0_

```
#define NEG0_ 3 /* integer <= 0 */
```

Definition at line 742 of file [ftypes.h](#).

4.51.2.498 EVEN_

```
#define EVEN_ 4 /* integer (even) */
```

Definition at line 743 of file [ftypes.h](#).

4.51.2.499 ODD_

```
#define ODD_ 5 /* integer (odd) */
```

Definition at line 744 of file [ftypes.h](#).

4.51.2.500 Z_

```
#define Z_ 6 /* integer */
```

Definition at line 745 of file [ftypes.h](#).

4.51.2.501 SYMBOL_

```
#define SYMBOL_ 7 /* symbol only */
```

Definition at line 746 of file [ftypes.h](#).

4.51.2.502 FIXED_

```
#define FIXED_ 8 /* fixed index */
```

Definition at line 747 of file [ftypes.h](#).

4.51.2.503 INDEX_

```
#define INDEX_ 9 /* index only */
```

Definition at line 748 of file [ftypes.h](#).

4.51.2.504 Q_

```
#define Q_ 10 /* rational */
```

Definition at line 749 of file [ftypes.h](#).

4.51.2.505 DUMMYINDEX_

```
#define DUMMYINDEX_ 11 /* dummy index only */
```

Definition at line 750 of file [ftypes.h](#).

4.51.2.506 VECTOR_

```
#define VECTOR_ 12 /* vector only */
```

Definition at line 751 of file [ftypes.h](#).

4.51.2.507 GAMMA1

```
#define GAMMA1 0
```

Definition at line 757 of file [ftypes.h](#).

4.51.2.508 GAMMA5

```
#define GAMMA5 -1
```

Definition at line 758 of file [ftypes.h](#).

4.51.2.509 GAMMA6

```
#define GAMMA6 -2
```

Definition at line 759 of file [ftypes.h](#).

4.51.2.510 GAMMA7

```
#define GAMMA7 -3
```

Definition at line 760 of file [ftypes.h](#).

4.51.2.511 FUNNYVEC

```
#define FUNNYVEC -4
```

Definition at line 761 of file [ftypes.h](#).

4.51.2.512 FUNNYWILD

```
#define FUNNYWILD -5
```

Definition at line 762 of file [ftypes.h](#).

4.51.2.513 SUMMEDIND

```
#define SUMMEDIND -6
```

Definition at line 763 of file [ftypes.h](#).

4.51.2.514 NOINDEX

```
#define NOINDEX -7
```

Definition at line 764 of file [ftypes.h](#).

4.51.2.515 FUNNYDOLLAR

```
#define FUNNYDOLLAR -8
```

Definition at line 765 of file [ftypes.h](#).

4.51.2.516 EMPTYINDEX

```
#define EMPTYINDEX -9
```

Definition at line 766 of file [ftypes.h](#).

4.51.2.517 MINSPEC

```
#define MINSPEC -10
```

Definition at line 772 of file [ftypes.h](#).

4.51.2.518 USEDFLAG

```
#define USEDFLAG 2
```

Definition at line 774 of file [ftypes.h](#).

4.51.2.519 DUMMYFLAG

```
#define DUMMYFLAG 1
```

Definition at line 775 of file [ftypes.h](#).

4.51.2.520 MAINSORT

```
#define MAINSORT 0
```

Definition at line 777 of file [ftypes.h](#).

4.51.2.521 FUNCTIONSORT

```
#define FUNCTIONSORT 1
```

Definition at line 778 of file [ftypes.h](#).

4.51.2.522 SUBSORT

```
#define SUBSORT 2
```

Definition at line 779 of file [ftypes.h](#).

4.51.2.523 FLOATMODE

```
#define FLOATMODE 1
```

Definition at line 781 of file [ftypes.h](#).

4.51.2.524 RATIONALMODE

```
#define RATIONALMODE 0
```

Definition at line 782 of file [ftypes.h](#).

4.51.2.525 NUMSPECSETS

```
#define NUMSPECSETS 10
```

Definition at line 784 of file [ftypes.h](#).

4.51.2.526 ISZERO

```
#define ISZERO 1
```

Definition at line 786 of file [ftypes.h](#).

4.51.2.527 ISUNMODIFIED

```
#define ISUNMODIFIED 2
```

Definition at line 787 of file [ftypes.h](#).

4.51.2.528 ISCOMPRESSED

```
#define ISCOMPRESSED 4
```

Definition at line 788 of file [ftypes.h](#).

4.51.2.529 ISINRHS

```
#define ISINRHS 8
```

Definition at line 789 of file [ftypes.h](#).

4.51.2.530 ISFACTORIZED

```
#define ISFACTORIZED 16
```

Definition at line 790 of file [ftypes.h](#).

4.51.2.531 TOBEFACTORED

```
#define TOBEFACTORED 32
```

Definition at line 791 of file [ftypes.h](#).

4.51.2.532 TOBEUNFACTORED

```
#define TOBEUNFACTORED 64
```

Definition at line 792 of file [ftypes.h](#).

4.51.2.533 KEEPZERO

```
#define KEEPZERO 128
```

Definition at line 793 of file [ftypes.h](#).

4.51.2.534 VARTYPENONE

```
#define VARTYPENONE 0
```

Definition at line 795 of file [ftypes.h](#).

4.51.2.535 VARTYPECOMPLEX

```
#define VARTYPECOMPLEX 1
```

Definition at line 796 of file [ftypes.h](#).

4.51.2.536 VARTYPEIMAGINARY

```
#define VARTYPEIMAGINARY 2
```

Definition at line 797 of file [ftypes.h](#).

4.51.2.537 VARTPEROOTOFUNITY

```
#define VARTPEROOTOFUNITY 4
```

Definition at line 798 of file [ftypes.h](#).

4.51.2.538 VARTYPEMINUS

```
#define VARTYPEMINUS 8
```

Definition at line 799 of file [ftypes.h](#).

4.51.2.539 CYCLESYMMETRIC

```
#define CYCLESYMMETRIC 1
```

Definition at line 800 of file [ftypes.h](#).

4.51.2.540 RCYCLESYMMETRIC

```
#define RCYCLESYMMETRIC 2
```

Definition at line 801 of file [ftypes.h](#).

4.51.2.541 SYMMETRIC

```
#define SYMMETRIC 3
```

Definition at line 802 of file [ftypes.h](#).

4.51.2.542 ANTISYMMETRIC

```
#define ANTISYMMETRIC 4
```

Definition at line 803 of file [ftypes.h](#).

4.51.2.543 REVERSEORDER

```
#define REVERSEORDER 256
```

Definition at line 804 of file [ftypes.h](#).

4.51.2.544 SUBMULTI

```
#define SUBMULTI 1
```

Definition at line 810 of file [ftypes.h](#).

4.51.2.545 SUBONCE

```
#define SUBONCE 2
```

Definition at line 811 of file [ftypes.h](#).

4.51.2.546 SUBONLY

```
#define SUBONLY 3
```

Definition at line 812 of file [ftypes.h](#).

4.51.2.547 SUBMANY

```
#define SUBMANY 4
```

Definition at line 813 of file [ftypes.h](#).

4.51.2.548 SUBVECTOR

```
#define SUBVECTOR 5
```

Definition at line 814 of file [ftypes.h](#).

4.51.2.549 SUBSELECT

```
#define SUBSELECT 6
```

Definition at line 815 of file [ftypes.h](#).

4.51.2.550 SUBALL

```
#define SUBALL 7
```

Definition at line 816 of file [ftypes.h](#).

4.51.2.551 SUBMASK

```
#define SUBMASK 15
```

Definition at line 817 of file [ftypes.h](#).

4.51.2.552 SUBDISORDER

```
#define SUBDISORDER 16
```

Definition at line 818 of file [ftypes.h](#).

4.51.2.553 SUBAFTER

```
#define SUBAFTER 32
```

Definition at line 819 of file [ftypes.h](#).

4.51.2.554 SUBAFTERNOT

```
#define SUBAFTERNOT 64
```

Definition at line 820 of file [ftypes.h](#).

4.51.2.555 IDHEAD

```
#define IDHEAD 6
```

Definition at line 822 of file [ftypes.h](#).

4.51.2.556 DOLLARFLAG

```
#define DOLLARFLAG 1
```

Definition at line 824 of file [ftypes.h](#).

4.51.2.557 NORMALIZEFLAG

```
#define NORMALIZEFLAG 2
```

Definition at line 825 of file [ftypes.h](#).

4.51.2.558 GIDENT

```
#define GIDENT 1
```

Definition at line 827 of file [ftypes.h](#).

4.51.2.559 GFIVE

```
#define GFIVE 4
```

Definition at line 828 of file [ftypes.h](#).

4.51.2.560 GPLUS

```
#define GPLUS 3
```

Definition at line 829 of file [ftypes.h](#).

4.51.2.561 GMINUS

```
#define GMINUS 2
```

Definition at line 830 of file [ftypes.h](#).

4.51.2.562 LONGNUMBER

```
#define LONGNUMBER 1
```

Definition at line 836 of file [ftypes.h](#).

4.51.2.563 MATCH

```
#define MATCH 2
```

Definition at line 837 of file [ftypes.h](#).

4.51.2.564 COEFFI

```
#define COEFFI 3
```

Definition at line 838 of file [ftypes.h](#).

4.51.2.565 SUBEXPR

```
#define SUBEXPR 4
```

Definition at line 839 of file [ftypes.h](#).

4.51.2.566 MULTIPLEOF

```
#define MULTIPLEOF 5
```

Definition at line 840 of file [ftypes.h](#).

4.51.2.567 IFDOLLAR

```
#define IFDOLLAR 6
```

Definition at line 841 of file [ftypes.h](#).

4.51.2.568 IFEXPRESSION

```
#define IFEXPRESSION 7
```

Definition at line 842 of file [ftypes.h](#).

4.51.2.569 IFDOLLAREXTRA

```
#define IFDOLLAREXTRA 8
```

Definition at line 843 of file [ftypes.h](#).

4.51.2.570 IFISFACTORIZED

```
#define IFISFACTORIZED 9
```

Definition at line 844 of file [ftypes.h](#).

4.51.2.571 IFOCCURS

```
#define IFOCCURS 10
```

Definition at line 845 of file [ftypes.h](#).

4.51.2.572 IFUSERFLAG

```
#define IFUSERFLAG 11
```

Definition at line 846 of file [ftypes.h](#).

4.51.2.573 IFFLOATNUMBER

```
#define IFFLOATNUMBER 12
```

Definition at line 847 of file [ftypes.h](#).

4.51.2.574 GREATER

```
#define GREATER 0
```

Definition at line 848 of file [ftypes.h](#).

4.51.2.575 GREATEREQUAL

```
#define GREATEREQUAL 1
```

Definition at line 849 of file [ftypes.h](#).

4.51.2.576 LESS

```
#define LESS 2
```

Definition at line 850 of file [ftypes.h](#).

4.51.2.577 LESSEQUAL

```
#define LESSEQUAL 3
```

Definition at line 851 of file [ftypes.h](#).

4.51.2.578 EQUAL

```
#define EQUAL 4
```

Definition at line 852 of file [ftypes.h](#).

4.51.2.579 NOTEQUAL

```
#define NOTEQUAL 5
```

Definition at line 853 of file [ftypes.h](#).

4.51.2.580 ORCOND

```
#define ORCOND 6
```

Definition at line 854 of file [ftypes.h](#).

4.51.2.581 ANDCOND

```
#define ANDCOND 7
```

Definition at line 855 of file [ftypes.h](#).

4.51.2.582 DUMMY

```
#define DUMMY 1
```

Definition at line 856 of file [ftypes.h](#).

4.51.2.583 SORT

```
#define SORT 1
```

Definition at line 857 of file [ftypes.h](#).

4.51.2.584 STORE

```
#define STORE 2
```

Definition at line 858 of file [ftypes.h](#).

4.51.2.585 END

```
#define END 3
```

Definition at line 859 of file [ftypes.h](#).

4.51.2.586 GLOBAL

```
#define GLOBAL 4
```

Definition at line 860 of file [ftypes.h](#).

4.51.2.587 CLEAR

```
#define CLEAR 5
```

Definition at line 861 of file [ftypes.h](#).

4.51.2.588 VECTBIT

```
#define VECTBIT 1
```

Definition at line 863 of file [ftypes.h](#).

4.51.2.589 DOTPBIT

```
#define DOTPBIT 2
```

Definition at line 864 of file [ftypes.h](#).

4.51.2.590 FUNBIT

```
#define FUNBIT 4
```

Definition at line 865 of file [ftypes.h](#).

4.51.2.591 SETBIT

```
#define SETBIT 8
```

Definition at line 866 of file [ftypes.h](#).

4.51.2.592 EXTRAPARAMETER

```
#define EXTRAPARAMETER 0x4000
```

Definition at line 868 of file [ftypes.h](#).

4.51.2.593 GENCOMMUTE

```
#define GENCOMMUTE 0
```

Definition at line 869 of file [ftypes.h](#).

4.51.2.594 GENNONCOMMUTE

```
#define GENNONCOMMUTE 0x2000
```

Definition at line 870 of file [ftypes.h](#).

4.51.2.595 NAMENOTFOUND

```
#define NAMENOTFOUND -9
```

Definition at line 872 of file [ftypes.h](#).

4.51.2.596 DOLUNDEFINED

```
#define DOLUNDEFINED 0
```

Definition at line 878 of file [ftypes.h](#).

4.51.2.597 DOLNUMBER

```
#define DOLNUMBER 1
```

Definition at line 879 of file [ftypes.h](#).

4.51.2.598 DOLARGUMENT

```
#define DOLARGUMENT 2
```

Definition at line 880 of file [ftypes.h](#).

4.51.2.599 DOLSUBTERM

```
#define DOLSUBTERM 3
```

Definition at line 881 of file [ftypes.h](#).

4.51.2.600 DOLTERMS

```
#define DOLTERMS 4
```

Definition at line 882 of file [ftypes.h](#).

4.51.2.601 DOLWILDARGS

```
#define DOLWILDARGS 5
```

Definition at line 883 of file [ftypes.h](#).

4.51.2.602 DOLINDEX

```
#define DOLINDEX 6
```

Definition at line 884 of file [ftypes.h](#).

4.51.2.603 DOLZERO

```
#define DOLZERO 7
```

Definition at line 885 of file [ftypes.h](#).

4.51.2.604 FINDLOOP

```
#define FINDLOOP 0
```

Definition at line 887 of file [ftypes.h](#).

4.51.2.605 REPLACELOOP

```
#define REPLACELOOP 1
```

Definition at line 888 of file [ftypes.h](#).

4.51.2.606 NOFUNPOWERS

```
#define NOFUNPOWERS 0
```

Definition at line 890 of file [ftypes.h](#).

4.51.2.607 COMFUNPOWERS

```
#define COMFUNPOWERS 1
```

Definition at line 891 of file [ftypes.h](#).

4.51.2.608 ALLFUNPOWERS

```
#define ALLFUNPOWERS 2
```

Definition at line 892 of file [ftypes.h](#).

4.51.2.609 PROPERORDERFLAG

```
#define PROPERORDERFLAG 0
```

Definition at line 894 of file [ftypes.h](#).

4.51.2.610 REGULAR

```
#define REGULAR 0
```

Definition at line 896 of file [ftypes.h](#).

4.51.2.611 FINISH

```
#define FINISH 1
```

Definition at line 897 of file [ftypes.h](#).

4.51.2.612 POLYADD

```
#define POLYADD 1
```

Definition at line 899 of file [ftypes.h](#).

4.51.2.613 POLYSUB

```
#define POLYSUB 2
```

Definition at line 900 of file [ftypes.h](#).

4.51.2.614 POLYMUL

```
#define POLYMUL 3
```

Definition at line 901 of file [ftypes.h](#).

4.51.2.615 POLYDIV

```
#define POLYDIV 4
```

Definition at line 902 of file [ftypes.h](#).

4.51.2.616 POLYREM

```
#define POLYREM 5
```

Definition at line 903 of file [ftypes.h](#).

4.51.2.617 POLYGCD

```
#define POLYGCD 6
```

Definition at line 904 of file [ftypes.h](#).

4.51.2.618 POLYINTFAC

```
#define POLYINTFAC 7
```

Definition at line 905 of file [ftypes.h](#).

4.51.2.619 POLYNORM

```
#define POLYNORM 8
```

Definition at line 906 of file [ftypes.h](#).

4.51.2.620 MODNONE

```
#define MODNONE 0
```

Definition at line 908 of file [ftypes.h](#).

4.51.2.621 MODSUM

```
#define MODSUM 1
```

Definition at line 909 of file [ftypes.h](#).

4.51.2.622 MODMAX

```
#define MODMAX 2
```

Definition at line 910 of file [ftypes.h](#).

4.51.2.623 MODMIN

```
#define MODMIN 3
```

Definition at line 911 of file [ftypes.h](#).

4.51.2.624 MODLOCAL

```
#define MODLOCAL 4
```

Definition at line 912 of file [ftypes.h](#).

4.51.2.625 ELEMENTUSED

```
#define ELEMENTUSED 1
```

Definition at line 914 of file [ftypes.h](#).

4.51.2.626 ELEMENTLOADED

```
#define ELEMENTLOADED 2
```

Definition at line 915 of file [ftypes.h](#).

4.51.2.627 POSNEG

```
#define POSNEG 0x1
```

Definition at line 920 of file [ftypes.h](#).

4.51.2.628 INVERSETABLE

```
#define INVERSETABLE 0x2
```

Definition at line 921 of file [ftypes.h](#).

4.51.2.629 COEFFICIENTSONLY

```
#define COEFFICIENTSONLY 0x4
```

Definition at line 922 of file [ftypes.h](#).

4.51.2.630 ALSOPOWERS

```
#define ALSOPOWERS 0x8
```

Definition at line 923 of file [ftypes.h](#).

4.51.2.631 ALSOFUNARGS

```
#define ALSOFUNARGS 0x10
```

Definition at line 924 of file [ftypes.h](#).

4.51.2.632 ALSODOLLARS

```
#define ALSODOLLARS 0x20
```

Definition at line 925 of file [ftypes.h](#).

4.51.2.633 NOINVERSES

```
#define NOINVERSES 0x40
```

Definition at line 926 of file [ftypes.h](#).

4.51.2.634 POSITIVEONLY

```
#define POSITIVEONLY 0
```

Definition at line 928 of file [ftypes.h](#).

4.51.2.635 UNPACK

```
#define UNPACK 0x80
```

Definition at line 929 of file [ftypes.h](#).

4.51.2.636 NOUNPACK

```
#define NOUNPACK 0
```

Definition at line 930 of file [ftypes.h](#).

4.51.2.637 FROMFUNCTION

```
#define FROMFUNCTION 0x100
```

Definition at line 931 of file [ftypes.h](#).

4.51.2.638 VARNames

```
#define VARNames 0
```

Definition at line 933 of file [ftypes.h](#).

4.51.2.639 AUTONAMES

```
#define AUTONAMES 1
```

Definition at line 934 of file [ftypes.h](#).

4.51.2.640 EXPRNames

```
#define EXPRNames 2
```

Definition at line 935 of file [ftypes.h](#).

4.51.2.641 DOLLARNAMES

```
#define DOLLARNAMES 3
```

Definition at line 936 of file [ftypes.h](#).

4.51.2.642 DUMMYBUFFER

```
#define DUMMYBUFFER 1
```

Definition at line 991 of file [ftypes.h](#).

4.51.2.643 ALLARGS

```
#define ALLARGS 1
```

Definition at line 993 of file [ftypes.h](#).

4.51.2.644 NUMARG

```
#define NUMARG 2
```

Definition at line 994 of file [ftypes.h](#).

4.51.2.645 ARGRANGE

```
#define ARGRANGE 3
```

Definition at line 995 of file [ftypes.h](#).

4.51.2.646 MAKEARGS

```
#define MAKEARGS 4
```

Definition at line 996 of file [ftypes.h](#).

4.51.2.647 MAXRANGEINDICATOR

```
#define MAXRANGEINDICATOR 4
```

Definition at line 997 of file [ftypes.h](#).

4.51.2.648 REPLACEARG

```
#define REPLACEARG 5
```

Definition at line 998 of file [ftypes.h](#).

4.51.2.649 ENCODEARG

```
#define ENCODEARG 6
```

Definition at line 999 of file [ftypes.h](#).

4.51.2.650 DECODEARG

```
#define DECODEARG 7
```

Definition at line 1000 of file [ftypes.h](#).

4.51.2.651 IMplodeARG

```
#define IMplodeARG 8
```

Definition at line 1001 of file [ftypes.h](#).

4.51.2.652 EXPLODEARG

```
#define EXPLODEARG 9
```

Definition at line 1002 of file [ftypes.h](#).

4.51.2.653 PERMUTEARG

```
#define PERMUTEARG 10
```

Definition at line 1003 of file [ftypes.h](#).

4.51.2.654 REVERSEARG

```
#define REVERSEARG 11
```

Definition at line 1004 of file [ftypes.h](#).

4.51.2.655 CYCLEARG

```
#define CYCLEARG 12
```

Definition at line 1005 of file [ftypes.h](#).

4.51.2.656 ISLYNDON

```
#define ISLYNDON 13
```

Definition at line 1006 of file [ftypes.h](#).

4.51.2.657 ISLYNDONR

```
#define ISLYNDONR 14
```

Definition at line 1007 of file [ftypes.h](#).

4.51.2.658 TOLYNDON

```
#define TOLYNDON 15
```

Definition at line 1008 of file [ftypes.h](#).

4.51.2.659 TOLYNDONR

```
#define TOLYNDONR 16
```

Definition at line 1009 of file [ftypes.h](#).

4.51.2.660 ADDARG

```
#define ADDARG 17
```

Definition at line 1010 of file [ftypes.h](#).

4.51.2.661 MULTIPLYARG

```
#define MULTIPLYARG 18
```

Definition at line 1011 of file [ftypes.h](#).

4.51.2.662 DROPARG

```
#define DROPARG 19
```

Definition at line 1012 of file [ftypes.h](#).

4.51.2.663 SELECTARG

```
#define SELECTARG 20
```

Definition at line 1013 of file [ftypes.h](#).

4.51.2.664 DEDUPARG

```
#define DEDUPARG 21
```

Definition at line 1014 of file [ftypes.h](#).

4.51.2.665 ZTOHARG

```
#define ZTOHARG 22
```

Definition at line 1015 of file [ftypes.h](#).

4.51.2.666 HTOZARG

```
#define HTOZARG 23
```

Definition at line 1016 of file [ftypes.h](#).

4.51.2.667 BASECODE

```
#define BASECODE 1
```

Definition at line 1024 of file [ftypes.h](#).

4.51.2.668 YESLYNDON

```
#define YESLYNDON 1
```

Definition at line 1025 of file [ftypes.h](#).

4.51.2.669 NOLYNDON

```
#define NOLYNDON 2
```

Definition at line 1026 of file [ftypes.h](#).

4.51.2.670 TOPOLYNOMIALFLAG

```
#define TOPOLYNOMIALFLAG 1
```

Definition at line 1028 of file [ftypes.h](#).

4.51.2.671 FACTARGFLAG

```
#define FACTARGFLAG 2
```

Definition at line 1029 of file [ftypes.h](#).

4.51.2.672 OLDFACTARG

```
#define OLDFACTARG 1
```

Definition at line 1031 of file [ftypes.h](#).

4.51.2.673 NEWFACTARG

```
#define NEWFACTARG 0
```

Definition at line 1032 of file [ftypes.h](#).

4.51.2.674 FROMMODULEOPTION

```
#define FROMMODULEOPTION 0
```

Definition at line 1034 of file [ftypes.h](#).

4.51.2.675 FROMPOINTINSTRUCTION

```
#define FROMPOINTINSTRUCTION 1
```

Definition at line 1035 of file [ftypes.h](#).

4.51.2.676 EXTRASYMBOL

```
#define EXTRASYMBOL 0
```

Definition at line 1037 of file [ftypes.h](#).

4.51.2.677 REGULARSYMBOL

```
#define REGULARSYMBOL 1
```

Definition at line 1038 of file [ftypes.h](#).

4.51.2.678 EXPRESSIONNUMBER

```
#define EXPRESSIONNUMBER 2
```

Definition at line 1039 of file [ftypes.h](#).

4.51.2.679 O_NONE

```
#define O_NONE 0
```

Definition at line 1041 of file [ftypes.h](#).

4.51.2.680 O_CSE

```
#define O_CSE 1
```

Definition at line 1042 of file [ftypes.h](#).

4.51.2.681 O_CSEGREEDY

```
#define O_CSEGREEDY 2
```

Definition at line 1043 of file [ftypes.h](#).

4.51.2.682 O_GREEDY

```
#define O_GREEDY 3
```

Definition at line 1044 of file [ftypes.h](#).

4.51.2.683 O_OCCURRENCE

```
#define O_OCCURRENCE 0
```

Definition at line 1046 of file [ftypes.h](#).

4.51.2.684 O_MCTS

```
#define O_MCTS 1
```

Definition at line 1047 of file [ftypes.h](#).

4.51.2.685 O_SIMULATED_ANNEALING

```
#define O_SIMULATED_ANNEALING 2
```

Definition at line 1048 of file [ftypes.h](#).

4.51.2.686 O_FORWARD

```
#define O_FORWARD 0
```

Definition at line 1050 of file [ftypes.h](#).

4.51.2.687 O_BACKWARD

```
#define O_BACKWARD 1
```

Definition at line 1051 of file [ftypes.h](#).

4.51.2.688 O_FORWARDORBACKWARD

```
#define O_FORWARDORBACKWARD 2
```

Definition at line 1052 of file [ftypes.h](#).

4.51.2.689 O_FORWARDANDBACKWARD

```
#define O_FORWARDANDBACKWARD 3
```

Definition at line 1053 of file [ftypes.h](#).

4.51.2.690 OPTHEAD

```
#define OPTHEAD 3
```

Definition at line 1055 of file [ftypes.h](#).

4.51.2.691 DOALL

```
#define DOALL 1
```

Definition at line 1056 of file [ftypes.h](#).

4.51.2.692 ONLYFUNCTIONS

```
#define ONLYFUNCTIONS 2
```

Definition at line 1057 of file [ftypes.h](#).

4.51.2.693 INUSE

```
#define INUSE 1
```

Definition at line 1059 of file [ftypes.h](#).

4.51.2.694 COULDCOMMUTE

```
#define COULDCOMMUTE 2
```

Definition at line 1060 of file [ftypes.h](#).

4.51.2.695 DOESNOTCOMMUTE

```
#define DOESNOTCOMMUTE 4
```

Definition at line 1061 of file [ftypes.h](#).

4.51.2.696 DICT_NONNUMBERS

```
#define DICT_NONNUMBERS 0
```

Definition at line 1063 of file [ftypes.h](#).

4.51.2.697 DICT_INTEGERONLY

```
#define DICT_INTEGERONLY 1
```

Definition at line 1064 of file [ftypes.h](#).

4.51.2.698 DICT_RATIONALONLY

```
#define DICT_RATIONALONLY 2
```

Definition at line 1065 of file [ftypes.h](#).

4.51.2.699 DICT_ALLNUMBERS

```
#define DICT_ALLNUMBERS 3
```

Definition at line 1066 of file [ftypes.h](#).

4.51.2.700 DICT_NOVARIABLES

```
#define DICT_NOVARIABLES 0
```

Definition at line 1067 of file [ftypes.h](#).

4.51.2.701 DICT_DOVARIABLES

```
#define DICT_DOVARIABLES 1
```

Definition at line 1068 of file [ftypes.h](#).

4.51.2.702 DICT_NOSPECIALS

```
#define DICT_NOSPECIALS 0
```

Definition at line 1069 of file [ftypes.h](#).

4.51.2.703 DICT_DOSPECIALS

```
#define DICT_DOSPECIALS 1
```

Definition at line 1070 of file [ftypes.h](#).

4.51.2.704 DICT_NOFUNWITHARGS

```
#define DICT_NOFUNWITHARGS 0
```

Definition at line 1071 of file [ftypes.h](#).

4.51.2.705 DICT_DOFUNWITHARGS

```
#define DICT_DOFUNWITHARGS 1
```

Definition at line 1072 of file [ftypes.h](#).

4.51.2.706 DICT_NOTINDOLLARS

```
#define DICT_NOTINDOLLARS 0
```

Definition at line 1073 of file [ftypes.h](#).

4.51.2.707 DICT_INDOLLARS

```
#define DICT_INDOLLARS 1
```

Definition at line 1074 of file [ftypes.h](#).

4.51.2.708 DICT_INTEGERNUMBER

```
#define DICT_INTEGERNUMBER 1
```

Definition at line 1076 of file [ftypes.h](#).

4.51.2.709 DICT_RATIONALNUMBER

```
#define DICT_RATIONALNUMBER 2
```

Definition at line 1077 of file [ftypes.h](#).

4.51.2.710 DICT_SYMBOL

```
#define DICT_SYMBOL 3
```

Definition at line 1078 of file [ftypes.h](#).

4.51.2.711 DICT_VECTOR

```
#define DICT_VECTOR 4
```

Definition at line 1079 of file [ftypes.h](#).

4.51.2.712 DICT_INDEX

```
#define DICT_INDEX 5
```

Definition at line 1080 of file [ftypes.h](#).

4.51.2.713 DICT_FUNCTION

```
#define DICT_FUNCTION 6
```

Definition at line 1081 of file [ftypes.h](#).

4.51.2.714 DICT_FUNCTION_WITH_ARGUMENTS

```
#define DICT_FUNCTION_WITH_ARGUMENTS 7
```

Definition at line 1082 of file [ftypes.h](#).

4.51.2.715 DICT_SPECIALCHARACTER

```
#define DICT_SPECIALCHARACTER 8
```

Definition at line 1083 of file [ftypes.h](#).

4.51.2.716 DICT_RANGE

```
#define DICT_RANGE 9
```

Definition at line 1084 of file [ftypes.h](#).

4.51.2.717 READSPECTATORFLAG

```
#define READSPECTATORFLAG 3
```

Definition at line 1086 of file [ftypes.h](#).

4.51.2.718 GLOBALSPECTATORFLAG

```
#define GLOBALSPECTATORFLAG 1
```

Definition at line 1087 of file [ftypes.h](#).

4.51.2.719 ORDEREDSET

```
#define ORDEREDSET 1
```

Definition at line 1089 of file [ftypes.h](#).

4.51.2.720 DENSETABLE

```
#define DENSETABLE 1
```

Definition at line 1091 of file [ftypes.h](#).

4.51.2.721 SPARSETABLE

```
#define SPARSETABLE 0
```

Definition at line 1092 of file [ftypes.h](#).

4.51.2.722 TOPOLOGIESONLY

```
#define TOPOLOGIESONLY 1
```

Definition at line 1098 of file [ftypes.h](#).

4.51.2.723 WITHOUTNODES

```
#define WITHOUTNODES 2
```

Definition at line 1099 of file [ftypes.h](#).

4.51.2.724 WITHEDGES

```
#define WITHEDGES 4
```

Definition at line 1100 of file [ftypes.h](#).

4.51.2.725 WITHBLOCKS

```
#define WITHBLOCKS 8
```

Definition at line 1101 of file [ftypes.h](#).

4.51.2.726 WITHONEPISETS

```
#define WITHONEPISETS 16
```

Definition at line 1102 of file [ftypes.h](#).

4.51.2.727 WITHSYMMETRIZEI

```
#define WITHSYMMETRIZEI 32
```

Definition at line 1103 of file [ftypes.h](#).

4.51.2.728 WITHSYMMETRIZEF

```
#define WITHSYMMETRIZEF 64
```

Definition at line 1104 of file [ftypes.h](#).

4.51.2.729 CHECKEXTERN

```
#define CHECKEXTERN 128
```

Definition at line 1106 of file [ftypes.h](#).

4.51.2.730 ONEPARTI

```
#define ONEPARTI 256
```

Definition at line 1108 of file [ftypes.h](#).

4.51.2.731 ONEPARTR

```
#define ONEPARTR 512
```

Definition at line 1109 of file [ftypes.h](#).

4.51.2.732 ONSHELL

```
#define ONSHELL 1024
```

Definition at line 1110 of file [ftypes.h](#).

4.51.2.733 OFFSHELL

```
#define OFFSHELL 2048
```

Definition at line 1111 of file [ftypes.h](#).

4.51.2.734 NOSIGMA

```
#define NOSIGMA 4096
```

Definition at line 1112 of file [ftypes.h](#).

4.51.2.735 SIGMA

```
#define SIGMA 8192
```

Definition at line 1113 of file [ftypes.h](#).

4.51.2.736 NOSNAIL

```
#define NOSNAIL 16384
```

Definition at line 1114 of file [ftypes.h](#).

4.51.2.737 SNAIL

```
#define SNAIL 32768
```

Definition at line 1115 of file [ftypes.h](#).

4.51.2.738 NOTADPOLE

```
#define NOTADPOLE 65536
```

Definition at line 1116 of file [ftypes.h](#).

4.51.2.739 TADPOLE

```
#define TADPOLE 131072
```

Definition at line 1117 of file [ftypes.h](#).

4.51.2.740 SIMPLE

```
#define SIMPLE 262144
```

Definition at line 1118 of file [ftypes.h](#).

4.51.2.741 NOTSIMPLE

```
#define NOTSIMPLE 524288
```

Definition at line 1119 of file [ftypes.h](#).

4.51.2.742 BIPART

```
#define BIPART 1048576
```

Definition at line 1120 of file [ftypes.h](#).

4.51.2.743 NONBIPART

```
#define NONBIPART 2097152
```

Definition at line 1121 of file [ftypes.h](#).

4.51.2.744 CYCLI

```
#define CYCLI 4194304
```

Definition at line 1122 of file [ftypes.h](#).

4.51.2.745 CYCLR

```
#define CYCLR 8388608
```

Definition at line 1123 of file [ftypes.h](#).

4.51.2.746 FLOOP

```
#define FLOOP 16777216
```

Definition at line 1124 of file [ftypes.h](#).

4.51.2.747 NOTFLOOP

```
#define NOTFLOOP 33554432
```

Definition at line 1125 of file [ftypes.h](#).

4.51.3 Typedef Documentation

4.51.3.1 PVFUNWP

```
typedef void(* PVFUNWP) (WORD *)
```

Definition at line 171 of file [ftypes.h](#).

4.51.3.2 PVFUNV

```
typedef void(* PVFUNV) (void)
```

Definition at line 172 of file [ftypes.h](#).

4.51.3.3 CFUN

```
typedef int(* CFUN) (void)
```

Definition at line 173 of file [ftypes.h](#).

4.51.3.4 TFUN

```
typedef int(* TFUN) (UBYTE *)
```

Definition at line 174 of file [ftypes.h](#).

4.51.3.5 TFUN1

```
typedef int(* TFUN1) (UBYTE *, int)
```

Definition at line 175 of file [ftypes.h](#).

4.52 ftypes.h

[Go to the documentation of this file.](#)

```

00001
00009 /* #[ License : */
00010 /*
00011 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00012 *   When using this file you are requested to refer to the publication
00013 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014 *   This is considered a matter of courtesy as the development was paid
00015 *   for by FOM the Dutch physics granting agency and we would like to
00016 *   be able to track its scientific use to convince FOM of its value
00017 *   for the community.
00018 *
00019 *   This file is part of FORM.
00020 *
00021 *   FORM is free software: you can redistribute it and/or modify it under the
00022 *   terms of the GNU General Public License as published by the Free Software
00023 *   Foundation, either version 3 of the License, or (at your option) any later
00024 *   version.
00025 *
00026 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00029 *   details.
00030 *
00031 *   You should have received a copy of the GNU General Public License along
00032 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00033 */
00034 /* #[ License : */
00035
00051 #define WITHOUTERROR 0
00052 #define WITHERROR 1
00053
00054 /*
00055     The various streams. (look also in tools.c)
00056 */
00057
00058 #define FILESTREAM 0
00059 #define PREVARSTREAM 1
00060 #define PREREADSTREAM 2
00061 #define PIPESTREAM 3
00062 #define PRECALCSTREAM 4
00063 #define DOLLARSTREAM 5
00064 #define PREREADSTREAM2 6
00065 #define EXTERNALCHANNELSTREAM 7
00066 #define PREREADSTREAM3 8
00067 #define REVERSEFILESTREAM 9
00068 #define INPUTSTREAM 10
00069
00070 #define ENDOFSTREAM 0xFF
00071 #define ENDOFINPUT 0xFF
00072
00073 /*
00074     Types of files
00075 */
00076
00077 #define SUBROUTINEFILE 0
00078 #define PROCEDUREFILE 1
00079 #define HEADERFILE 2
00080 #define SETUPFILE 3
00081 #define TABLEBASEFILE 4
00082
00083 /*
00084     Types of modules
00085 */
00086
00087 #define FIRSTMODULE -1
00088 #define GLOBALMODULE 0
00089 #define SORTMODULE 1
00090 #define STOREMODULE 2
00091 #define CLEARMODULE 3
00092 #define ENDMODULE 4
00093
00094 #define POLYFUN 0
00095
00096 #define NOPARALLEL_DOLLAR 0x0001
00097 #define NOPARALLEL_RHS 0x0002
00098 #define NOPARALLEL_CONVPOLY 0x0004
00099 #define NOPARALLEL_SPECTATOR 0x0008
00100 #define NOPARALLEL_USER 0x0010
00101 #define NOPARALLEL_TBLDOLLAR 0x0100
00102 #define NOPARALLEL_NPROC 0x0200
00103 #define PARALLELFLAG 0x0000
00104

```

```

00105 #define PRENOACTION 0
00106 #define PRERAISEAFTER 1
00107 #define PRELOWERAFTER 2
00108 /*
00109 #define ELIUMOD 1
00110 #define ELIZMOD 2
00111 #define SKIUMOD 3
00112 #define SKIZMOD 4
00113 */
00114 #define WITHSEMICOLON 0
00115 #define WITHOUTSEMICOLON 1
00116 #define MODULEINSTACK 8
00117 #define EXECUTINGIF 0
00118 #define LOOKINGFORELSE 1
00119 #define LOOKINGFORENDIF 2
00120 #define NEWSTATEMENT 1
00121 #define OLDSTATEMENT 0
00122
00123 #define EXECUTINGPRESWITCH 0
00124 #define SEARCHINGPRECASE 1
00125 #define SEARCHINGPREENDSWITCH 2
00126
00127 #define PREPROONLY 1
00128 #define DUMPTOCOMPILER 2
00129 #define DUMPOUTTERMS 4
00130 #define DUMPINTERMS 8
00131 #define DUMPTOSORT 16
00132 #define DUMPTOPARALLEL 32
00133 #define THREADSDEBUG 64
00134
00135 #define ERROROUT 0
00136 #define INPUTOUT 1
00137 #define STATSOUT 2
00138 #define EXPRSOUT 3
00139 #define WRITEOUT 4
00140
00141 #define EXTERNALCHANNELOUT 5
00142
00143 #define NUMERICALLOOP 0
00144 #define LISTEDLOOP 1
00145 #define ONEEXPRESSION 2
00146
00147 #define PRETYPENONE 0
00148 #define PRETYPEIF 1
00149 #define PRETYPEDO 2
00150 #define PRETYPEPROCEDURE 3
00151 #define PRETYPESWITCH 4
00152 #define PRETYPEINSIDE 5
00153
00154 /*
00155      Type of statement. Used to make sure that the statements are in proper order
00156 */
00157
00158 #define DECLARATION 1
00159 #define SPECIFICATION 2
00160 #define DEFINITION 3
00161 #define STATEMENT 4
00162 #define TOOUTPUT 5
00163 #define ATENDOFMODULE 6
00164 #define MIXED2 8 /* unused */
00165 #define MIXED 9
00166
00167 /*
00168      The typedefs are to allow the compilers to do better error checking.
00169 */
00170
00171 typedef void (*PVFUNWP)(WORD *);
00172 typedef void (*PVFUNV)(void);
00173 typedef int (*CFUN)(void);
00174 typedef int (*TFUN)(UBYTE *);
00175 typedef int (*TFUN1)(UBYTE *,int);
00176
00177 /*
00178      Flags used in the compiler's KEYWORD tables.
00179 */
00180
00181 #define NOAUTO 0
00182 #define PARTEST 1 /* parentheses test */
00183 #define WITHAUTO 2 /* auto-declaration */
00184
00185 #define ALLVARIABLES -1
00186 #define SYMBOLONLY 1
00187 #define INDEXONLY 2
00188 #define VECTORONLY 4
00189 #define FUNCTIONONLY 8
00190 #define SETONLY 16
00191 #define EXPRESSIONONLY 32

```

```
00192
00193
00201
00202 #define CDELETE -1
00203 #define ANYTYPE -1
00204 #define CSYMBOL 0
00205 #define CINDEX 1
00206 #define CVECTOR 2
00207 #define CFUNCTION 3
00208 #define CSET 4
00209 #define CEXPRESSION 5
00210 #define CDOTPRODUCT 6
00211 #define CNUMBER 7
00212 #define CSUBEXP 8
00213 #define CDELTA 9
00214 #define CDOLLAR 10
00215 #define CDUBIOUS 11
00216 #define CRANGE 12
00217 #define CMODEL 13
00218 #define CVECTOR1 21
00219 #define CDOUBLEDOT 22
00220
00223 /*
00224      Types of tokens in the tokenizer.
00225 */
00226
00227 #define TSYMBOL -1
00228 #define TINDEX -2
00229 #define TVECTOR -3
00230 #define TFUNCTION -4
00231 #define TSET -5
00232 #define TEXPRESSSION -6
00233 #define TDOTPRODUCT -7
00234 #define TNUMBER -8
00235 #define TSUBEXP -9
00236 #define TDELTA -10
00237 #define TDOLLAR -11
00238 #define TDUBIOUS -12
00239 #define LPARENTHESIS -13
00240 #define RPARENTHESIS -14
00241 #define TWILDCARD -15
00242 #define TWILDARG -16
00243 #define TDOT -17
00244 #define LBRACE -18
00245 #define RBRACE -19
00246 #define TCOMMA -20
00247 #define TFUNOPEN -21
00248 #define TFUNCLOSE -22
00249 #define TMULTIPLY -23
00250 #define TDIVIDE -24
00251 #define TPOWER -25
00252 #define TPLUS -26
00253 #define TMINUS -27
00254 #define TNOT -28
00255 #define TENDOFIT -29
00256 #define TSETOPEN -30
00257 #define TSETCLOSE -31
00258 #define TGENINDEX -32
00259 #define TCONJUGATE -33
00260 #define LRPARENTHESSES -34
00261 #define TNUMBER1 -35
00262 #define TPOWER1 -36
00263 #define TEMPTY -37
00264 #define TSETNUM -38
00265 #define TSGAMMA -39
00266 #define TSETDOL -40
00267 #define TFLOAT -41
00268
00269 #define TYPEISFUN 0
00270 #define TYPEISSUB 1
00271 #define TYPEISMYSTERY -1
00272
00273 #define LHSEXIDEX 2
00274 #define LHSIDE 1
00275 #define RHSIDE 0
00276
00277 /*
00278      Output modes
00279 */
00280
00281 #define FORTRANMODE 1
00282 #define REDUCEMODE 2
00283 #define MAPLEMODE 3
00284 #define MATHEMATICAMODE 4
00285 #define CMODE 5
00286 #define VORTRANMODE 6
00287 #define PFORTRANMODE 7
```

```

00288 #define DOUBLEFORTRANMODE 33
00289 #define DOUBLEPRECISIONFLAG 32
00290 #define NODOUBLEMASK 31
00291 #define QUADRUPLEFORTRANMODE 65
00292 #define QUADRUPLEPRECISIONFLAG 64
00293 #define ALLINTEGERDOUBLE 128
00294 #define NOQUADMASK 63
00295 #define NORMALFORMAT 0
00296 #define NOSPACEFORMAT 1
00297
00298 #define ISNOTFORTRAN90 0
00299 #define ISFORTRAN90 1
00300
00301 #define ALSOREVERSE 1
00302 #define CHISHOLM 2
00303 #define NOTRICK 16
00304
00305 #define SORTLOWFIRST 0
00306 #define SORTHIGHFIRST 1
00307 #define SORTPOWERFIRST 2
00308 #define SORTANTIPOWER 3
00309
00310 /* These control the output of WriteStats. The different sort
00311  * types have differently formatted stats. These variables should
00312  * not have arbitrary values since they are also used as array
00313  * indices by WriteStats. */
00314 #define STATSSPLITMERGE 0
00315 #define STATSMERGETOFILE 1
00316 #define STATSPOSTSORT 2
00317 /* CHECKLOGTYPE implies that statistics will not be printed in
00318  * the log file if AM.LogType = 1 (when running with -ll or -F). */
00319 #define NOCHECKLOGTYPE 0
00320 #define CHECKLOGTYPE 1
00321
00322 #define NMIN4SHIFT 4
00323 /*
00324     The next are the main codes.
00325     Note: SETSET is not allowed to be 4*n+1
00326     We use those codes in CoIdExpression for function information
00327     after the pattern. Because SETSET also stands there we have to
00328     be careful!!
00329     Don't forget MAXBUILTINFUNCTION when adding codes!
00330     The object FUNCTION is at the start of the functions that are in regular
00331     notation. Anything below it is in special notation.
00332
00333     Remark: HAAKJE0 is for compression purposes and should only occur
00334     at moments that ARGWILD cannot occur.
00335 */
00336 #define SYMBOL 1
00337 #define DOTPRODUCT 2
00338 #define VECTOR 3
00339 #define INDEX 4
00340 #define EXPRESSION 5
00341 #define SUBEXPRESSION 6
00342 #define DOLLAREXPRESSION 7
00343 #define SETSET 8
00344 #define ARGWILD 9
00345 #define MINVECTOR 10
00346 #define SETEXP 11
00347 #define DOLLAREXP2 12
00348 #define HAAKJE0 9
00349 #define FUNCTION 20
00350
00351 #define TMPPOLYFUN 14
00352 #define ARGFIELD 15
00353 #define SNUMBER 16
00354 #define LNUMBER 17
00355 #define HAAKJE 18
00356 #define DELTA 19
00357 #define EXPONENT 20
00358 #define DENOMINATOR 21
00359 #define SETFUNCTION 22
00360 #define GAMMA 23
00361 #define GAMMAI 24
00362 #define GAMMAFIVE 25
00363 #define GAMMASIX 26
00364 #define GAMMASEVEN 27
00365 #define SUMF1 28
00366 #define SUMF2 29
00367 #define DUMFUN 30
00368 #define REPLACEMENT 31
00369 #define REVERSEFUNCTION 32
00370 #define DISTRIBUTION 33
00371 #define DELTA3 34
00372 #define DUMMYFUN 35
00373 #define DUMMYTEN 36
00374 #define LEVICIVITA 37

```

```
00375 #define FACTORIAL 38
00376 #define INVERSEFACTORIAL 39
00377 #define BINOMIAL 40
00378 #define NUMARGSFUN 41
00379 #define SIGNFUN 42
00380 #define MODFUNCTION 43
00381 #define MOD2FUNCTION 44
00382 #define MINFUNCTION 45
00383 #define MAXFUNCTION 46
00384 #define ABSFUNCTION 47
00385 #define SIGFUNCTION 48
00386 #define INTFUNCTION 49
00387 #define THETA 50
00388 #define THETA2 51
00389 #define DELTA2 52
00390 #define DELTAP 53
00391 #define BERNOULLIFUNCTION 54
00392 #define COUNTFUNCTION 55
00393 #define MATCHFUNCTION 56
00394 #define PATTERNFUNCTION 57
00395 #define TERMFUNCTION 58
00396 #define CONJUGATION 59
00397 #define ROOTFUNCTION 60
00398 #define TABLEFUNCTION 61
00399 #define FIRSTBRACKET 62
00400 #define TERMSINEXPR 63
00401 #define NUMTERMSFUN 64
00402 #define GCDFUNCTION 65
00403 #define DIVFUNCTION 66
00404 #define REMFUNCTION 67
00405 #define MAXPOWEROF 68
00406 #define MINPOWEROF 69
00407 #define TABLESTUB 70
00408 #define FACTORIN 71
00409 #define TERMSINBRACKET 72
00410 #define WILDARGFUN 73
00411 /*
00412     In the past we would add new functions here and raise the numbers
00413     on the special reserved names. This is impractical in the light of
00414     the .sav files. The .sav files need a mechanism that contains the
00415     value of MAXBUILTINFUNCTION at the moment of writing. This allows
00416     form corrections if this value has changed in the mean time.
00417 */
00418 #define SQRTFUNCTION 74
00419 #define LNFUNCTION 75
00420 #define SINFUNCTION 76
00421 #define COSFUNCTION 77
00422 #define TANFUNCTION 78
00423 #define ASINFUNCTION 79
00424 #define ACOSFUNCTION 80
00425 #define ATANFUNCTION 81
00426 #define ATAN2FUNCTION 82
00427 #define SINHFUNCTION 83
00428 #define COSHFUNCTION 84
00429 #define TANHFUNCTION 85
00430 #define ASINHFUNCTION 86
00431 #define ACOSHFUNCTION 87
00432 #define ATANHFUNCTION 88
00433 #define LI2FUNCTION 89
00434 #define LINFUNCTION 90
00435
00436 #define EXTRASYMFUN 91
00437 #define RANDOMFUNCTION 92
00438 #define RANPERM 93
00439 #define NUMFACTORS 94
00440 #define FIRSTTERM 95
00441 #define CONTENTTERM 96
00442 #define PRIMENUMBER 97
00443 #define EXTEUCLIDEAN 98
00444 #define MAKERATIONAL 99
00445 #define INVERSEFUNCTION 100
00446 #define IDFUNCTION 101
00447 #define PUTFIRST 102
00448 #define PERMUTATIONS 103
00449 #define PARTITIONS 104
00450 #define MULFUNCTION 105
00451 #define MOEBIUS 106
00452 #define TOPOLOGIES 107
00453 #define DIAGRAMS 108
00454 #define TOPO 109
00455 #define NODEFUNCTION 110
00456 #define EDGE 111
00457 #define SIZEOFFUNCTION 112
00458 #define BLOCK 113
00459 #define ONEPI 114
00460 #define PHI 115
00461 #define FLOATFUN 116
```

```
00462 #define TOFLOAT 117
00463 #define TORAT 118
00464 #define MZV 119
00465 #define EULER 120
00466 #define MZVHALF 121
00467 #define AGMFUNCTION 122
00468 #define GAMMAFUN 123
00469 #define EXPFUNCTION 124
00470 #define HPLFUNCTION 125
00471 #define MPLFUNCTION 126
00472 #define MAXBUILTINFUNCTION 126
00473
00474 #define FIRSTUSERFUNCTION 150
00475
00476 #define ALLFUNCTIONS -1
00477 #define ALLMZVFUNCTIONS -2
00478
00479 /*
00480     Note: if we add a builtin table we have to look also inside names.c
00481     in the routine Globalize because there we assume there does not exist
00482     such an object
00483 */
00484
00485 #define ISYMBOL 0
00486 #define PISYMBOL 1
00487 #define COEFFSYMBOL 2
00488 #define NUMERATORSYMBOL 3
00489 #define DENOMINATORSYMBOL 4
00490 #define WILDARGSYMBOL 5
00491 #define DIMENSIONSYMBOL 6
00492 #define FACTORSYMBOL 7
00493 #define SEPARATESYMBOL 8
00494 #define EESYMBOL 9
00495 #define EMSYMBOL 10
00496
00497 #define BUILTINSYMBOLS 11
00498 #define FIRSTUSERSYMBOL 20
00499
00500 #define BUILTININDICES 1
00501 #define BUILTINVECTORS 1
00502 #define BUILTINDOLLARS 1
00503
00504 #define WILDARGVECTOR 0
00505 #define WILDARGINDEX 0
00506
00507 /*
00508     The objects that have a name that starts with TYPE are codes of statements
00509     made by the compiler. Each statement starts with such a code, followed by
00510     its size. For how most of these statements are used can be seen in the
00511     Generator function in the file proces.c
00512     TYPEOPERATION is an anachronism that remains used only for the statements
00513     that are executed in the file opera.c (like traces and contractions).
00514 */
00515
00516 #define TYPEEXPRESSION 0
00517 #define TYPEIDNEW 1
00518 #define TYPEIDOLD 2
00519 #define TYPEOPERATION 3
00520 #define TYPEPERPEAT 4
00521 #define TYPEENDREPEAT 5
00522 /*
00523     The next counts must be higher than the ones before
00524 */
00525 #define TYPECOUNT 20
00526 #define TYPEMULT 21
00527 #define TYPEGOTO 22
00528 #define TYPEDISCARD 23
00529 #define TYPEIF 24
00530 #define TYPEELSE 25
00531 #define TYPEELIF 26
00532 #define TYPEENDIF 27
00533 #define TYPESUM 28
00534 #define TYPECHISHOLM 29
00535 #define TYPEPERVERSE 30
00536 #define TYPEARG 31
00537 #define TYPENORM 32
00538 #define TYPENORM2 33
00539 #define TYPENORM3 34
00540 #define TYPEEXIT 35
00541 #define TYPESETEXIT 36
00542 #define TYPEPRINT 37
00543 #define TYPEFPRINT 38
00544 #define TYPEREDEFPRE 39
00545 #define TYPESPLITARG 40
00546 #define TYPESPLITARG2 41
00547 #define TYPEFACTARG 42
00548 #define TYPEFACTARG2 43
```

```
00549 #define TYPETRY 44
00550 #define TYPEASSIGN 45
00551 #define TYPERENUMBER 46
00552 #define TYPESUMFIX 47
00553 #define TYPEFINDLOOP 48
00554 #define TYPEUNRAVEL 49
00555 #define TYPEADJUSTBOUNDS 50
00556 #define TYPEINSIDE 51
00557 #define TYPETERM 52
00558 #define TYPESORT 53
00559 #define TYPEDETCURDUM 54
00560 #define TYPEINEXPRESSION 55
00561 #define TYPESPLITFIRSTARG 56
00562 #define TYPESPLITLASTARG 57
00563 #define TYPEMERGE 58
00564 #define TYPETESTUSE 59
00565 #define TYPEAPPLY 60
00566 #define TYPEAPPLYRESET 61
00567 #define TYPECHAININ 62
00568 #define TYPECHAINOUT 63
00569 #define TYPEENORM4 64
00570 #define TYPEFACTOR 65
00571 #define TYPEARGIMPLode 66
00572 #define TYPEARGEXPLODE 67
00573 #define TYPEDENOMINATORS 68
00574 #define TYPESTUFFLE 69
00575 #define TYPEDROPCEFFICIENT 70
00576 #define TYPETRANSFORM 71
00577 #define TYPETOPOLYNOMIAL 72
00578 #define TYPEFROMPOLYNOMIAL 73
00579 #define TYPEDOLOOP 74
00580 #define TYPEENDDOLOOP 75
00581 #define TYPEDROPSYMBOLS 76
00582 #define TYPEPUTINSIDE 77
00583 #define TYPETOSPECTATOR 78
00584 #define TYPEARGTOEXTRASymbol 79
00585 #define TYPECANONICALIZE 80
00586 #define TYPESWITCH 81
00587 #define TYPEENDSWITCH 82
00588 #define TYPESETUSERFLAG 83
00589 #define TYPECLEARUSERFLAG 84
00590 #define TYPEALLLOOPS 85
00591 #define TYPEALLPATHS 86
00592 #ifdef WITHFLOAT
00593 #define TYPEEVALUATE 87
00594 #define TYPEEXPAND 88
00595 #define TYPETOFloat 89
00596 #define TYPETORAT 90
00597 #define TYPESTRICTROUNDING 91
00598 #define TYPECHOP 92
00599 #endif
00600
00601 /*
00602     The codes for the 'operations' that are part of TYPEOPERATION.
00603 */
00604
00605 #define TAKETRACE 1
00606 #define CONTRACT 2
00607 #define RATIO 3
00608 #define SYMMETRIZE 4
00609 #define TENVEC 5
00610 #define SUMNUM1 6
00611 #define SUMNUM2 7
00612
00613 /*
00614     The various types of wildcards.
00615 */
00616
00617 #define WILDDUMMY 0
00618 #define SYMTONUM 1
00619 #define SYMTOSYM 2
00620 #define SYMTOSUB 3
00621 #define VECTOMIN 4
00622 #define VECTOVEC 5
00623 #define VECTOSUB 6
00624 #define INDTOIND 7
00625 #define INDTOSUB 8
00626 #define FUNTOFUN 9
00627 #define ARGTOARG 10
00628 #define ARLTOARL 11
00629 #define EXPTOEXP 12
00630 #define FROMBRAC 13
00631 #define FROMSET 14
00632 #define SETTONUM 15
00633 #define WILDCARDS 16
00634 #define SETNUMBER 17
00635 #define LOADDOLLAR 18
```



```
00636 /*
00637     Some new types of wildcards that hold only for function arguments.
00638 */
00639 #define NUMTONUM 20
00640 #define NUMTOSYM 21
00641 #define NUMTOIND 22
00642 #define NUMTOSUB 23
00643 /*
00644     Flag or-ed with ARGTOARG to give the number of arguments.
00645 */
00646 #define EATTENSOR 0x2000
00647
00648 /*
00649     Dirty flags (introduced when functions got a field with a dirty flag)
00650 */
00651
00652 #define CLEANFLAG 0
00653 #define DIRTYFLAG 1
00654 #define DIRTYSYMLFLAG 2
00655 #define MUSTCLEANPRF 4
00656 #define SUBTERMUSED1 8
00657 #define SUBTERMUSED2 16
00658 #define ALLDIRTY (DIRTYFLAG|DIRTYSYMLFLAG)
00659
00660 #define ARGHEAD 2
00661 #define FUNHEAD 3
00662 #define SUBEXPSIZE 5
00663 #define EXPRHEAD 5
00664 #define TYPEARGHEADSIZE 6
00665
00666 /*
00667     Actions to be taken with expressions. They are marked with these objects
00668     during compilation.
00669 */
00670
00671 #define SKIP 1
00672 #define DROP 2
00673 #define HIDE 3
00674 #define UNHIDE 4
00675 #define INTOHIDE 5
00676
00677 /*
00678     Types of expressions
00679 */
00680
00681 #define LOCALEXPRESSSION 0
00682 #define SKIPLEXPRESSION 1
00683 #define DROPLEXPRESSSION 2
00684 #define DROPEDEXPRESSION 3
00685 #define GLOBALEXPRESSION 4
00686 #define SKIPGEXPRESSION 5
00687 #define DROPGEXPRESSION 6
00688 #define STOREDEXPRESSION 8
00689 #define HIDDENLEXPRESSSION 9
00690 #define HIDDENGEXPRESSION 13
00691 #define INCXPRESSSION 9
00692 #define HIDELEXPRESSSION 10
00693 #define HIDEGEXPRESSION 14
00694 #define DROPHLEXPRESSSION 11
00695 #define DROPHGEXPRESSION 15
00696 #define UNHIDELEXPRESSSION 12
00697 #define UNHIDEGEXPRESSION 16
00698 #define INTOHIDELEXPRESSSION 17
00699 #define INTOHIDEGEXPRESSION 18
00700 #define SPECTATOREXPRESSSION 19
00701 #define DROPSPECTATOREXPRESSSION 20
00702 #define SKIPUNHIDELEXPRESSSION 21
00703 #define SKIPUNHIDEGEXPRESSION 22
00704
00705 #define PRINTOFF 0
00706 #define PRINTON 1
00707 #define PRINTCONTENTS 2
00708 #define PRINTCONTENT 3
00709 #define PRINTLFILE 4
00710 #define PRINTONETERM 8
00711 #define PRINTONEFUNCTION 16
00712 #define PRINTALL 32
00713
00714 /*
00715     Special codes for the replace variable in the EXPRESSIONS struct
00716 */
00717
00718 #define REGULAREXPRESSSION -1
00719 #define REDEFINEDEXPRESSSION -2
00720 #define NEWLYDEFINEDEXPRESSSION -3
00721
00722 #define VERTEXFUNCTION -1
```

```

00730 #define GENERALFUNCTION 0
00731 #define TENSORFUNCTION 2
00732 #define GAMMAFUNCTION 4
00735 /*
00736     Special sets
00737 */
00738
00739 #define POS_      0 /* integer > 0 */
00740 #define POS0_    1 /* integer >= 0 */
00741 #define NEG_     2 /* integer < 0 */
00742 #define NEG0_   3 /* integer <= 0 */
00743 #define EVEN_   4 /* integer (even) */
00744 #define ODD_    5 /* integer (odd) */
00745 #define Z_      6 /* integer */
00746 #define SYMBOL_ 7 /* symbol only */
00747 #define FIXED_  8 /* fixed index */
00748 #define INDEX_  9 /* index only */
00749 #define Q_      10 /* rational */
00750 #define DUMMYINDEX_ 11 /* dummy index only */
00751 #define VECTOR_  12 /* vector only */
00752
00753 /*
00754     Special indices.
00755 */
00756
00757 #define GAMMA1 0
00758 #define GAMMA5 -1
00759 #define GAMMA6 -2
00760 #define GAMMA7 -3
00761 #define FUNNYVEC -4
00762 #define FUNNYWILD -5
00763 #define SUMMEDIND -6
00764 #define NOINDEX -7
00765 #define FUNNYDOLLAR -8
00766 #define EMPTYINDEX -9
00767
00768 /*
00769     The next one should be less than all of the above special indices.
00770 */
00771
00772 #define MINSPEC -10
00773
00774 #define USEDFLAG 2
00775 #define DUMMYFLAG 1
00776
00777 #define MAINSORT 0
00778 #define FUNCTIONSORT 1
00779 #define SUBSORT 2
00780
00781 #define FLOATMODE 1
00782 #define RATIONALMODE 0
00783
00784 #define NUMSPECSETS 10
00785
00786 #define ISZERO 1
00787 #define ISUNMODIFIED 2
00788 #define ISCOMPRESSED 4
00789 #define ISINRHS 8
00790 #define ISFACTORIZED 16
00791 #define TOBEFACTORED 32
00792 #define TOBEUNFACTORED 64
00793 #define KEEPZERO 128
00794
00795 #define VARTYPENONE 0
00796 #define VARTYPECOMPLEX 1
00797 #define VARTYPEIMAGINARY 2
00798 #define VARTYPEROOTOFUNITY 4
00799 #define VARTYPEMINUS 8
00800 #define CYCLESYMMETRIC 1
00801 #define RCYCLESYMMETRIC 2
00802 #define SYMMETRIC 3
00803 #define ANTISYMMETRIC 4
00804 #define REVERSEORDER 256
00805
00806 /*
00807     Types of id statements (substitutions)
00808 */
00809
00810 #define SUBMULTI 1
00811 #define SUBONCE 2
00812 #define SUBONLY 3
00813 #define SUBMANY 4
00814 #define SUBVECTOR 5
00815 #define SUBSELECT 6
00816 #define SUBALL 7
00817 #define SUBMASK 15
00818 #define SUBDISORDER 16

```

```
00819 #define SUBAFTER 32
00820 #define SUBAFTERNOT 64
00821
00822 #define IDHEAD 6
00823
00824 #define DOLLARFLAG 1
00825 #define NORMALIZEFLAG 2
00826
00827 #define GIDENT 1
00828 #define GFIVE 4
00829 #define GPLUS 3
00830 #define GMINUS 2
00831
00832 /*
00833      Types of objects inside an if clause.
00834 */
00835
00836 #define LONGNUMBER 1
00837 #define MATCH 2
00838 #define COEFFI 3
00839 #define SUBEXPR 4
00840 #define MULTIPLEOF 5
00841 #define IFDOLLAR 6
00842 #define IFEXPRESSION 7
00843 #define IFDOLLAREXTRA 8
00844 #define IFISFACTORIZED 9
00845 #define IFOCCURS 10
00846 #define IFUSERFLAG 11
00847 #define IFFLOATNUMBER 12
00848 #define GREATER 0
00849 #define GREATEREQUAL 1
00850 #define LESS 2
00851 #define LESSEQUAL 3
00852 #define EQUAL 4
00853 #define NOTEQUAL 5
00854 #define ORCOND 6
00855 #define ANDCOND 7
00856 #define DUMMY 1
00857 #define SORT 1
00858 #define STORE 2
00859 #define END 3
00860 #define GLOBAL 4
00861 #define CLEAR 5
00862
00863 #define VECTBIT 1
00864 #define DOTPBIT 2
00865 #define FUNBIT 4
00866 #define SETBIT 8
00867
00868 #define EXTRAPARAMETER 0x4000
00869 #define GENCOMMUTE 0
00870 #define GENNONCOMMUTE 0x2000
00871
00872 #define NAMENOTFOUND -9
00873
00874 /*
00875      Types of dollar expressions.
00876 */
00877
00878 #define DOLUNDEFINED 0
00879 #define DOLNUMBER 1
00880 #define DOLARGUMENT 2
00881 #define DOLSUBTERM 3
00882 #define DOLTERMS 4
00883 #define DOLWILDARGS 5
00884 #define DOLINDEX 6
00885 #define DOLZERO 7
00886
00887 #define FINDLOOP 0
00888 #define REPLACELOOP 1
00889
00890 #define NOFUNPOWERS 0
00891 #define COMFUNPOWERS 1
00892 #define ALLFUNPOWERS 2
00893
00894 #define PROPERORDERFLAG 0
00895
00896 #define REGULAR 0
00897 #define FINISH 1
00898
00899 #define POLYADD 1
00900 #define POLYSUB 2
00901 #define POLYMUL 3
00902 #define POLYDIV 4
00903 #define POLYREM 5
00904 #define POLYGCD 6
00905 #define POLYINTFAC 7
```

```

00906 #define POLYNORM 8
00907
00908 #define MODNONE 0
00909 #define MODSUM 1
00910 #define MODMAX 2
00911 #define MODMIN 3
00912 #define MODLOCAL 4
00913
00914 #define ELEMENTUSED 1
00915 #define ELEMENTLOADED 2
00916 /*
00917     Variables for the modulus statement, flags in AC.modmode
00918     For explanation, see CoModulus
00919 */
00920 #define POSNEG 0x1
00921 #define INVERSETABLE 0x2
00922 #define COEFFICIENTSONLY 0x4
00923 #define ALSOPOWERS 0x8
00924 #define ALSOFUNARGS 0x10
00925 #define ALSODOLLARS 0x20
00926 #define NOINVERSES 0x40
00927
00928 #define POSITIVEONLY 0
00929 #define UNPACK 0x80
00930 #define NOUNPACK 0
00931 #define FROMFUNCTION 0x100
00932
00933 #define VARNAMES 0
00934 #define AUTONAMES 1
00935 #define EXPRNAMES 2
00936 #define DOLLARNAMES 3
00937
00938 #ifdef WITHPTHREADS
00939 /*
00940     Signals that the workers have to react to
00941 */
00942
00943 #define TERMINATETHREAD -1
00944 #define STARTNEWEXPRESSION 1
00945 #define LOWESTLEVELGENERATION 2
00946 #define FINISHEXPRESSION 3
00947 #define CLEANUPEXPRESSION 4
00948 #define HIGHERLEVELGENERATION 5
00949 #define STARTNEWMODULE 6
00950 #define CLAIMOUTPUT 7
00951 #define FINISHEXPRESSION2 8
00952 #define INISORTBOT 7
00953 #define RUNSORTBOT 8
00954 #define DOONEEXPRESSION 9
00955 #define DOBRACKETS 10
00956 #define CLEARCLOCK 11
00957 #define MCTSEXPANDTREE 12
00958 #define OPTIMIZEEXPRESSION 13
00959
00960 #define MASTERBUFFERISFULL 1
00961
00962 /*
00963     Bucket states
00964 */
00965
00966 #define BUCKETFREE 1
00967 #define BUCKETINUSE 0
00968 #define BUCKETCOMINGFREE 2
00969 #define BUCKETFILLED -1
00970 #define BUCKETATEND -2
00971 #define BUCKETTERMINATED 3
00972 #define BUCKETRELEASED 4
00973
00974 #define NUMBEROFBLOCKSINSORT 10
00975 #define MINIMUMNUMBEROFTERMS 10
00976
00977 #define BUCKETDOINGTERM 1
00978 #define BUCKETASSIGNED -1
00979 #define BUCKETTOBERELEASED -2
00980 #define BUCKETPREPARINGTERM 0
00981
00982 #define BUCKETDOINGTERMS 0
00983 #define BUCKETDOINGBRACKET 1
00984 #endif
00985
00986 /*
00987     The next variable is because there is some use of cbufnum that is
00988     probably irrelevant. We use here DUMMYBUFNUM instead of AC.cbufnum
00989     just in case we run into trouble later.
00990 */
00991 #define DUMMYBUFFER 1
00992

```

```
00993 #define ALLARGS      1
00994 #define NUMARG         2
00995 #define ARGRANGE       3
00996 #define MAKEARGS       4
00997 #define MAXRANGEINDICATOR 4
00998 #define REPLACEARG     5
00999 #define ENCODEARG      6
01000 #define DECODEARG      7
01001 #define IMplodeARG     8
01002 #define EXPLODEARG     9
01003 #define PERMUTEARG     10
01004 #define REVERSEARG     11
01005 #define CYCLEARG       12
01006 #define ISLYNDON       13
01007 #define ISLYNDONR      14
01008 #define TOLYNDON       15
01009 #define TOLYNDONR      16
01010 #define ADDARG         17
01011 #define MULTIPLYARG    18
01012 #define DROPARG        19
01013 #define SELECTARG      20
01014 #define DEDUPARG       21
01015 #define ZTOHARG        22
01016 #define HTOZARG        23
01017 /*
01018 #define ZTOSARG         24
01019 #define STOZARG         25
01020 #define HTOSARG         26
01021 #define STOARG          27
01022 */
01023
01024 #define BASECODE 1
01025 #define YESLYNDON 1
01026 #define NOLYNDON 2
01027
01028 #define TOPOLYNOMIALFLAG 1
01029 #define FACTARGFLAG 2
01030
01031 #define OLDFACTARG 1
01032 #define NEWFACTARG 0
01033
01034 #define FROMMODULEOPTION 0
01035 #define FROMPOINTINSTRUCTION 1
01036
01037 #define EXTRASYMBOL 0
01038 #define REGULARSYMBOL 1
01039 #define EXPRESSIONNUMBER 2
01040
01041 #define O_NONE 0
01042 #define O_CSE 1
01043 #define O_CSEGREEDY 2
01044 #define O_GREEDY 3
01045
01046 #define O_OCCURRENCE 0
01047 #define O_MCTS 1
01048 #define O_SIMULATED_ANNEALING 2
01049
01050 #define O_FORWARD 0
01051 #define O_BACKWARD 1
01052 #define O_FORWARDORBACKWARD 2
01053 #define O_FORWARDANDBACKWARD 3
01054
01055 #define OPTHEAD 3
01056 #define DOALL 1
01057 #define ONLYFUNCTIONS 2
01058
01059 #define INUSE 1
01060 #define COULDCOMMUTE 2
01061 #define DOESNOTCOMMUTE 4
01062
01063 #define DICT_NONUMBERS 0
01064 #define DICT_INTEGERONLY 1
01065 #define DICT_RATIONALONLY 2
01066 #define DICT_ALLNUMBERS 3
01067 #define DICT_NOVARIABLES 0
01068 #define DICT_DOVARIABLES 1
01069 #define DICT_NOSPECIALS 0
01070 #define DICT_DOSPECIALS 1
01071 #define DICT_NOFUNWITHARGS 0
01072 #define DICT_DOFUNWITHARGS 1
01073 #define DICT_NOTINDOLLARS 0
01074 #define DICT_INDOLLARS 1
01075
01076 #define DICT_INTEGERNUMBER 1
01077 #define DICT_RATIONALNUMBER 2
01078 #define DICT_SYMBOL 3
01079 #define DICT_VECTOR 4
```

```

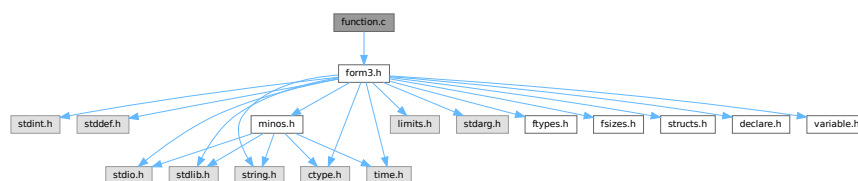
01080 #define DICT_INDEX 5
01081 #define DICT_FUNCTION 6
01082 #define DICT_FUNCTION_WITH_ARGUMENTS 7
01083 #define DICT_SPECIALCHARACTER 8
01084 #define DICT_RANGE 9
01085
01086 #define READSPECTATORFLAG 3
01087 #define GLOBALSPECTATORFLAG 1
01088
01089 #define ORDEREDSET 1
01090
01091 #define DENSETABLE 1
01092 #define SPARSETABLE 0
01093
01094 // Diagram generator flags. They should be powers of two, since they are added
01095 // to pass to diagrams and masked in diawrap.cc to check the flags.
01096 // We use a "stringified" version of these when defining the FORM preprocessor
01097 // variables, so can't neatly use "1<10" etc here.
01098 #define TOPOLOGIESONLY 1
01099 #define WITHOUTNODES 2
01100 #define WITHEDGES 4
01101 #define WITHBLOCKS 8
01102 #define WITHONEPISETS 16
01103 #define WITHSYMMETRIZEI 32
01104 #define WITHSYMMETRIZEF 64
01105 // This is not an "option" preprocessor var but is set by "external" particle definitions
01106 #define CHECKEXTERN 128
01107 // The "ggraf compatible" filtering flags:
01108 #define ONEPARTI 256
01109 #define ONEPARTR 512
01110 #define ONSHELL 1024
01111 #define OFFSHELL 2048
01112 #define NOSIGMA 4096
01113 #define SIGMA 8192
01114 #define NOSNAIL 16384
01115 #define SNAIL 32768
01116 #define NOTADPOLE 65536
01117 #define TADPOLE 131072
01118 #define SIMPLE 262144
01119 #define NOTSIMPLE 524288
01120 #define BIPART 1048576
01121 #define NONBIPART 2097152
01122 #define CYCLI 4194304
01123 #define CYCLR 8388608
01124 #define FLOOP 16777216
01125 #define NOTFLOOP 33554432
01126

```

4.53 function.c File Reference

```
#include "form3.h"
```

Include dependency graph for function.c:



Functions

- int [MakeDirty](#) (WORD *term, WORD *x, WORD par)
- void [MarkDirty](#) (WORD *term, WORD flags)
- void [PolyFunDirty](#) (PHEAD WORD *term)
- void [PolyFunClean](#) (PHEAD WORD *term)
- WORD [Symmetrize](#) (PHEAD WORD *func, WORD *Lijst, WORD ngroups, WORD gsize, WORD type)

- int [CompGroup](#) (PHEAD WORD type, WORD **args, WORD *a1, WORD *a2, WORD num)
- int [FullSymmetrize](#) (PHEAD WORD *fun, int type)
- int [SymGen](#) (PHEAD WORD *term, WORD *params, WORD num, WORD level)
- int [SymFind](#) (PHEAD WORD *term, WORD *params)
- int [ChainIn](#) (PHEAD WORD *term, WORD funnum)
- int [ChainOut](#) (PHEAD WORD *term, WORD funnum)
- int [MatchFunction](#) (PHEAD WORD *pattern, WORD *interm, WORD *wilds)
- int [ScanFunctions](#) (PHEAD WORD *inpat, WORD *inter, WORD par)

4.53.1 Detailed Description

The file with the central routines for the pattern matching of functions and their arguments. The file also contains the routines for the execution of the Symmetrize statement and its variations (like antisymmetrize etc).

Definition in file [function.c](#).

4.53.2 Function Documentation

4.53.2.1 MakeDirty()

```
int MakeDirty (  
    WORD * term,  
    WORD * x,  
    WORD par )
```

Definition at line 51 of file [function.c](#).

4.53.2.2 MarkDirty()

```
void MarkDirty (  
    WORD * term,  
    WORD flags )
```

Definition at line 97 of file [function.c](#).

4.53.2.3 PolyFunDirty()

```
void PolyFunDirty (  
    PHEAD WORD * term )
```

Definition at line 139 of file [function.c](#).

4.53.2.4 PolyFunClean()

```
void PolyFunClean (  
    PHEAD WORD * term )
```

Definition at line 174 of file [function.c](#).

4.53.2.5 Symmetrize()

```
WORD Symmetrize (
    PHEAD WORD * func,
    WORD * Lijst,
    WORD ngroups,
    WORD gsize,
    WORD type )
```

Definition at line [213](#) of file [function.c](#).

4.53.2.6 CompGroup()

```
int CompGroup (
    PHEAD WORD type,
    WORD ** args,
    WORD * a1,
    WORD * a2,
    WORD num )
```

Definition at line [373](#) of file [function.c](#).

4.53.2.7 FullSymmetrize()

```
int FullSymmetrize (
    PHEAD WORD * fun,
    int type )
```

Definition at line [473](#) of file [function.c](#).

4.53.2.8 SymGen()

```
int SymGen (
    PHEAD WORD * term,
    WORD * params,
    WORD num,
    WORD level )
```

Definition at line [518](#) of file [function.c](#).

4.53.2.9 SymFind()

```
int SymFind (
    PHEAD WORD * term,
    WORD * params )
```

Definition at line [633](#) of file [function.c](#).

4.53.2.10 ChainIn()

```
int ChainIn (
    PHEAD WORD * term,
    WORD funnum )
```

Definition at line 691 of file [function.c](#).

4.53.2.11 ChainOut()

```
int ChainOut (
    PHEAD WORD * term,
    WORD funnum )
```

Definition at line 748 of file [function.c](#).

4.53.2.12 MatchFunction()

```
int MatchFunction (
    PHEAD WORD * pattern,
    WORD * interm,
    WORD * wilds )
```

Definition at line 826 of file [function.c](#).

4.53.2.13 ScanFunctions()

```
int ScanFunctions (
    PHEAD WORD * inpat,
    WORD * inter,
    WORD par )
```

Definition at line 1618 of file [function.c](#).

4.54 function.c

[Go to the documentation of this file.](#)

```
00001
00008 /* # [ License : */
00009 /*
00010 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00011 * When using this file you are requested to refer to the publication
00012 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013 * This is considered a matter of courtesy as the development was paid
00014 * for by FOM the Dutch physics granting agency and we would like to
00015 * be able to track its scientific use to convince FOM of its value
00016 * for the community.
00017 *
00018 * This file is part of FORM.
00019 *
00020 * FORM is free software: you can redistribute it and/or modify it under the
00021 * terms of the GNU General Public License as published by the Free Software
00022 * Foundation, either version 3 of the License, or (at your option) any later
00023 * version.
00024 *
00025 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
```

```

00027 *   FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
00028 *   details.
00029 *
00030 *   You should have received a copy of the GNU General Public License along
00031 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00032 */
00033 /* #] License : */
00034 /*
00035     #[ Includes : function.c
00036 */
00037
00038 #include "form3.h"
00039
00040 /*
00041     #] Includes :
00042     #[ Utilities :
00043         #[ MakeDirty :
00044
00045         Routine finds the function with the address x in it
00046         and mark all arguments that contain x as dirty.
00047         if par == 0 term is a full term, else term is the start of a
00048         function
00049 */
00050
00051 int MakeDirty(WORD *term, WORD *x, WORD par)
00052 {
00053     WORD *next, *n;
00054     if ( !par ) {
00055         next = term; next += *term;
00056         next -= ABS(next[-1]);
00057         term++;
00058         if ( x < term ) return(0);
00059         if ( x >= next ) return(0);
00060         while ( term < next ) {
00061             n = term + term[1];
00062             if ( x < n ) break;
00063             term = n;
00064         }
00065         /*     next = n; */
00066     }
00067     else {
00068         next = term + term[1];
00069         if ( x < term || x >= next ) return(0);
00070     }
00071     if ( *term < FUNCTION ) return(0);
00072     if ( functions[*term-FUNCTION].spec >= TENSORFUNCTION ) return(0);
00073     term += FUNHEAD;
00074     if ( x < term ) return(0);
00075     next = term; NEXTARG(next)
00076     while ( x >= next ) { term = next; NEXTARG(next) }
00077     if ( *term < 0 ) return(0);
00078     term[1] = 1;
00079     term += ARGHEAD;
00080     if ( x < term ) return(1);
00081     next = term + *term;
00082     while ( x >= next ) { term = next; next += *next; }
00083     MakeDirty(term,x,0);
00084     return(1);
00085 }
00086
00087 /*
00088     #] MakeDirty :
00089     #[ MarkDirty :
00090
00091     Routine marks all functions dirty with the given flags.
00092     Is to be used when there is a possibility that symmetrization
00093     properties of functions may have changed. In that case we play
00094     it safe.
00095 */
00096
00097 void MarkDirty(WORD *term, WORD flags)
00098 {
00099     WORD *t, *r, *m, *tstop;
00100     GETSTOP(term,tstop);
00101     t = term+1;
00102     while ( t < tstop ) {
00103         if ( *t < FUNCTION ) { t += t[1]; continue; }
00104         t[2] |= flags;
00105         if ( *t < FUNCTION+WILDOFFSET && functions[*t-FUNCTION].spec > 0 ) {
00106             t += t[1]; continue;
00107         }
00108         if ( *t >= FUNCTION+WILDOFFSET && functions[*t-FUNCTION-WILDOFFSET].spec > 0 ) {
00109             t += t[1]; continue;
00110         }
00111         r = t + FUNHEAD;
00112         t += t[1];
00113         while ( r < t ) {

```

```

00114         if ( *r <= 0 ) {
00115             if ( *r <= -FUNCTION ) r++;
00116             else r += 2;
00117             continue;
00118         }
00119         r[1] |= DIRTYFLAG;
00120         m = r + ARGHEAD;
00121         r += *r;
00122         while ( m < r ) {
00123             MarkDirty(m,flags);
00124             m += *m;
00125         }
00126     }
00127 }
00128 }
00129
00130 /*
00131     #] MarkDirty :
00132     #[ PolyFunDirty :
00133
00134     Routine marks the PolyFun or the PolyRatFun dirty.
00135     This is used when there is modular calculus and the modulus
00136     has changed for the current module.
00137 */
00138
00139 void PolyFunDirty(PHEAD WORD *term)
00140 {
00141     GETBIDENTITY
00142     WORD *t, *tstop, *endarg;
00143     tstop = term + *term;
00144     tstop -= ABS(tstop[-1]);
00145     t = term+1;
00146     while ( t < tstop ) {
00147         if ( *t == AR.PolyFun ) {
00148             if ( AR.PolyFunType == 2 ) t[2] |= MUSTCLEANPRF;
00149             endarg = t + t[1];
00150             t[2] |= DIRTYFLAG;
00151             t += FUNHEAD;
00152             while ( t < endarg ) {
00153                 if ( *t > 0 ) {
00154                     t[1] |= DIRTYFLAG;
00155                 }
00156                 NEXTARG(t);
00157             }
00158         }
00159         else {
00160             t += t[1];
00161         }
00162     }
00163 }
00164
00165 /*
00166     #] PolyFunDirty :
00167     #[ PolyFunClean :
00168
00169     Routine marks the PolyFun or the PolyRatFun clean.
00170     This is used when there is modular calculus and the modulus
00171     has changed for the current module.
00172 */
00173
00174 void PolyFunClean(PHEAD WORD *term)
00175 {
00176     GETBIDENTITY
00177     WORD *t, *tstop;
00178     tstop = term + *term;
00179     tstop -= ABS(tstop[-1]);
00180     t = term+1;
00181     while ( t < tstop ) {
00182         if ( *t == AR.PolyFun ) {
00183             t[2] &= ~MUSTCLEANPRF;
00184         }
00185         t += t[1];
00186     }
00187 }
00188
00189 /*
00190     #] PolyFunClean :
00191     #[ Symmetrize :
00192
00193     (Anti)Symmetrizes the arguments of a function.
00194     Nlist tells of how many arguments are involved.
00195     Nlist == 0      All arguments must be sorted.
00196     Nlist > 0      Arguments mentioned are to be sorted, rest skipped.
00197     type = SYMMETRIC      Full symmetrization
00198     type = ANTISYMMETRIC: Full symmetrization
00199     type = CYCLESYMMETRIC: Cyclic
00200     type = RCYCLESYMMETRIC: Cyclic or reverse

```

```

00201         Return value: OR of:
00202             0 even, 1 odd
00203             2 equal groups
00204             4 there was a permutation.
00205
00206         The information in Lijst tells what grouping is to be applied.
00207         The information is:
00208         ngroups number of groups
00209         gsize size of groups
00210         Lijst[0].... The groups.
00211     */
00212
00213 WORD Symmetrize(PHEAD WORD *func, WORD *Lijst, WORD ngroups, WORD gsize,
00214               WORD type)
00215 {
00216     GETBIDENTITY
00217     WORD **args,**arg,nargs;
00218     WORD *to, *r, *fstop;
00219     WORD i, j, k, ff, exch, nexch, neq;
00220     WORD *a1, *a2, *a3;
00221     WORD reverseorder;
00222     if ( ( type & REVERSEORDER ) != 0 ) reverseorder = -1;
00223     else reverseorder = 1;
00224     type &= ~REVERSEORDER;
00225
00226     ff = ( *func > FUNCTION ) ? functions[*func-FUNCTION].spec: 0;
00227
00228     if ( 2*func[1] > AN.arglistsize ) {
00229         if ( AN.arglist ) M_free(AN.arglist,"Symmetrize");
00230         AN.arglistsize = 2*func[1] + 8;
00231         AN.arglist = (WORD **)Malloc1(AN.arglistsize*sizeof(WORD *),"Symmetrize");
00232     }
00233     arg = args = AN.arglist;
00234     to = AT.WorkPointer;
00235     r = func;
00236     fstop = r + r[1];
00237     r += FUNHEAD;
00238     nargs = 0;
00239     while ( r < fstop ) { /* Make list of arguments */
00240         *arg++ = r;
00241         nargs++;
00242         if ( ff ) {
00243             if ( *r == FUNNYWILD ) r++;
00244             r++;
00245         }
00246         else { NEXTARG(r); }
00247     }
00248     exch = 0;
00249     nexch = 0;
00250     neq = 0;
00251     a1 = Lijst;
00252     if ( type == SYMMETRIC || type == ANTISYMMETRIC ) {
00253         for ( i = 1; i < ngroups; i++ ) {
00254             a3 = a2 = a1 + gsize;
00255             k = reverseorder*CompGroup(BHEAD ff,args,a1,a2,gsiz);
00256             if ( k < 0 ) {
00257                 j = i-1;
00258                 for(;;) {
00259                     for ( k = 0; k < gsize; k++ ) {
00260                         r = args[a1[k]]; args[a1[k]] = args[a2[k]]; args[a2[k]] = r;
00261                     }
00262                     exch ^= 1;
00263                     nexch = 4;
00264                     if ( j <= 0 ) break;
00265                     a1 -= gsize;
00266                     a2 -= gsize;
00267                     k = reverseorder*CompGroup(BHEAD ff,args,a1,a2,gsiz);
00268                     if ( k == 0 ) neq = 2;
00269                     if ( k >= 0 ) break;
00270                     j--;
00271                 }
00272             }
00273             else if ( k == 0 ) neq = 2;
00274             a1 = a3;
00275         }
00276     }
00277     else if ( type == CYCLESYMMETRIC || type == RCYCLESYMMETRIC ) {
00278         WORD rev = 0, jmin = 0, ii, iimin;
00279     recycle:
00280         for ( j = 1; j < ngroups; j++ ) {
00281             for ( i = 0; i < ngroups; i++ ) {
00282                 iimin = jmin + i;
00283                 if ( iimin >= ngroups ) iimin -= ngroups;
00284                 ii = j + i;
00285                 if ( ii >= ngroups ) ii -= ngroups;
00286                 k = reverseorder*CompGroup(BHEAD ff,args,Lijst+gsiz*iimin,Lijst+gsiz*ii,gsiz);
00287                 if ( k > 0 ) break;

```

```

00288         if ( k < 0 ) { jmin = j; nexch = 4; break; }
00289     }
00290 }
00291 if ( type == RCYCLESYMMETRIC && rev == 0 && ngroups > 1 ) {
00292     for ( j = 0; j < ngroups; j++ ) {
00293         for ( i = 0; i < ngroups; i++ ) {
00294             iimin = jmin + i;
00295             if ( iimin >= ngroups ) iimin -= ngroups;
00296             ii = j - i;
00297             if ( ii < 0 ) ii += ngroups;
00298             k = reverseorder*CompGroup(BHEAD ff,args,Lijst+gsize*iimin,Lijst+gsize*ii,gsize);
00299             if ( k > 0 ) break;
00300             if ( k < 0 ) {
00301                 nexch = 4;
00302                 jmin = 0;
00303                 a1 = Lijst;
00304                 a2 = Lijst + gsize * (ngroups-1);
00305                 while ( a2 > a1 ) {
00306                     for ( k = 0; k < gsize; k++ ) {
00307                         r = args[a1[k]];
00308                         args[a1[k]] = args[a2[k]];
00309                         args[a2[k]] = r;
00310                     }
00311                     a1 += gsize; a2 -= gsize;
00312                 }
00313                 rev = 1;
00314                 goto recycle;
00315             }
00316         }
00317     }
00318 }
00319 if ( jmin != 0 ) {
00320     arg = AN.arglist + func[1];
00321     a1 = Lijst + gsize * jmin;
00322     k = gsize * ngroups;
00323     a2 = Lijst + k;
00324     for ( i = 0; i < k; i++ ) {
00325         if ( a1 >= a2 ) a1 = Lijst;
00326         *arg++ = args[*a1++];
00327     }
00328     arg = AN.arglist + func[1];
00329     a1 = Lijst;
00330     for ( i = 0; i < k; i++ ) args[*a1++] = *arg++;
00331 }
00332 }
00333 r = func;
00334 i = FUNHEAD;
00335 NCOPY(to,r,i);
00336 for ( i = 0; i < nargs; i++ ) {
00337     if ( ff ) {
00338         if ( *(args[i]) == FUNNYWILD ) {
00339             *to++ = *(args[i]);
00340             *to++ = args[i][1];
00341         }
00342         else *to++ = *(args[i]);
00343     }
00344     else if ( ( j = *args[i] ) < 0 ) {
00345         *to++ = j;
00346         if ( j > -FUNCTION ) *to++ = args[i][1];
00347     }
00348     else {
00349         r = args[i];
00350         NCOPY(to,r,j);
00351     }
00352 }
00353 i = func[1];
00354 to = func;
00355 r = AT.WorkPointer;
00356 NCOPY(to,r,i);
00357 return ( exch | nexch | neq );
00358 }
00359
00360 /*
00361     #] Symmetrize :
00362     #[ CompGroup :
00363
00364     Routine compares two groups of arguments
00365     The arguments are in args[a1[i]] and args[a2[i]]
00366     for i = 0 to num
00367     type indicates the type of function.
00368     return value: -1 if there should be an exchange
00369     0 if they are equal
00370     1 if they are OK.
00371 */
00372
00373 int CompGroup(PHEAD WORD type, WORD **args, WORD *a1, WORD *a2, WORD num)
00374 {

```

```

00375     GETBIDENTITY
00376     WORD *t1, *t2, i1, i2, n, k;
00377
00378     for ( n = 0; n < num; n++ ) {
00379         t1 = args[a1[n]]; t2 = args[a2[n]];
00380         if ( type >= TENSORFUNCTION ) {
00381             if ( AR.Eside == LHSIDE || AR.Eside == LHSIDEX ) {
00382                 if ( *t1 == FUNNYWILD ) {
00383                     if ( *t2 == FUNNYWILD ) {
00384                         if ( t1[1] < t2[1] ) return(1);
00385                         if ( t1[1] > t2[1] ) return(-1);
00386                     }
00387                     return(-1);
00388                 }
00389                 else if ( *t2 == FUNNYWILD ) {
00390                     return(1);
00391                 }
00392                 else {
00393                     if ( *t1 < *t2 ) return(1);
00394                     if ( *t1 > *t2 ) return(-1);
00395                 }
00396             }
00397             else {
00398                 if ( *t1 < *t2 ) return(1);
00399                 if ( *t1 > *t2 ) return(-1);
00400             }
00401         }
00402         else if ( type == 0 ) {
00403             if ( AC.properorderflag ) {
00404                 k = CompArg(t1,t2);
00405                 if ( k < 0 ) return(1);
00406                 if ( k > 0 ) return(-1);
00407                 NEXTARG(t1)
00408                 NEXTARG(t2)
00409             }
00410             else {
00411                 if ( *t1 > 0 ) {
00412                     i1 = *t1 - ARGHEAD - 1;
00413                     t1 += ARGHEAD + 1;
00414                     if ( *t2 > 0 ) {
00415                         i2 = *t2 - ARGHEAD - 1;
00416                         t2 += ARGHEAD + 1;
00417                         while ( i1 > 0 && i2 > 0 ) {
00418                             if ( *t1 > *t2 ) return(-1);
00419                             else if ( *t1 < *t2 ) return(1);
00420                             i1--; i2--; t1++; t2++;
00421                         }
00422                         if ( i1 > 0 ) return(-1);
00423                         else if ( i2 > 0 ) return(1);
00424                     }
00425                 }
00426                 /*
00427                 This seems to be a bug. Reported by Aneesh Monahar, 28-sep-2005
00428                 else return(1);
00429                 */
00430                 else return(-1);
00431             }
00432             else if ( *t2 > 0 ) return(1);
00433             else {
00434                 if ( *t1 != *t2 ) {
00435                     if ( *t1 <= -FUNCTION && *t2 <= -FUNCTION ) {
00436                         if ( *t1 < *t2 ) return(-1);
00437                         return(1);
00438                     }
00439                     else {
00440                         if ( *t1 < *t2 ) return(1);
00441                         return(-1);
00442                     }
00443                 }
00444                 if ( *t1 > -FUNCTION ) {
00445                     if ( t1[1] != t2[1] ) {
00446                         if ( t1[1] < t2[1] ) return(1);
00447                         return(-1);
00448                     }
00449                 }
00450             }
00451         }
00452     }
00453     return(0);
00454 }
00455
00456 /*
00457     #] CompGroup :
00458     #[ FullSymmetrize :
00459
00460     Relay function for Normalize to execute a full symmetrization
00461     of a function fun. It hooks into Symmetrize according to the

```

```

00462         calling conventions for it.
00463         type = 0: Symmetrize
00464         type = 1: AntiSymmetrize
00465         type = 2: CycleSymmetrize
00466         type = 3: RCycleSymmetrize
00467         Return values:
00468         bit 0: odd permutation
00469         bit 1: identical arguments
00470         bit 2: there was a permutation.
00471 */
00472
00473 int FullSymmetrize(PHEAD WORD *fun, int type)
00474 {
00475     GETBIDENTITY
00476     WORD *Lijst, count = 0;
00477     WORD *t, *funstop, i;
00478     int retval;
00479
00480     if ( functions[*fun-FUNCTION].spec > 0 ) {
00481         count = fun[1] - FUNHEAD;
00482         for ( i = fun[1]-1; i >= FUNHEAD; i-- ) {
00483             if ( fun[i] == FUNNYWILD ) count--;
00484         }
00485     }
00486     else {
00487         funstop = fun + fun[1];
00488         t = fun + FUNHEAD;
00489         while ( t < funstop ) { count++; NEXTARG(t) }
00490     }
00491     if ( count < 2 ) {
00492         fun[2] &= ~DIRTYSYMFLAG;
00493         return(0);
00494     }
00495     Lijst = AT.WorkPointer;
00496     for ( i = 0; i < count; i++ ) Lijst[i] = i;
00497     AT.WorkPointer += count;
00498     retval = Symmetrize(BHEAD fun,Lijst,count,1,type);
00499     fun[2] &= ~DIRTYSYMFLAG;
00500     AT.WorkPointer = Lijst;
00501     return(retval);
00502 }
00503
00504 /*
00505     #] FullSymmetrize :
00506     #[ SymGen :
00507
00508     Routine does the outer work in the symmetrization.
00509     It locates the function(s) and loads up the parameters.
00510     It also studies the result.
00511
00512     if params[4] = -1 and no extra -> all
00513         extra -> strip groups with elements too large
00514         0 -> if group with element too large: nofun
00515         >0 -> must have right number of arguments
00516 */
00517
00518 int SymGen(PHEAD WORD *term, WORD *params, WORD num, WORD level)
00519 {
00520     GETBIDENTITY
00521     WORD *t, *r, *m;
00522     WORD i, j, k, c1, c2, ngroup;
00523     WORD *rstop, Nlist, *inLijst, *Lijst, sign = 1, sumch = 0, count;
00524     DUMMYUSE(num);
00525     c1 = params[3]; /* function number */
00526     c2 = FUNCTION + WILDOFFSET;
00527     Nlist = params[4];
00528     if ( Nlist < 0 ) Nlist = 0;
00529     else Nlist = params[0] - 7;
00530     t = term;
00531     m = t + *t;
00532     m -= ABS(m[-1]);
00533     t++;
00534     while ( t < m ) {
00535         if ( *t == c1 || c1 > c2 ) { /* Candidate function */
00536             if ( *t >= FUNCTION && functions[*t-FUNCTION].spec
00537                 >= TENSORFUNCTION ) {
00538                 count = t[1] - FUNHEAD;
00539             }
00540             else {
00541                 count = 0;
00542                 r = t;
00543                 rstop = t + t[1];
00544                 r += FUNHEAD;
00545                 while ( r < rstop ) { count++; NEXTARG(r) }
00546             }
00547             if ( ( j = params[4] ) > 0 && j != count ) goto NextFun;
00548             if ( j == 0 ) {

```

```

00549         inLijst = params+7;
00550         for ( i = 0; i < Nlist; i++ )
00551             if ( inLijst[i] > count-1 ) goto NextFun;
00552     }
00553
00554     if ( Nlist > (params[0] - 7) ) Nlist = params[0] - 7;
00555     Lijst = AT.WorkPointer;
00556     inLijst = params + 7;
00557     ngroup = params[5];
00558     if ( Nlist > 0 && j < 0 ) {
00559         k = 0;
00560         for ( i = 0; i < ngroup; i++ ) {
00561             for ( j = 0; j < params[6]; j++ ) {
00562                 if ( inLijst[j] > count+1 ) {
00563                     inLijst += params[6];
00564                     goto NextGroup;
00565                 }
00566             }
00567             j = params[6];
00568             NCOPY(Lijst,inLijst,j);
00569             k++;
00570 NextGroup:;
00571         }
00572         if ( k <= 1 ) goto NextFun;
00573         ngroup = k;
00574         inLijst = AT.WorkPointer;
00575         AT.WorkPointer = Lijst;
00576         Lijst = inLijst;
00577     }
00578     else if ( Nlist == 0 ) {
00579         for ( i = 0; i < count; i++ ) Lijst[i] = i;
00580         AT.WorkPointer += count;
00581         ngroup = count;
00582     }
00583     else {
00584         for ( i = 0; i < Nlist; i++ ) Lijst[i] = inLijst[i];
00585         AT.WorkPointer += Nlist;
00586     }
00587     j = Symmetrize(BHEAD t,Lijst,ngroup,params[6],params[2]);
00588     AT.WorkPointer = Lijst;
00589     if ( params[2] == 4 ) { /* antisymmetric */
00590         if ( ( j & 1 ) != 0 ) sign = -sign;
00591         if ( ( j & 2 ) != 0 ) return(0); /* equal arguments */
00592     }
00593     if ( ( j & 4 ) != 0 ) sumch++;
00594     t[2] &= ~DIRTYSYMLAG;
00595 }
00596 NextFun:
00597     t += t[1];
00598 }
00599 if ( sign < 0 ) {
00600     t = term;
00601     t += *t - 1;
00602     *t = -*t;
00603 }
00604 if ( sumch ) {
00605     if ( Normalize(BHEAD term) ) {
00606         MLOCK(ErrorMessageLock);
00607         MesCall("SymGen");
00608         MUNLOCK(ErrorMessageLock);
00609         return(-1);
00610     }
00611     if ( !*term ) return(0);
00612     *AN.RepPoint = 1;
00613     AR.expchanged = 1;
00614     if ( AR.CurDum > AM.IndDum && AR.sLevel <= 0 ) ReNumber(BHEAD term);
00615 }
00616 return(Generator(BHEAD term,level));
00617 }
00618
00619 /*
00620     #] SymGen :
00621     #[ SymFind :
00622
00623     There is a certain amount of double work here, as this routine
00624     finds the function to be treated, while the SymGen routine has
00625     to find it again. Note however that this way things remain
00626     uniform and simple. Moreover this avoids problems with actions
00627     on more than one function simultaneously.
00628     Output in AT.TMout:
00629     Number,sym/anti,fun,lenpar,ngroups,gsize,fields
00630
00631 */
00632
00633 int SymFind(PHEAD WORD *term, WORD *params)
00634 {
00635     GETBIDENTITY

```



```

00636     WORD *t, *r, *m;
00637     WORD j, c1, c2, count;
00638     WORD *rstop;
00639     c1 = params[4];      /* function number */
00640     c2 = FUNCTION + WILDOFFSET;
00641     t = term;
00642     m = t + *t;
00643     m -= ABS(m[-1]);
00644     t++;
00645     while ( t < m ) {
00646         if ( *t == c1 || c1 > c2 ) {      /* Candidate function */
00647             if ( *t >= FUNCTION && functions[*t-FUNCTION].spec
00648                 >= TENSORFUNCTION ) { count = t[1] - FUNHEAD; }
00649             else {
00650                 count = 0;
00651                 r = t;
00652                 rstop = t + t[1];
00653                 r += FUNHEAD;
00654                 while ( r < rstop ) { count++; NEXTARG(r) }
00655             }
00656             if ( ( j = params[5] ) > 0 && j != count ) goto NextFun;
00657             if ( j == 0 ) {
00658                 r = params + 8;
00659                 rstop = params + params[1];
00660                 while ( r < rstop ) {
00661                     if ( *r > count + 1 ) goto NextFun;
00662                     r++;
00663                 }
00664             }
00665
00666             t = AT.TMout;
00667             r = params;
00668             j = r[1] - 1;
00669             *t++ = j;
00670             *t++ = SYMMETRIZE;
00671             r += 3;
00672             j--;
00673             NCOPY(t,r,j);
00674             return(1);
00675         }
00676     NextFun:
00677         t += t[1];
00678     }
00679     return(0);
00680 }
00681
00682 /*
00683     #] SymFind :
00684     #[ ChainIn :
00685
00686     Equivalent to repeat id f(?a)*f(?b) = f(?a,?b);
00687
00688     This one always takes less space.
00689 */
00690
00691 int ChainIn(PHEAD WORD *term, WORD funnum)
00692 {
00693     GETBIDENTITY
00694     WORD *t, *tend, *m, *tt, *ts;
00695     int action, normFlag = 0;
00696     if ( funnum < 0 ) { /* Dollar to be expanded */
00697         funnum = DolToFunction(BHEAD -funnum);
00698         if ( AN.ErrorInDollar || funnum <= 0 ) {
00699             MLOCK(ErrorMessageLock);
00700             MesPrint("Dollar variable does not evaluate to function in ChainIn statement");
00701             MUNLOCK(ErrorMessageLock);
00702             return(-1);
00703         }
00704     }
00705     do {
00706         action = 0;
00707         tend = term+*term;
00708         tend -= ABS(tend[-1]);
00709         t = term+1;
00710         while ( t < tend ) {
00711             if ( *t != funnum ) { t += t[1]; continue; }
00712             m = t;
00713             t += t[1];
00714             tt = t;
00715             if ( t >= tend || *t != funnum ) continue;
00716             action = 1;
00717             normFlag = 1;
00718             while ( t < tend && *t == funnum ) {
00719                 ts = t + t[1];
00720                 t += FUNHEAD;
00721                 while ( t < ts ) *tt++ = *t++;
00722             }

```

```

00723         m[1] = tt - m;
00724         ts = term + *term;
00725         while ( t < ts ) *tt++ = *t++;
00726         *term = tt - term;
00727         break;
00728     }
00729 } while ( action );
00730
00731 if ( normFlag ) {
00732     /* We need to check the newly-constructed arguments w.r.t symmetry properties */
00733     MarkDirty(term, DIRTYSYMFLAG);
00734     AT.WorkPointer = term + *term;
00735     Normalize(BHEAD term);
00736 }
00737
00738 return(0);
00739 }
00740
00741 /*
00742     #] ChainIn :
00743     #[ ChainOut :
00744
00745     Equivalent to repeat id f(x1?,x2?,?a) = f(x1)*f(x2,?a);
00746 */
00747
00748 int ChainOut(PHEAD WORD *term, WORD funnum)
00749 {
00750     GETBIDENTITY
00751     WORD *t, *tend, *tt, *ts, *w, *ws;
00752     int flag = 0, i;
00753     if ( funnum < 0 ) { /* Dollar to be expanded */
00754         funnum = DolToFunction(BHEAD -funnum);
00755         if ( AN.ErrorInDollar || funnum <= 0 ) {
00756             MLOCK(ErrorMessageLock);
00757             MesPrint("Dollar variable does not evaluate to function in ChainOut statement");
00758             MUNLOCK(ErrorMessageLock);
00759             return(-1);
00760         }
00761     }
00762     tend = term+*term;
00763     if ( AT.WorkPointer < tend ) AT.WorkPointer = tend;
00764     tend -= ABS(tend[-1]);
00765     t = term+1; tt = term; w = AT.WorkPointer;
00766     while ( t < tend ) {
00767         if ( *t != funnum || t[1] == FUNHEAD ) { t += t[1]; continue; }
00768         flag = 1;
00769         while ( tt < t ) *w++ = *tt++;
00770         ts = t + t[1];
00771         t += FUNHEAD;
00772         while ( t < ts ) {
00773             ws = w;
00774             for ( i = 0; i < FUNHEAD; i++ ) *w++ = tt[i];
00775             if ( functions[*tt-FUNCTION].spec >= TENSORFUNCTION ) {
00776                 *w++ = *tt++;
00777             }
00778             else if ( *t < 0 ) {
00779                 if ( *t <= -FUNCTION ) *w++ = *t++;
00780                 else { *w++ = *t++; *w++ = *t++; }
00781             }
00782             else {
00783                 i = *t; NCOPY(w,t,i);
00784             }
00785             ws[1] = w - ws;
00786         }
00787         tt = t;
00788     }
00789     if ( flag == 1 ) {
00790         ts = term + *term;
00791         while ( tt < ts ) *w++ = *tt++;
00792         *AT.WorkPointer = w - AT.WorkPointer;
00793         t = term; w = AT.WorkPointer; i = *w;
00794         NCOPY(t,w,i)
00795         AT.WorkPointer = term + *term;
00796         Normalize(BHEAD term);
00797     }
00798     return(0);
00799 }
00800
00801 /*
00802     #] ChainOut :
00803     #] Utilities :
00804     #[ Patterns :
00805         #[ MatchFunction :          WORD MatchFunction(pattern,interm,wilds)
00806
00807         The routine assumes that the function numbers are the same.
00808         The contents are compared and a possible wildcard assignment
00809         is made. Note that it may be necessary to use a wildcard

```

```

00810         assignment stack to do things right.
00811         The routine can become arbitrarily complicated as there is
00812         no end to the possible wildcarding.
00813         Examples:
00814         - a: No wildcarding -> straight match
00815         - b: Individual arguments (object -> object)
00816         - c: whole arguments (object to subexpression)
00817         - d: any argumentlist
00818         - e: part of an argument (object inside subexpression)
00819
00820         The ones with a minus sign in front have been implemented.
00821
00822         Note: the argument wilds allows backtracking when multiple
00823         ?a,?b give a match that later turns out to be useless.
00824 */
00825
00826 int MatchFunction(PHEAD WORD *pattern, WORD *interm, WORD *wilds)
00827 {
00828     GETBIDENTITY
00829     WORD *m, *t, *r, i;
00830     WORD *mstop = 0, *tstop = 0;
00831     WORD *argmstop, *argtstop;
00832     WORD *mtrmstop, *ttrmstop;
00833     WORD *msubstop, *mnextsub;
00834     WORD msizcoef, mcount, tcount, newvalue, j;
00835     WORD *oldm, *oldt;
00836     WORD *OldWork, numofwildarg;
00837     WORD nwstore, tobeeaten, reservevalue = 0, resernum = 0, withwild;
00838     WORD *wildargtaken;
00839     CBUF *C = cbuf+AT.ebufnum;
00840     int ntw = AN.NumTotWildArgs;
00841     LONG oldcpointer = C->Pointer - C->Buffer;
00842 #ifdef WITHFLOAT
00843     // Pattern matching against float_ functions is currently disabled.
00844     // To relax this in the future, move this early return down to where gamma
00845     // functions and tensors are handled specially.
00846     if ( *interm == FLOATFUN ) return(0);
00847 #endif
00848 /*
00849     Test first for a straight match
00850 */
00851     AN.RepFunList[AN.RepFunNum+1] = 0;
00852     if ( *wilds == 0 ) {
00853         m = pattern; t = interm;
00854
00855         if ( *m != *t ) {
00856             if ( *m < (FUNCTION + WILDOFFSET) ) return(0);
00857             if ( *t < FUNCTION ) return(0);
00858             if ( functions[*t-FUNCTION].spec !=
00859                 functions[*m-FUNCTION-WILDOFFSET].spec ) return(0);
00860         }
00861         i = m[1];
00862         if ( *m >= (FUNCTION + WILDOFFSET) ) { i--; m++; t++; }
00863         do { if ( *m++ != *t++ ) break; } while ( --i > 0 );
00864         if ( i <= 0 ) { /* Arguments match */
00865             if ( AN.SignCheck && AN.ExpectedSign ) return(0);
00866             i = *pattern - WILDOFFSET;
00867             if ( i >= FUNCTION ) {
00868                 if ( *interm != GAMMA
00869                     && !CheckWild(BHEAD i,FUNTOFUN,*interm,&newvalue) ) {
00870                     AddWild(BHEAD i,FUNTOFUN,newvalue);
00871                     return(1);
00872                 }
00873                 return(0);
00874             }
00875             else return(1);
00876         }
00877     }
00878 /*
00879     Store the current Wildcard assignments
00880 */
00881     t = wildargtaken = OldWork = AT.WorkPointer;
00882     t += ntw;
00883     m = AN.WildValue;
00884     nwstore = i = (m[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
00885     if ( i > 0 ) {
00886         r = AT.WildMask;
00887         do {
00888             *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
00889         } while ( --i > 0 );
00890         *t++ = C->numrhs;
00891     }
00892     if ( t >= AT.WorkTop ) {
00893         MLOCK(ErrorMessageLock);
00894         MesWork();
00895         MUNLOCK(ErrorMessageLock);
00896         Terminate(-1);

```

```

00897     }
00898     AT.WorkPointer = t;
00899
00900     if ( *wilds ) {
00901         if ( *wilds == 1 ) goto endloop;
00902         else goto enloop;          /* tensors = 2 */
00903     }
00904     m = pattern; t = interm;
00905 /*
00906     Single out the specials
00907 */
00908     if ( *t == GAMMA ) {
00909 /*
00910         #[ GAMMA :
00911
00912         For the gamma's we need to do two things:
00913         a: Find that there is a match
00914         b: Find where the match occurs in the string
00915         This last thing cannot be stored in the current conventions,
00916         but once the wildcard assignments have been made it is much
00917         easier to find it back.
00918         Alternative: replace the function number in the term temporarily
00919         by the offset inside the string. This makes things maybe easier.
00920 */
00921         if ( *m != GAMMA ) goto NoCaseB;
00922         i = t[1] - m[1];
00923         if ( m[1] == FUNHEAD+1 ) {
00924             if ( i ) goto NoCaseB;
00925             if ( m[FUNHEAD] < (AM.OffsetIndex+WILDOFFSET) ||
00926                 t[FUNHEAD] >= (AM.OffsetIndex+WILDOFFSET) ) goto NoCaseB;
00927
00928             if ( CheckWild(BHEAD m[FUNHEAD]-WILDOFFSET,INDTOIND,t[FUNHEAD],&newvalue) ) goto NoCaseB;
00929             AddWild(BHEAD m[FUNHEAD]-WILDOFFSET,INDTOIND,newvalue);
00930
00931             AT.WorkPointer = OldWork;
00932             if ( AN.SignCheck && AN.ExpectedSign ) return(0);
00933             return(1);          /* m was eaten. we have a match! */
00934         }
00935         if ( i < 0 ) goto NoCaseB; /* Pattern longer than target */
00936         mstop = m + m[1];
00937         tstop = t + t[1];
00938         m += FUNHEAD; t += FUNHEAD;
00939         if ( *m >= (AM.OffsetIndex+WILDOFFSET) && *t < (AM.OffsetIndex+WILDOFFSET) ) {
00940             if ( CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,*t,&newvalue) ) goto NoCaseB;
00941             reservevalue = newvalue;
00942             withwild = 1;
00943             resernum = *m-WILDOFFSET;
00944             AddWild(BHEAD *m-WILDOFFSET,INDTOIND,newvalue);
00945         }
00946         else if ( *m != *t ) goto NoCaseB;
00947         else withwild = 0;
00948         m++; t++;
00949         oldm = m; argtstop = oldt = t;
00950         j = 0;          /* No wildcard assignments yet */
00951         while ( i >= 0 ) {
00952             if ( *m == *t ) {
00953 WithGamma:
00954                 m++; t++;
00955                 if ( m >= mstop ) {
00956                     if ( t < tstop && mstop < AN.patstop ) {
00957                         WORD k;
00958                         mnxtsub = pattern + pattern[1];
00959                         k = *mnxtsub;
00960                         while ( k == GAMMA && mnxtsub[FUNHEAD]
00961                             != pattern[FUNHEAD] ) {
00962                             mnxtsub += mnxtsub[1];
00963                             if ( mnxtsub >= AN.patstop ) goto FullOK;
00964                             k = *mnxtsub;
00965                         }
00966                         if ( k >= FUNCTION ) {
00967                             if ( k > (FUNCTION + WILDOFFSET) ) k -= WILDOFFSET;
00968                             if ( functions[k-FUNCTION].commute ) goto NoGamma;
00969                         }
00970                     }
00971 FullOK:
00972                     if ( AN.SignCheck && AN.ExpectedSign ) goto NoGamma;
00973                     AN.RepFunList[AN.RepFunNum+1] = WORDDIF(oldt, argtstop);
00974                     return(1);
00975                 }
00976                 if ( t >= tstop ) goto NoCaseB;
00977             }
00978             else if ( *m >= (AM.OffsetIndex+WILDOFFSET)
00979                 && *m < (AM.OffsetIndex + (WILDOFFSET<1)) && ( *t >= 0 ||
00980                 *t < MINSPEC ) ) {          /* Wildcard index */
00981                 if ( !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,*t,&newvalue) ) {
00982                     AddWild(BHEAD *m-WILDOFFSET,INDTOIND,newvalue);
00983                     j = 1;
00984                     goto WithGamma;
00985                 }
00986             }
00987         }

```

```

00984         else goto NoGamma;
00985     }
00986     else if ( *m < MINSPEC && *m >= (AM.OffsetVector+WILDOFFSET)
00987     && *t < MINSPEC ) {          /* Wildcard vector */
00988         if ( !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*t,&newvalue) ) {
00989             AddWild(BHEAD *m-WILDOFFSET,VECTOVEC,newvalue);
00990             j = 1;
00991             goto WithGamma;
00992         }
00993         else goto NoGamma;
00994     }
00995     else {
00996 NoGamma:
00997         if ( j ) {          /* Undo wildcards */
00998             m = AN.WildValue;
00999             t = OldWork + AN.NumTotWildArgs; r = AT.WildMask; j = nwstore;
01000             if ( j > 0 ) {
01001                 do {
01002                     *m++ = *t++; *m++ = *t++;
01003                     *m++ = *t++; *m++ = *t++; *r++ = *t++;
01004                 } while ( --j > 0 );
01005                 C->numrhs = *t++;
01006                 C->Pointer = C->Buffer + oldcpointer;
01007             }
01008             j = 0;
01009         }
01010         m = oldm; t = ++oldt; i--;
01011         if ( withwild ) {
01012             AddWild(BHEAD resernum,INDTOIND,reservevalue);
01013         }
01014     }
01015 }
01016 goto NoCaseB;
01017 /*
01018 #] GAMMA :
01019 #[ Tensors :
01020 */
01021 }
01022 else if ( *t >= FUNCTION && functions[*t-FUNCTION].spec >= TENSORFUNCTION ) {
01023     mstop = m + m[1];
01024     tstop = t + t[1];
01025     mcount = 0;
01026     m += FUNHEAD;
01027     t += FUNHEAD;
01028     AN.WildArgs = 0;
01029     tcount = WORDDIF(tstop,t);
01030     while ( m < mstop ) {
01031         if ( *m == FUNNYWILD ) { m++; AN.WildArgs++; }
01032         m++; mcount++;
01033     }
01034     tobeeaten = tcount - mcount + AN.WildArgs;
01035     if ( tobeeaten ) {
01036         if ( tobeeaten < 0 || AN.WildArgs == 0 ) {
01037             AT.WorkPointer = OldWork;
01038             return(0); /* Cannot match */
01039         }
01040     }
01041     AT.WildArgTaken[0] = AN.WildEat = tobeeaten;
01042     for ( i = 1; i < AN.WildArgs; i++ ) AT.WildArgTaken[i] = 0;
01043 toploop:
01044     numofwildarg = 0;
01045
01046     m = pattern; t = interm;
01047     mstop = m + m[1];
01048     if ( *m != *t ) {
01049         i = *m - WILDOFFSET;
01050         if ( CheckWild(BHEAD i,FUNTOFUN,*t,&newvalue) ) goto NoCaseB;
01051         AddWild(BHEAD i,FUNTOFUN,newvalue);
01052     }
01053     m += FUNHEAD;
01054     t += FUNHEAD;
01055     while ( m < mstop ) {
01056 /*
01057         First test for an exact match
01058 */
01059         if ( *m == *t ) { m++; t++; continue; }
01060 /*
01061         No exact match. Try ARGWILD
01062 */
01063         AN.argaddress = t;
01064         if ( *m == FUNNYWILD ) {
01065             tobeeaten = AT.WildArgTaken[numofwildarg++];
01066             if ( CheckWild(BHEAD m[1],ARGTOARG|EATTENSOR,tobeeaten,t) ) goto endloop;
01067             AddWild(BHEAD m[1],ARGTOARG|EATTENSOR,tobeeaten);
01068             m += 2;
01069             t += tobeeaten;
01070             continue;

```

```

01071     }
01072     /*
01073     Now the various cases:
01074     */
01075     i = *m;
01076     if ( i < MINSPEC ) {
01077         if ( *t != i ) {
01078             if ( *t >= MINSPEC ) goto endloop;
01079             i -= WILDOFFSET;
01080             if ( i < AM.OffsetVector ) goto endloop;
01081             if ( CheckWild(BHEAD i, VECTOVEC, *t, &newvalue) )
01082                 goto endloop;
01083             AddWild(BHEAD i, VECTOVEC, newvalue);
01084         }
01085     }
01086     else if ( i >= AM.OffsetIndex ) { /* Index */
01087         if ( i < ( AM.OffsetIndex + WILDOFFSET ) ) goto endloop;
01088         if ( i >= ( AM.OffsetIndex + (WILDOFFSET*1) ) ) {
01089             /* Summed over index */
01090             goto endloop; /* For the moment */
01091         }
01092         i -= WILDOFFSET;
01093         if ( CheckWild(BHEAD i, INDTOIND, *t, &newvalue) )
01094             goto endloop; /* Assignment not allowed */
01095         AddWild(BHEAD i, INDTOIND, newvalue);
01096     }
01097     else goto endloop;
01098     m++; t++;
01099 }
01100 if ( AN.SignCheck && AN.ExpectedSign ) goto endloop;
01101 AT.WorkPointer = OldWork;
01102 if ( AN.WildArgs > 1 ) *wilds = 2;
01103 return(1); /* m was eaten. we have a match! */
01104
01105 endloop;;
01106 /*
01107 restore the current Wildcard assignments
01108 */
01109 i = nwstore;
01110 if ( i > 0 ) {
01111     m = AN.WildValue;
01112     t = OldWork + ntw; r = AT.WildMask;
01113     do {
01114         *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01115     } while ( --i > 0 );
01116     C->numrhs = *t++;
01117     C->Pointer = C->Buffer + oldcpointer;
01118 }
01119 enloop;;
01120 i = AN.WildArgs - 1;
01121 if ( i <= 0 ) {
01122     AT.WorkPointer = OldWork;
01123     return(0);
01124 }
01125 while ( --i >= 0 ) {
01126     if ( AT.WildArgTaken[i] == 0 ) {
01127         if ( i == 0 ) {
01128             AT.WorkPointer = OldWork;
01129             *wilds = 0;
01130             return(0);
01131         }
01132     }
01133     else {
01134         (AT.WildArgTaken[i])--;
01135         numofwildarg = 0;
01136         for ( j = 0; j <= i; j++ ) {
01137             numofwildarg += AT.WildArgTaken[j];
01138         }
01139         AT.WildArgTaken[j] = AN.WildEat-numofwildarg;
01140         for ( j++; j < AN.WildArgs; j++ ) AT.WildArgTaken[j] = 0;
01141         break;
01142     }
01143 }
01144 goto toploop;
01145 /*
01146 #] Tensors :
01147 */
01148 }
01149 /*
01150 Count the number of arguments. Either equal or an argument wildcard.
01151 */
01152 mstop = m + m[1];
01153 tstop = t + t[1];
01154 mcount = 0; tcount = 0;
01155 m += FUNHEAD; t += FUNHEAD;
01156 while ( t < tstop ) { tcount++; NEXTARG(t) }
01157 AN.WildArgs = 0;

```

```

01158     while ( m < mstop ) {
01159         mcount++;
01160         if ( *m == -ARGWILD ) AN.WildArgs++;
01161         NEXTARG(m)
01162     }
01163     tobeeaten = tcount - mcount + AN.WildArgs;
01164     if ( tobeeaten ) {
01165         if ( tobeeaten < 0 || AN.WildArgs == 0 ) {
01166             AT.WorkPointer = OldWork;
01167             return(0); /* Cannot match */
01168         }
01169     }
01170 /*
01171     Set up the array AT.WildArgTaken for the number of arguments that each
01172     wildarg eats.
01173 */
01174     AT.WildArgTaken[0] = AN.WildEat = tobeeaten;
01175     for ( i = 1; i < AN.WildArgs; i++ ) AT.WildArgTaken[i] = 0;
01176 topofloop:
01177     numofwildarg = 0;
01178 /*
01179     Test for single wildcard object/argument
01180 */
01181     m = pattern; t = interm;
01182     if ( *m != *t ) {
01183         i = *m - WILDOFFSET;
01184         if ( CheckWild(BHEAD i,FUNTOFUN,*t,&newvalue) ) goto NoCaseB;
01185         AddWild(BHEAD i,FUNTOFUN,newvalue);
01186     }
01187     mstop = m + m[1];
01188 /* tstop = t + t[1]; */
01189     m += FUNHEAD;
01190     t += FUNHEAD;
01191     while ( m < mstop ) {
01192         argmstop = oldm = m;
01193         argtstop = oldt = t;
01194         NEXTARG(argmstop)
01195         NEXTARG(argtstop)
01196         if ( t == tstop ) { /* This concerns a very rare bug */
01197             if ( *m == -ARGWILD ) goto ArgAll;
01198             goto endofloop;
01199         }
01200         if ( *m < 0 && *t < 0 ) {
01201             if ( *t <= -FUNCTION ) {
01202                 if ( *t == *m ) {}
01203                 else if ( *m <= -FUNCTION-WILDOFFSET
01204                     && functions[*t-FUNCTION].spec
01205                     == functions[-*m-FUNCTION-WILDOFFSET].spec ) {
01206                     i = -*m - WILDOFFSET;
01207                     if ( CheckWild(BHEAD i,FUNTOFUN,*t,&newvalue) ) goto endofloop;
01208                     AddWild(BHEAD i,FUNTOFUN,newvalue);
01209                 }
01210                 else if ( *m == -SYMBOL && m[1] >= 2*MAXPOWER ) {
01211                     i = m[1] - 2*MAXPOWER;
01212                     AN.argaddress = AT.FunArg;
01213                     AT.FunArg[ARGHEAD+1] = -*t;
01214                     if ( CheckWild(BHEAD i,SYMTOSUB,1,AN.argaddress) ) goto endofloop;
01215                     AddWild(BHEAD i,SYMTOSUB,0);
01216                 }
01217                 else if ( *m == -ARGWILD ) {
01218 ArgAll:
01219                     i = AT.WildArgTaken[numofwildarg++];
01220                     AN.argaddress = t;
01221                     if ( CheckWild(BHEAD m[1],ARGTOARG,i,t) ) goto endofloop;
01222                     AddWild(BHEAD m[1],ARGTOARG,i);
01223                     m += 2; /*
01224                     while ( --i >= 0 ) { NEXTARG(t) }
01225                     argtstop = t;
01226                 }
01227                 else goto endofloop;
01228             }
01229             else if ( *t == *m ) {
01230                 if ( t[1] == m[1] ) {}
01231                 else if ( *t == -SYMBOL ) {
01232                     j = SYMTOSYM;
01233 SymAll:
01234                     if ( ( i = m[1] - 2*MAXPOWER ) < 0 ) goto endofloop;
01235                     if ( CheckWild(BHEAD i,j,t[1],&newvalue) ) goto endofloop;
01236                     AddWild(BHEAD i,j,newvalue);
01237                 }
01238                 else if ( *t == -INDEX ) {
01239 IndAll:
01240                     i = m[1] - WILDOFFSET;
01241                     if ( i < AM.OffsetIndex || i >= WILDOFFSET+AM.OffsetIndex )
01242                         goto endofloop;
01243                     /* We kill the summed over indices here */
01244                     if ( CheckWild(BHEAD i,INDTOIND,t[1],&newvalue) ) goto endofloop;
01245                     AddWild(BHEAD i,INDTOIND,newvalue);
01246                 }
01247             }
01248         }
01249     }

```

```

01245         else if ( *t == -VECTOR || *t == -MINVECTOR ) {
01246             i = m[1] - WILDOFFSET;
01247             if ( i < AM.OffsetVector ) goto endofloop;
01248             if ( CheckWild(BHEAD i, VECTOVEC, t[1], &newvalue) ) goto endofloop;
01249             AddWild(BHEAD i, VECTOVEC, newvalue);
01250         }
01251         else goto endofloop;
01252     }
01253     else if ( *m == -ARGWILD ) goto ArgAll;
01254     else if ( *m == -INDEX && m[1] >= AM.OffsetIndex+WILDOFFSET
01255 && m[1] < AM.OffsetIndex+(WILDOFFSET<1) ) {
01256         if ( *t == -VECTOR ) goto IndAll;
01257         if ( *t == -SNUMBER && t[1] >= 0 && t[1] < AM.OffsetIndex ) goto IndAll;
01258         if ( *t == -MINVECTOR ) {
01259             i = m[1] - WILDOFFSET;
01260             AN.argaddress = AT.MinVecArg;
01261             AT.MinVecArg[ARGHEAD+3] = t[1];
01262             if ( CheckWild(BHEAD i, INDTO SUB, 1, AN.argaddress) ) goto endofloop;
01263             AddWild(BHEAD i, INDTO SUB, (WORD)0);
01264         }
01265         else goto endofloop;
01266     }
01267     else if ( *m == -SYMBOL && m[1] >= 2*MAXPOWER && *t == -SNUMBER ) {
01268         j = SYMTONUM;
01269         goto SymAll;
01270     }
01271     else if ( *m == -VECTOR && *t == -MINVECTOR &&
01272 ( i = m[1] - WILDOFFSET ) >= AM.OffsetVector ) {
01273 /*
01274 =====
01275         AN.argaddress = AT.MinVecArg;
01276         AT.MinVecArg[ARGHEAD+3] = t[1];
01277         if ( CheckWild(BHEAD i, VECTOSUB, 1, AN.argaddress) ) goto endofloop;
01278         AddWild(BHEAD i, VECTOSUB, (WORD)0);
01279 =====
01280 */
01281         if ( CheckWild(BHEAD i, VECTOMIN, t[1], &newvalue) ) goto endofloop;
01282         AddWild(BHEAD i, VECTOMIN, newvalue);
01283     }
01284 }
01285 else if ( *m == -MINVECTOR && *t == -VECTOR &&
01286 ( i = m[1] - WILDOFFSET ) >= AM.OffsetVector ) {
01287 /*
01288 =====
01289         AN.argaddress = AT.MinVecArg;
01290         AT.MinVecArg[ARGHEAD+3] = t[1];
01291         if ( CheckWild(BHEAD i, VECTOSUB, 1, AN.argaddress) ) goto endofloop;
01292         AddWild(BHEAD i, VECTOSUB, (WORD)0);
01293 =====
01294 */
01295         if ( CheckWild(BHEAD i, VECTOMIN, t[1], &newvalue) ) goto endofloop;
01296         AddWild(BHEAD i, VECTOMIN, newvalue);
01297     }
01298     else goto endofloop;
01299 }
01300 else if ( *t <= -FUNCTION && *m > 0 ) {
01301     if ( ( m[ARGHEAD]+ARGHEAD == *m ) && m[*m-1] == 3
01302 && m[*m-2] == 1 && m[*m-3] == 1 && m[ARGHEAD+1] >= FUNCTION
01303 && m[ARGHEAD+2] == *m-ARGHEAD-4 ) { /* Check for f(?a) etc */
01304         WORD *mmmst, *mmm;
01305         if ( m[ARGHEAD+1] >= FUNCTION+WILDOFFSET ) {
01306             i = *m - WILDOFFSET; /*
01307             i = m[ARGHEAD+1] - WILDOFFSET;
01308             if ( CheckWild(BHEAD i, FUNTOFUN, -*t, &newvalue) ) goto endofloop;
01309             AddWild(BHEAD i, FUNTOFUN, newvalue);
01310         }
01311         else if ( m[ARGHEAD+1] != -*t ) goto endofloop;
01312 /*
01313         Only arguments allowed are ?a etc.
01314 */
01315         mmmst = m+*m-3;
01316         mmm = m + ARGHEAD + FUNHEAD + 1;
01317         while ( mmm < mmmst ) {
01318             if ( *mmm != -ARGWILD ) goto endofloop;
01319             i = 0;
01320             AN.argaddress = t;
01321             if ( CheckWild(BHEAD mmm[1], ARGTOARG, i, t) ) goto endofloop;
01322             AddWild(BHEAD mmm[1], ARGTOARG, i);
01323             mmm += 2;
01324         }
01325     }
01326     else goto endofloop;
01327 }
01328 else if ( *m < 0 && *t > 0 ) {
01329     if ( *m == -SYMBOL ) { /* SYMTOSUB */
01330         if ( m[1] < 2*MAXPOWER ) goto endofloop;
01331         i = m[1] - 2*MAXPOWER;

```



```

01332         AN.argaddress = t;
01333         if ( CheckWild(BHEAD i, SYMTOSUB, 1, AN.argaddress) ) goto endofloop;
01334         AddWild(BHEAD i, SYMTOSUB, 0);
01335     }
01336     else if ( *m == -VECTOR ) {
01337         if ( ( i = m[1] - WILDOFFSET ) < AM.OffsetVector )
01338             goto endofloop;
01339         AN.argaddress = t;
01340         if ( CheckWild(BHEAD i, VECTOSUB, 1, t) ) goto endofloop;
01341         AddWild(BHEAD i, VECTOSUB, (WORD)0);
01342     }
01343     else if ( *m == -INDEX ) {
01344         if ( ( i = m[1] - WILDOFFSET ) < AM.OffsetIndex ) goto endofloop;
01345         if ( i >= AM.OffsetIndex + WILDOFFSET ) goto endofloop;
01346         AN.argaddress = t;
01347         if ( CheckWild(BHEAD i, INDTOSUB, 1, AN.argaddress) ) goto endofloop;
01348         AddWild(BHEAD i, INDTOSUB, (WORD)0);
01349     }
01350     else if ( *m == -ARGWILD ) goto ArgAll;
01351     else goto endofloop;
01352 }
01353 else if ( *m > 0 && *t > 0 ) {
01354     WORD ii = *t-*m;
01355     i = *m;
01356     do { if ( *m++ != *t++ ) break; } while ( --i > 0 );
01357     if ( i == 1 && ii == 0 ) { /* sign difference */
01358         goto endofloop;
01359     }
01360     else if ( i > 0 ) {
01361         WORD *cto, *cfrom, *csav, ci;
01362         WORD oRepFunNum;
01363         WORD *oRepFunList;
01364         WORD *oterstart, *oterstop, *opatstop;
01365         WORD oExpectedSign;
01366         WORD wildargs, wildeat;
01367         /*
01368         Not an exact match here.
01369         We have to hope that the pattern contains a composite wildcard.
01370         */
01371         m = oldm; t = oldt;
01372         m += ARGHEAD; t += ARGHEAD; /* Point at (first?) term */
01373         mtrmstop = m + *m;
01374         ttrmstop = t + *t;
01375         if ( mtrmstop < argmstop ) goto endofloop; /* More than one term */
01376         msizcoef = mtrmstop[-1];
01377         if ( msizcoef < 0 ) msizcoef = -msizcoef;
01378         msubstop = mtrmstop - msizcoef;
01379         m++;
01380         if ( m >= msubstop ) goto endofloop; /* Only coefficient */
01381         /*
01382         Here we have a composite term. It can match provided it
01383         matches the entire argument. This argument must be a
01384         single term also and the coefficients should match
01385         (more or less).
01386         The matching takes:
01387         1: Match the functions etc. Nothing can be left.
01388         2: Match dotproducts and symbols. ONLY must match
01389            and nothing may be left.
01390         For safety it is best to take the term out and put it
01391         in workspace.
01392         */
01393         if ( argtstop > ttrmstop ) goto endofloop;
01394         m--;
01395         oterstart = AN.terstart;
01396         oterstop = AN.terstop;
01397         opatstop = AN.patstop;
01398         oRepFunList = AN.RepFunList;
01399         oRepFunNum = AN.RepFunNum;
01400         AN.RepFunNum = 0;
01401         AN.RepFunList = AT.WorkPointer;
01402         AT.WorkPointer = (WORD *)(((UBYTE *) (AT.WorkPointer)) + AM.MaxTer);
01403         if ( AT.WorkPointer+*t+5 > AT.WorkTop ) {
01404             MLOCK(ErrorMessageLock);
01405             MesWork();
01406             MUNLOCK(ErrorMessageLock);
01407             return(-1);
01408         }
01409     }
01410     csav = cto = AT.WorkPointer;
01411     cfrom = t;
01412     ci = *t;
01413     while ( --ci >= 0 ) *cto++ = *cfrom++;
01414     AT.WorkPointer = cto;
01415     ci = msizcoef;
01416     cfrom = mtrmstop;
01417     --ci;
01418     if ( abs(*--cfrom) != abs(*--cto) ) {

```

```

01419         AT.WorkPointer = csav;
01420         AN.RepFunList = oRepFunList;
01421         AN.RepFunNum = oRepFunNum;
01422         AN.terstart = oterstart;
01423         AN.terstop = oterstop;
01424         AN.patstop = opatstop;
01425         goto endofloop;
01426     }
01427     i = (*cfrom != *cto) ? 1 : 0; /* buffer AN.ExpectedSign until we are beyond the goto
*/
01428     while ( --ci >= 0 ) {
01429         if ( *--cfrom != *--cto ) {
01430             AT.WorkPointer = csav;
01431             AN.RepFunList = oRepFunList;
01432             AN.RepFunNum = oRepFunNum;
01433             AN.terstart = oterstart;
01434             AN.terstop = oterstop;
01435             AN.patstop = opatstop;
01436             goto endofloop;
01437         }
01438     }
01439     oExpectedSign = AN.ExpectedSign; /* buffer AN.ExpectedSign until we are beyond
FindRest/FindOnly */
01440     AN.ExpectedSign = i;
01441     *m -= msizcoef;
01442     wildargs = AN.WildArgs;
01443     wildeat = AN.WildEat;
01444     for ( i = 0; i < wildargs; i++ ) wildargtaken[i] = AT.WildArgTaken[i];
01445     AN.ForFindOnly = 0; AN.UseFindOnly = 1;
01446     AN.nogroundlevel++;
01447     if ( FindRest(BHEAD csav,m) && ( AN.UsedOtherFind || FindOnly(BHEAD csav,m) ) ) {}
01448     else {
01449 nomatch:
01450         *m += msizcoef;
01451         AT.WorkPointer = csav;
01452         AN.RepFunList = oRepFunList;
01453         AN.RepFunNum = oRepFunNum;
01454         AN.terstart = oterstart;
01455         AN.terstop = oterstop;
01456         AN.patstop = opatstop;
01457         AN.WildArgs = wildargs;
01458         AN.WildEat = wildeat;
01459         AN.ExpectedSign = oExpectedSign;
01460         AN.nogroundlevel--;
01461         for ( i = 0; i < wildargs; i++ ) AT.WildArgTaken[i] = wildargtaken[i];
01462         goto endofloop;
01463     }
01464     /* if ( *m == 1 || m[1] < FUNCTION || functions[m[1]-FUNCTION].spec >= TENSORFUNCTION ) {
*/
01465     if ( *m == 1 || m[1] < FUNCTION ) {
01466         if ( AN.ExpectedSign ) goto nomatch;
01467     }
01468     else {
01469         if ( m[1] > FUNCTION + WILDOFFSET ) {
01470             if ( functions[m[1]-FUNCTION-WILDOFFSET].spec >= TENSORFUNCTION ) {
01471                 if ( AN.ExpectedSign != AN.RepFunList[AN.RepFunNum-1] ) goto nomatch;
01472             }
01473         }
01474         else {
01475             if ( AN.ExpectedSign != AN.RepFunList[AN.RepFunNum-1] ) goto nomatch;
01476         /*
01477             if ( functions[m[1]-FUNCTION].spec >= TENSORFUNCTION ) {
01478                 if ( AN.ExpectedSign != AN.RepFunList[AN.RepFunNum-1] ) goto nomatch;
01479             }
01480         */
01481         }
01482     }
01483     AN.nogroundlevel--;
01484     AN.ExpectedSign = oExpectedSign;
01485     AN.WildArgs = wildargs;
01486     AN.WildEat = wildeat;
01487     for ( i = 0; i < wildargs; i++ ) AT.WildArgTaken[i] = wildargtaken[i];
01488     Substitute(BHEAD csav,m,1);
01489     cto = csav;
01490     cfrom = cto + *cto - msizcoef;
01491     cto++;
01492     *m += msizcoef;
01493     AT.WorkPointer = csav;
01494     AN.RepFunList = oRepFunList;
01495     AN.RepFunNum = oRepFunNum;
01496     AN.terstart = oterstart;
01497     AN.terstop = oterstop;
01498     AN.patstop = opatstop;
01499     if ( *cto != SUBEXPRESSION ) goto endofloop;
01500     cto += cto[1];
01501     if ( cto < cfrom ) goto endofloop;
01502 }

```

```

01503     }
01504     else goto endofloop;
01505
01506     t = argtstop;                      /* Next argument */
01507     m = argmstop;
01508 }
01509 if ( AN.SignCheck && AN.ExpectedSign ) goto endofloop;
01510 AT.WorkPointer = OldWork;
01511 if ( AN.WildArgs > 1 ) *wilds = 1;
01512 if ( AN.SignCheck && AN.ExpectedSign ) return(0);
01513 return(1);      /* m was eaten. we have a match! */
01514
01515 endofloop;;
01516 /*
01517     restore the current Wildcard assignments
01518 */
01519 i = nwstore;
01520 if ( i > 0 ) {
01521     m = AN.WildValue;
01522     t = OldWork + ntw; r = AT.WildMask;
01523     do {
01524         *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01525     } while ( --i > 0 );
01526     C->numrhs = *t++;
01527     C->Pointer = C->Buffer + oldcpointer;
01528 }
01529
01530 endoloop;;
01531 i = AN.WildArgs-1;
01532 if ( i <= 0 ) {
01533     AT.WorkPointer = OldWork;
01534     return(0);
01535 }
01536 while ( --i >= 0 ) {
01537     if ( AT.WildArgTaken[i] == 0 ) {
01538         if ( i == 0 ) {
01539             AT.WorkPointer = OldWork;
01540             return(0);
01541         }
01542     }
01543     else {
01544         (AT.WildArgTaken[i])--;
01545         numofwildarg = 0;
01546         for ( j = 0; j <= i; j++ ) {
01547             numofwildarg += AT.WildArgTaken[j];
01548         }
01549         AT.WildArgTaken[j] = AN.WildEat-numofwildarg;
01550 /* ----> bug to be replaced in other source code */
01551         for ( j++; j < AN.WildArgs; j++ ) AT.WildArgTaken[j] = 0;
01552         break;
01553     }
01554 }
01555 goto topofloop;
01556 NoCaseB:
01557 /*
01558     Restore the old Wildcard assignments
01559 */
01560 i = nwstore;
01561 if ( i > 0 ) {
01562     m = AN.WildValue;
01563     t = OldWork + ntw; r = AT.WildMask;
01564     do {
01565         *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01566     } while ( --i > 0 );
01567     C->numrhs = *t++;
01568     C->Pointer = C->Buffer + oldcpointer;
01569 }
01570 AT.WorkPointer = OldWork;
01571 return(0);      /* no match */
01572 }
01573
01574 /*
01575     #] MatchFunction :
01576     #[ ScanFunctions :          WORD ScanFunctions(inpat,inter,par)
01577
01578     Finds in which functions to look for a match.
01579     inpat is the start of the pattern still to be matched.
01580     inter is the start of the term still to be matched.
01581     par gives information about commutativity.
01582         par = 0: nothing special
01583         par = 1: regular noncommuting function
01584         par = 2: GAMMA function
01585
01586     AN.patstop: end of the functions field in the search pattern
01587     AN.terstop: end of the functions field in the target pattern
01588     AN.terstart: address of entire term;
01589

```

```

01590         The actual matching of the functions and their arguments is done
01591         in a number of different routines. Mainly MatchFunction when there
01592         are no symmetry properties.
01593         Also: MatchE
01594             MatchCy
01595             FunMatchSy
01596             FunMatchCy
01597
01598         The main problem here is backtracking, ie continuing with wildcard
01599         possibilities when a first assignment doesn't work.
01600         Important note: this was completely forgotten in the symmetric
01601         functions till 6-jan-2009. As of the moment this still has to
01602         be fixed. ?????21-mar-2023???? Is this still unfixed????
01603
01604         Functions inside functions can cause problems when antisymmetric
01605         functions are involved. The sign of the term may be at stake.
01606         At the lowest level this is no problem but in f(-fas(n2,n1)) this
01607         plays a role. Next is when we have a product of functions inside
01608         an argument. The strategy must be that we test the sign only at the
01609         last function. Hence, when input+input[1] >= AN.patstop.
01610         We might relax that to the last antisymmetric function at a later stage.
01611
01612         New scheme to be implemented for non-commuting objects:
01613         When we are matching a second (or higher) function, any match can only
01614         be directly after the last matched non-commuting function or a commuting
01615         function. This will take care of whatever happens in MatchE etc.
01616     */
01617
01618     int ScanFunctions(PHEAD WORD *input, WORD *inter, WORD par)
01619     {
01620         GETBIDENTITY
01621         WORD i, *m, *t, *r, sym, psym;
01622         WORD *newpat, *newter, *instart, *oinpat = 0, *ointer = 0;
01623         WORD nwstore, offset, *OldWork, SetStop = 0, oRepFunNum = AN.RepFunNum;
01624         WORD wilds, wildargs = 0, wildeat = 0, *wildargtaken;
01625         WORD *Oterfirstcomm = AN.terfirstcomm;
01626         CBUF *C = cbuf+AT.ebufnum;
01627         int ntw = AN.NumTotWildArgs;
01628         LONG oldcpointer = C->Pointer - C->Buffer;
01629         WORD oldSignCheck = AN.SignCheck;
01630         instart = inter;
01631     /*
01632         Only active for the last function in the pattern.
01633         The actual test on the sign is in MatchFunction or the symmetric functions
01634     */
01635         if ( AN.nogroundlevel ) {
01636             AN.SignCheck = ( input + input[1] >= AN.patstop ) ? 1 : 0;
01637         }
01638         else {
01639             AN.SignCheck = 0;
01640         }
01641     /*
01642         Store the current Wildcard assignments
01643     */
01644         t = wildargtaken = OldWork = AT.WorkPointer;
01645         t += ntw;
01646         m = AN.WildValue;
01647         nwstore = i = (m[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
01648         if ( i > 0 ) {
01649             r = AT.WildMask;
01650             do {
01651                 *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
01652             } while ( --i > 0 );
01653             *t++ = C->numrhs;
01654         }
01655         if ( t >= AT.WorkTop ) {
01656             MLOCK(ErrorMessageLock);
01657             MesWork();
01658             MUNLOCK(ErrorMessageLock);
01659             Terminate(-1);
01660         }
01661         AT.WorkPointer = t;
01662         do {
01663 #ifndef NEWCOMMUTE
01664     /*
01665         Find an eligible unsubstituted function
01666     */
01667         if ( AN.RepFunNum > 0 ) {
01668     /*
01669             First try a non-commuting function, just after the last
01670             substituted non-commuting function.
01671     */
01672             if ( *inter >= FUNCTION && functions[*inter-FUNCTION].commute ) {
01673                 do {
01674                     offset = WORDDIF(inter,AN.terstart);
01675                     for ( i = 0; i < AN.RepFunNum; i += 2 ) {
01676                         if ( AN.RepFunList[i] >= offset ) break;

```

```

01677         }
01678         if ( i >= AN.RepFunNum ) break;
01679         inter += inter[1];
01680     } while ( inter < AN.terfirstcomm );
01681     if ( inter < AN.terfirstcomm ) { /* Check that it is directly after */
01682         for ( i = 0; i < AN.RepFunNum; i += 2 ) {
01683             if ( functions[AN.terstart[AN.RepFunList[i]]-FUNCTION].commute
01684                 && AN.RepFunList[i]+AN.terstart[AN.RepFunList[i]+1] == offset ) break;
01685         }
01686         if ( i < AN.RepFunNum ) goto trythis;
01687     }
01688     inter = AN.terfirstcomm;
01689 }
01690 /*
01691     Now try one of the commuting functions
01692 */
01693 while ( inter < AN.terstop ) {
01694     offset = WORDDIF(inter,AN.terstart);
01695     for ( i = 0; i < AN.RepFunNum; i += 2 ) {
01696         if ( AN.RepFunList[i] == offset ) break;
01697     }
01698     if ( i >= AN.RepFunNum ) break;
01699     inter += inter[1];
01700 }
01701 if ( inter >= AN.terstop ) goto Failure;
01702 trythis:;
01703 }
01704 else {
01705     /*
01706         The first function can be anywhere. We have no problems.
01707     */
01708     offset = WORDDIF(inter,AN.terstart);
01709 }
01710 #else
01711     /* first find an unsubstituted function */
01712     do {
01713         offset = WORDDIF(inter,AN.terstart);
01714         for ( i = 0; i < AN.RepFunNum; i += 2 ) {
01715             if ( AN.RepFunList[i] == offset ) break;
01716         }
01717         if ( i >= AN.RepFunNum ) break;
01718         inter += inter[1];
01719     } while ( inter < AN.terstop );
01720     if ( inter >= AN.terstop ) goto Failure;
01721 #endif
01722     wilds = 0;
01723     /* We found one */
01724     if ( *inter >= FUNCTION && *inpat >= FUNCTION ) {
01725         if ( *inpat == *inter || *inpat >= FUNCTION + WILDOFFSET ) {
01726             /*
01727                 if ( inter[1] == FUNHEAD ) goto rewild;
01728             */
01729             if ( functions[*inter-FUNCTION].spec >= TENSORFUNCTION
01730                 && ( *inter == *inpat ||
01731                     functions[*inpat-FUNCTION-WILDOFFSET].spec >= TENSORFUNCTION ) ) {
01732                 sym = functions[*inter-FUNCTION].symmetric & ~REVERSEORDER;
01733                 if ( *inpat == *inter ) psym = sym;
01734                 else psym = functions[*inpat-FUNCTION-WILDOFFSET].symmetric & ~REVERSEORDER;
01735                 if ( sym == ANTISYMMETRIC || sym == SYMMETRIC
01736                     || psym == SYMMETRIC || psym == ANTISYMMETRIC ) {
01737                     if ( sym == ANTISYMMETRIC && psym == SYMMETRIC ) goto rewild;
01738                     if ( sym == SYMMETRIC && psym == ANTISYMMETRIC ) goto rewild;
01739                 }
01740                 /*
01741                     Special function call for (anti)symmetric tensors
01742                 */
01743                 if ( MatchE(BHEAD inpat,inter,instart,par) ) goto OnSuccess;
01744             }
01745             else if ( sym == CYCLESYMMETRIC || sym == RCYCLESYMMETRIC
01746                 || psym == CYCLESYMMETRIC || psym == RCYCLESYMMETRIC ) {
01747                 /*
01748                     Special function call for (r)cyclic tensors
01749                 */
01750                 if ( MatchCy(BHEAD inpat,inter,instart,par) ) goto OnSuccess;
01751             }
01752             else goto rewild;
01753         }
01754     }
01755     else if ( functions[*inter-FUNCTION].spec <= 0
01756         && ( *inter == *inpat ||
01757             functions[*inpat-FUNCTION-WILDOFFSET].spec <= 0 ) ) {
01758         sym = functions[*inter-FUNCTION].symmetric & ~REVERSEORDER;
01759         if ( *inpat == *inter ) psym = sym;
01760         else psym = functions[*inpat-FUNCTION-WILDOFFSET].symmetric & ~REVERSEORDER;
01761         if ( psym == SYMMETRIC || sym == SYMMETRIC
01762             || psym == ANTISYMMETRIC || sym == ANTISYMMETRIC ) {
01763             /*
01764                 The next statement was commented out. Why???
01765                 Werkt nog niet. Teken wordt nog niet bijgehouden.
01766                 5-nov-2001

```

```

01764 */
01765         || psym == ANTISYMMETRIC || sym == ANTISYMMETRIC
01766     ) {
01767         if ( sym == ANTISYMMETRIC && psym == SYMMETRIC ) goto rewild;
01768         if ( sym == SYMMETRIC && psym == ANTISYMMETRIC ) goto rewild;
01769         if ( FunMatchSy(BHEAD inpat,inter,instart,par) ) goto OnSuccess;
01770     }
01771     else
01772         if ( sym == CYCLESYMMETRIC || sym == RCYCLESYMMETRIC
01773         || psym == CYCLESYMMETRIC || psym == RCYCLESYMMETRIC ) {
01774             if ( FunMatchCy(BHEAD inpat,inter,instart,par) ) goto OnSuccess;
01775         }
01776         else goto rewild;
01777     }
01778     else goto rewild;
01779     AN.terfirstcomm = Oterfirstcomm;
01780 }
01781 else if ( par > 0 ) { SetStop = 1; goto maybenext; }
01782 }
01783 else {
01784 rewild:
01785     AN.terfirstcomm = Oterfirstcomm;
01786     if ( *inter != SUBEXPRESSION && MatchFunction(BHEAD inpat,inter,&wilds) ) {
01787         AN.terfirstcomm = Oterfirstcomm;
01788         if ( wilds ) {
01789             /*
01790              Store wildcards to continue in MatchFunction if the current
01791              wildcards do not work out.
01792             */
01793             wildargs = AN.WildArgs;
01794             wildeat = AN.WildEat;
01795             for ( i = 0; i < wildargs; i++ ) wildargtaken[i] = AT.WildArgTaken[i];
01796             oinpat = inpat; ointer = inter;
01797         }
01798         if ( par && *inter == GAMMA && AN.RepFunList[AN.RepFunNum+1] ) {
01799             SetStop = 1; goto NoMat;
01800         }
01801         if ( par == 2 ) {
01802             if ( *inter < FUNCTION || functions[*inter-FUNCTION].commute ) {
01803                 goto NoMat;
01804             }
01805             par = 1;
01806         }
01807         AN.RepFunList[AN.RepFunNum] = offset;
01808         AN.RepFunNum += 2;
01809         newpat = inpat + inpat[1];
01810         if ( newpat >= AN.patstop ) {
01811             if ( AN.UseFindOnly == 0 ) {
01812                 if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
01813                     AN.UsedOtherFind = 1;
01814                     goto OnSuccess;
01815                 }
01816                 AN.RepFunNum -= 2;
01817                 goto NoMat;
01818             }
01819             goto OnSuccess;
01820         }
01821         if ( *inter < FUNCTION || functions[*inter-FUNCTION].commute ) {
01822             newter = inter + inter[1];
01823             if ( newter >= AN.terstop ) goto Failure;
01824             if ( *inter == GAMMA && inpat[1] <
01825             inter[1] - AN.RepFunList[AN.RepFunNum-1] ) {
01826                 if ( ScanFunctions(BHEAD newpat,newter,2) ) goto OnSuccess;
01827                 AN.terfirstcomm = Oterfirstcomm;
01828             }
01829             else if ( *newter == SUBEXPRESSION ) {}
01830             else if ( functions[*inter-FUNCTION].commute ) {
01831                 if ( ScanFunctions(BHEAD newpat,newter,1) ) goto OnSuccess;
01832                 AN.terfirstcomm = Oterfirstcomm;
01833                 if ( ( *newpat < (FUNCTION+WILDOFFSET)
01834                 && ( functions[*newpat-FUNCTION].commute == 0 ) ) ||
01835                 ( *newpat >= (FUNCTION+WILDOFFSET)
01836                 && ( functions[*newpat-FUNCTION-WILDOFFSET].commute == 0 ) ) ) {
01837                     newter = AN.terfirstcomm;
01838                     if ( newter < AN.terstop && ScanFunctions(BHEAD newpat,newter,1) ) goto
01839                     OnSuccess;
01840                 }
01841             }
01842             else {
01843                 if ( ScanFunctions(BHEAD newpat,instart,1) ) goto OnSuccess;
01844                 AN.terfirstcomm = Oterfirstcomm;
01845             }
01846             SetStop = par;
01847         }
01848         else {
01849             /*
01850              Shouldn't this be newpat instead of inpat?????

```

```

01850 */
01851         if ( par && inter > instart && ( ( *newpat < (FUNCTION+WILDOFFSET)
01852         && functions[*newpat-FUNCTION].commute ) ||
01853         ( *newpat >= (FUNCTION+WILDOFFSET)
01854         && functions[*newpat-FUNCTION-WILDOFFSET].commute ) ) ) {
01855             SetStop = 1;
01856         }
01857         else {
01858             newter = instart;
01859             if ( ScanFunctions(BHEAD newpat,newter,par) ) goto OnSuccess;
01860             AN.terfirstcomm = Oterfirstcomm;
01861         }
01862     }
01863 /*
01864     Restore the old Wildcard assignments
01865 */
01866 NoMat:
01867     i = nwstore;
01868     if ( i > 0 ) {
01869         m = AN.WildValue;
01870         t = OldWork + ntwa; r = AT.WildMask;
01871         do {
01872             *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01873         } while ( --i > 0 );
01874         C->numrhs = *t++;
01875         C->Pointer = C->Buffer + oldcpointer;
01876     }
01877 /*
01878     AN.RepFunNum -= 2; */
01879     AN.RepFunNum = oRepFunNum;
01880     if ( wilds ) {
01881         inter = ointer; inpat = oinpat;
01882         AN.WildArgs = wildargs;
01883         AN.WildEat = wildeat;
01884         for ( i = 0; i < wildargs; i++ ) AT.WildArgTaken[i] = wildargtaken[i];
01885         goto rewild;
01886     }
01887     if ( SetStop ) break;
01888 }
01889 else if ( par ) {
01890     maybenext:
01891     if ( *inpat < (FUNCTION+WILDOFFSET) ) {
01892         if ( *inpat < FUNCTION ||
01893             functions[*inpat-FUNCTION].commute ) break;
01894     }
01895     else {
01896         if ( functions[*inpat-FUNCTION-WILDOFFSET].commute ) break;
01897     }
01898     inter += inter[1];
01899 } while ( inter < AN.terstop );
01900 Failure:
01901     AN.SignCheck = oldSignCheck;
01902     AT.WorkPointer = OldWork;
01903     return(0);
01904 OnSuccess:
01905     if ( AT.idallflag && AN.nogroundlevel <= 0 ) {
01906         if ( AT.idallmaxnum > 0 && AT.idallnum >= AT.idallmaxnum ) {
01907             AN.terfirstcomm = Oterfirstcomm;
01908             AN.SignCheck = oldSignCheck;
01909             AT.WorkPointer = OldWork;
01910             return(0);
01911         }
01912         SubsInAll(BHEAD0);
01913         AT.idallnum++;
01914         if ( AT.idallmaxnum == 0 || AT.idallnum < AT.idallmaxnum ) goto NoMat;
01915     }
01916     AN.terfirstcomm = Oterfirstcomm;
01917     AN.SignCheck = oldSignCheck;
01918 /*
01919     Now the disorder test
01920 */
01921     if ( AN.DisOrderFlag && AN.RepFunNum >= 4 ) {
01922         WORD k, kk;
01923         for ( i = 2; i < AN.RepFunNum; i += 2 ) {
01924 /*
01925 -----> We still have to copy the code from Normalize wrt properorderflag
01926 */
01927             m = AN.terstart + AN.RepFunList[i-2];
01928             t = AN.terstart + AN.RepFunList[i];
01929             if ( *m != *t ) {
01930                 if ( *m > *t ) continue;
01931                 goto doesmatch;
01932             }
01933             if ( *m >= FUNCTION && functions[*m-FUNCTION].spec >=
01934             TENSORFUNCTION ) {
01935                 k = m[1] - FUNHEAD;
01936                 kk = t[1] - FUNHEAD;

```

```

01937             m += FUNHEAD;
01938             t += FUNHEAD;
01939         }
01940         else {
01941             k = m[1] - FUNHEAD;
01942             kk = t[1] - FUNHEAD;
01943             m += FUNHEAD;
01944             t += FUNHEAD;
01945         }
01946         while ( k > 0 && kk > 0 ) {
01947             if ( *m < *t ) goto NextFor;
01948             else if ( *m++ > *t++ ) goto doesmatch;
01949             k--; kk--;
01950         }
01951         if ( k > 0 ) goto doesmatch;
01952 NextFor;;
01953     }
01954     SetStop = 1;
01955     goto NoMat;
01956 }
01957 doesmatch:
01958     AT.WorkPointer = OldWork;
01959     return(1);
01960 }
01961
01962 /*
01963     #] ScanFunctions :
01964     #] Patterns :
01965 */

```

4.55 fwin.h File Reference

Macros

- `#define` [LINEFEED](#) `'\n'`
- `#define` [CARRIAGERETURN](#) `0x0D`
- `#define` [WITHRETURN](#)
- `#define` [WITHSYSTEM](#)
- `#define` [P_term](#)(code) `exit((int)(code<0?-code:code))`
- `#define` [SEPARATOR](#) `'\'`
- `#define` [ALTSEPARATOR](#) `'/'`
- `#define` [PATHSEPARATOR](#) `','`
- `#define` [WITH_ENV](#)

4.55.1 Detailed Description

Settings for Windows computers.

Definition in file [fwin.h](#).

4.55.2 Macro Definition Documentation

4.55.2.1 LINEFEED

```
#define LINEFEED '\n'
```

Definition at line 33 of file [fwin.h](#).

4.55.2.2 CARRIAGEReturn

```
#define CARRIAGEReturn 0x0D
```

Definition at line 34 of file [fwin.h](#).

4.55.2.3 WITHReturn

```
#define WITHReturn
```

Definition at line 35 of file [fwin.h](#).

4.55.2.4 WITHSystem

```
#define WITHSystem
```

Definition at line 37 of file [fwin.h](#).

4.55.2.5 P_term

```
#define P_term(  
    code ) exit((int) (code<0?-code:code))
```

Definition at line 39 of file [fwin.h](#).

4.55.2.6 SEPARATOR

```
#define SEPARATOR '\\'
```

Definition at line 41 of file [fwin.h](#).

4.55.2.7 ALTSEPARATOR

```
#define ALTSEPARATOR '/'
```

Definition at line 42 of file [fwin.h](#).

4.55.2.8 PATHSEPARATOR

```
#define PATHSEPARATOR ';' 
```

Definition at line 43 of file [fwin.h](#).

4.55.2.9 WITH_ENV

```
#define WITH_ENV
```

Definition at line 44 of file [fwin.h](#).

4.56 fwin.h

[Go to the documentation of this file.](#)

```
00001
00006 /* #[ License : */
00007 /*
00008 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009 *   When using this file you are requested to refer to the publication
00010 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011 *   This is considered a matter of courtesy as the development was paid
00012 *   for by FOM the Dutch physics granting agency and we would like to
00013 *   be able to track its scientific use to convince FOM of its value
00014 *   for the community.
00015 *
00016 *   This file is part of FORM.
00017 *
00018 *   FORM is free software: you can redistribute it and/or modify it under the
00019 *   terms of the GNU General Public License as published by the Free Software
00020 *   Foundation, either version 3 of the License, or (at your option) any later
00021 *   version.
00022 *
00023 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00026 *   details.
00027 *
00028 *   You should have received a copy of the GNU General Public License along
00029 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00030 */
00031 /* #[ License : */
00032
00033 #define LINEFEED '\n'
00034 #define CARRIAGERETURN 0x0D
00035 #define WITHRETURN
00036
00037 #define WITHSYSTEM
00038
00039 #define P_term(code)    exit((int)(code<0?-code:code))
00040
00041 #define SEPARATOR '\\\
00042 #define ALTSEPARATOR '/'
00043 #define PATHSEPARATOR ';'
00044 #define WITH_ENV
```

4.57 grcc.cc

```
00001 // { ( [
00002 //*****
00003 // grcc.cc
00004
00005 /* #[ License : */
00006 /*
00007 *   Copyright (C) 2023-2026 T. Kaneko
00008 *   When using this file you are requested to refer to the publication
00009 *   Comput.Phys.Commun. 92 (1995) 127-152
00010 *
00011 *   This file is part of FORM.
00012 *
00013 *   FORM is free software: you can redistribute it and/or modify it under the
00014 *   terms of the GNU General Public License as published by the Free Software
00015 *   Foundation, either version 3 of the License, or (at your option) any later
00016 *   version.
00017 *
00018 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00019 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00020 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00021 *   details.
00022 *
```

```

00023 *   You should have received a copy of the GNU General Public License along
00024 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00025 */
00026 /* #] License : */
00027
00028 #ifndef NOFORM
00029 extern "C" {
00030 #include "form3.h"
00031 }
00032 #endif
00033
00034 #include <iostream>
00035 #include <iomanip>
00036 #include <sstream>
00037 #include <cstdio>
00038 #include <stdlib.h>
00039 #include <string.h>
00040 #include <time.h>
00041
00042 //=====
00043 // new default values of options
00044 // #define OLDDFAULT
00045
00046 // new function for the output in python format
00047 // #define NEWOUTPY
00048
00049 //=====
00050 // For debugging
00051 //
00052 // #define MONITOR      // in graph elimination
00053 // #define CHECK        // check consistency
00054
00055 //=====
00056 // optimization : best combination depends on process by process
00057 #define SIMPSEL
00058
00059 // optimization : lower and upper bound of deg of target node of connection.
00060 #define MINMAXLEG
00061
00062 // optimization : optimization with respect to 'extonly' particles
00063 #define OPTEXTONLY
00064
00065 // check unique interaction by name or by code
00066 #define UNIQINTR
00067
00068 // possible extensions of the program
00069 // #define ORBITS
00070
00071 // reverse direction of an edge of agraph when anti-particle flows on it.
00072 // In this case the direction of edges will be different
00073 // from ones of original mgraph.
00074 // #define EDGEORDER
00075
00076 //-----
00077 #include "grcc.h"
00078
00079 #ifdef GRCC_NAMESPACE
00080 using namespace Grcc;
00081 #endif
00082
00083 #define MAXSTR 1024
00084
00085 //-----
00086 // Macro functions
00087 #define CLWIGHTD(x) (5*(x))
00088 #define CLWIGHTO(x) (3*(x)-2)
00089 #define MAXSTRLEN 81
00090
00091 #define MASK(n) ((1ul) << (n))
00092
00093 #define BOOLSTR(x) ((x) ? "True" : "False")
00094
00095 //-----
00096 // Static variables
00097
00098 static OptDef optDef0[] = {
00099 {"Step", "Generate particle assigned graphs", GRCC_AGraph, 0},
00100 {"Outgrf", "Output to file (out.grf)", False, 0},
00101 {"Outgrp", "Output to file (out.grp)", False, 0},
00102 {"OPI", "Only 1PI graphs", True, 0},
00103 {"NoSelfLoop", "Exclude graphs with loops consist of 1 edge", True, 0},
00104 {"NoTadpole", "Exclude graphs with tadpoles (2 edge connected)", True, 0},
00105 {"No1PtBlock", "Exclude graphs with tadpole blocks (2 node connected)", False, 0},
00106 {"No2Pt1PI", "Exclude graphs with 2-point subgraphs", False, 0},
00107 {"NoExtSelf", "Exclude graphs with 2-pt subgraphs at ext. particles", False, 0},
00108 {"NoAdj2PtV", "Exclude graphs with an edge connecting 2-pt vertices", False, 0},
00109 {"Block", "Exclude graphs with more than one block", False, 0},

```

```

00110 {"NoMultiEdge", "Exclude graphs with multi-edges", False, 0},
00111 {"SymmInitial", "Symmetrize initial particles", False, 0},
00112 {"SymmFinal", "Symmetrize final particles", False, 0},
00113 };
00114 static OptDef optDef1[] = {
00115 {"Step", "Generate particle assigned graphs", GRCC_AGraph, 0},
00116 {"Outgrf", "Output to file (out.grf)", False, 0},
00117 {"Outgrp", "Output to file (out.grp)", False, 0},
00118 {"OPI", "Only 1PI graphs", False, 0},
00119 {"NoSelfLoop", "Exclude graphs with loops consist of 1 edge", False, 0},
00120 {"NoTadpole", "Exclude graphs with tadpoles (2 edge connected)", False, 0},
00121 {"No1PtBlock", "Exclude graphs with tadpole blocks (2 node connected)", False, 0},
00122 {"No2PtL1PI", "Exclude graphs with 2-point subgraphs", False, 0},
00123 {"NoExtSelf", "Exclude graphs with 2-pt subgraphs at ext. particles", False, 0},
00124 {"NoAdj2PtV", "Exclude graphs with an edge connecting 2-pt vertices", False, 0},
00125 {"Block", "Exclude graphs with more than one block", False, 0},
00126 {"NoMultiEdge", "Exclude graphs with multi-edges", False, 0},
00127 {"SymmInitial", "Symmetrize initial particles", False, 0},
00128 {"SymmFinal", "Symmetrize final particles", False, 0},
00129 };
00130 #ifdef OLDDEFAULT
00131 static OptDef *optDef = &(optDef0[0]);
00132 static int nOptDef = sizeof(optDef0)/sizeof(OptDef);
00133 #else
00134 static OptDef *optDef = &(optDef1[0]);
00135 static int nOptDef = sizeof(optDef1)/sizeof(OptDef);
00136 #endif
00137
00138
00139 static OptQGDef optQGDef[] = {
00140 {"onepi", "onepr", "one-particle illreducible"},
00141 {"onshell", "offshell", "without self-energy part at external particles"},
00142 {"nosigma", "sigma", "no 2-point functions"},
00143 {"nosnail", "snail", "without snail"},
00144 {"notadpole", "tadpole", "without tadpole"},
00145 {"simple", "notsimple", "without self-loop nor multi-edge"},
00146 {"bipart", "nonbipart", "only bipartite graph"},
00147 {"cycli", "cyclr", "???"},
00148 {"floop", "", "without fermion loops of odd length"},
00149 #ifdef GRCC_QGRAF_OPT_TOPOL
00150 {"topol", "?", "topology"},
00151 #endif
00152 };
00153
00154 static int nOptQGDef = sizeof(optQGDef)/sizeof(OptQGDef);
00155
00156 static int prlevel = 2;
00157 static ErExit *erExit = NULL;
00158 static void *erExitArg = NULL;
00159
00160 //*****
00161 // Utility functions
00162 //=====
00163
00164 static void grcc_fprintf(FILE* out, const char* fmt, ...);
00165 static void erEnd(const char *msg);
00166 static Bool nextPart(int nelem, int nclist, int *clist, int *nl, int *r);
00167 static void prllist(int n, const int *a, const char *msg);
00168 static int *newArray(int size, int val);
00169 static int *deleteArray(int *a);
00170 static int **newMat(int n0, int n1, int val);
00171 static int **deleteMat(int **m, int n0);
00172 static void prIntArray(int n, int *p, const char *msg);
00173 static void prIntArrayErr(int n, int *p, const char *msg);
00174 static int *intdup(int n, int *v);
00175 static int *delintdup(int *v);
00176 static void bsort(int n, int *ord, int *a);
00177 static void sorti(int n, int *a);
00178 static int sortb(int n, int *a);
00179 static int toSList(int n, int *a);
00180 static int intSetAdd(int n, int *a, int v, const int size);
00181 static int intSListAdd(int n, int *a, int v, const int size);
00182 static int cmpArray(int na, int *a, int nb, int *b);
00183 static Bool leqArray(int n, int *a, int *b);
00184 static void prMomStr(int mom, const char *ms, int mn);
00185 static void listDiff(int na, int *a, int nb, int *b, int *np, int *p, int *nq, int *q, int *nr, int
    *r);
00186 static Bool isSubSList(int na, int *a, int nb, int *b);
00187 static int subtrSet(int na, int *a, int nb, int *b, int *c, int size);
00188 static int nextPerm(int nelem, int nclass, int *cl, int *r, int *q, int *p, int count);
00189 static BigInt ipow(int n, int p);
00190 static BigInt factorial(int n);
00191
00192 // for debugging
00193 #ifdef CHECK
00194 static Bool isIn(int n, int *a, int v);
00195 #endif

```

```

00196
00197 //=====
00198 // class Options
00199 //-----
00200 Options::Options(void)
00201 {
00202     if (nOptDef != GRCC_OPT_Size) {
00203         grcc_fprintf(GRCC_Stderr, "*** Options: inconsistent default values\n");
00204         grcc_fprintf(GRCC_Stderr, "nOptDef=%d, GRCC_OPT_Size=%d\n",
00205             nOptDef, GRCC_OPT_Size);
00206         GRCC_ABORT();
00207     }
00208
00209     model = NULL;
00210     proc = NULL;
00211     sproc = NULL;
00212
00213     outmg = NULL;
00214     endmg = NULL;
00215     outag = NULL;
00216
00217     argmg = NULL;
00218     argemg = NULL;
00219     argag = NULL;
00220
00221     setDefaultValues();
00222
00223     // QGraf options
00224     if (nOptQGDef != GRCC_QGRAFOPT_Size) {
00225         grcc_fprintf(GRCC_Stderr, "*** Options: inconsistent default values\n");
00226         grcc_fprintf(GRCC_Stderr, "nOptQGDef=%d, GRCC_QGRAFOPT_Size=%d\n",
00227             nOptQGDef, GRCC_QGRAFOPT_Size);
00228         GRCC_ABORT();
00229     }
00230     nqgopt = 0;
00231     for (int j = 0; j < nOptQGDef; j++) {
00232         qgref[nqgopt].name = optQGDef[j].name;
00233         qgref[nqgopt].index = j;
00234         qgref[nqgopt].sign = +1;
00235         nqgopt++;
00236         if (strlen(optQGDef[j].cname) > 0) {
00237             qgref[nqgopt].name = optQGDef[j].cname;
00238             qgref[nqgopt].index = j;
00239             qgref[nqgopt].sign = -1;
00240             nqgopt++;
00241         }
00242     }
00243
00244     // default values
00245     for (int j = 0; j < GRCC_QGRAFOPT_Size; j++) {
00246         qgopt[j] = 0;
00247     }
00248
00249     // for output
00250     out = new Output(this);
00251
00252     // subrpocess
00253     sproc = NULL;
00254
00255     // measuring time
00256     time0 = 0.0;
00257     time1 = 0.0;
00258 }
00259
00260 //-----
00261 Options::~Options(void)
00262 {
00263     outmg = NULL;
00264     outag = NULL;
00265
00266     argmg = NULL;
00267     argag = NULL;
00268
00269     delete out;
00270 }
00271
00272 //-----
00273 void Options::setDefaultValues(void)
00274 {
00275     int j;
00276
00277     // default values
00278     for (j = 0; j < GRCC_OPT_Size; j++) {
00279         values[j] = optDef[j].defaultv;
00280     }
00281
00282     values[GRCC_OPT_Step] = GRCC_AGraph;

```

```

00283
00284     // print level
00285     prlevel = 1;
00286 }
00287
00288 //-----
00289 void Options::setOldDefaultValues(void)
00290 {
00291     int j;
00292
00293     // default values
00294     for (j = 0; j < GRCC_OPT_Size; j++) {
00295         values[j] = optDef0[j].defaultv;
00296     }
00297
00298     values[GRCC_OPT_Step] = GRCC_AGraph;
00299
00300     // print level
00301     prlevel = 1;
00302 }
00303
00304 //-----
00305 void Options::setOutMG(OutEGB *omg, void *pt)
00306 {
00307     outmg = omg;
00308     argmg = pt;
00309 }
00310
00311 //-----
00312 void Options::setEndMG(OutEGB *emg, void *pt)
00313 {
00314     endmg = emg;
00315     argemg = pt;
00316 }
00317
00318 //-----
00319 void Options::setOutAG(OutEG *oag, void *pt)
00320 {
00321     outag = oag;
00322     argag = pt;
00323 }
00324
00325 //-----
00326 void Options::setErExit(ErExit *ere, void *erearg)
00327 {
00328     erExit = ere;
00329     erExitArg = erearg;
00330 }
00331
00332 //-----
00333 void Options::printLevel(int l)
00334 {
00335     prlevel = l;
00336 }
00337
00338 //-----
00339 void Options::setValue(int ind, int val)
00340 {
00341     if (0 <= ind && ind < GRCC_OPT_Size) {
00342         values[ind] = val;
00343     } else {
00344         if (prlevel > 0) {
00345             grcc_fprintf(GRCC_Stderr, "*** Options::setValue : invalid index=%d ",
00346                 ind);
00347             grcc_fprintf(GRCC_Stderr, "(val=%d)\n", val);
00348         }
00349         erEnd("Options::setValue : invalid index");
00350     }
00351 }
00352
00353 //-----
00354 int Options::getValue(int ind)
00355 {
00356     if (0 <= ind && ind < GRCC_OPT_Size) {
00357         return values[ind];
00358     } else {
00359         if (prlevel > 0) {
00360             grcc_fprintf(GRCC_Stderr, "*** Options::getValue : invalid index=%d\n", ind);
00361         }
00362         erEnd("Options::getValue : invalid index");
00363     }
00364     return -1;
00365 }
00366
00367 //-----
00368 void Options::setQGrafOpt(int *qg)
00369 {

```

```

00370 // reset options : default is to generate all connected graphs
00371
00372 #ifdef OLDDDEFAULT
00373     values[GRCC_OPT_1PI] = False;
00374     values[GRCC_OPT_NoSelfLoop] = False;
00375     values[GRCC_OPT_NoTadpole] = False;
00376     values[GRCC_OPT_No1PtBlock] = False;
00377     values[GRCC_OPT_No2PtL1PI] = False;
00378     values[GRCC_OPT_NoExtSelf] = False;
00379     values[GRCC_OPT_NoAdj2PtV] = False;
00380     values[GRCC_OPT_Block] = False;
00381     values[GRCC_OPT_SymmInitial] = False;
00382     values[GRCC_OPT_SymmFinal] = False;
00383 #endif
00384
00385     for (int j = 0; j < GRCC_QGRAF_OPT_Size; j++) {
00386         qgopt[j] = qg[j];
00387     }
00388
00389     // {"onepi", "onepr"},
00390     if (qgopt[GRCC_QGRAF_OPT_ONEPI] > 0) {
00391         values[GRCC_OPT_1PI] = True;
00392     } else if (qgopt[GRCC_QGRAF_OPT_ONEPI] < 0) {
00393         values[GRCC_OPT_1PI] = False;
00394     }
00395
00396     // {"onshell", "offshell"}
00397     if (qgopt[GRCC_QGRAF_OPT_ONSHELL] > 0) {
00398         values[GRCC_OPT_NoExtSelf] = 1;
00399     } else if (qgopt[GRCC_QGRAF_OPT_ONSHELL] < 0) {
00400         values[GRCC_OPT_NoExtSelf] = 0;
00401     }
00402
00403     // {"nosigma", "sigma"}
00404
00405     // {"nosnail", "snail"},
00406     if (qgopt[GRCC_QGRAF_OPT_NOSNAIL] > 0) {
00407         values[GRCC_OPT_NoTadpole] = True;
00408         values[GRCC_OPT_No1PtBlock] = True;
00409     } else if (qgopt[GRCC_QGRAF_OPT_NOSNAIL] < 0) {
00410         values[GRCC_OPT_NoTadpole] = False;
00411         values[GRCC_OPT_No1PtBlock] = False;
00412     }
00413
00414     // {"notadpole", "tadpole"}
00415     if (qgopt[GRCC_QGRAF_OPT_NOTADPOLE] > 0) {
00416         values[GRCC_OPT_NoTadpole] = True;
00417     } else if (qgopt[GRCC_QGRAF_OPT_NOTADPOLE] < 0) {
00418         values[GRCC_OPT_NoTadpole] = False;
00419     }
00420
00421     // {"simple", "notsimple"},
00422     if (qgopt[GRCC_QGRAF_OPT_SIMPLE] > 0) {
00423         values[GRCC_OPT_NoSelfLoop] = True;
00424         values[GRCC_OPT_NoMultiEdge] = True;
00425     } else if (qgopt[GRCC_QGRAF_OPT_SIMPLE] < 0) {
00426         values[GRCC_OPT_NoSelfLoop] = False;
00427         values[GRCC_OPT_NoMultiEdge] = False;
00428     }
00429
00430     // {"bipart", "nonbipart"},
00431     // {"cycli", "cyclr"},
00432
00433 #ifdef GRCC_QGRAF_OPT_TOPOL
00434     // {"topol", ""},
00435     if (qgopt[GRCC_QGRAF_OPT_TOPOL] > 0) {
00436         values[GRCC_OPT_SymmInitial] = True;
00437         values[GRCC_OPT_SymmFinal] = True;
00438     }
00439 #endif
00440 }
00441
00442 //-----
00443 void Options::print(void)
00444 {
00445     int j;
00446
00447     grcc_fprintf(GRCC_Stdout, "Options\n");
00448     grcc_fprintf(GRCC_Stdout, "+++ GRCC_OPT_Size=%d, print level=%d: ",
00449         GRCC_OPT_Size, prlevel);
00450     grcc_fprintf(GRCC_Stdout, "symbol = value (default)\n");
00451     for (j=0; j < GRCC_OPT_Size; j++) {
00452         grcc_fprintf(GRCC_Stdout, "    %4d GRCC_OPT_%-15s = %2d (%2d)\n",
00453             j, optDef[j].name, values[j], optDef[j].defaultv);
00454     }
00455     grcc_fprintf(GRCC_Stdout, "    outgrf=%s, outgrp=%s\n", out->outgrf, out->outgrp);
00456     grcc_fprintf(GRCC_Stdout, "    GRCC_QGRAF_OPT_Size=%d:\n", GRCC_QGRAF_OPT_Size);

```

```

00457     for (j=0; j < GRCC_QGRAFOPT_Size; j++) {
00458         grcc_fprintf(GRCC_Stdout, "    %4d %-10s = %2d\n", j, optQGDef[j].name, qgopt[j]);
00459     }
00460 }
00461
00462 //-----
00463 const OptDef *Options::getDef(void)
00464 {
00465     return optDef;
00466 }
00467
00468 //-----
00469 const OptDef *Options::getOldDef(void)
00470 {
00471     return optDef0;
00472 }
00473
00474 //-----
00475 const OptQGDef *Options::getQGDef(void)
00476 {
00477     return optQGDef;
00478 }
00479
00480 //-----
00481 void Options::setOutputF(Bool outf, const char *fname)
00482 {
00483     values[GRCC_OPT_Outgrf] = outf;
00484     out->setOutgrf(fname);
00485 }
00486
00487 //-----
00488 void Options::setOutputP(Bool outp, const char *fname)
00489 {
00490     values[GRCC_OPT_Outgrp] = outp;
00491     out->setOutgrp(fname);
00492 }
00493
00494 //-----
00495 void Options::printModel(void)
00496 {
00497     if (prlevel > 0) {
00498         if (model != NULL) {
00499             model->prModel();
00500         } else {
00501             grcc_fprintf(GRCC_Stdout, "*** model is not defined\n");
00502         }
00503     }
00504 }
00505
00506 //-----
00507 void Options::outModel(void)
00508 {
00509     if (out != NULL && model != NULL) {
00510         out->outModelF();
00511         out->outModelP();
00512     }
00513 }
00514
00515 //-----
00516 void Options::begin(Model *mdl)
00517 {
00518     model = mdl;
00519     if (out != NULL) {
00520         if (values[GRCC_OPT_Outgrf]) {
00521             out->outBeginF(model, values[GRCC_OPT_Outgrf]);
00522             if (model != NULL) {
00523                 out->outModelF();
00524             }
00525         }
00526         if (values[GRCC_OPT_Outgrp]) {
00527             out->outBeginP(model, values[GRCC_OPT_Outgrp]);
00528             if (model != NULL) {
00529                 out->outModelP();
00530             }
00531         }
00532     }
00533 }
00534
00535 //-----
00536 void Options::end(void)
00537 {
00538     if (prlevel > 1) {
00539         grcc_fprintf(GRCC_Stdout, "Optimization: ");
00540 #ifdef SIMPSEL
00541         grcc_fprintf(GRCC_Stdout, "SIMPSEL=1 ");
00542 #else
00543         grcc_fprintf(GRCC_Stdout, "SIMPSEL=0 ");
00544 #endif
00545     }
00546 }

```



```

00544 #endif
00545 #ifdef MINMAXLEG
00546     grcc_fprintf(GRCC_Stdout, "MINMAXLEG=1 ");
00547 #else
00548     grcc_fprintf(GRCC_Stdout, "MINMAXLEG=0 ");
00549 #endif
00550 #ifdef OPTEXTONLY
00551     grcc_fprintf(GRCC_Stdout, "OPTEXTONLY=1 ");
00552 #else
00553     grcc_fprintf(GRCC_Stdout, "OPTEXTONLY=0 ");
00554 #endif
00555     grcc_fprintf(GRCC_Stdout, "\n");
00556 }
00557
00558     if (out != NULL && values[GRCC_OPT_Outgrf]) {
00559         out->outEndF();
00560     }
00561     if (out != NULL && values[GRCC_OPT_Outgrp]) {
00562         out->outEndP();
00563     }
00564     model = NULL;
00565 }
00566 //
00567 //-----
00568 void Options::beginProc(Process *prc)
00569 {
00570     proc = prc;
00571     if (out != NULL && values[GRCC_OPT_Outgrf]) {
00572         out->outProcBeginF(prc);
00573     }
00574     if (out != NULL && values[GRCC_OPT_Outgrp]) {
00575         out->outProcBeginP(prc);
00576     }
00577     if (proc != NULL) {
00578         proc->mgrcount = 0;
00579         proc->agrcount = 0;
00580     }
00581 }
00582
00583 //-----
00584 void Options::endProc(void)
00585 {
00586     int k;
00587
00588     if (proc == NULL) {
00589         if (out != NULL && values[GRCC_OPT_Outgrf]) {
00590             out->outProcEndF();
00591         }
00592         if (out != NULL && values[GRCC_OPT_Outgrp]) {
00593             out->outProcEndP();
00594         }
00595         return;
00596     }
00597     if (prlevel > 0) {
00598         grcc_fprintf(GRCC_Stdout, "\n");
00599         grcc_fprintf(GRCC_Stdout, "+++ Proc %d: ext=%d, loop=%d, ",
00600             proc->id, proc->nExtern, proc->loop);
00601         if (model != NULL) {
00602             grcc_fprintf(GRCC_Stdout, "order=");
00603             prIntArray(model->ncouple, proc->clist, ": ");
00604             model->prParticleArray(proc->ninitl, proc->initlPart, "-->");
00605             model->prParticleArray(proc->nfinal, proc->finalPart, "");
00606         }
00607         grcc_fprintf(GRCC_Stdout, " (%8.2f sec)\n", proc->sec);
00608
00609         grcc_fprintf(GRCC_Stdout, "      Proc      %d: Total M-Graphs=%ld, M-Graphs=",
00610             proc->id, proc->nMGraphs);
00611         proc->wMGraphs.print(" (Conn)\n");
00612
00613         grcc_fprintf(GRCC_Stdout, "      Proc      %d: Total M-Graphs=%ld, M-Graphs=",
00614             proc->id, proc->nMOPI);
00615         proc->wMOPI.print(" (1PI)\n");
00616
00617         grcc_fprintf(GRCC_Stdout, "      Proc      %d: Total A-Graphs=%ld, A-Graphs=",
00618             proc->id, proc->nAGraphs);
00619         proc->wAGraphs.print(" (Conn)\n");
00620
00621         grcc_fprintf(GRCC_Stdout, "      Proc      %d: Total A-Graphs=%ld, A-Graphs=",
00622             proc->id, proc->nAOPI);
00623         proc->wAOPI.print(" (1PI)\n");
00624
00625         grcc_fprintf(GRCC_Stdout, "# { %d,{", proc->ninitl);
00626         for (k = 0; k < proc->ninitl; k++) {
00627             if (k != 0) {
00628                 grcc_fprintf(GRCC_Stdout, ", ");
00629             }
00630         }

```

```

00631         if (proc->model != NULL) {
00632             grcc_fprintf(GRCC_Stdout, "\\%s\\", proc->model->particleName(proc->initlPart[k]));
00633         } else {
00634             grcc_fprintf(GRCC_Stdout, "%d", proc->initlPart[k]);
00635         }
00636     }
00637     grcc_fprintf(GRCC_Stdout, "}, %d,{", proc->nfinal);
00638     for (k = 0; k < proc->nfinal; k++) {
00639         if (k != 0) {
00640             grcc_fprintf(GRCC_Stdout, ", ");
00641         }
00642         if (proc->model != NULL) {
00643             grcc_fprintf(GRCC_Stdout, "\\%s\\", proc->model->particleName(proc->finalPart[k]));
00644         } else {
00645             grcc_fprintf(GRCC_Stdout, "%d", proc->initlPart[k]);
00646         }
00647     }
00648     grcc_fprintf(GRCC_Stdout, "}, ");
00649     if (model != NULL) {
00650         grcc_fprintf(GRCC_Stdout, "{");
00651         for (k = 0; k < model->ncouple; k++) {
00652             if (k != 0) {
00653                 grcc_fprintf(GRCC_Stdout, ", ");
00654             }
00655             grcc_fprintf(GRCC_Stdout, "%d", proc->clist[k]);
00656         }
00657         grcc_fprintf(GRCC_Stdout, "},");
00658     }
00659     grcc_fprintf(GRCC_Stdout, "},%6ldL,%6ldL,%3ldL, -1.0, %4.2f},\\n",
00660                 proc->nAOPI, proc->wAOPI.num, proc->wAOPI.den, proc->sec);
00661 }
00662
00663 if (out != NULL && values[GRCC_OPT_Outgrf]) {
00664     out->outProcEndF();
00665 }
00666 if (out != NULL && values[GRCC_OPT_Outgrp]) {
00667     out->outProcEndP();
00668 }
00669 proc = NULL;
00670 }
00671
00672 //-----
00673 void Options::beginSubProc(SProcess *sprc)
00674 {
00675     sprc = sprc;
00676
00677     // 'beginProc' is called before
00678     if (out != NULL && values[GRCC_OPT_Outgrf]) {
00679         out->outSProcBeginF(sprc);
00680     }
00681     if (out != NULL && values[GRCC_OPT_Outgrp]) {
00682         out->outSProcBeginP(sprc);
00683     }
00684     if (proc != NULL) {
00685         return;
00686     }
00687     if (sprc != NULL) {
00688         sprc->mgrcount = 0;
00689         sprc->agrcount = 0;
00690     }
00691 }
00692
00693 //-----
00694 void Options::endSubProc(void)
00695 {
00696     MGraph *mgraph;
00697
00698     if (sprc == NULL) {
00699         if (out != NULL && values[GRCC_OPT_Outgrf]) {
00700             out->outProcEndF();
00701         }
00702         if (out != NULL && values[GRCC_OPT_Outgrp]) {
00703             out->outProcEndP();
00704         }
00705         return;
00706     }
00707     mgraph = sprc->mgraph;
00708     if (sprc != NULL) {
00709         sprc->resultMGraph(mgraph->cDiag, mgraph->wscon,
00710                           mgraph->c1PI, mgraph->wsopi);
00711     }
00712
00713     if (prlevel > 1) {
00714         grcc_fprintf(GRCC_Stdout, "\\n");
00715         grcc_fprintf(GRCC_Stdout, "+++ Subproc %d: ext=%d, loop=%d, nodes=%d, edges=%d\\n",
00716                     sprc->id, sprc->nExtern, sprc->loop,
00717                     sprc->nNodes, sprc->nEdges);

```

```

00718
00719     grcc_fprintf(GRCC_Stdout, "    Subproc %d: Total M-Graphs=%ld, M-Wsum=",
00720                   sproc->id, sproc->nMGraphs);
00721     sproc->wMGraphs.print(" (Conn)\n");
00722
00723     grcc_fprintf(GRCC_Stdout, "    Subproc %d: Total M-Graphs=%ld, M-Wsum=",
00724                   sproc->id, sproc->nMOPI);
00725     sproc->wMOPI.print(" (1PI)\n");
00726
00727     grcc_fprintf(GRCC_Stdout, "    Subproc %d: Total A-Graphs=%ld, A-Wsum=",
00728                   sproc->id, sproc->nAGraphs);
00729     sproc->wAGraphs.print(" (Conn)\n");
00730
00731     grcc_fprintf(GRCC_Stdout, "    Subproc %d: Total A-Graphs=%ld, A-Wsum=",
00732                   sproc->id, sproc->nAOPI);
00733     sproc->wAOPI.print(" (1PI)\n");
00734 }
00735 if (prlevel > 0) {
00736     grcc_fprintf(GRCC_Stdout, "\n");
00737     grcc_fprintf(GRCC_Stdout, "* Total %ld MGraphs; %ld 1PI", mgraph->cDiag, mgraph->c1PI);
00738     grcc_fprintf(GRCC_Stdout, " wscon = ");
00739     mgraph->wscon.print(" ( ");
00740     mgraph->wsopi.print(" 1PI)\n");
00741
00742 #ifdef MONITOR
00743     grcc_fprintf(GRCC_Stdout, "* refine: %ld\n", mgraph->nCallRefine);
00744     grcc_fprintf(GRCC_Stdout, "* discarded for refinement: %ld\n", mgraph->discardRefine);
00745     grcc_fprintf(GRCC_Stdout, "* discarded for disconnected: %ld\n", mgraph->discardDisc);
00746     grcc_fprintf(GRCC_Stdout, "* discarded for duplication: %ld\n", mgraph->discardIso);
00747 #endif
00748 }
00749
00750 if (proc != NULL) {
00751     return;
00752 }
00753 if (out != NULL && values[GRCC_OPT_Outgrf]) {
00754     out->outProcEndF();
00755 }
00756 if (out != NULL && values[GRCC_OPT_Outgrp]) {
00757     out->outProcEndP();
00758 }
00759 }
00760
00761 //-----
00762 void Options::newMGraph(MGraph *mgr)
00763 {
00764     MGraph *mgraph = mgr;
00765
00766     if (proc != NULL) {
00767         proc->mgrcount++;
00768         mgr->egraph->mId = proc->mgrcount;
00769     } else if (sproc != NULL) {
00770         sproc->mgrcount++;
00771         mgr->egraph->mId = sproc->mgrcount;
00772     }
00773
00774     if (out != NULL
00775         && values[GRCC_OPT_Step] == GRCC_MGraph) {
00776         if (values[GRCC_OPT_Outgrf]) {
00777             out->outEGraphF(mgraph->egraph);
00778         }
00779         if (values[GRCC_OPT_Outgrp]) {
00780             out->outEGraphP(mgraph->egraph);
00781         }
00782     }
00783     if (sproc != NULL) {
00784         sproc->endMGraph(mgr);
00785     }
00786     if (endmg != NULL) {
00787         (*endmg)(mgr->egraph, argemg);
00788     }
00789 }
00790
00791 //-----
00792 void Options::newAGraph(EGraph *egraph)
00793 {
00794     Fraction sf, zr;
00795
00796     if (proc != NULL) {
00797         proc->agrcount++;
00798         egraph->sId = proc->agrcount;
00799     } else if (sproc != NULL) {
00800         sproc->agrcount++;
00801         egraph->sId = sproc->agrcount;
00802     }
00803
00804     if (sproc != NULL) {

```

```

00805         zr = Fraction(0, 1);
00806         if ((egraph->nsym)*(egraph->esym) != 0) {
00807             sf = Fraction(egraph->extperm, egraph->nsym*egraph->esym);
00808         } else {
00809             sf = zr;
00810         }
00811         if (egraph->mgraph->opi) {
00812             sproc->resultAGraph(1, sf, 1, sf);
00813         } else {
00814             sproc->resultAGraph(1, sf, 0, zr);
00815         }
00816     }
00817
00818     if (out != NULL && values[GRCC_OPT_Outgrf]) {
00819         out->outEGraphF(egraph);
00820     }
00821     if (out != NULL && values[GRCC_OPT_Outgrp]) {
00822         out->outEGraphP(egraph);
00823     }
00824     if (outag != NULL) {
00825         (*outag)(egraph, argag);
00826     }
00827
00828     sproc->endAGraph(egraph);
00829 }
00830
00831 //=====
00832 Output::Output(Options *optn)
00833 {
00834     opt      = optn;
00835     model    = NULL;
00836     proc     = NULL;
00837     sproc    = NULL;
00838     outgrfp  = NULL;
00839     outgrpp  = NULL;
00840     procId   = -1;
00841     outgrf   = NULL;
00842     outgrp   = NULL;
00843     outproc  = False;
00844 }
00845
00846 //-----
00847 Output::~Output(void)
00848 {
00849     if (outgrfp != NULL) {
00850         if (prlevel > 0) {
00851             grcc_fprintf(GRCC_Stderr, "*** file has not been closed : \"%s\"\\n",
00852                 outgrf);
00853         }
00854         fclose(outgrfp);
00855         outgrfp = NULL;
00856         erEnd("file has not been closed");
00857     }
00858     if (outgrf != NULL) {
00859         free(outgrf);
00860         outgrf = NULL;
00861     }
00862     if (outgrpp != NULL) {
00863         if (prlevel > 0) {
00864             grcc_fprintf(GRCC_Stderr, "*** file has not been closed : \"%s\"\\n",
00865                 outgrp);
00866         }
00867         fclose(outgrpp);
00868         outgrpp = NULL;
00869         erEnd("file has not been closed");
00870     }
00871     if (outgrp != NULL) {
00872         free(outgrp);
00873         outgrp = NULL;
00874     }
00875 }
00876
00877 //-----
00878 void Output::setOutgrf(const char *fname)
00879 {
00880     if (outgrf != NULL && strcmp(outgrf, fname) == 0) {
00881         return;
00882     }
00883     if (outgrf != NULL) {
00884         // delete outgrf;
00885         free(outgrf);
00886     }
00887     if (fname == NULL || strlen(fname) < 1) {
00888         outgrf = NULL;
00889     } else {
00890

```

```

00892         outgrp = strdup(fname);
00893     }
00894 }
00895
00896 //-----
00897 void Output::setOutgrp(const char *fname)
00898 {
00899     if (outgrp != NULL && strcmp(outgrp, fname) == 0) {
00900         return;
00901     }
00902
00903     if (outgrp != NULL) {
00904         free(outgrp);
00905     }
00906     if (fname == NULL || strlen(fname) < 1) {
00907         outgrp = NULL;
00908     } else {
00909         outgrp = strdup(fname);
00910     }
00911 }
00912
00913 //-----
00914 Bool Output::outBeginF(Model *mdl, Bool pr)
00915 {
00916     time_t tp;
00917     struct tm *tm;
00918     char sdate[MAXSTRLEN];
00919
00920     model = mdl;
00921
00922     if (outgrp == NULL || strlen(outgrp) < 1) {
00923         outgrp = NULL;
00924         return True;
00925     } else if (outgrp != NULL) {
00926         if (prlevel > 0) {
00927             grcc_fprintf(GRCC_Stderr, "*** outBegin: \"%s\" is already opened\n",
00928                 outgrp);
00929         }
00930         erEnd("outBegin: file is already opened\n");
00931     }
00932
00933     if (!pr) {
00934         return True;
00935     }
00936     if ((outgrfp = fopen(outgrp, "w")) == NULL) {
00937         if (prlevel > 0) {
00938             grcc_fprintf(GRCC_Stderr, "*** outBegin: cannot open %s\n", outgrp);
00939         }
00940         return False;
00941     }
00942     tp = time(NULL);
00943     tm = localtime(&tp);
00944     strftime(sdate, MAXSTRLEN, "%Y/%m/%d %H:%M:%S", tm);
00945
00946     fprintf(outgrfp, "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
00947     fprintf(outgrfp, "% Generated by grcc at \"%s\"\n", sdate);
00948     fprintf(outgrfp, "Version={2,2,0,0};\n");
00949     if (model != NULL) {
00950         fprintf(outgrfp, "Model=\"%s.mdl\";\n", model->name);
00951     } else {
00952         fprintf(outgrfp, "Model=\"%s.mdl\";\n");
00953     }
00954
00955     return True;
00956 }
00957
00958 //-----
00959 Bool Output::outBeginP(Model *mdl, Bool pr)
00960 {
00961     time_t tp;
00962     struct tm *tm;
00963     char sdate[MAXSTRLEN];
00964
00965     model = mdl;
00966
00967     if (outgrp == NULL || strlen(outgrp) < 1) {
00968         outgrpp = NULL;
00969         return True;
00970     } else if (outgrpp != NULL) {
00971         if (prlevel > 0) {
00972             grcc_fprintf(GRCC_Stderr, "*** outBegin: \"%s\" is already opened\n",
00973                 outgrpp);
00974         }
00975         erEnd("outBegin: file is already opened\n");
00976     }
00977 }
00978

```

```

00979     if (!pr) {
00980         return True;
00981     }
00982     if ((outgrpp = fopen(outgrp, "w")) == NULL) {
00983         if (prlevel > 0) {
00984             grcc_fprintf(GRCC_Stderr, "*** outBegin: cannot open %s\n", outgrp);
00985         }
00986         return False;
00987     }
00988     tp = time(NULL);
00989     tm = localtime(&tp);
00990     strftime(sdate, MAXSTRLEN, "%Y/%m/%d %H:%M:%S", tm);
00991
00992     fprintf(outgrpp, "#####\n");
00993     fprintf(outgrpp, "# Generated by 'grcc' at \"%s\"\n", sdate);
00994
00995     return True;
00996 }
00997
00998 //-----
00999 void Output::outEndF(void)
01000 {
01001     if (outgrfp == NULL) {
01002         return;
01003     }
01004     fprintf(outgrfp, "#####\n");
01005     fprintf(outgrfp, "End=1;\n");
01006     fprintf(outgrfp, "#####\n");
01007     fclose(outgrfp);
01008     outgrfp = NULL;
01009
01010     free(outgrf);
01011     outgrf = NULL;
01012 }
01013
01014 //-----
01015 void Output::outEndP(void)
01016 {
01017     if (outgrpp == NULL) {
01018         return;
01019     }
01020     fprintf(outgrpp, "#####\n");
01021     fprintf(outgrpp, "# End\n");
01022     fprintf(outgrpp, "#####\n");
01023     fclose(outgrpp);
01024     outgrpp = NULL;
01025
01026     free(outgrp);
01027     outgrp = NULL;
01028 }
01029
01030 //-----
01031 void Output::outProcBeginF(Process *prc)
01032 {
01033     int k, ex;
01034
01035     if (prc == NULL) {
01036         return;
01037     }
01038
01039     proc = prc;
01040     sproc = NULL;
01041     procId = proc->id;
01042
01043     if (outgrfp == NULL) {
01044         return;
01045     }
01046     if (outproc) {
01047         return;
01048     }
01049     outproc = True;
01050
01051     fprintf(outgrfp, "#####\n");
01052     fprintf(outgrfp, "Process=%d;\n", proc->id);
01053     fprintf(outgrfp, "External=%d;\n", proc->ninitl+proc->nfinal);
01054     for (k = 0, ex = 0; k < proc->ninitl; k++, ex++) {
01055         fprintf(outgrfp, "%4d= initial %s;\n", ex,
01056             proc->model->particleName(proc->initlPart[k]));
01057     }
01058     for (k = 0; k < proc->nfinal; k++, ex++) {
01059         fprintf(outgrfp, "%4d= final %s;\n", ex,
01060             proc->model->particleName(proc->finalPart[k]));
01061     }
01062     fprintf(outgrfp, "End;\n");
01063     for (k = 0; k < proc->model->ncouple; k++) {
01064         fprintf(outgrfp, "%s=%d; ", proc->model->cnlist[k], proc->clist[k]);
01065     }

```

```

01066     fprintf(outgrfp, "Loop=%d;\n", proc->loop);
01067     fprintf(outgrfp, "OPI=%s;\n", (opt->values[GRCC_OPT_1PI] > 0? "Yes" : "No"));
01068     fprintf(outgrfp, "Assign=%s;\n", (opt->values[GRCC_OPT_Step]==GRCC_AGraph ? "Yes" : "No"));
01069     fprintf(outgrfp, "% Options : dummy values\n");
01070     fprintf(outgrfp, "Expand=Yes; ExpMin=0; ExpMax=-1;\n");
01071     fprintf(outgrfp, "Block=No; Tadpole=Yes; Extself=Yes;\n");
01072     fprintf(outgrfp, "Selfe=Yes; Countert=No; AnyCT=No;Undefp=No;\n");
01073 }
01074
01075 //-----
01076 void Output::outProcBeginP(Process *prc)
01077 {
01078     if (prc == NULL) {
01079         return;
01080     }
01081     proc = prc;
01082     sproc = NULL;
01083     procId = proc->id;
01084
01085     if (outgrpp == NULL) {
01086         return;
01087     }
01088     if (outproc) {
01089         return;
01090     }
01091     outproc = True;
01092
01093     fprintf(outgrpp, "# Process=%d;\n", proc->id);
01094 }
01095
01096 //-----
01097 void Output::outProcBegin0(int next, int couple, int loop)
01098 {
01099     int k, ex;
01100
01101     procId = 0;
01102
01103     if (outgrfp == NULL) {
01104         return;
01105     }
01106     if (outproc) {
01107         return;
01108     }
01109     outproc = True;
01110
01111     fprintf(outgrfp, "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
01112     fprintf(outgrfp, "Process=%d;\n", procId);
01113     fprintf(outgrfp, "External=%d;\n", next);
01114     int ninit = (next + 1) / 2;
01115     for (k = 0, ex = 0; k < ninit; k++, ex++) {
01116         fprintf(outgrfp, "%4d= initial undef;\n", ex);
01117     }
01118     for (k = 0; k < next - ninit; k++, ex++) {
01119         fprintf(outgrfp, "%4d= final   undef;\n", ex);
01120     }
01121     fprintf(outgrfp, "End;\n");
01122     fprintf(outgrfp, "GRCC_PHI=%d; ", couple);
01123     // fprintf(outgrfp, "PHI=%d; ", couple);
01124     fprintf(outgrfp, "Loop=%d;\n", loop);
01125     fprintf(outgrfp, "OPI=%s;\n", (opt->values[GRCC_OPT_1PI] > 0? "Yes" : "No"));
01126     fprintf(outgrfp, "Assign=No;\n");
01127     fprintf(outgrfp, "% Options : dummy values\n");
01128     fprintf(outgrfp, "Expand=Yes; ExpMin=0; ExpMax=-1;\n");
01129     fprintf(outgrfp, "Block=No; Tadpole=Yes; Extself=Yes;\n");
01130     fprintf(outgrfp, "Selfe=Yes; Countert=No; AnyCT=No;Undefp=No;\n");
01131 }
01132
01133 //-----
01134 void Output::outSProcBeginF(SProcess *sprc)
01135 {
01136     int j, k, ex;
01137     PNodeClass *pnc;
01138
01139     if (sprc == NULL) {
01140         return;
01141     }
01142
01143     sproc = sprc;
01144     procId = sproc->id;
01145     pnc = sproc->pnclass;
01146
01147     if (outgrfp == NULL) {
01148         return;
01149     }
01150
01151     if (outproc) {
01152         return;

```

```

01153     }
01154     outproc = True;
01155     fprintf(outgrfp, "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
01156     fprintf(outgrfp, "Process=%d;\n", sproc->id);
01157     fprintf(outgrfp, "External=%d;\n", sproc->nExtern);
01158
01159     ex = 0;
01160     for (j = 0; j < pnc->nclass; j++) {
01161         if (pnc->type[j] == GRCC_AT_Initial) {
01162             for (k = 0; k < pnc->count[j]; k++) {
01163                 fprintf(outgrfp, "%4d= initial %s;\n", ex,
01164                     model->particleName(pnc->particle[j]));
01165                 ex++;
01166             }
01167         } else if (pnc->type[j] == GRCC_AT_Final) {
01168             for (k = 0; k < pnc->count[j]; k++) {
01169                 fprintf(outgrfp, "%4d= final %s;\n", ex,
01170                     model->particleName(-pnc->particle[j]));
01171                 ex++;
01172             }
01173         }
01174     }
01175
01176     fprintf(outgrfp, "End;\n");
01177     for (k = 0; k < sproc->model->ncouple; k++) {
01178         fprintf(outgrfp, "%s=%d; ", sproc->model->cnlist[k], sproc->clist[k]);
01179     }
01180     fprintf(outgrfp, "Loop=%d;\n", sproc->loop);
01181     fprintf(outgrfp, "OPI=%s;\n", (opt->values[GRCC_OPT_lPI] > 0 ? "Yes" : "No"));
01182     fprintf(outgrfp, "Assign=%s;\n", (opt->values[GRCC_OPT_Step]==GRCC_AGraph ? "Yes" : "No"));
01183     fprintf(outgrfp, "% Options : dummy values\n");
01184     fprintf(outgrfp, "Expand=Yes; ExpMin=0; ExpMax=-1;\n");
01185     fprintf(outgrfp, "Block=No; Tadpole=Yes; Extself=Yes;\n");
01186     fprintf(outgrfp, "Selfe=Yes; Countert=No; AnyCT=No;Undefp=No;\n");
01187 }
01188
01189 //-----
01190 void Output::outSProcBeginP(SProcess *sproc)
01191 {
01192     if (sproc == NULL) {
01193         return;
01194     }
01195
01196     sproc = sproc;
01197     procId = sproc->id;
01198
01199     if (outgrpp == NULL) {
01200         return;
01201     }
01202     if (outproc) {
01203         return;
01204     }
01205     outproc = True;
01206     fprintf(outgrpp, "# SProcess=%d;\n", sproc->id);
01207 }
01208
01209 //-----
01210 void Output::outProcEndF(void)
01211 {
01212     if (outgrfp == NULL) {
01213         return;
01214     }
01215     if (proc != NULL) {
01216         fprintf(outgrfp, "%-----\n");
01217         fprintf(outgrfp, "Pend=%d;\n", proc->id);
01218     } else if (sproc != NULL) {
01219         fprintf(outgrfp, "%-----\n");
01220         fprintf(outgrfp, "Pend=%d;\n", sproc->id);
01221     } else {
01222         fprintf(outgrfp, "%-----\n");
01223         fprintf(outgrfp, "Pend=1;\n");
01224     }
01225     fflush(outgrfp);
01226 }
01227
01228 //-----
01229 void Output::outProcEndP(void)
01230 {
01231     if (outgrpp == NULL) {
01232         return;
01233     }
01234     fflush(outgrpp);
01235 }
01236
01237 //-----
01238 void Output::outEGraphF(EGraph *egraph)
01239 {

```



```

01240     int nd, lg, ptcl, ed, intr, j, k, loop;
01241     Model *mdl = egraph->model;
01242     Bool popt;
01243     EFLine *fl;
01244
01245     if (outgrfp == NULL) {
01246         return;
01247     }
01248     fprintf(outgrfp, "%s-----\n");
01249     if (egraph->assigned) {
01250         fprintf(outgrfp, "Graph=%ld;\n", egraph->sId);
01251         fprintf(outgrfp, "%s AGraph=%ld;\n", egraph->aId);
01252     } else {
01253         fprintf(outgrfp, "Graph=%ld;\n", egraph->mId);
01254     }
01255     fprintf(outgrfp, "Gtype=%ld;\n", egraph->mId);
01256     if (egraph->assigned) {
01257         fprintf(outgrfp, "Sfactor=%ld;\n",
01258             egraph->nsym * egraph->esym * egraph->fsign);
01259     } else {
01260         fprintf(outgrfp, "Sfactor=%ld;\n", egraph->nsym * egraph->esym);
01261     }
01262     fprintf(outgrfp, "%s ExtPerm=%ld; NSym=%ld; ESym=%ld; NSym1=%ld; "
01263         " Multp=%ld;\n",
01264         egraph->extperm, egraph->nsym, egraph->esym, egraph->nsym1,
01265         egraph->multp);
01266
01267     fprintf(outgrfp, "Vertex=%d;\n", egraph->nNodes - egraph->nExtern);
01268
01269     for (nd = 0; nd < egraph->nNodes; nd++) {
01270         fprintf(outgrfp, "%4d", nd);
01271         if (mdl != NULL && egraph->assigned && !egraph->isExternal(nd)) {
01272             intr = egraph->nodes[nd]->intrct;
01273             loop = mdl->interacts[intr]->loop;
01274             popt = False;
01275
01276             // not tree vertex
01277             if (loop > 0) {
01278                 fprintf(outgrfp, "[loop=%d", loop);
01279                 popt = True;
01280             }
01281
01282             // multiple coupling constants
01283             if (mdl->ncouple > 1) {
01284                 if (popt) {
01285                     fprintf(outgrfp, ";order={");
01286                 } else {
01287                     fprintf(outgrfp, "[order={");
01288                     popt = True;
01289                 }
01290                 for (j = 0; j < mdl->interacts[intr]->nclist; j++) {
01291                     if (j != 0) {
01292                         fprintf(outgrfp, ",");
01293                     }
01294                     fprintf(outgrfp, "%d", mdl->interacts[intr]->clist[j]);
01295                 }
01296                 fprintf(outgrfp, "}");
01297             }
01298             if (popt) {
01299                 fprintf(outgrfp, "]");
01300             }
01301         } else if (!egraph->isExternal(nd)) {
01302             loop = egraph->nodes[nd]->extloop;
01303             // not tree vertex
01304             if (loop > 0) {
01305                 fprintf(outgrfp, "[loop=%d", loop);
01306             }
01307         }
01308         fprintf(outgrfp, "={");
01309
01310         // list of legs
01311         for (lg = 0; lg < egraph->nodes[nd]->deg; lg++) {
01312             if (lg != 0) {
01313                 fprintf(outgrfp, ", ");
01314             }
01315             ed = V2Iedge(egraph->nodes[nd]->edges[lg]);
01316             // fprintf(outgrfp, "%4d", egraph->nodes[nd]->edges[lg]);
01317             fprintf(outgrfp, "%4d", abs(egraph->nodes[nd]->edges[lg]));
01318             ptcl = egraph->edges[ed]->ptcl;
01319             if (mdl != NULL && ptcl != 0 && egraph->assigned) {
01320                 if (egraph->nodes[nd]->edges[lg] < 0) {
01321                     ptcl = mdl->normalParticle(-ptcl);
01322                 } else {
01323                     ptcl = mdl->normalParticle(ptcl);
01324                 }
01325                 fprintf(outgrfp, "[%s]", mdl->particleName(ptcl));
01326             } else {

```

```

01327         fprintf(outgrfp, "[undef]");
01328     }
01329 }
01330 fprintf(outgrfp, ";\n");
01331 }
01332 // print Fermion line as comment line
01333 if (egraph->nFlines >= 0) {
01334     fprintf(outgrfp, "%s FLines=%d; FSign=%d; sId=%ld;\n",
01335         egraph->nFlines, egraph->fsign, egraph->sId);
01336 }
01337 for (j = 0; j < egraph->nFlines; j++) {
01338     fl = egraph->flines[j];
01339     fprintf(outgrfp, "%s %4d", j);
01340     if (fl->ftype == FL_Open) {
01341         fprintf(outgrfp, "[FOpen]=[");
01342     } else if (fl->ftype == FL_Closed) {
01343         fprintf(outgrfp, "[FLoop]=[");
01344     } else {
01345         fprintf(outgrfp, " ?%d", fl->ftype);
01346     }
01347     for (k = 0; k < fl->nlist; k++) {
01348         if (k != 0) {
01349             fprintf(outgrfp, ", ");
01350         }
01351         fprintf(outgrfp, "%d", fl->elist[k]);
01352     }
01353     fprintf(outgrfp, "];\n");
01354 }
01355 fprintf(outgrfp, "Vend;\n");
01356 fprintf(outgrfp, "Gend;\n");
01357 }
01358
01359 //-----
01360 #ifdef NEWOUTPY
01361 void Output::outEGraphP(EGraph *eg)
01362 {
01363     if (outgrpp == NULL) {
01364         return;
01365     }
01366
01367     if (eg->assigned) {
01368         gcc_fprintf(GRCC_Stdout, "outEGraphP:egraph:egraph[%ld]\n", eg->mId-1);
01369         eg->printPy(outgrpp, eg->mId);
01370         return;
01371     } else {
01372         gcc_fprintf(GRCC_Stdout, "outEGraphP:mgraph:egraph[%ld]\n", eg->mId-1);
01373         eg->mgraph->printPy(outgrpp, eg->mId);
01374         return;
01375     }
01376 }
01377 #else
01378 void Output::outEGraphP(EGraph *eg)
01379 {
01380     int j, nd, lg, ed, k;
01381     ENode *node;
01382     static char undef[] = "Undef";
01383     char *s;
01384     EFLine *fl;
01385
01386     if (outgrpp == NULL) {
01387         return;
01388     }
01389
01390     if (eg->assigned) {
01391         fprintf(outgrpp, "egraph[%ld] = {\n", eg->sId-1);
01392     } else {
01393         fprintf(outgrpp, "egraph[%ld] = {\n", eg->mId-1);
01394     }
01395
01396     // process
01397     if (proc != NULL) {
01398         fprintf(outgrpp, "  \"Process\": {\n");
01399         proc->outProcP(outgrpp);
01400         fprintf(outgrpp, "  },\n");
01401     }
01402
01403     // graph
01404     fprintf(outgrpp, "  \"GId\": [%ld, %ld, %ld],\n",
01405         eg->mId, eg->aId, eg->sId);
01406     fprintf(outgrpp, "  \"GParam\": {\"Assigned\":%d, \"
01407         \"NNodes\":%d, \"NEEdges\":%d, \"NFlines\":%d},\n",
01408         eg->assigned, eg->nNodes, eg->nEdges, eg->nFlines);
01409     fprintf(outgrpp, "  \"Sym\": {\"FSign\":%d, \"NSym\":%ld, \"
01410         \"ESym\":%ld, \"NSym1\":%ld, \"Multp\":%ld},\n",
01411         eg->fsign, eg->nsym, eg->esym, eg->nsym1, eg->multp);
01412
01413     // nodes

```

```

01414     fprintf(outgrpp, "  \Nodes\": [\n");
01415     for (nd = 0; nd < eg->nNodes; nd++) {
01416         node = eg->nodes[nd];
01417         if (model==NULL) {
01418             s = undef;
01419         } else if (!eg->assigned) {
01420             if (eg->isExternal(nd)) {
01421                 ed = Abs(eg->nodes[nd]->edges[0])-1;
01422                 s = model->particleName(eg->edges[ed]->ptcl);
01423             } else {
01424                 s = undef;
01425             }
01426         } else if (eg->isExternal(nd)) {
01427             s = model->particleName(node->intrct);
01428         } else {
01429             s = model->interacts[node->intrct]->name;
01430         }
01431         fprintf(outgrpp, "      [%d, \"%s\", %d, [",
01432             nd, s, node->extloop);
01433         for (lg = 0; lg < eg->nodes[nd]->deg; lg++) {
01434             if (lg != 0) {
01435                 fprintf(outgrpp, ", ");
01436             }
01437             fprintf(outgrpp, "%d", eg->nodes[nd]->edges[lg]);
01438         }
01439         fprintf(outgrpp, "]],\n");
01440     }
01441     fprintf(outgrpp, " ],\n");
01442
01443     // edges
01444     fprintf(outgrpp, "  \Edges\": [\n");
01445     for (ed = 0; ed < eg->nEdges; ed++) {
01446         if (model != NULL) {
01447             s = model->particleName(eg->edges[ed]->ptcl);
01448             fprintf(outgrpp, "      [%d, \"%s\", [%d, %d]],\n",
01449                 ed+1, s, eg->edges[ed]->nodes[0],
01450                 eg->edges[ed]->nodes[1]);
01451         } else {
01452             fprintf(outgrpp, "      [%d, \"Undef\", [%d, %d]],\n",
01453                 ed+1, eg->edges[ed]->nodes[0],
01454                 eg->edges[ed]->nodes[1]);
01455         }
01456     }
01457     fprintf(outgrpp, " ],\n");
01458
01459     if (eg->nFlines >= 0) {
01460         // print Fermion line as comment line
01461         fprintf(outgrpp, "  \FLines\": [\n");
01462         for (j = 0; j < eg->nFlines; j++) {
01463             fl = eg->flines[j];
01464             fprintf(outgrpp, "      [%d, ", j);
01465             if (fl->ftype == FL_Open) {
01466                 fprintf(outgrpp, "\"FOpen\", ");
01467             } else if (fl->ftype == FL_Closed) {
01468                 fprintf(outgrpp, "\"FLoop\", ");
01469             } else {
01470                 fprintf(outgrpp, "\"Undef%d\", ", fl->ftype);
01471             }
01472             fprintf(outgrpp, "[");
01473             for (k = 0; k < fl->nlist; k++) {
01474                 if (k != 0) {
01475                     fprintf(outgrpp, ", ");
01476                 }
01477                 fprintf(outgrpp, "%d", fl->elist[k]);
01478             }
01479             fprintf(outgrpp, "]],\n");
01480         }
01481         fprintf(outgrpp, " ]\n");
01482     }
01483
01484     // end of a graph
01485     fprintf(outgrpp, "];\n");
01486 }
01487 #endif
01488
01489 //-----
01490 void Output::outModelF(void)
01491 {
01492     char      *mdlfn;
01493     FILE      *mdlfp;
01494     Particle  *pt;
01495     Interaction *vt;
01496     int       j, k, l, ln;
01497     const char *ptn;
01498
01499     if (model == NULL) {
01500

```

```

01501         if (prlevel > 0) {
01502             gcc_fprintf(GRCC_Stderr, "*** Output::outModel : model is not defined\n");
01503         }
01504         return;
01505     }
01506     ln = strlen(model->name)+5;
01507     mdlfn = new char[ln];
01508     sprintf(mdlfn, ln, "%s.mdl", model->name);
01509
01510     if ((mdlfp = fopen(mdlfn, "w")) == NULL) {
01511         if (prlevel > 0) {
01512             gcc_fprintf(GRCC_Stderr, "*** Output::outModel : cannot open \"%s\"\n",
01513                 mdlfn);
01514         }
01515         return;
01516     }
01517
01518     for (l = 0; l < 60; l++) { putc('% ', mdlfp); } putc('\n', mdlfp);
01519     fprintf(mdlfp, "% File \"%s\"\n", mdlfn);
01520     fprintf(mdlfp, "% Generated by graph generator\n");
01521     fprintf(mdlfp, " Version={2,2,0};\n");
01522
01523     // coupling constants
01524     fprintf(mdlfp, " Order={");
01525     for (j = 0; j < model->ncouple; j++) {
01526         if (j != 0) {
01527             fprintf(mdlfp, ", ");
01528         }
01529         fprintf(mdlfp, "%s", model->cnlist[j]);
01530     }
01531     fprintf(mdlfp, "};\n");
01532
01533     for (l = 0; l < 60; l++) { putc('% ', mdlfp); } putc('\n', mdlfp);
01534     fprintf(mdlfp, "% particles\n");
01535     for (l = 0; l < 60; l++) { putc('% ', mdlfp); } putc('\n', mdlfp);
01536     for (j = 1; j < model->nParticles; j++) {
01537         pt = model->particles[j];
01538         fprintf(mdlfp, " Particle=%s; Antiparticle=%s;\n",
01539             pt->name, pt->aname);
01540         ptn = pt->typeGName();
01541         fprintf(mdlfp, " PType=%s; Charge=0; Color=1; Mass=0; Width=0;\n", ptn);
01542         fprintf(mdlfp, " PCode=%d; Massless; \n", pt->pcode);
01543         fprintf(mdlfp, " Pend;\n");
01544         fprintf(mdlfp, "%s\n");
01545     }
01546     for (l = 0; l < 60; l++) { putc('% ', mdlfp); } putc('\n', mdlfp);
01547     fprintf(mdlfp, "% interactions\n");
01548     for (l = 0; l < 60; l++) { putc('% ', mdlfp); } putc('\n', mdlfp);
01549     for (j = 0; j < model->nInteracts; j++) {
01550         vt = model->interacts[j];
01551         fprintf(mdlfp, " Vertex={");
01552         for (k = 0; k < vt->nplist; k++) {
01553             if (k != 0) {
01554                 fprintf(mdlfp, ", ");
01555             }
01556             fprintf(mdlfp, "%s", model->particleName(vt->plist[k]));
01557         }
01558         fprintf(mdlfp, "}; ");
01559         for (k = 0; k < model->ncouple; k++) {
01560             fprintf(mdlfp, "%s=%d; ", model->cnlist[k], vt->clist[k]);
01561         }
01562         fprintf(mdlfp, "FName=%s; Vend;\n", vt->name);
01563     }
01564     for (l = 0; l < 60; l++) { putc('% ', mdlfp); } putc('\n', mdlfp);
01565     fprintf(mdlfp, " Mend;\n");
01566     for (l = 0; l < 60; l++) { putc('% ', mdlfp); } putc('\n', mdlfp);
01567     fclose(mdlfp);
01568     delete[] mdlfn;
01569 }
01570
01571 //-----
01572 void Output::outModelP(void)
01573 {
01574     char      *mdlfn;
01575     FILE      *mdlfp;
01576     Particle   *pt;
01577     Interaction *vt;
01578     int        j, k, l, ln;
01579     const char *ptn;
01580
01581     if (model == NULL) {
01582         if (prlevel > 0) {
01583             gcc_fprintf(GRCC_Stderr, "*** Output::outModel : ");
01584             gcc_fprintf(GRCC_Stderr, "model is not defined\n");
01585         }
01586         return;
01587     }

```

```

01588     ln = strlen(model->name)+5;
01589     mdlfn = new char[ln];
01590     sprintf(mdlfn, ln, "%s.mdp", model->name);
01591
01592     if ((mdlfp = fopen(mdlfn, "w")) == NULL) {
01593         if (prlevel > 0) {
01594             grcc_fprintf(GRCC_Stderr, "*** Output::outModel : cannot open \"%s\\\"\\n\",
01595                 mdlfn);
01596         }
01597         return;
01598     }
01599
01600     for (l = 0; l < 60; l++) { putc('#', mdlfp); } putc('\n', mdlfp);
01601     fprintf(mdlfp, "# File \"%s\\\"\\n\", mdlfn);
01602     fprintf(mdlfp, "# Generated by graph generater grcc\\n");
01603
01604     // coupling constants
01605     fprintf(mdlfp, "Model={\\n");
01606     fprintf(mdlfp, "  \\\"Model\\\": {");
01607     fprintf(mdlfp, "\\\"%s\\\", %d, {",
01608         model->name, model->ncouple);
01609     for (j = 0; j < model->ncouple; j++) {
01610         if (j != 0) {
01611             fprintf(mdlfp, ", ");
01612         }
01613         fprintf(mdlfp, "\\\"%s\\\"\\n\", model->cnlist[j]);
01614     }
01615     fprintf(mdlfp, "], ");
01616     if (model->defpart == GRCC_DEFBYCODE) {
01617         fprintf(mdlfp, "\\\"ByCode\\\"");
01618     } else {
01619         fprintf(mdlfp, "\\\"ByName\\\"");
01620     }
01621     fprintf(mdlfp, "],\\n");
01622
01623     fprintf(mdlfp, "  \\\"Particles\\\": {\\n");
01624     for (j = 1; j < model->nParticles; j++) {
01625         pt = model->particles[j];
01626         ptn = pt->typeName();
01627         fprintf(mdlfp, "    [\\\"%s\\\", %d, \\\"%s\\\", %d, \\\"%s\\\", %d],\\n",
01628             pt->name, pt->pcode, pt->aname, pt->acode,
01629             ptn, pt->extonly);
01630     }
01631     fprintf(mdlfp, "  ],\\n");
01632     fprintf(mdlfp, "  \\\"Interactions\\\": {\\n");
01633     for (j = 0; j < model->nInteracts; j++) {
01634         vt = model->interacts[j];
01635         fprintf(mdlfp, "    [\\\"%s\\\", %d, %d, {", vt->name, vt->id, vt->nplist);
01636         for (k = 0; k < vt->nplist; k++) {
01637             if (k != 0) {
01638                 fprintf(mdlfp, ", ");
01639             }
01640             if (model->defpart == GRCC_DEFBYCODE) {
01641                 fprintf(mdlfp, "%d", vt->plist[k]);
01642             } else {
01643                 fprintf(mdlfp, "\\\"%s\\\"\\n\", model->particleName(vt->plist[k]));
01644             }
01645         }
01646         fprintf(mdlfp, "], {");
01647         for (k = 0; k < model->ncouple; k++) {
01648             if (k != 0) {
01649                 fprintf(mdlfp, ", ");
01650             }
01651             fprintf(mdlfp, "%d", vt->clist[k]);
01652         }
01653         fprintf(mdlfp, "}],\\n");
01654     }
01655     fprintf(mdlfp, "  ],\\n");
01656     fprintf(mdlfp, "};\\n");
01657     fclose(mdlfp);
01658     delete[] mdlfn;
01659 }
01660
01661 //*****
01662 // model.cc
01663
01664 //=====
01665
01666 static const char *ptypenames[] = GRCC_PT_Table;
01667 static const char *pgtypenames[] = GRCC_PT_GTable;
01668
01669 //=====
01670 // class Particle
01671 //-----
01672 Particle::Particle(Model *modl, int pid, PInput *pinp)
01673 {
01674     static char buff[100];

```

```

01675     static int  nbuff=100;
01676     int j;
01677
01678     mdl      = mdl;
01679     id       = pid;
01680     if (pinp->name == NULL || strlen(pinp->name) < 1) {
01681         if (pinp->pcode != 0) {
01682             snprintf(buff, nbuff, "pa%d", pinp->pcode);
01683             name = strdup(buff);
01684         } else {
01685             snprintf(buff, nbuff, "pi%d", pid);
01686             name = strdup(buff);
01687         }
01688     } else {
01689         name = strdup(pinp->name);
01690     }
01691     if (pinp->aname == NULL) {
01692         if (pinp->acode != 0) {
01693             snprintf(buff, nbuff, "ap%d", Abs(pinp->acode));
01694             aname = strdup(buff);
01695         } else {
01696             snprintf(buff, nbuff, "ai%d", pid);
01697             aname = strdup(buff);
01698         }
01699     } else {
01700         aname = strdup(pinp->aname);
01701     }
01702     pcode = pinp->pcode;
01703     acode = pinp->acode;
01704     if (Abs(mdl->defpart) == GRCC_DEFBYCODE) {
01705         neutral = (pcode == acode);
01706     } else if (Abs(mdl->defpart) == GRCC_DEFBYNAME) {
01707         neutral = ((strcmp(name, aname)==0) ? True : False);
01708     }
01709
01710     // convert ptype string to its code.
01711     if (Abs(mdl->defpart) == GRCC_DEFBYCODE) {
01712         if (pinp->ptypec < GRCC_PT_Undef || pinp->ptypec > GRCC_PT_Size) {
01713             if (prlevel > 0) {
01714                 grcc_fprintf(GRCC_Stderr, "*** particle type is not defined: %d\n",
01715                     pinp->ptypec);
01716             }
01717             erEnd("particle type is not defined (illegal code)");
01718         }
01719         ptype = pinp->ptypec;
01720     } else if (Abs(mdl->defpart) == GRCC_DEFBYNAME) {
01721         if (pinp->ptypen == NULL) {
01722             erEnd("particle type is not defined (NULL)");
01723         }
01724         ptype = -1;
01725         for (j = 0; j < GRCC_PT_Size; j++) {
01726             if (strcmp(pinp->ptypen, ptypenames[j]) == 0) {
01727                 ptype = j;
01728                 break;
01729             }
01730         }
01731         if (ptype < 0) {
01732             if (prlevel > 0) {
01733                 grcc_fprintf(GRCC_Stderr, "*** particle type \"%s\" is not defined\n",
01734                     pinp->ptypen);
01735             }
01736             erEnd("particle type is not defined (name)");
01737         }
01738     }
01739     if (ptype == GRCC_PT_Undef && ptype != 0) {
01740         erEnd("illegal ptype");
01741     }
01742
01743     extonly = pinp->extonly;
01744     cmindeg = 0;
01745     cmaxdeg = 0;
01746 }
01747
01748 //-----
01749 Particle::~Particle(void)
01750 {
01751     free(aname);
01752     free(name);
01753 }
01754
01755 //-----
01756 char *Particle::particleName(int p)
01757 {
01758     if (p < 0) {
01759         return aname;
01760     } else {
01761         return name;
01762     }

```

```

01762     }
01763 }
01764
01765 //-----
01766 int Particle::particleCode(int p)
01767 {
01768     if (p < 0) {
01769         return acode;
01770     } else {
01771         return pcode;
01772     }
01773 }
01774
01775 //-----
01776 int Particle::isNeutral(void)
01777 {
01778     return neutral;
01779 }
01780
01781 //-----
01782 const char *Particle::typeName(void)
01783 {
01784     return ptypenames[ptype];
01785 }
01786
01787 //-----
01788 const char *Particle::typeGName(void)
01789 {
01790     return pgtypenames[ptype];
01791 }
01792
01793 //-----
01794 char *Particle::aparticle(void)
01795 {
01796     static char prtcl[] = "Particle";
01797
01798     if (isNeutral()) {
01799         return prtcl;
01800     } else {
01801         return aname;
01802     }
01803 }
01804
01805 //-----
01806 void Particle::prParticle(void)
01807 {
01808     if (isNeutral()) {
01809         grcc_fprintf(GRCC_Stdout, "pid=%d, name=\"%s\\", real_field, ", id, name);
01810     } else {
01811         grcc_fprintf(GRCC_Stdout, "pid=%d, name=\"%s\\", aname=\"%s\\", ", id, name, aname);
01812     }
01813     grcc_fprintf(GRCC_Stdout, "ptype=%s, pcode=%d, acode=%d, extonly=%d, cdeg=(%d,%d)\\n",
01814         ptypenames[ptype], pcode, acode, extonly, cmindeg, cmaxdeg);
01815 }
01816
01817 //=====
01818 // class Interaction
01819 //-----
01820 Interaction::Interaction(Model *mdl, int iid, const char *nam, int icd, int *cpl, int nlgs, int
    *plst, int csm, int lp)
01821 {
01822     static char buff[100];
01823     static int nbuff=100;
01824     static Bool prmsg = True;
01825     Particle *p;
01826     int j, ndir, sdir, jdir, nmaj, jmaj, ngho, sgho, jgho, ptcl;
01827     Bool ok;
01828
01829     mdl = mdl; // the model
01830     id = iid; // id of this interaction
01831     icode = icd; // id of this interaction
01832     csum = csm; // the total orders of coupling constants
01833     nlgs = nlgs; // the size of plist[]
01834     loop = lp; // the number of loops
01835
01836     if (nam == NULL || strlen(nam) < 1) {
01837         snprintf(buff, nbuff, "vt%d", icd);
01838         name = strdup(buff);
01839     } else {
01840         name = strdup(nam); // the name of the interaction
01841     }
01842     nclist = mdl->ncouple;
01843     clist = new int[nclist];
01844     icode = icd;
01845     for (j = 0; j < mdl->ncouple; j++) {
01846         clist[j] = cpl[j];
01847     }

```

```

01848     plist = new int[nlegs];
01849     nplist = nlegs;
01850     for (j = 0; j < nlegs; j++) {
01851         plist[j] = plst[j];
01852     }
01853     nslist = 0;
01854     slist = NULL;
01855
01856     // check fermion number conservation
01857     ndir = 0;
01858     sdir = 0;
01859     jdir = 0;
01860     nmaj = 0;
01861     jmaj = 0;
01862     ngho = 0;
01863     sgho = 0;
01864     jgho = 0;
01865     ok = True;
01866     for (j = 0; j < nplist; j++) {
01867         if (plist[j] > 0) {
01868             p = mdl->particles[plist[j]];
01869             ptcl = 1;
01870         } else {
01871             p = mdl->particles[-plist[j]];
01872             ptcl = -1;
01873         }
01874
01875         if (p->ptype == GRCC_PT_Dirac) {
01876             ndir++;
01877             if (ptcl > 0) {
01878                 sdir++;
01879             } else {
01880                 sdir--;
01881             }
01882             if (Abs(sdir) == 1) {
01883                 jdir = j;
01884             } else if ((Abs(sdir) == 0 && j != jdir + 1) || Abs(sdir) > 1) {
01885                 grcc_fprintf(GRCC_Stderr, "*** Interaction: Dirac and anti-Dirac should "
01886                     "be arranged in pairs in an interaction.\n");
01887                 ok = False;
01888             }
01889         } else if (p->ptype == GRCC_PT_Ghost) {
01890             ngho++;
01891             if (ptcl > 0) {
01892                 sgho++;
01893             } else {
01894                 sgho--;
01895             }
01896             if (Abs(sgho) == 1) {
01897                 jgho = j;
01898             } else if ((Abs(sgho) == 0 && j != jgho + 1) || Abs(sgho) > 1) {
01899                 grcc_fprintf(GRCC_Stderr, "*** Interaction: Ghost and anti-Ghost should "
01900                     "be arranged in pairs in an interaction.\n");
01901                 ok = False;
01902             }
01903         } else if (p->ptype == GRCC_PT_Majorana) {
01904             nmaj++;
01905             if (nmaj % 2 == 1) {
01906                 jmaj = j;
01907             } else if (j != jmaj + 1) {
01908                 grcc_fprintf(GRCC_Stderr, "*** Interaction: two Majoranas should "
01909                     "be arranged in pairs in an interaction.\n");
01910                 ok = False;
01911             }
01912         }
01913     }
01914     if (sdir != 0) {
01915         grcc_fprintf(GRCC_Stderr, "*** Interaction: Dirac number is not conserved\n");
01916         ok = False;
01917     }
01918     if (sgho != 0) {
01919         grcc_fprintf(GRCC_Stderr, "*** Interaction: Ghost number is not conserved\n");
01920         ok = False;
01921     }
01922     if (nmaj % 2 != 0) {
01923         grcc_fprintf(GRCC_Stderr, "*** Interaction: odd number of Majorana particles\n");
01924         ok = False;
01925     }
01926     if (ndir + nmaj + ngho > 2 && prmsg) {
01927         grcc_fprintf(GRCC_Stderr, "+++ Interaction: more than 2 "
01928             "Dirac/Majorana/Ghost "
01929             "particles are interacting.\n");
01930         grcc_fprintf(GRCC_Stderr, "    Interaction has %d Dirac, %d Ghost "
01931             "and %d Majorana particles.\n",
01932             ndir, nmaj, ngho);
01933         grcc_fprintf(GRCC_Stderr, "    Sign factors related to fermions may "
01934             "be inconsistent with your calculation method.\n");

```



```

01935     prmsg = False;
01936 }
01937 if (!ok) {
01938     prInteraction();
01939     erEnd("illegal Dirac/Majorana/Ghost interaction");
01940 }
01941 }
01942
01943 //-----
01944 Interaction::~Interaction(void)
01945 {
01946     if (slist != NULL) {
01947         delete[] slist;
01948     }
01949     if (plist != NULL) {
01950         delete[] plist;
01951     }
01952     if (clist != NULL) {
01953         delete[] clist;
01954     }
01955     free(name);
01956 }
01957
01958 //-----
01959 void Interaction::prInteraction(void)
01960 {
01961     int j;
01962
01963     grcc_fprintf(GRCC_Stdout, "vid=%d, icode=%d, name=\"%s\\", loop=%d, csum=%d, cpl=",
01964                 id, icode, name, loop, csum);
01965     prIntArray(mdl->ncouple, clist, " ", legs=["");
01966     for (j = 0; j < nplist; j++) {
01967         if (j != 0) {
01968             grcc_fprintf(GRCC_Stdout, " ");
01969         }
01970         grcc_fprintf(GRCC_Stdout, "%s", mdl->particleName(plist[j]));
01971     }
01972     grcc_fprintf(GRCC_Stdout, "];\\n");
01973 }
01974 }
01975
01976 //=====
01977 // class Model
01978 //-----
01979 Model::Model(MInput *minp)
01980 {
01981     static PInput pundef = {"undef", "undef", 0, 0, "undef", 0, 0};
01982     int j;
01983
01984     if (minp->name == NULL || strlen(minp->name) < 1) {
01985         erEnd("model should have a name");
01986     }
01987     name = strdup(minp->name);
01988     ncouple = minp->ncouple;
01989     if (ncouple >= GRCC_MAXNCPLG) {
01990         erEnd("too many coupling constants in the model (GRCC_MAXNCPLG)");
01991     }
01992     // cnlist = minp->cnamlist;
01993     cnlist = new char*[ncouple];
01994     for (j = 0; j < ncouple; j++) {
01995         cnlist[j] = strdup(minp->cnamlist[j]);
01996     }
01997
01998     nParticles = 0;
01999     particles = new Particle*[GRCC_MAXMPARTICLES];
02000     for (j = 0; j < GRCC_MAXMPARTICLES; j++) {
02001         particles[j] = NULL;
02002     }
02003     pdef = False;
02004
02005     nInteracts = 0;
02006     interacts = new Interaction*[GRCC_MAXMINTERACT];
02007     for (j = 0; j < GRCC_MAXMINTERACT; j++) {
02008         interacts[j] = NULL;
02009     }
02010     vdef = False;
02011
02012     if (particles == NULL || interacts == NULL) {
02013         erEnd("memory allocation failed");
02014     }
02015
02016 #ifdef UNIQINTR
02017     if (minp->defpart != GRCC_DEFBYNAME
02018         && minp->defpart != GRCC_DEFBYCODE) {
02019         erEnd("illegal value of defpart");
02020     }
02021 #else

```

```

02022     if (Abs(minp->defpart) != GRCC_DEFBYNAME
02023         && Abs(minp->defpart) != GRCC_DEFBYCODE) {
02024         erEnd("illegal value of defpart");
02025     }
02026 #endif
02027
02028     defpart = minp->defpart;
02029
02030     maxnlegs = 0;
02031     maxcpl = 0;
02032     maxloop = 0;
02033     ncplgcp = 0;
02034     cplgvl = NULL;
02035
02036     addParticle(&pundef);
02037 }
02038
02039 //-----
02040 Model::~Model(void)
02041 {
02042     int j;
02043
02044     if (ncplgcp > 0) {
02045         for (j = ncplgcp-1; j >= 0; j--) {
02046             cplgvl[j] = delintdup(cplgvl[j]);
02047         }
02048         delete[] cplgvl;
02049         cplgnvl = delintdup(cplgnvl);
02050         cplglg = delintdup(cplglg);
02051         cplgcp = delintdup(cplgcp);
02052     }
02053
02054     for (j = GRCC_MAXMINTERACT-1; j >= 0; j--) {
02055         if (interacts[j] != NULL) {
02056             interacts[j]->slist = delintdup(interacts[j]->slist);
02057             delete interacts[j];
02058             interacts[j] = NULL;
02059         }
02060     }
02061     delete[] interacts;
02062     interacts = NULL;
02063
02064     for (j = GRCC_MAXMPARTICLES-1; j >= 0; j--) {
02065         if (particles[j] != NULL) {
02066             delete particles[j];
02067             particles[j] = NULL;
02068         }
02069     }
02070     delete[] particles;
02071     particles = NULL;
02072
02073     for (j=ncouple-1; j>=0; j--) {
02074         free(cnlist[j]);
02075     }
02076     delete[] cnlist;
02077
02078     free(name);
02079 }
02080
02081 //-----
02082 void Model::prModel(void)
02083 {
02084     static char hdr[] = "#=====\n";
02085     static char hdl[] = "#-----\n";
02086     int j;
02087
02088     grcc_fprintf(GRCC_Stdout, "%s", hdr);
02089     grcc_fprintf(GRCC_Stdout, "Model=\"%s\\", " ", name);
02090     grcc_fprintf(GRCC_Stdout, "coupling=[");
02091     for (j = 0; j < ncouple; j++) {
02092         if (j != 0) {
02093             grcc_fprintf(GRCC_Stdout, ", ");
02094         }
02095         grcc_fprintf(GRCC_Stdout, "\\\"%s\\\"", cnlist[j]);
02096     }
02097     grcc_fprintf(GRCC_Stdout, "];\n");
02098     grcc_fprintf(GRCC_Stdout, "%s", hdl);
02099     grcc_fprintf(GRCC_Stdout, "Particles=%d;\n", nParticles);
02100     for (j = 1; j < nParticles; j++) {
02101         particles[j]->prParticle();
02102     }
02103     grcc_fprintf(GRCC_Stdout, "EndParticle;\n");
02104     grcc_fprintf(GRCC_Stdout, "allPart (%d) = [", nallPart);
02105     for (j = 0; j < nallPart; j++) {
02106         if (j != 0) {
02107             grcc_fprintf(GRCC_Stdout, ", ");
02108         }

```

```

02109         grcc_fprintf(GRCC_Stdout, "%d", allPart[j]);
02110     }
02111     grcc_fprintf(GRCC_Stdout, "]\n");
02112
02113     grcc_fprintf(GRCC_Stdout, "%s", hdl);
02114     grcc_fprintf(GRCC_Stdout, "Interactions=%d;\n", nInteracts);
02115     for (j = 0; j < nInteracts; j++) {
02116         interacts[j]->prInteraction();
02117     }
02118     grcc_fprintf(GRCC_Stdout, "EndInteraction;\n");
02119     grcc_fprintf(GRCC_Stdout, "%s", hdl);
02120     grcc_fprintf(GRCC_Stdout, "InteractionTable;\n");
02121     grcc_fprintf(GRCC_Stdout, "# %d class in (total coupling, degree)\n", ncplgcp);
02122     for (j = 0; j < ncplgcp; j++) {
02123         grcc_fprintf(GRCC_Stdout, "    class=%d: cp=%d, lg=%d, vl=", j, cplgcp[j], cplglg[j]);
02124         prIntArray(cplgnvl[j], cplgvl[j], "\n");
02125     }
02126     grcc_fprintf(GRCC_Stdout, "EndInteractionTable;\n");
02127     grcc_fprintf(GRCC_Stdout, "%s", hdr);
02128 }
02129
02130 //-----
02131 void Model::addParticle(PInput *pinp)
02132 {
02133     int nid, aid, pid, pcd, acd;
02134
02135     if (nParticles >= GRCC_MAXMPARTICLES) {
02136         erEnd("particle table overflow (GRCC_MAXMPARTICLES)");
02137     }
02138
02139     // uniqueness check
02140     if (Abs(defpart) == GRCC_DEFBYCODE) {
02141         pcd = findParticleCode(pinp->pcode);
02142         acd = findParticleCode(pinp->acode);
02143         if (pcd > 0 || acd > 0) {
02144             if (prlevel > 0) {
02145                 grcc_fprintf(GRCC_Stderr, "*** particle code [%d, %d] ",
02146                     pinp->pcode, pinp->acode);
02147                 grcc_fprintf(GRCC_Stderr, "is already used.\n");
02148             }
02149             erEnd("particle code is already defined");
02150         }
02151     } else if (Abs(defpart) == GRCC_DEFBYNAME) {
02152         if (pinp->name == NULL || pinp->aname == NULL ||
02153             strlen(pinp->name) < 1 || strlen(pinp->aname) < 1) {
02154             erEnd("no name of particle or anti-particle.");
02155         }
02156         nid = findParticleName(pinp->name);
02157         aid = findParticleName(pinp->aname);
02158         if (nid > 0 || aid > 0) {
02159             if (prlevel > 0) {
02160                 grcc_fprintf(GRCC_Stderr, "*** particle name [%s, %s] ",
02161                     pinp->name, pinp->aname);
02162                 grcc_fprintf(GRCC_Stderr, "is already used.\n");
02163             }
02164             erEnd("particle name is already defined.");
02165         }
02166     }
02167
02168     pid = nParticles++;
02169     particles[pid] = new Particle(this, pid, pinp);
02170 }
02171
02172 //-----
02173 void Model::addParticleEnd(void)
02174 {
02175     int p, ap;
02176
02177     pdef = True;
02178
02179     #ifdef OPTEXTONLY
02180     nallPart = 0;
02181     for (p = nParticles-1; p >= 1; p--) {
02182         if (!particles[p]->extonly) {
02183             ap = antiParticle(p);
02184             if (ap != p) {
02185                 allPart[nallPart++] = ap;
02186             }
02187         }
02188     }
02189     for (p = 1; p < nParticles; p++) {
02190         if (!particles[p]->extonly) {
02191             allPart[nallPart++] = p;
02192         }
02193     }
02194     sorti(nallPart, allPart);
02195 #else

```

```

02196     nallPart = 0;
02197     for (p = nParticles-1; p >= 1; p--) {
02198         ap = antiParticle(p);
02199         if (ap != p) {
02200             allPart[nallPart++] = ap;
02201         }
02202     }
02203     for (p = 1; p < nParticles; p++) {
02204         allPart[nallPart++] = p;
02205     }
02206     sorti(nallPart, allPart);
02207 #endif
02208 }
02209
02210 //-----
02211 void Model::addInteraction(IInput *iinp)
02212 {
02213     int vid, nlegs, j, k, c;
02214     int csum, lp2, lp;
02215     int plist[GRCC_MAXLEGS];
02216
02217     if (!pdef) {
02218         erEnd("call 'addParticleEnd' before 'addInteraction'");
02219     }
02220     if (nInteracts >= GRCC_MAXMINTERACT) {
02221         erEnd("too many interactions in the model (GRCC_MAXMINTERACT)");
02222     }
02223     nlegs = iinp->nplistn;
02224     if (nlegs >= GRCC_MAXLEGS) {
02225         erEnd("too many legs in an interaction (GRCC_MAXLEGS)");
02226     }
02227
02228     // check identifier
02229     if (defpart == GRCC_DEFBYCODE) {
02230         if (iinp->icode < 0) {
02231             if (prlevel > 0) {
02232                 grcc_fprintf(GRCC_Stderr, "*** vertex code %d should be positive\n",
02233                     iinp->icode);
02234             }
02235             erEnd("vertex code should be positive");
02236         } else {
02237             vid = findInteractionCode(iinp->icode);
02238             if (vid >= 0) {
02239                 if (prlevel > 0) {
02240                     grcc_fprintf(GRCC_Stderr, "*** vertex code %d is already used\n",
02241                         iinp->icode);
02242                 }
02243                 erEnd("vertex code is already used");
02244             }
02245         }
02246     } else if (defpart == GRCC_DEFBYNAME) {
02247         if (iinp->name == NULL || strlen(iinp->name) < 1) {
02248             erEnd("no name of vertex\n");
02249         }
02250         vid = findInteractionName(iinp->name);
02251         if (vid >= 0) {
02252             if (prlevel > 0) {
02253                 grcc_fprintf(GRCC_Stderr, "*** vertex name %s is already used\n",
02254                     iinp->name);
02255                 erEnd("vertex name is already used");
02256             }
02257         }
02258     } else {
02259 #ifdef UNIQINTR
02260         // defpart =< 0
02261         erEnd("illegal value of defpart");
02262 #else
02263         // don't care about duplicated name/code of interaction
02264         ;
02265 #endif
02266     }
02267     vid = nInteracts++;
02268
02269     for (j = 0; j < nlegs; j++) {
02270         if (Abs(defpart) == GRCC_DEFBYCODE) {
02271             plist[j] = findParticleCode(iinp->plistc[j]);
02272             if (plist[j] == 0) {
02273                 if (prlevel > 0) {
02274                     grcc_fprintf(GRCC_Stderr, "*** particle code %d ",
02275                         iinp->plistc[j]);
02276                     grcc_fprintf(GRCC_Stderr, "is not defined\n");
02277                 }
02278                 erEnd("particle is not defined (code)");
02279             }
02280         } else if (Abs(defpart) == GRCC_DEFBYNAME) {
02281             plist[j] = findParticleName(iinp->plistn[j]);
02282             if (plist[j] == 0) {

```

```

02283         if (prlevel > 0) {
02284             grcc_fprintf(GRCC_Stderr, "*** particle name %s ",
02285                 iinp->plistn[j]);
02286             grcc_fprintf(GRCC_Stderr, "is not defined\n");
02287         }
02288         erEnd("particle is not defined (name)");
02289     }
02290 }
02291 }
02292
02293 csum = 0;
02294 for (j = 0; j < ncouple; j++) {
02295     c = iinp->cvalist[j];
02296     if (c < 0) {
02297         if (prlevel > 0) {
02298             grcc_fprintf(GRCC_Stderr, "*** illegal value of c-constants \n");
02299             for (k = 0; k < ncouple; k++) {
02300                 grcc_fprintf(GRCC_Stderr, "      %s = %d\n",
02301                     cnlist[k], iinp->cvalist[k]);
02302             }
02303         }
02304         erEnd("illegal value of c-constants");
02305     }
02306     csum += c;
02307 }
02308 if (csum < 0) {
02309     if (prlevel > 0) {
02310         grcc_fprintf(GRCC_Stderr, "*** illegal total coupling-constants %d\n",
02311             csum);
02312         for (k = 0; k < ncouple; k++) {
02313             grcc_fprintf(GRCC_Stderr, "      %s = %d\n",
02314                 cnlist[k], iinp->cvalist[k]);
02315         }
02316     }
02317     erEnd("illegal value of c-constants");
02318 }
02319
02320 // assume : csum = nlegs - 2 + 2*loop
02321 lp2 = csum - nlegs + 2;
02322 if (lp2 % 2 != 0 || lp2 < 0) {
02323     if (prlevel > 0) {
02324         grcc_fprintf(GRCC_Stderr, "*** illegal coupling const : ");
02325         grcc_fprintf(GRCC_Stderr, "nlegs - 2 + 2*loop: 2*loop = %d\n", lp2);
02326     }
02327 }
02328 lp = lp2/2;
02329
02330 maxnlegs = Max(maxnlegs, nlegs);
02331 maxloop = Max(maxloop, lp);
02332 maxcpl = Max(maxcpl, csum);
02333
02334 interacts[vid] = new Interaction(this, vid, iinp->name, iinp->icode,
02335     iinp->cvalist, nlegs, plist, csum, lp);
02336 }
02337
02338 //-----
02339 void Model::addInteractionEnd(void)
02340 {
02341     int lst[GRCC_MAXMINTERACT];
02342     int tcp[GRCC_MAXMINTERACT], tlg[GRCC_MAXMINTERACT];
02343     int tnv1[GRCC_MAXMINTERACT], *tv1[GRCC_MAXMINTERACT];
02344     Interaction *vt;
02345     int cp, lg, j, k, p;
02346
02347     vdef = True;
02348     ncplgcp = 0;
02349     for (cp = 0; cp <= maxcpl; cp++) {
02350
02351         for (lg = 0; lg <= maxnlegs; lg++) {
02352             k = 0;
02353             for (j = 0; j < nInteracts; j++) {
02354                 vt = interacts[j];
02355                 if (vt->csum == cp && vt->nlegs == lg) {
02356                     lst[k++] = j;
02357                 }
02358             }
02359             if (k > 0) {
02360                 tcp[ncplgcp] = cp;
02361                 tlg[ncplgcp] = lg;
02362                 tnv1[ncplgcp] = k;
02363                 tv1[ncplgcp] = intdup(k, lst);
02364                 ncplgcp++;
02365             }
02366         }
02367     }
02368     cplgcp = intdup(ncplgcp, tcp);
02369     cplglg = intdup(ncplgcp, tlg);

```

```

02370     cplgnvl = intdup(ncplgcp, tnv1);
02371     cplgv1 = new int*[ncplgcp];
02372     for (j = 0; j < ncplgcp; j++) {
02373         if (cplgnvl[j] > 0) {
02374             cplgv1[j] = tv1[j];
02375         } else {
02376             cplgv1[j] = NULL;
02377         }
02378     }
02379
02380     for (j = 0; j < nInteracts; j++) {
02381         vt = interacts[j];
02382         if (vt->slist != NULL) {
02383             erEnd("addInteractionEnd: vt->nslist != NULL");
02384         }
02385         vt->slist = intdup(vt->nplist, vt->plist);
02386         vt->nslist = toSList(vt->nplist, vt->slist);
02387     }
02388
02389     for (p = 0; p < nParticles; p++) {
02390         particles[p]->cmindeg = 0;
02391         particles[p]->cmaxdeg = 0;
02392     }
02393     for (j = 0; j < nInteracts; j++) {
02394         vt = interacts[j];
02395         for (k = 0; k < vt->nplist; k++) {
02396             p = abs(vt->plist[k]);
02397             if (particles[p]->cmindeg == 0) {
02398                 particles[p]->cmindeg = vt->nlegs;
02399                 particles[p]->cmaxdeg = vt->nlegs;
02400             } else {
02401                 particles[p]->cmindeg = Min(particles[p]->cmindeg, vt->nlegs);
02402                 particles[p]->cmaxdeg = Max(particles[p]->cmaxdeg, vt->nlegs);
02403             }
02404         }
02405     }
02406 }
02407
02408 //-----
02409 int Model::findParticleName(const char *name)
02410 {
02411     int j;
02412
02413     for (j = 0; j < nParticles; j++) {
02414         if (strcmp(name, particles[j]->name) == 0) {
02415             return j;
02416         }
02417     }
02418     for (j = 0; j < nParticles; j++) {
02419         if (strcmp(name, particles[j]->aname) == 0) {
02420             return -j;
02421         }
02422     }
02423     return 0;
02424 }
02425
02426 //-----
02427 int Model::findParticleCode(int pcd)
02428 {
02429     int j;
02430
02431     for (j = 0; j < nParticles; j++) {
02432         if (pcd == particles[j]->pcode) {
02433             return j;
02434         } else if (pcd == particles[j]->acode) {
02435             return -j;
02436         }
02437     }
02438     return 0;
02439 }
02440
02441 //-----
02442 char *Model::particleName(int p)
02443 {
02444     int q;
02445
02446     if (Abs(p) >= nParticles) {
02447         if (prlevel > 0) {
02448             grcc_fprintf(GRCC_Stderr, "\n*** Model::particleName: ");
02449             grcc_fprintf(GRCC_Stderr, "illegal particle id=%d\n", p);
02450         }
02451         erEnd("Model::particleName: illegal particle");
02452     }
02453     if (p < 0) {
02454         q = -p;
02455     } else {
02456         q = p;

```

```

02457     }
02458     return particles[q]->particleName(p);
02459 }
02460
02461 //-----
02462 int Model::particleCode(int p)
02463 {
02464     int q;
02465
02466     if (Abs(p) >= nParticles) {
02467         grcc_fprintf(GRCC_Stderr, "\n*** Model::particleCode: illegal particle id=%d\n",
02468             p);
02469         erEnd("Model::particleCode: illegal particle");
02470     }
02471     if (p < 0) {
02472         q = - p;
02473     } else {
02474         q = p;
02475     }
02476     return particles[q]->particleCode(p);
02477 }
02478
02479 //-----
02480 void Model::prParticleArray(int n, int *a, const char *msg)
02481 {
02482     int j;
02483
02484     grcc_fprintf(GRCC_Stdout, "[");
02485     for (j = 0; j < n; j++) {
02486         if (j != 0) {
02487             grcc_fprintf(GRCC_Stdout, ", ");
02488         }
02489         grcc_fprintf(GRCC_Stdout, "%s", particleName(a[j]));
02490     }
02491     grcc_fprintf(GRCC_Stdout, "]%s", msg);
02492 }
02493
02494 //-----
02495 int Model::findMClass(const int cpl, const int dgr)
02496 {
02497     int j;
02498
02499     for (j = 0; j < ncplgcp; j++) {
02500         if (cplgcp[j] == cpl && cplglg[j] == dgr) {
02501             return j;
02502         }
02503     }
02504     return -1;
02505 }
02506
02507 //-----
02508 int Model::normalParticle(int pt)
02509 {
02510     int ptc = Abs(pt);
02511     Particle *p;
02512
02513     if (ptc >= nParticles) {
02514         return 0;
02515     }
02516     p = particles[ptc];
02517     if (p->isNeutral()) {
02518         return ptc;
02519     } else {
02520         return pt;
02521     }
02522 }
02523
02524 //-----
02525 int Model::antiParticle(int pt)
02526 {
02527     int ptc = Abs(pt);
02528     Particle *p;
02529
02530     p = particles[ptc];
02531     if (p->isNeutral()) {
02532         return ptc;
02533     } else {
02534         return -pt;
02535     }
02536 }
02537
02538 //-----
02539 int *Model::allParticles(int *len)
02540 {
02541     *len = nallPart;
02542     return allPart;
02543 }

```

```

02544
02545 //-----
02546 int Model::findInteractionName(const char *name)
02547 {
02548     int j;
02549
02550     for (j = 0; j < nInteracts; j++) {
02551         if (strcmp(name, interacts[j]->name) == 0) {
02552             return j;
02553         }
02554     }
02555     return -1;
02556 }
02557
02558 //-----
02559 int Model::findInteractionCode(int icd)
02560 {
02561     int j;
02562
02563     for (j = 0; j < nInteracts; j++) {
02564         if (icd == interacts[j]->icode) {
02565             return j;
02566         }
02567     }
02568     return -1;
02569 }
02570
02571 //-----
02572 void Model::printMInput(MInput *min)
02573 {
02574     grcc_fprintf(GRCC_Stdout, "  \\Model\\": [\\%s\\", %d, [",
02575                 min->name, min->ncouple);
02576     for (int j = 0; j < min->ncouple; j++) {
02577         if (j != 0) {
02578             grcc_fprintf(GRCC_Stdout, ", ");
02579         }
02580         grcc_fprintf(GRCC_Stdout, "\\%s\\", min->cnamlist[j]);
02581     }
02582     grcc_fprintf(GRCC_Stdout, "], ");
02583     if (min->defpart == GRCC_DEFBYCODE) {
02584         grcc_fprintf(GRCC_Stdout, "\\ByCode\\");
02585     } else {
02586         grcc_fprintf(GRCC_Stdout, "\\ByName\\");
02587     }
02588     grcc_fprintf(GRCC_Stdout, "],\\n");
02589 }
02590
02591 //-----
02592 void Model::printPInput(PInput *pin)
02593 {
02594     grcc_fprintf(GRCC_Stdout, "    [\\%s\\"(%d), \\%s\\"(%d), \\%s\\"(%d), %d],\\n",
02595                 pin->name, pin->pcode, pin->aname, pin->acode,
02596                 pin->ptypen, pin->ptypec, pin->extonly);
02597 }
02598
02599 //-----
02600 void Model::printIInput(IInput *iin)
02601 {
02602     int j;
02603
02604     grcc_fprintf(GRCC_Stdout, "    [\\%s\\"(%d), %d, [", iin->name, iin->icode, iin->nplistn);
02605     for (j = 0; j < iin->nplistn; j++) {
02606         if (j != 0) {
02607             grcc_fprintf(GRCC_Stdout, ", ");
02608         }
02609         grcc_fprintf(GRCC_Stdout, "\\%s\\"(%d)", iin->plistn[j], iin->plistc[j]);
02610     }
02611     grcc_fprintf(GRCC_Stdout, "], [");
02612     for (j = 0; j < GRCC_MAXNCPLG; j++) {
02613         if (j != 0) {
02614             grcc_fprintf(GRCC_Stdout, ", ");
02615         }
02616         grcc_fprintf(GRCC_Stdout, "%d", iin->cvallist[j]);
02617     }
02618     grcc_fprintf(GRCC_Stdout, "],\\n");
02619 }
02620
02621 //*****
02622 // proc.cc
02623
02624 //=====
02625 // class PNodeClass
02626 //-----
02627 PNodeClass::PNodeClass(SProcess *spc, int nnodes, int nclss, NCInput *cls)
02628 {
02629     // Create a class of nodes
02630     // nnodes : # nodes

```



```

02631 // nclass : # classes
02632 // cls : table of class inputs
02633 //
02634 int j, k, nn, lp2;
02635 Bool ok = True;
02636
02637 sproc = spc;
02638 nnodes = nnods;
02639 nclass = nclss;
02640 deg = new int[nclass];
02641 type = new int[nclass];
02642 particle = new int[nclass];
02643 count = new int[nclass];
02644 couple = new int[nclass];
02645 cmindeg = new int[nclass];
02646 cmaxdeg = new int[nclass];
02647 for (j = 0; j < nclass; j++) {
02648     deg[j] = cls[j].cldeg;
02649     count[j] = cls[j].clnum;
02650     type[j] = cls[j].cltyp;
02651     cmindeg[j] = cls[j].cmindeg;
02652     cmaxdeg[j] = cls[j].cmaxdeg;
02653     particle[j] = cls[j].ptcl;
02654     couple[j] = cls[j].cpl;
02655     lp2 = (couple[j]-deg[j]+2);
02656     if (!isATEXternal(type[j]) && (lp2 % 2 != 0 || lp2 < 0)) {
02657         if (prlevel > 0) {
02658             grcc_fprintf(GRCC_Stderr, "*** PNodeClass: illegal loop: "
02659                 ": 2*loop=cpl[%d] (%d)-deg[%d] (%d)+2 =2*loop = %d\n",
02660                 j, couple[j], j, deg[j], lp2);
02661             for (k = 0; k < nclss; k++) {
02662                 grcc_fprintf(GRCC_Stderr, "k=%d, deg=%d, typ=%d, ptcl=%d, ",
02663                     k, deg[k], type[k], particle[k]);
02664                 grcc_fprintf(GRCC_Stderr, "cpl=%d, cnt=%d\n",
02665                     couple[k], count[k]);
02666             }
02667         }
02668         ok = False;
02669     }
02670 }
02671 if (!ok) {
02672     erEnd("PNodeClass: illegal loop");
02673 }
02674 cl2nd = new int[nclass+1];
02675 nd2cl = new int[nnodes];
02676 cl2mcl = new int[nnodes];
02677
02678 nn = 0;
02679 for (j = 0; j < nclass; j++) {
02680     cl2nd[j] = nn;
02681     for (k = 0; k < count[j]; k++, nn++) {
02682         nd2cl[nn] = j;
02683     }
02684     if (isATEXternal(type[j])) {
02685         cl2mcl[j] = -1;
02686     } else {
02687         cl2mcl[j] = spc->model->findMClass(couple[j], deg[j]);
02688         if (cl2mcl[j] < 0) {
02689             if (prlevel > 0) {
02690                 grcc_fprintf(GRCC_Stderr, "*** PNodeClass : no vertex : ");
02691                 grcc_fprintf(GRCC_Stderr, "coupling=%d, degree=%d\n",
02692                     couple[j], deg[j]);
02693             }
02694             erEnd("PNodeClass : no vertex");
02695         }
02696     }
02697 }
02698 cl2nd[nclass] = nnodes;
02699 }
02700 //-----
02701 PNodeClass::PNodeClass(SProcess *spc, int nnods, int nclss, int *dgs, int *typ, int *ptcl, int *cpl,
    int *cnt, int *cmindeg, int *cmaxdeg)
02702 {
02703     // Create a class of nodes
02704     // nnodes : # nodes
02705     // nclss : # classes
02706     // dgs : degree of a node, which is common to the vertices in a class
02707     // typ : initial/final/vertex
02708     // ptcl : particle code or list of possible interactions
02709     // cpl : values of coupling constants.
02710     // cnt : the number of nodes in this class
02711     //
02712     int j, k, nn, lp2;
02713     Bool ok = True;
02714
02715     sproc = spc;
02716     nnodes = nnods;

```

```

02717     nclass    = nclss;
02718     deg       = new int[nclass];
02719     type      = new int[nclass];
02720     particle  = new int[nclass];
02721     count     = new int[nclass];
02722     couple    = new int[nclass];
02723     cmindeg   = new int[nclass];
02724     cmaxdeg   = new int[nclass];
02725     for (j = 0; j < nclass; j++) {
02726         deg[j]    = dgs[j];
02727         type[j]   = typ[j];
02728         particle[j] = ptcl[j];
02729         count[j]  = cnt[j];
02730         couple[j] = cpl[j];
02731         cmindeg[j] = cmind[j];
02732         cmaxdeg[j] = cmaxd[j];
02733         lp2 = (cpl[j]-dgs[j]+2);
02734         if (!isATEXternal(type[j]) && (lp2 % 2 != 0 || lp2 < 0)) {
02735             if (prlevel > 0) {
02736                 grcc_fprintf(GRCC_Stderr, "*** PNodeClass: illegal loop: "
02737                     ": 2*loop=cpl[%d](%d)-deg[%d](%d)+2 =2*loop = %d\n",
02738                     j, cpl[j], j, dgs[j], lp2);
02739                 for (k = 0; k < nclss; k++) {
02740                     grcc_fprintf(GRCC_Stderr, "k=%d, dgs=%d, typ=%d, ptcl=%d, ",
02741                         k, dgs[k], typ[k], ptcl[k]);
02742                     grcc_fprintf(GRCC_Stderr, "cpl=%d, cnt=%d\n", cpl[k], cnt[k]);
02743                 }
02744             }
02745             ok = False;
02746         }
02747     }
02748     if (!ok) {
02749         erEnd("PNodeClass: illegal loop");
02750     }
02751     cl2nd    = new int[nclass+1];
02752     nd2cl    = new int[nnodes];
02753     cl2mcl   = new int[nnodes];
02754
02755     nn = 0;
02756     for (j = 0; j < nclass; j++) {
02757         cl2nd[j] = nn;
02758         for (k = 0; k < count[j]; k++, nn++) {
02759             nd2cl[nn] = j;
02760         }
02761         if (isATEXternal(type[j])) {
02762             cl2mcl[j] = -1;
02763         } else {
02764             cl2mcl[j] = spc->model->findMClass(couple[j], deg[j]);
02765             if (cl2mcl[j] < 0) {
02766                 if (prlevel > 0) {
02767                     grcc_fprintf(GRCC_Stderr, "*** PNodeClass : no vertex : ");
02768                     grcc_fprintf(GRCC_Stderr, "coupling=%d, degree=%d\n",
02769                         couple[j], deg[j]);
02770                 }
02771                 erEnd("PNodeClass : no vertex");
02772             }
02773         }
02774     }
02775     cl2nd[nclass] = nnodes;
02776 }
02777
02778 //-----
02779 PNodeClass::~PNodeClass(void)
02780 {
02781     delete[] cl2mcl;
02782     delete[] nd2cl;
02783     delete[] cl2nd;
02784     delete[] cmaxdeg;
02785     delete[] cmindeg;
02786     delete[] couple;
02787     delete[] count;
02788     delete[] particle;
02789     delete[] type;
02790     delete[] deg;
02791 }
02792
02793 //-----
02794 void PNodeClass::prPNodeClass(void)
02795 {
02796     int j;
02797
02798     grcc_fprintf(GRCC_Stdout, "+++ PNodeClass: nclass=%d, nnodes=%d\n", nclass, nnodes);
02799     for (j = 0; j < nclass; j++) {
02800         prElem(j);
02801     }
02802     grcc_fprintf(GRCC_Stdout, "\n");
02803 }

```

```

02804
02805 //-----
02806 void PNodeClass::prElem(int j)
02807 {
02808     int tp;
02809
02810     grcc_fprintf(GRCC_Stdout, "%3d: %-7s(%2d), ", j, GRCC_AT_NdStr(type[j]), type[j]);
02811     grcc_fprintf(GRCC_Stdout, "deg=%d, count=%d, nodes[%d--%d], couple=%d, cmindeg=%d, cmaxdeg=%d",
02812         deg[j], count[j], cl2nd[j], cl2nd[j+1], couple[j], cmindeg[j], cmaxdeg[j]);
02813     tp = type[j];
02814     if (isATExternal(tp)) {
02815         if (sproc->model != NULL) {
02816             grcc_fprintf(GRCC_Stdout, ", ptcl=%s ", sproc->model->particleName(particle[j]));
02817         } else {
02818             grcc_fprintf(GRCC_Stdout, ", ptcl=%d ", particle[j]);
02819         }
02820     }
02821     grcc_fprintf(GRCC_Stdout, "\n");
02822 }
02823 }
02824
02825 //=====
02826 // class SProcess
02827 //-----
02828 SProcess::SProcess(Model *mdl, Process *prc, Options *opts, int sid, int *clst, int ncls, int *cdeg,
02829     int *ctyp, int *ptcl, int *cpl, int *cnum, int *cmin, int *cmaxd)
02830 {
02831     // Construct a Subprocess object
02832     //   mdl          : model
02833     //   prc          : process (may be NULL)
02834     //   opts         : options
02835     //   sid          : id of the sprocess
02836     //   ncls         : the number of classes
02837     //   cdeg[ncls]   : the table of degrees of nodes.
02838     //   ctyp[ncls]   : the table of nodes in the classes
02839     //   ptcl[ncls]   : particle(External)/interaction code(Internal)
02840     //   cpl[ncls]    : the table of total order of coupling constants.
02841     //   cnum[ncls]   : the table of nodes in the classes
02842     //   cmin[ncls]   : the table of min(deg of connectable vertex)
02843     //   cmaxd[ncls]  : the table of max(deg of connectable vertex)
02844
02845     int j, cp, lp2, ndeg, nvrt, tcpl0;
02846     bool ok;
02847
02848     if (ncls < 1) {
02849         grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: no class %d\n", ncls);
02850         erEnd("SProcess::SProcess: no Class");
02851     }
02852     id      = sid;
02853     model   = mdl;
02854     proc    = prc;
02855     opt     = opts;
02856     nclass  = ncls;
02857
02858     nNodes  = 0;
02859     nEdges  = 0;
02860     nExtern = 0;
02861     tCouple = 0;
02862     loop    = 0;
02863     mgraph  = NULL;
02864     egraph  = NULL;
02865     astack  = NULL;
02866
02867     mgrcount = 0;
02868     agrcount = 0;
02869     extperm  = 1;
02870
02871     // the results of the graph generation
02872     nMGraphs = 0; // the number of generated M-graphs
02873     nMOPI    = 0; // the number of 1PI M-graphs
02874     wMGraphs.setValue(0,1); // the weighted sum of M-graphs
02875     wMOPI.setValue(0,1); // the weighted sum of 1PI M-graphs
02876
02877     nAGraphs = 0; // the number of generated A-graphs
02878     nAOPI    = 0; // the number of 1PI A-graphs
02879     wAGraphs.setValue(0,1); // the weighted sum of A-graphs
02880     wAOPI.setValue(0,1); // the weighted sum of 1PI A-graphs
02881
02882     // check input
02883     nNodes = 0;
02884     nExtern = 0;
02885
02886     ndeg    = 0;
02887     nvrt    = 0;
02888     ok      = True;
02889
02890     tcpl0 = 0;

```

```

02890     for (j = 0; j < model->ncouple; j++) {
02891         clist[j] = clst[j];
02892         tcpl0 += clist[j];
02893     }
02894     for (j = 0; j < nclass; j++) {
02895         cp = cpl[j];
02896         if (cp > 0) {
02897             lp2 = cpl[j] - cdeg[j] + 2;
02898             if (lp2 % 2 != 0 || lp2 < 0) {
02899                 if (prlevel > 0) {
02900                     grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: illegal loop: "
02901                                 "class %d, cp=%d, deg=%d, lp2=%d\n",
02902                                 j, cp, cdeg[j], lp2);
02903                 }
02904                 ok = False;
02905             }
02906         }
02907         tCouple += cp*cnum[j];
02908         ndeg += cdeg[j]*cnum[j];
02909
02910         if (!isATEXternal(ctyp[j])) {
02911             nvrt += cnum[j];
02912         } else if (isATEXternal(ctyp[j])) {
02913             if (model->normalParticle(ptcl[j]) == 0) {
02914                 if (prlevel > 0) {
02915                     grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
02916                     grcc_fprintf(GRCC_Stderr, "illegal particle code: ");
02917                     grcc_fprintf(GRCC_Stderr, "code: class %d, ptcl=%d\n", j, ptcl[j]);
02918                 }
02919                 ok = False;
02920             } else {
02921                 nExtern += cnum[j];
02922             }
02923             extperm *= factorial(cnum[j]);
02924         } else {
02925             if (prlevel > 0) {
02926                 grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: illegal type: ");
02927                 grcc_fprintf(GRCC_Stderr, "class %d, type=%d\n", j, ctyp[j]);
02928             }
02929             ok = False;
02930         }
02931     }
02932 }
02933 if (tcpl0 != tCouple) {
02934     if (prlevel > 0) {
02935         grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
02936         grcc_fprintf(GRCC_Stderr, "illegal coupling constants:");
02937         grcc_fprintf(GRCC_Stderr, " %d != %d\n", tcpl0, tCouple);
02938     }
02939     ok = False;
02940 }
02941 if (ndeg == nExtern) {
02942     if (prlevel > 0) {
02943         grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
02944         grcc_fprintf(GRCC_Stderr, "no vertices: ");
02945         grcc_fprintf(GRCC_Stderr, "nExtern=%d, ndeg=%d\n", nExtern, ndeg);
02946     }
02947     ok = False;
02948 }
02949 if (ndeg % 2 == 0) {
02950     nEdges = ndeg/2;
02951 } else {
02952     if (prlevel > 0) {
02953         grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
02954         grcc_fprintf(GRCC_Stderr, "illegal total deg = %d (not even)\n", ndeg);
02955     }
02956     ok = False;
02957 }
02958 nNodes = nvrt + nExtern;
02959 if (nNodes >= GRCC_MAXNODES) {
02960     if (prlevel > 0) {
02961         grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
02962         grcc_fprintf(GRCC_Stderr, "too many nodes = %d\n", nNodes);
02963         grcc_fprintf(GRCC_Stderr, " nExtern=%d, nvrt=%d (GRCC_MAXNODES)\n",
02964                     nExtern, nvrt);
02965     }
02966     ok = False;
02967 }
02968 if (!ok) {
02969     prSProcess();
02970     erEnd("SProcess::SProcess: illegal input");
02971 }
02972 if (proc == NULL) {
02973     opt->beginProc(NULL);
02974 }
02975
02976 lp2 = tCouple - nExtern + 2;

```

```

02977     if (lp2 % 2 != 0 || lp2 < 0) {
02978         if (prlevel > 0) {
02979             grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: illegal loop: "
02980                 "tCouple=%d, nExtern=%d, 2*loop=%d\n",
02981                 tCouple, nExtern, lp2);
02982         }
02983         ok = False;
02984     }
02985     loop = lp2/2;
02986
02987     // stack for assignment
02988     if (opt->values[GRCC_OPT_Step] == GRCC_AGraph) {
02989         astack = new AStack(GRCC_MAXNSTACK, GRCC_MAXESTACK);
02990     }
02991
02992     // save to PNodeClass object
02993     pnclass = new PNodeClass(this, nNodes, nclass, cdeg, ctyp, ptcl, cpl, cnum, cmind, cmaxd);
02994 }
02995
02996 //-----
02997 SProcess::SProcess(Model *mdl, Process *prc, Options *opts, int sid, int *clst, int ncls, NCInput
    *cls)
02998 {
02999     // Construct a Subprocess object
03000     //     mdl          : model
03001     //     prc          : process (may be NULL)
03002     //     opts         : options
03003     //     sid          : id of the sprocess
03004     //     ncls         : the number of classes
03005     //     cls          : input data of classes
03006
03007     int j, cp, lp2, ndeg, nvrt, tcpl0;
03008     bool ok;
03009
03010     if (ncls < 1) {
03011         grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: no class %d\n", ncls);
03012         erEnd("SProcess::SProcess: no Class");
03013     }
03014     id      = sid;
03015     model   = mdl;
03016     proc    = prc;
03017     opt     = opts;
03018     nclass  = ncls;
03019
03020     nNodes  = 0;
03021     nEdges  = 0;
03022     nExtern = 0;
03023     tCouple = 0;
03024     loop    = 0;
03025     mgraph  = NULL;
03026     egraph  = NULL;
03027     astack  = NULL;
03028
03029     mgrcount = 0;
03030     agrcount = 0;
03031     extperm  = 1;
03032
03033     // the results of the graph generation
03034     nMGraphs = 0; // the number of generated M-graphs
03035     nMOPI    = 0; // the number of 1PI M-graphs
03036     wMGraphs.setValue(0,1); // the weighted sum of M-graphs
03037     wMOPI.setValue(0,1); // the weighted sum of 1PI M-graphs
03038
03039     nAGraphs = 0; // the number of generated A-graphs
03040     nAOPI    = 0; // the number of 1PI A-graphs
03041     wAGraphs.setValue(0,1); // the weighted sum of A-graphs
03042     wAOPI.setValue(0,1); // the weighted sum of 1PI A-graphs
03043
03044     // check input
03045     nNodes = 0;
03046     nExtern = 0;
03047
03048     ndeg    = 0;
03049     nvrt    = 0;
03050     ok      = True;
03051
03052     tcpl0 = 0;
03053     for (j = 0; j < model->ncouple; j++) {
03054         clist[j] = clst[j];
03055         tcpl0 += clist[j];
03056     }
03057     for (j = 0; j < nclass; j++) {
03058         cp = cls[j].cple;
03059         if (cp > 0) {
03060             lp2 = cls[j].cple - cls[j].cldeg + 2;
03061             if (lp2 % 2 != 0 || lp2 < 0) {
03062                 if (prlevel > 0) {

```

```

03063         grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: illegal loop: "
03064                     "class %d, cp=%d, deg=%d, lp2=%d\n",
03065                     j, cp, cls[j].cldeg, lp2);
03066     }
03067     ok = False;
03068 }
03069 }
03070 tCouple += cp*cls[j].clnum;
03071 ndeg    += cls[j].cldeg*cls[j].clnum;
03072
03073 if (!isATEXternal(cls[j].cltyp)) {
03074     nvrt += cls[j].clnum;
03075 } else if (isATEXternal(cls[j].cltyp)) {
03076     if (model->normalParticle(cls[j].ptcl) == 0) {
03077         if (prlevel > 0) {
03078             grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
03079             grcc_fprintf(GRCC_Stderr, "illegal particle code: ");
03080             grcc_fprintf(GRCC_Stderr, "code: class %d, ptcl=%d\n", j, cls[j].ptcl);
03081         }
03082         ok = False;
03083     } else {
03084         nExtern += cls[j].clnum;
03085     }
03086     extperm *= factorial(cls[j].clnum);
03087
03088 } else {
03089     if (prlevel > 0) {
03090         grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: illegal type: ");
03091         grcc_fprintf(GRCC_Stderr, "class %d, type=%d\n", j, cls[j].cltyp);
03092     }
03093     ok = False;
03094 }
03095 }
03096 if (tcpl0 != tCouple) {
03097     if (prlevel > 0) {
03098         grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
03099         grcc_fprintf(GRCC_Stderr, "illegal coupling constants:");
03100         grcc_fprintf(GRCC_Stderr, " %d != %d\n", tcpl0, tCouple);
03101     }
03102     ok = False;
03103 }
03104 if (ndeg == nExtern) {
03105     if (prlevel > 0) {
03106         grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
03107         grcc_fprintf(GRCC_Stderr, "no vertices: ");
03108         grcc_fprintf(GRCC_Stderr, "nExtern=%d, ndeg=%d\n", nExtern, ndeg);
03109     }
03110     ok = False;
03111 }
03112 if (ndeg % 2 == 0) {
03113     nEdges = ndeg/2;
03114 } else {
03115     if (prlevel > 0) {
03116         grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
03117         grcc_fprintf(GRCC_Stderr, "illegal total deg = %d (not even)\n", ndeg);
03118     }
03119     ok = False;
03120 }
03121 nNodes = nvrt + nExtern;
03122 if (nNodes >= GRCC_MAXNODES) {
03123     if (prlevel > 0) {
03124         grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: ");
03125         grcc_fprintf(GRCC_Stderr, "too many nodes = %d\n", nNodes);
03126         grcc_fprintf(GRCC_Stderr, "    nExtern=%d, nvrt=%d (GRCC_MAXNODES)\n",
03127                     nExtern, nvrt);
03128     }
03129     ok = False;
03130 }
03131 if (!ok) {
03132     erEnd("SProcess::SProcess: illegal input");
03133 }
03134 if (proc == NULL) {
03135     opt->beginProc(NULL);
03136 }
03137
03138 lp2 = tCouple - nExtern + 2;
03139 if (lp2 % 2 != 0 || lp2 < 0) {
03140     if (prlevel > 0) {
03141         grcc_fprintf(GRCC_Stderr, "*** SProcess::SProcess: illegal loop: "
03142                     "tCouple=%d, nExtern=%d, 2*loop=%d\n",
03143                     tCouple, nExtern, lp2);
03144     }
03145     ok = False;
03146 }
03147 loop = lp2/2;
03148
03149 // stack for assignment

```

```

03150     if (opt->values[GRCC_OPT_Step] == GRCC_AGraph) {
03151         astack = new AStack(GRCC_MAXNSTACK, GRCC_MAXESTACK);
03152     }
03153
03154     // save to PNodeClass object
03155     pnclass = new PNodeClass(this, nNodes, nclass, cls);
03156 }
03157
03158 //-----
03159 SProcess::~SProcess(void)
03160 {
03161     delete pnclass;
03162     delete astack;
03163     delete mgraph;
03164     model = NULL;
03165 }
03166
03167 //-----
03168 void SProcess::prSProcess(void)
03169 {
03170     grcc_fprintf(GRCC_Stdout, "\n");
03171     grcc_fprintf(GRCC_Stdout, "+++ Subprocess %d, class=%d\n", id, nclass);
03172     if (pnclass != NULL) {
03173         pnclass->prPNodeClass();
03174     } else {
03175         grcc_fprintf(GRCC_Stdout, " pnclass = NULL\n");
03176     }
03177 }
03178
03179 //-----
03180 BigInt SProcess::generate(void)
03181 {
03182     // Entry point of Feynman graph generation
03183     int cldeg[GRCC_MAXNODES], clnum[GRCC_MAXNODES], cltyp[GRCC_MAXNODES];
03184     int cmind[GRCC_MAXNODES], cmaxd[GRCC_MAXNODES];
03185     int ncl;
03186     BigInt ng;
03187
03188     ncl = toMNodeClass(cltyp, cldeg, clnum, cmind, cmaxd);
03189
03190     opt->beginSubProc(this);
03191
03192     mgraph = new MGraph(id, ncl, cldeg, clnum, cltyp, cmind, cmaxd, opt);
03193
03194     ng = mgraph->generate();
03195
03196     opt->endSubProc();
03197
03198     delete mgraph;
03199     mgraph = NULL;
03200
03201     return ng;
03202 }
03203
03204 //-----
03205 int SProcess::toMNodeClass(int *cltyp, int *cldeg, int *clnum, int *cmind, int *cmaxd)
03206 {
03207     int j;
03208
03209     for (j = 0; j < nclass; j++) {
03210         if (isATExternal(pnclass->type[j])) {
03211             cltyp[j] = -1;
03212         } else {
03213             cltyp[j] = (pnclass->couple[j]-pnclass->deg[j]+2)/2;
03214         }
03215         cldeg[j] = pnclass->deg[j];
03216         clnum[j] = pnclass->count[j];
03217         cmind[j] = pnclass->cminddeg[j];
03218         cmaxd[j] = pnclass->cmaxdeg[j];
03219     }
03220     return nclass;
03221 }
03222
03223 //-----
03224 void SProcess::assign(MGraph *mgr)
03225 {
03226     PNodeClass *pnc;
03227
03228     pnc = match(mgr);
03229
03230     agraph = new Assign(this, mgr, pnc);
03231
03232 #ifdef CHECK
03233     agraph->checkAG("SProcess::assign");
03234 #endif
03235     delete pnc;
03236     delete agraph;

```

```

03237     agraph = NULL;
03238 }
03239
03240 //-----
03241 PNodeClass *SProcess::match(MGraph *mgr)
03242 {
03243     PNodeClass *pnc = NULL;
03244
03245     int typ[GRCC_MAXNODES], dgs[GRCC_MAXNODES];
03246     int cnt[GRCC_MAXNODES], ptcl[GRCC_MAXNODES];
03247     int cpl[GRCC_MAXNODES];
03248     int n2m[GRCC_MAXNODES];
03249     int cmind[GRCC_MAXNODES], cmaxd[GRCC_MAXNODES];
03250     int j, k, l, nd, mc, mcl, mclss;
03251
03252     if (mgr->nNodes != nNodes) {
03253         if (prlevel > 0) {
03254             gcc_fprintf(GRCC_Stderr, "*** SProcess::match: different nNodes=%d != %d\n",
03255                 nNodes, mgr->nNodes);
03256         }
03257         erEnd("SProcess::match: different nNodes");
03258     }
03259
03260     mcl = toMNodeClass(typ, dgs, cnt, cmind, cmaxd);
03261
03262     nd = 0;
03263     for (j = 0; j < mcl; j++) {
03264         for (k = 0; k < cnt[j]; k++, nd++) {
03265             n2m[nd] = j;
03266         }
03267     }
03268
03269     nd = 0;
03270     mclss = mgr->curcl->nClasses;
03271     for (j = 0; j < mclss; j++) {
03272         for (k = 0; k < mgr->curcl->clist[j]; k++, nd++) {
03273             mc = mgr->nodes[nd]->clss;
03274             if (mc != n2m[nd]) {
03275                 if (prlevel > 0) {
03276                     gcc_fprintf(GRCC_Stderr, "*** SProcess::match: ");
03277                     gcc_fprintf(GRCC_Stderr, "inconsistent class\n");
03278                     for (l = 0; l < nNodes; l++) {
03279                         gcc_fprintf(GRCC_Stderr, "    %d: %d, %d\n",
03280                             j, mgr->nodes[l]->clss, n2m[l]);
03281                     }
03282                 }
03283                 erEnd("SProcess::match: inconsistent class");
03284             }
03285         }
03286     }
03287     for (j = 0; j < mcl; j++) {
03288         dgs[j] = pnclass->deg[j];
03289         typ[j] = pnclass->type[j];
03290         ptcl[j] = pnclass->particle[j];
03291         cnt[j] = pnclass->count[j];
03292         cpl[j] = pnclass->couple[j];
03293         cmind[j] = pnclass->cminddeg[j];
03294         cmaxd[j] = pnclass->cmaxdeg[j];
03295     }
03296     pnc = new PNodeClass(this, nNodes, mcl, dgs, typ, ptcl, cpl, cnt, cmind, cmaxd);
03297
03298     return pnc;
03299 }
03300
03301 //-----
03302 void SProcess::resultMGraph(BigInt nmgraphs, Fraction mwsum, BigInt nmopi, Fraction mwopi)
03303 {
03304     nMGraphs += nmgraphs;
03305     nMOPI += nmopi;
03306     wMGraphs.add(mwsum);
03307     wMOPI.add(mwopi);
03308 }
03309
03310 //-----
03311 void SProcess::resultAGraph(BigInt nmgraphs, Fraction mwsum, BigInt nmopi, Fraction mwopi)
03312 {
03313     nAGraphs += nmgraphs;
03314     nAOPI += nmopi;
03315     wAGraphs.add(mwsum);
03316     wAOPI.add(mwopi);
03317 }
03318
03319 //-----
03320 void SProcess::endMGraph(MGraph *mgr)
03321 {
03322     egraph = mgr->egraph;
03323 }

```



```

03324     if (opt->values[GRCC_OPT_Step] != GRCC_MGraph) {
03325         assign(mgr);
03326     }
03327 }
03328
03329 //-----
03330 void SProcess::endAGraph(EGraph *egr)
03331 {
03332     egraph = egr;
03333 }
03334
03335 //=====
03336 // class Process
03337 //-----
03338
03339 Process::Process(int pid, Model *modl, Options *optn, int nini, int *intlPrt, int nfin, int *finlPrt,
03340 int *coupling)
03341 {
03342     // Define a process and construct a set of sprocesses
03343     // model      : Model object
03344     // optn       : Option object
03345     // intlPrt    : list of particle-id of intlPart particles
03346     // finalPart  : list of particle-id of final particles
03347     // coupling   : list of coupling constants
03348
03349     int j, lp2;
03350     Bool ok;
03351     char buff[MAXSTR];
03352
03353     id      = pid;
03354     model    = modl;
03355     opt      = optn;
03356     ninitl   = nini;
03357     nfinal   = nfin;
03358     intlPart = intdup(ninitl, intlPrt);
03359     finalPart = intdup(nfinal, finlPrt);
03360
03361     // count total number of coupling constants.
03362     cttotal = 0;
03363     for (j = 0; j < GRCC_MAXNCPLG; j++) {
03364         if (j < model->ncouple) {
03365             clist[j] = coupling[j];
03366             cttotal += coupling[j];
03367         } else {
03368             clist[j] = 0;
03369         }
03370     }
03371     if (cttotal < 1) {
03372         erEnd("Process: coupling constant = 0");
03373     }
03374
03375     nExtern = 0;
03376     loop    = -1;
03377     maxnlegs = 0;
03378     mgrcount = 0;
03379     agrcount = 0;
03380
03381     // table of sprocesses
03382     nSubproc = 0;
03383     sproc    = NULL;
03384
03385     // the results of the graph generation
03386     nMGraphs = 0; // the number of generated M-graphs
03387     nMOPI    = 0; // the number of 1PI M-graphs
03388     wMGraphs.setValue(0,1); // the weighted sum of M-graphs
03389     wMOPI.setValue(0,1); // the weighted sum of 1PI M-graphs
03390
03391     nAGraphs = 0; // the number of generated A-graphs
03392     nAOPI    = 0; // the number of 1PI A-graphs
03393     wAGraphs.setValue(0,1); // the weighted sum of A-graphs
03394     wAOPI.setValue(0,1); // the weighted sum of 1PI A-graphs
03395
03396     ok = True;
03397
03398     // numbers
03399     nExtern = ninitl + nfinal;
03400     maxnlegs = Max(nExtern, model->maxnlegs);
03401
03402     // count loops
03403     lp2 = cttotal - nExtern + 2;
03404     if (lp2 % 2 != 0) {
03405         if (prlevel > 0) {
03406             grcc_fprintf(GRCC_Stderr, "*** cannot generate : 2*loop is odd : "
03407                 "2*loop = %d, cttotal=%d, nExtern=%d\n",
03408                 lp2, cttotal, nExtern);
03409         }
03410         ok = False;
03411     }

```

```

03410     } else if (lp2 < 0) {
03411         if (prlevel > 0) {
03412             grcc_fprintf(GRCC_Stderr, "*** cannot make a connected graph : "
03413                 "2*loop=%d, ctotat=%d, nExtern=%d\n",
03414                 lp2, ctotat, nExtern);
03415         }
03416         ok = False;
03417     }
03418
03419     loop = lp2 / 2;
03420
03421     if (!ok) {
03422         if (prlevel > 0) {
03423             grcc_fprintf(GRCC_Stderr, "*** Process: illegal input: lp2 = %d\n", lp2);
03424         }
03425         erEnd("Process: illegal input");
03426     }
03427
03428     if (opt->out->outgrp != NULL) {
03429         snprintf(buff, MAXSTR, "out%d.prp", pid);
03430         prProcessP(buff);
03431     }
03432
03433     // construct sprocesses
03434     mkSProcess();
03435 }
03436
03437 //-----
03438 Process::~Process(void)
03439 {
03440     initlPart = delintdup(initlPart);
03441     finalPart = delintdup(finalPart);
03442     for (int j = 0; j < nSubproc; j++) {
03443         delete sptbl[j];
03444     }
03445     // delete sproc;
03446 }
03447
03448 //-----
03449 void Process::prProcess(void)
03450 {
03451     grcc_fprintf(GRCC_Stdout, "+++ process options : OPI = %d, (Step) = %d, coupling=%d: ",
03452         opt->values[GRCC_OPT_1PI], opt->values[GRCC_OPT_Step], ctotat);
03453     prIntArray(model->ncouple, clist, "\n");
03454 }
03455
03456 //-----
03457 void Process::prProcessP(const char *fname)
03458 {
03459     FILE *fp;
03460
03461     if ((fp = fopen(fname, "w")) == NULL) {
03462         grcc_fprintf(GRCC_Stderr, "*** cannot open \"%s\"\n", fname);
03463         return;
03464     }
03465     fprintf(fp, "Process = {\n");
03466     outProcP(fp);
03467     fprintf(fp, "};\n");
03468     fclose(fp);
03469 }
03470
03471 //-----
03472 void Process::outProcP(FILE *fp)
03473 {
03474     int j;
03475
03476     if (model == NULL) {
03477         return;
03478     }
03479     fprintf(fp, "    \"Model\": \"%s\\\", \n", model->name);
03480     fprintf(fp, "    \"Initial\": [");
03481     for (j = 0; j < ninitl; j++) {
03482         if (j != 0) {
03483             fprintf(fp, ", ");
03484         }
03485         fprintf(fp, "\"%s\\\"", model->particleName(initlPart[j]));
03486     }
03487     fprintf(fp, "], \n");
03488     fprintf(fp, "    \"Final\": [");
03489     for (j = 0; j < nfinal; j++) {
03490         if (j != 0) {
03491             fprintf(fp, ", ");
03492         }
03493         fprintf(fp, "\"%s\\\"", model->particleName(finalPart[j]));
03494     }
03495     fprintf(fp, "], \n");
03496     fprintf(fp, "    \"Couple\": [");

```

```

03497     for (j = 0; j < model->ncouple; j++) {
03498         if (j != 0) {
03499             fprintf(fp, ", ");
03500         }
03501         fprintf(fp, "%d", clist[j]);
03502     }
03503     fprintf(fp, "],\n");
03504     fprintf(fp, "    \\"Options\\": {\n");
03505     for (j = 0; j < GRCC_OPT_Size; j++) {
03506         if (j == GRCC_OPT_Outgrf || j == GRCC_OPT_Outgrp) {
03507             continue;
03508         }
03509         fprintf(fp, "        \\"%s\\":%d,\n", optDef[j].name, opt->values[j]);
03510     }
03511     fprintf(fp, "        \\"%s\\":", optDef[GRCC_OPT_Outgrf].name);
03512     if (opt->out->outgrf != NULL) {
03513         fprintf(fp, "\\"%s\\",\n", opt->out->outgrf);
03514     } else {
03515         fprintf(fp, "0,\n");
03516     }
03517     fprintf(fp, "        \\"%s\\":", optDef[GRCC_OPT_Outgrp].name);
03518     if (opt->out->outgrp != NULL) {
03519         fprintf(fp, "\\"%s\\",\n", opt->out->outgrp);
03520     } else {
03521         fprintf(fp, "0,\n");
03522     }
03523     fprintf(fp, "    }\n");
03524 }
03525
03526 //-----
03527 void Process::mkSProcess(void)
03528 {
03529     // Construct sprocesses
03530     // Each sprocess is a set of nodes whose degree and order of
03531     // coupling constants are determined.
03532     // Only the total coupling constants are considered here even if
03533     // two or more coupling constants are defined in the model
03534
03535     NCInput cls[GRCC_MAXMINTERACT];
03536     int r, j, k, lin2, nvtx, nleg, nc, n0;
03537     int nl[GRCC_MAXMINTERACT];
03538     double proct0 = 0, proct1 = 0;
03539
03540     if (model->ncplgcp < 1) {
03541         erEnd("function 'addInteractionEnd' has not been called");
03542     }
03543     // output file to start process
03544     opt->beginProc(this);
03545
03546     // starting time
03547     Second(&proct0);
03548
03549     // generate all possible number of {nl[i]},
03550     // \sum_i nl[i]*cplgcp[i] = (total number of coupling constants)
03551
03552     r = -1;
03553     nSubproc = 0;
03554     ngraphs = 0;
03555     nopi = 0;
03556     wgraphs = 0;
03557     wopi = 0;
03558
03559     // for partitions of (leg, order)
03560
03561     while (nextPart(ctotal, model->ncplgcp, model->cplgcp, nl, &r)) {
03562         // count the total number of vertices and legs.
03563         nvtx = 0;
03564         nleg = 0;
03565         for (j = 0; j < model->ncplgcp; j++) {
03566             nvtx += nl[j];
03567             nleg += nl[j]*model->cplglg[j];
03568         }
03569
03570         // count loops
03571         lin2 = nExtern + nleg;
03572         if (lin2 % 2 != 0) {
03573             continue;
03574         }
03575         loop = lin2/2 - nExtern - nvtx + 1;
03576         if (loop < 0) {
03577             continue;
03578         }
03579
03580         nc = 0;
03581         if (opt->values[GRCC_OPT_SymmInitial]) {
03582             n0 = nc;
03583             for (j = 0; j < ninitl; j++) {

```

```

03584         for (k = n0; k < nc; k++) {
03585             if (cls[k].ptcl == initlPart[j]) {
03586                 cls[k].clnum++;
03587                 break;
03588             }
03589         }
03590         if (k >= nc) {
03591             cls[nc].ptcl = initlPart[j];
03592             cls[nc].clnum = 1;
03593             cls[nc].cple = 0;
03594             cls[nc].clnum = 1;
03595             cls[nc].cldeg = 1;
03596             cls[nc].cltyp = GRCC_AT_Initial;
03597             cls[nc].ptcl = initlPart[j];
03598             cls[nc].cmind
03599                 = model->particles[Abs(cls[nc].ptcl)]->cminddeg;
03600             cls[nc].cmaxd
03601                 = model->particles[Abs(cls[nc].ptcl)]->cmaxdeg;
03602             if (opt->values[GRCC_OPT_NoExtSelf] > 0 && nExtern > 2) {
03603                 cls[nc].cmind = Max(cls[nc].cmind, 3);
03604             }
03605             nc++;
03606         }
03607     }
03608 } else {
03609     for (j = 0; j < ninitl; j++) {
03610         cls[nc].ptcl = initlPart[j];
03611         cls[nc].clnum = 1;
03612         cls[nc].cple = 0;
03613         cls[nc].cldeg = 1;
03614         cls[nc].cltyp = GRCC_AT_Initial;
03615         cls[nc].cmind = model->particles[Abs(cls[nc].ptcl)]->cminddeg;
03616         cls[nc].cmaxd = model->particles[Abs(cls[nc].ptcl)]->cmaxdeg;
03617         if (opt->values[GRCC_OPT_NoExtSelf] > 0 && nExtern > 2) {
03618             cls[nc].cmind = Max(cls[nc].cmind, 3);
03619         }
03620         nc++;
03621     }
03622 }
03623 if (opt->values[GRCC_OPT_SymmFinal]) {
03624     n0 = nc;
03625     for (j = 0; j < nfinal; j++) {
03626         for (k = n0; k < nc; k++) {
03627             if (cls[k].ptcl == model->antiParticle(finalPart[j])) {
03628                 cls[k].clnum++;
03629                 break;
03630             }
03631         }
03632         if (k >= nc) {
03633             cls[nc].ptcl = model->antiParticle(finalPart[j]);
03634             cls[nc].clnum = 1;
03635             cls[nc].cple = 0;
03636             cls[nc].cldeg = 1;
03637             cls[nc].cltyp = GRCC_AT_Final;
03638             cls[nc].cmind
03639                 = model->particles[Abs(cls[nc].ptcl)]->cminddeg;
03640             cls[nc].cmaxd
03641                 = model->particles[Abs(cls[nc].ptcl)]->cmaxdeg;
03642             if (opt->values[GRCC_OPT_NoExtSelf] > 0 && nExtern > 2) {
03643                 cls[nc].cmind = Max(cls[nc].cmind, 3);
03644             }
03645             nc++;
03646         }
03647     }
03648 } else {
03649     for (j = 0; j < nfinal; j++) {
03650         cls[nc].ptcl = model->antiParticle(finalPart[j]);
03651         cls[nc].cldeg = 1;
03652         cls[nc].cple = 0;
03653         cls[nc].clnum = 1;
03654         cls[nc].cltyp = GRCC_AT_Final;
03655         cls[nc].cmind = model->particles[Abs(cls[nc].ptcl)]->cminddeg;
03656         cls[nc].cmaxd = model->particles[Abs(cls[nc].ptcl)]->cmaxdeg;
03657         if (opt->values[GRCC_OPT_NoExtSelf] > 0 && nExtern > 2) {
03658             cls[nc].cmind = Max(cls[nc].cmind, 3);
03659         }
03660         nc++;
03661     }
03662 }
03663 for (j = 0; j < model->ncplgcp; j++) {
03664     if (nl[j] > 0) {
03665         cls[nc].ptcl = 0;
03666         cls[nc].cple = model->cplgcp[j];
03667         cls[nc].cldeg = model->cplglg[j];
03668         cls[nc].clnum = nl[j];
03669         // number of loops
03670         cls[nc].cltyp = (model->cplgcp[j] - model->cplglg[j] + 2)/2;

```

```

03671         cls[nc].cmindeg = 0;
03672         cls[nc].cmaxdeg = 0;
03673         nc++;
03674     }
03675 }
03676 // create a sprocess ??? to be rewritten
03677 sproc = new SProcess(model, this, opt, nSubproc, clist, nc, cls);
03678
03679 // list of sprocesses
03680 if (nSubproc >= GRCC_MAXSUBPROCS) {
03681     erEnd("Subclass: too many sprocesses (GRCC_MAXSUBPROCS)");
03682 }
03683 sptbl[nSubproc++] = sproc;
03684
03685 // generate M-graphs
03686 sproc->generate();
03687
03688 // summation of the numbers and weighted numbers of graphs
03689 nMGraphs += sproc->nMGraphs;
03690 nMOPI += sproc->nMOPI;
03691 wMGraphs.add(sproc->wMGraphs);
03692 wMOPI.add(sproc->wMOPI);
03693
03694 nAGraphs += sproc->nAGraphs;
03695 nAOPI += sproc->nAOPI;
03696 wAGraphs.add(sproc->wAGraphs);
03697 wAOPI.add(sproc->wAOPI);
03698
03699 if (nAGraphs > 0) {
03700     ngraphs = nAGraphs;
03701 } else {
03702     ngraphs = nAGraphs;
03703 }
03704 // delete sproc;
03705 // sproc = NULL;
03706 }
03707
03708 // ending time
03709 Second(&proct1);
03710 sec = proct1-proct0;
03711
03712 // output file to finish process
03713 opt->endProc();
03714 }
03715
03716 //*****
03717 // mgraph.cc
03718 //=====
03719 //-----
03720 MNodeClass::MNodeClass(int nnodes, int nclasses)
03721 {
03722     // Input
03723     // nnodes : possible maximal number of nodes
03724     // nclasses : possible maximal number of classes
03725
03726     int j, k;
03727
03728     nNodes = nnodes;
03729     nClasses = nclasses;
03730     maxdeg = 0;
03731     for (j = 0; j < nNodes; j++) {
03732         for (k = 0; k < nNodes; k++) {
03733             clmat[j][k] = 0;
03734         }
03735         clist[j] = 0;
03736         ndcl[j] = 0;
03737         flist[j] = 0;
03738         clord[j] = 0;
03739         cmindeg[j] = 0;
03740         cmaxdeg[j] = 0;
03741     }
03742     flist[nNodes] = 0;
03743     flg0 = 0;
03744     flg1 = 0;
03745     flg2 = 0;
03746 }
03747
03748 //-----
03749 MNodeClass::~MNodeClass(void)
03750 {
03751 }
03752
03753 //=====
03754 // class MNodeClass
03755 //-----
03756 void MNodeClass::init(int *cl, int mxdeg, int **adjmat)
03757 {

```

```

03758 // initialize the object
03759 // Argument
03760 //   cl[j]           : list of number of nodes in the j-th class
03761 //   mxdeg           : maximum degree to nodes
03762 //   adjmat[j][k]    : adjacency matrix
03763
03764 int j;
03765
03766 for (j = 0; j < nClasses; j++) {
03767     clist[j] = cl[j];
03768     clord[j] = j;
03769 }
03770 mxdeg = mxdeg;
03771 mkNdCl();
03772 mkClMat(adjmat);
03773 mkFlist();
03774 flg0 = 0;
03775 flg1 = 0;
03776 flg2 = 0;
03777 }
03778
03779 //-----
03780 void MNodeClass::copy(MNodeClass* mnc)
03781 {
03782     // copy MNodeClass 'mnc' to this object
03783
03784     int j, k;
03785
03786     nClasses = mnc->nClasses;
03787     for (k = 0; k < nClasses; k++) {
03788         clist[k] = mnc->clist[k];
03789     }
03790     mxdeg = mnc->mxdeg;
03791     for (j = 0; j < nNodes; j++) {
03792         ndcl[j] = mnc->ndcl[j];
03793         clord[j] = mnc->clord[j];
03794         for (k = 0; k < nClasses; k++) {
03795             clmat[j][k] = mnc->clmat[j][k];
03796         }
03797     }
03798     for (k = 0; k < nClasses+1; k++) {
03799         flist[k] = mnc->flist[k];
03800     }
03801 }
03802
03803 //-----
03804 void MNodeClass::mkFlist(void)
03805 {
03806     // Construct flist
03807     //   The set of nodes in class 'c' is [flist[c],...,flist[c+1]-1]
03808
03809     int j, f;
03810
03811     f = 0;
03812     for (j = 0; j < nClasses; j++) {
03813         flist[j] = f;
03814         f += clist[j];
03815     }
03816     flist[nClasses] = f;
03817 }
03818
03819 //-----
03820 void MNodeClass::mkNdCl(void)
03821 {
03822     // Construct ndcl
03823     //   ndcl[nd] = (the class id in which node 'nd' belongs)
03824
03825     int c, k;
03826     int nd = 0;
03827
03828     for (c = 0; c < nClasses; c++) {
03829         for (k = 0; k < clist[c]; k++) {
03830             ndcl[nd++] = c;
03831         }
03832     }
03833 }
03834
03835 //-----
03836 int MNodeClass::clCmp(int nd0, int nd1, int cn)
03837 {
03838     // Comparison of two nodes 'nd0' and 'nd1'
03839     //   in lexicographic ordering of [class, connection configuration]
03840
03841     int cmp;
03842     // Wether two nodes are in a same class or not.
03843
03844     cmp = ndcl[nd0] - ndcl[nd1];

```

```

03845     if (cmp != 0) {
03846         return cmp;
03847     }
03848
03849     // Sign '-' signifies the reverse ordering
03850     cmp = - cmpMNCArray(clmat[nd0], clmat[nd1], cn);
03851     if (cmp != 0) {
03852         return cmp;
03853     }
03854     return cmp;
03855 }
03856
03857 //-----
03858 void MNodeClass::mkClMat(int **adjmat)
03859 {
03860     // Construct a matrix 'clmat[nd][tc]' which is the number
03861     // of edges connecting 'nd' and all nodes in class 'tc'.
03862
03863     int nd, td, tc;
03864
03865     for (nd = 0; nd < nNodes; nd++) {
03866         for (td = 0; td < nNodes; td++) {
03867             tc = ndcl[td]; // another node
03868             if (nd == td) {
03869                 clmat[nd][tc] += CLWIGHTD(adjmat[nd][td]);
03870             } else {
03871                 clmat[nd][tc] += CLWIGHTO(adjmat[nd][td]);
03872             }
03873         }
03874     }
03875 }
03876
03877 //-----
03878 void MNodeClass::incMat(int nd, int td, int val)
03879 {
03880     // Increase the number of edges by 'val' between 'nd' and 'td'.
03881
03882     int tdc = ndcl[td]; // class of 'td'
03883     clmat[nd][tdc] += val; // modify matrix 'clmat'.
03884 }
03885
03886 //-----
03887 void MNodeClass::printMat(void)
03888 {
03889     // Print configuration matrix.
03890
03891     int j1, j2;
03892
03893     grcc_fprintf(GRCC_Stdout, "\n");
03894
03895     grcc_fprintf(GRCC_Stdout, "nClasses=%d", nClasses);
03896     grcc_fprintf(GRCC_Stdout, " clord="); prIntArray(nClasses, clord, "");
03897     grcc_fprintf(GRCC_Stdout, " flist="); prIntArray(nClasses, flist, "\n");
03898     grcc_fprintf(GRCC_Stdout, "flg = (%d, %d, %d)\n", flg0, flg1, flg2);
03899
03900     // the first line
03901     grcc_fprintf(GRCC_Stdout, "nd: cl: ");
03902     for (j2 = 0; j2 < nClasses; j2++) {
03903         grcc_fprintf(GRCC_Stdout, "%2d ", j2);
03904     }
03905     grcc_fprintf(GRCC_Stdout, "\n");
03906
03907     // print raw
03908     for (j1 = 0; j1 < nNodes; j1++) {
03909         grcc_fprintf(GRCC_Stdout, "%2d; %2d: [", j1, ndcl[j1]);
03910         for (j2 = 0; j2 < nClasses; j2++) {
03911             grcc_fprintf(GRCC_Stdout, " %2d", clmat[j1][j2]);
03912         }
03913         grcc_fprintf(GRCC_Stdout, "]\n");
03914     }
03915 }
03916
03917 //-----
03918 int MNodeClass::cmpMNCArray(int *a0, int *a1, int ma)
03919 {
03920     int j, k;
03921
03922     for (k = 0; k < ma; k++) {
03923         j = clord[k];
03924         if (a0[j] < a1[j]) {
03925             return -1;
03926         } else if (a0[j] > a1[j]) {
03927             return 1;
03928         }
03929     }
03930     return 0;
03931 }

```

```

03932
03933 //-----
03934 void MNodeClass::reorder(MGraph *mg)
03935 {
03936     int flg[GRCC_MAXNODES];
03937     int co, cn;
03938     int f0, f1, f2;
03939
03940     f0 = 0;
03941     f1 = 0;
03942     f2 = 0;
03943     for (co = 0; co < nClasses; co++) {
03944         cn = flist[co];
03945         if (mg->nodes[cn]->freelg < 1) {
03946             flg[co] = 0;
03947             f0++;
03948         } else if (mg->nodes[cn]->freelg < mg->nodes[cn]->deg) {
03949             flg[co] = mg->nodes[cn]->deg + maxdeg*clist[co];
03950             f1++;
03951         } else {
03952             flg[co] = maxdeg*(mg->nodes[cn]->deg + maxdeg*clist[co]);
03953             f2++;
03954         }
03955     }
03956     if (f0 < nClasses) {
03957         bsort(nClasses, clord, flg);
03958     }
03959     flg0 = f0;
03960     flg1 = f0 + f1;
03961     flg2 = f0 + f1 + f2;
03962 }
03963
03964 //=====
03965 // class MNode : nodes in MGraph
03966 //-----
03967 MNode::MNode(int vid, int vclss, NCInput *mgi)
03968 {
03969     // Arguments
03970     // vid : identifier of the vertex
03971     // vdeg : degree of the node
03972     // vextlp : external node (-1) or looped vertex
03973     // vclss : class of the node
03974
03975     id = vid; // id of the node
03976     clss = vclss; // class to which the node belongs
03977     deg = mgi->cldeg; // degree of the node
03978     freelg = mgi->cldeg; // number of free legs
03979     extloop = mgi->cltyp; // external node or not
03980     cmindeg = mgi->cmin; // min(deg of connectable vertex)
03981     cmaxdeg = mgi->cmax; // max(deg of connectable vertex)
03982 }
03983
03984 //-----
03985 MNode::MNode(int vid, int vdeg, int vextlp, int vclss, int cmin, int cmax)
03986 {
03987     // Arguments
03988     // vid : identifier of the vertex
03989     // vdeg : degree of the node
03990     // vextlp : external node (-1) or looped vertex
03991     // vclss : class of the node
03992
03993     id = vid; // id of the node
03994     deg = vdeg; // degree of the node
03995     freelg = vdeg; // number of free legs
03996     clss = vclss; // class to which the node belongs
03997     extloop = vextlp; // external node or not
03998     cmindeg = cmin; // min(deg of connectable vertex)
03999     cmaxdeg = cmax; // max(deg of connectable vertex)
04000 }
04001
04002 //=====
04003 // class MGraph : scalar graph expressed by matrix form
04004 //-----
04005 MGraph::MGraph(int pid, int ncl, int *cldeg, int *clnum, int *cltyp, int *cmin, int *cmax, Options
04006 *opts)
04007 {
04008     int nn, ne, j, k;
04009     // initial conditions
04010     nClasses = ncl;
04011     clist = new int[ncl];
04012     opt = opts;
04013     mId = -1;
04014
04015     mindeg = -1;
04016     maxdeg = -1;
04017     ne = 0;

```



```

04018     nn = 0;
04019     for (j = 0; j < nClasses; j++) {
04020         clist[j] = clnum[j];
04021         nn += clnum[j];
04022         ne += cldeg[j]*clnum[j];
04023         if (mindeg < 0) {
04024             mindeg = cldeg[j];
04025         } else {
04026             mindeg = Min(mindeg, cldeg[j]);
04027         }
04028         if (maxdeg < 0) {
04029             maxdeg = cldeg[j];
04030         } else {
04031             maxdeg = Max(maxdeg, cldeg[j]);
04032         }
04033     }
04034     if (ne % 2 != 0) {
04035         if (prlevel > 0) {
04036             grcc_fprintf(GRCC_Stderr, "Sum of degrees are not even\n");
04037             for (j = 0; j < nClasses; j++) {
04038                 grcc_fprintf(GRCC_Stderr, "class %2d: %2d %2d %2d\n",
04039                     j, cldeg[j], clnum[j], cltyp[j]);
04040             }
04041         }
04042         erEnd("illegal degrees of nodes");
04043     }
04044     pId = pid;
04045     nNodes = nn;
04046     nodes = new MNode*[nNodes];
04047     group = new SGroup();
04048 #ifdef ORBITS
04049     orbits = new MORbits();
04050 #endif
04051
04052     nEdges = ne / 2;
04053     nLoops = nEdges - nNodes + 1;
04054
04055     egraph = new EGraph(nNodes, nEdges, maxdeg);
04056     nn = 0;
04057     nExtern = 0;
04058     for (j = 0; j < nClasses; j++) {
04059         for (k = 0; k < clist[j]; k++, nn++) {
04060             nodes[nn] = new MNode(nn, cldeg[j], cltyp[j], j, cmindeg[j], cmaxdeg[j]);
04061             egraph->setExtLoop(nn, cltyp[j]);
04062             if (cltyp[j] < 0) {
04063                 nExtern++;
04064             }
04065         }
04066     }
04067     egraph->endSetExtLoop();
04068
04069     init();
04070 }
04071
04072 //-----
04073 MGraph::MGraph(int pid, int ncl, NCInput *mgi, Options *opts)
04074 {
04075     int nn, ne, j, k;
04076
04077     // initial conditions
04078     nClasses = ncl;
04079     clist = new int[ncl];
04080     opt = opts;
04081     mId = -1;
04082
04083     mindeg = -1;
04084     maxdeg = -1;
04085     ne = 0;
04086     nn = 0;
04087     for (j = 0; j < nClasses; j++) {
04088         clist[j] = mgi[j].clnum;
04089         nn += mgi[j].clnum;
04090         ne += mgi[j].cldeg*mgi[j].clnum;
04091         if (mindeg < 0) {
04092             mindeg = mgi[j].cldeg;
04093         } else {
04094             mindeg = Min(mindeg, mgi[j].cldeg);
04095         }
04096         if (maxdeg < 0) {
04097             maxdeg = mgi[j].cldeg;
04098         } else {
04099             maxdeg = Max(maxdeg, mgi[j].cldeg);
04100         }
04101     }
04102     if (ne % 2 != 0) {
04103         if (prlevel > 0) {
04104             grcc_fprintf(GRCC_Stderr, "Sum of degrees are not even\n");

```

```

04105         for (j = 0; j < nClasses; j++) {
04106             grcc_fprintf(GRCC_Stderr, "class %2d: %2d %2d %2d\n",
04107                 j, mgi[j].cldeg, mgi[j].clnum, mgi[j].cltyp);
04108         }
04109     }
04110     erEnd("illegal degrees of nodes");
04111 }
04112 pid = pid;
04113 nNodes = nn;
04114 if (nNodes < 0) {
04115     grcc_fprintf(GRCC_Stdout, "*** nNodes = %d\n", nNodes);
04116     erEnd("MGraph::MGraph : nNodes < 0");
04117 }
04118 nodes = new MNode*[nNodes];
04119 group = new SGroup();
04120 #ifdef ORBITS
04121     orbits = new MOrbits();
04122 #endif
04123
04124     nEdges = ne / 2;
04125     nLoops = nEdges - nNodes + 1;
04126
04127     egraph = new EGraph(nNodes, nEdges, maxdeg);
04128     nn = 0;
04129     nExtern = 0;
04130     for (j = 0; j < nClasses; j++) {
04131         for (k = 0; k < clist[j]; k++, nn++) {
04132             nodes[nn] = new MNode(nn, j, mgi+j);
04133             egraph->setExtLoop(nn, mgi[j].cltyp);
04134             if (mgi[j].cltyp < 0) {
04135                 nExtern++;
04136             }
04137         }
04138     }
04139     egraph->endSetExtLoop();
04140
04141     init();
04142 }
04143
04144 //-----
04145 void MGraph::init(void)
04146 {
04147     // generated set of graphs
04148     cDiag = 0;
04149     c1PI = 0;
04150     cNoTadpole = 0;
04151     cNoTadBlock = 0;
04152     c1PINoTadBlock = 0;
04153
04154     // the current graph
04155     adjMat = newMat(nNodes, nNodes, 0);
04156     nsym = ToBigInt(0);
04157     esym = ToBigInt(0);
04158     wscon = Fraction(0, 1);
04159     wsopi = Fraction(0, 1);
04160
04161     // current node classification
04162     curcl = new MNodeClass(nNodes, nClasses);
04163
04164     // table of n-edge connected components
04165     mconn = new MConn(nNodes, nEdges);
04166
04167     // measures of efficiency
04168     ngen = 0;
04169     ngconn = 0;
04170
04171     #ifdef MONITOR
04172         nCallRefine = 0;
04173         discardRefine = 0;
04174         discardDisc = 0;
04175         discardIso = 0;
04176     #endif
04177
04178     // work space for isIsomorphic
04179     modmat = newMat(nNodes, nNodes, 0);
04180
04181     // work space for bisearchME
04182     bidef = newArray(nNodes, 0);
04183     bilow = newArray(nNodes, 0);
04184     bicol = newArray(nNodes, 0);
04185     bicount = 0;
04186 }
04187
04188 //-----
04189 MGraph::~MGraph(void)
04190 {
04191     int j;

```

```

04192
04193     // group->delGroup();
04194
04195     bicol = deleteArray(bicol);
04196     bilow = deleteArray(bilow);
04197     bidef = deleteArray(bidef);
04198
04199
04200     modmat = deleteMat(modmat, nNodes);
04201     delete mconn;
04202     delete curcl;
04203     adjMat = deleteMat(adjMat, nNodes);
04204 #ifdef ORBITS
04205     delete orbits;
04206 #endif
04207     delete egraph;
04208     delete group;
04209     for (j = 0; j < nNodes; j++) {
04210         delete nodes[j];
04211         nodes[j] = NULL;
04212     }
04213     delete[] nodes;
04214     delete[] clist;
04215
04216 }
04217
04218 //-----
04219 void MGraph::printAdjMat(MNodeClass *cl)
04220 {
04221     int j1, j2;
04222
04223     grcc_fprintf(GRCC_Stdout, " ");
04224     for (j2 = 0; j2 < nNodes; j2++) {
04225         grcc_fprintf(GRCC_Stdout, " %2d", j2);
04226     }
04227     grcc_fprintf(GRCC_Stdout, "\n");
04228     for (j1 = 0; j1 < nNodes; j1++) {
04229         grcc_fprintf(GRCC_Stdout, "%2d : [", j1);
04230         for (j2 = 0; j2 < nNodes; j2++) {
04231             grcc_fprintf(GRCC_Stdout, " %2d", adjMat[j1][j2]);
04232         }
04233         grcc_fprintf(GRCC_Stdout, "] %2d\n", cl->ndcl[j1]);
04234     }
04235 }
04236
04237 //-----
04238 void MGraph::print(void)
04239 {
04240     int j;
04241
04242     grcc_fprintf(GRCC_Stdout, "MGraph: pId=%d, cDiag=%ld, c1PI=%ld\n", pId, cDiag, c1PI);
04243     grcc_fprintf(GRCC_Stdout, "      nNodes=%d, nEdges=%d, nExtern=%d, nLoops=%d "
04244                 "mindeg=%d, maxdeg=%d, sym={%ld, %ld}\n",
04245                 nNodes, nEdges, nExtern, nLoops, mindeg, maxdeg, nsym, esym);
04246     grcc_fprintf(GRCC_Stdout, "      Nodes=%d\n", nNodes);
04247     for (j = 0; j < nNodes; j++) {
04248         grcc_fprintf(GRCC_Stdout, "          %2d: id=%d, deg=%d, clss=%d, extloop=%d, ",
04249                     j, nodes[j]->id, nodes[j]->deg, nodes[j]->clss,
04250                     nodes[j]->extloop);
04251         grcc_fprintf(GRCC_Stdout, "mind=%d, maxd=%d, freelg=%d\n",
04252                     nodes[j]->cmindeg, nodes[j]->cmaxdeg, nodes[j]->freelg);
04253     }
04254
04255     curcl->printMat();
04256     grcc_fprintf(GRCC_Stdout, "\n");
04257
04258     printAdjMat(curcl);
04259 }
04260
04261 //-----
04262 void MGraph::printPy(FILE *fp, long mId)
04263 {
04264     fprintf(fp, "#TGraph : (gseq, gid, nodes, amat)\n");
04265     fprintf(fp, " (%ld, (%d, %d),\n", mId, pId, -1);
04266     fprintf(fp, " [ # nodes: (cid, deg0, loop, part)\n");
04267     for (int j = 0; j < nNodes; j++) {
04268         fprintf(fp, "          (%2d,%2d,%2d,%2d), # %2d\n",
04269                 nodes[j]->clss, nodes[j]->deg, nodes[j]->extloop, 0, j);
04270     }
04271     fprintf(fp, " ],\n");
04272     fprintf(fp, " [\n");
04273     fprintf(fp, " #");
04274     for (int j2 = 0; j2 < nNodes; j2++) {
04275         fprintf(fp, " %4d", j2);
04276     }
04277     fprintf(fp, "\n");
04278     for (int j1 = 0; j1 < nNodes; j1++) {

```

```

04279         fprintf(fp, "    ");
04280         for (int j2 = 0; j2 < nNodes; j2++) {
04281             fprintf(fp, " %2d, ", adjMat[j1][j2]);
04282         }
04283         fprintf(fp, "],    # %2d\n", j1);
04284     }
04285     fprintf(fp, " ], # nodes\n");
04286     fprintf(fp, " [ # group of 2 elements\n");
04287     fprintf(fp, "    [],\n");
04288     fprintf(fp, " ], # group\n");
04289     fprintf(fp, ")\n");
04290 }
04291
04292 //-----
04293 Bool MGraph::isConnected(void)
04294 {
04295     // Check graph can be a connected one.
04296     // If a connected component without free leg is not the whole graph then
04297     // return False, otherwise return True.
04298
04299     int j, n, nv;
04300
04301     for (j = 0; j < nNodes; j++) {
04302         nodes[j]->visited = -1;
04303     }
04304     if (visit(0)) {
04305         return True;
04306     }
04307     nv = 0;
04308     for (n = 0; n < nNodes; n++) {
04309         if (nodes[n]->visited >= 0) {
04310             nv++;
04311         }
04312     }
04313     return (nv == nNodes);
04314 }
04315
04316 //-----
04317 Bool MGraph::visit(int nd)
04318 {
04319     // Visiting connected node used for 'isConnected'
04320     // If child nodes has free legs, then this function returns True.
04321     // otherwise it returns False.
04322
04323     int td;
04324
04325     // This node has free legs.
04326     if (nodes[nd]->freelg > 0) {
04327         return True;
04328     }
04329     nodes[nd]->visited = 0;
04330     for (td = 0; td < nNodes; td++) {
04331         if ((adjMat[nd][td] > 0) && (nodes[td]->visited < 0)) {
04332             if (visit(td)) {
04333                 return True;
04334             }
04335         }
04336     }
04337     // all the child nodes has no free legs.
04338     return False;
04339 }
04340
04341 //-----
04342 Bool MGraph::isIsomorphic(MNodeClass *cl)
04343 {
04344     // Check whether the current graph is the Representative
04345     // of a isomorphic class.
04346     // nsym = symmetry factor by the permutation of nodes.
04347     // esym = symmetry factor by the permutation of edge.
04348     // If this graph is not a representative, then returns False.
04349
04350     int j1, j2, cmp, nself;
04351     int *perm;
04352
04353     nsym = ToBigInt(0);
04354     esym = ToBigInt(1);
04355
04356     group->newGroup(nNodes, cl->nClasses, cl->clist);
04357
04358 #ifdef ORBITS
04359     orbits->initPerm(nNodes);
04360 #endif
04361
04362     while (True) {
04363         perm = group->genNext();
04364
04365         if (perm == NULL) {

```

```

04366         // calculate permutations of edges
04367         esym = ToBigInt(1);
04368         for (j1 = 0; j1 < nNodes; j1++) {
04369             if (adjMat[j1][j1] > 0) {
04370                 nself = adjMat[j1][j1]/2;
04371                 esym *= factorial(nself);
04372                 esym *= ipow(2, nself);
04373             }
04374             for (j2 = j1+1; j2 < nNodes; j2++) {
04375                 if (adjMat[j1][j2] > 0) {
04376                     esym *= factorial(adjMat[j1][j2]);
04377                 }
04378             }
04379         }
04380         return True;
04381     }
04382
04383     permMat(nNodes, perm, adjMat, modmat);
04384     cmp = compMat(nNodes, adjMat, modmat);
04385     if (cmp < 0) {
04386         return False;
04387     } else if (cmp == 0) {
04388         // if ngroup < 0, symmetry group is $S_{-group}$.
04389         group->addGroup(perm);
04390
04391         nsym = nsym + ToBigInt(1);
04392 #ifdef ORBITS
04393         orbits->fromPerm(perm);
04394 #endif
04395     }
04396 }
04397 }
04398
04399 //-----
04400 void MGraph::permMat(int size, int *perm, int **mat0, int **mat1)
04401 {
04402     // apply permutation to matrix 'mat0' and obtain 'mat1'
04403
04404     int j1, j2;
04405
04406     for (j1 = 0; j1 < size; j1++) {
04407         for (j2 = 0; j2 < size; j2++) {
04408             mat1[j1][j2] = mat0[perm[j1]][perm[j2]];
04409         }
04410     }
04411 }
04412
04413 //-----
04414 int MGraph::compMat(int size, int **mat0, int **mat1)
04415 {
04416     // comparison of matrix
04417
04418     int j1, j2, cmp;
04419
04420     for (j1 = 0; j1 < size; j1++) {
04421         for (j2 = 0; j2 < size; j2++) {
04422             cmp = mat0[j1][j2] - mat1[j1][j2];
04423             if (cmp != 0) {
04424                 return cmp;
04425             }
04426         }
04427     }
04428     return 0;
04429 }
04430
04431 //-----
04432 MNodeClass *MGraph::refineClass(MNodeClass *cl)
04433 {
04434     // Refine the classification
04435     // cl : the current 'MNodeClass' object
04436     // cn : the class number
04437     // Returns (the new class number corresponds to 'cn') if OK, or
04438     // 'None' if ordering condition is not satisfied
04439
04440     MNodeClass *ccl = cl;
04441     MNodeClass *xcl = NULL;
04442     MNodeClass *ncl = NULL;
04443     int ucl[GRCC_MAXNODES];
04444     int ccn = cl->nClasses;
04445     int nucl, nce;
04446     int td, cmp;
04447     int nccl;
04448
04449 #ifdef MONITOR
04450     nCallRefine++;
04451 #endif
04452     nucl = 0;

```

```

04453     while (ccl->nClasses != nucl) { // repeat refinement.
04454         nce = 0;
04455         nucl = 0;
04456         for (td = 1; td < nNodes; td++) {
04457             // 'td' is the next node and the current node is 'td-1'.
04458             // Count up the number of the elements in the current class
04459             // corresponding to the current node
04460             nce++;
04461             cmp = ccl->clCmp(td-1, td, ccn);
04462
04463             // the ordering condition is not satisfied.
04464             if (cmp > 0) {
04465                 if (ccl != cl) {
04466                     delete ccl;
04467                 }
04468                 return NULL;
04469             }
04470             // else if (cmp < 0) {
04471             //     // 'td' is in the next class to the current node.
04472             //     ucl[nucl++] = nce; // close the current class
04473             //
04474             //     // start new class
04475             //     nce = 0;
04476             // }
04477             // nothing to do for the case of 'cmp == 0'.
04478         }
04479     }
04480
04481     // close array 'ucl'.
04482     ucl[nucl++] = nce + 1;
04483
04484 #ifdef CHECK
04485     // inconsistent
04486     if (nucl < ccl->nClasses) {
04487         erEnd("refineClasses : smaller number of classes");
04488     }
04489 #endif
04490
04491     // preparation of the next repetition.
04492     xcl = new MNodeClass(nNodes, nucl);
04493     xcl->init(ucl, maxdeg, adjMat);
04494     ccn = xcl->nClasses;
04495
04496     nccl = ccl->nClasses;
04497
04498     if (ccl != cl) {
04499         delete ccl;
04500     }
04501     ccl = xcl;
04502     if (ccl->nClasses == nccl) {
04503         ccl->reorder(this);
04504         return ccl;
04505     }
04506
04507     nucl = 0;
04508 }
04509
04510 return ncl;
04511 }
04512
04513 //-----
04514 void MGraph::biconnME(void)
04515 {
04516     // Count the number of lPI components.
04517
04518     int j, jl, root, vr, next, nart;
04519     MCOpi mopi;
04520     MCBLOCK mblk;
04521     ULong momset;
04522
04523     // initialization
04524     bicount = 0;
04525     mconn->init();
04526     mconn->initCEdges(this);
04527
04528     for (j = 0; j < nNodes; j++) {
04529         bidef[j] = -1;
04530         bilow[j] = -1;
04531         bicol[j] = 0;
04532     }
04533     bipart = True;
04534
04535     // find a root for the root of bisearch
04536     // root must be an external node
04537     // or an vertex when there are not external nodes.
04538     root = -1;
04539     vr = -1;

```

```

04540     for (j = 0; j < nNodes; j++) {
04541         if (bidef[j] < 0) {
04542             if (isExternal(j)) {
04543                 root = j;
04544                 break;
04545             } else if (vr < 0) {
04546                 vr = j;
04547             }
04548         }
04549     }
04550     // case (2)
04551     if (root < 0) {
04552         root = vr;
04553     }
04554
04555     if (root >= 0) {
04556         bisearchME(root, -1, 0, 1, &mopi, &mblk, &momset, &next, &nart);
04557
04558         mconn->nlpopic = 0;
04559         mconn->nctopic = 0;
04560         for (j = 0; j < mconn->nopic; j++) {
04561             if (mconn->opics[j].loop > 0) {
04562                 mconn->nlpopic++;
04563             } else if (mconn->opics[j].ctloop > 0) {
04564                 mconn->nctopic++;
04565             }
04566         }
04567
04568         mconn->ne0bridges = 0;
04569         mconn->nelbridges = 0;
04570         for (j = 0; j < mconn->nbridges; j++) {
04571             if (mconn->bridges[j].next == 0) {
04572                 mconn->ne0bridges++;
04573             }
04574             if (mconn->bridges[j].next == 1) {
04575                 mconn->nelbridges++;
04576             }
04577         }
04578
04579         mconn->nselfloops = 0;
04580         for (j = 0; j < nNodes; j++) {
04581             mconn->nselfloops += adjMat[j][j]/2;
04582         }
04583
04584         mconn->nmultiedges = 0;
04585         for (j = 0; j < nNodes; j++) {
04586             for (j1 = j+1; j1 < nNodes; j1++) {
04587                 if (adjMat[j][j1] > 1) {
04588                     mconn->nmultiedges++;
04589                 }
04590             }
04591         }
04592
04593         mconn->nalblocks = 0;
04594         for (j = 0; j < mconn->nblocks; j++) {
04595             if (mconn->blocks[j].nartps == 1) {
04596                 mconn->nalblocks++;
04597             }
04598         }
04599         mconn->neblocks = mconn->nblocks + mconn->nctopic;
04600
04601 #ifdef CHECK
04602     if (True) {
04603         if (isExternal(root)) {
04604             momset |= MASK(root);
04605         }
04606         ULong m = 0;
04607         for (j = 0; j < nExtern; j++) {
04608             m |= MASK(j);
04609         }
04610         bool ok = True;
04611         if (momset != m) {
04612             grcc_fprintf(GRCC_Stdout, "*** momset = %ld != %ld\n", momset, m);
04613             ok = False;
04614         }
04615         if (mconn->nbacked != nLoops) {
04616             grcc_fprintf(GRCC_Stdout, "*** nbacked = %d != %c\n", mconn->nbacked, nLoops);
04617             ok = False;
04618         }
04619         if (! ok) {
04620             print();
04621             egraph->print();
04622             mconn->print();
04623             erEnd("biconnME: illegal connection");
04624         }
04625     }
04626 #endif

```

```

04627
04628 // no vertex.
04629 } else {
04630 // no vertices
04631 // Connected ==> only when ex=2, loop=0
04632 mconn->nopic = 0;
04633 mconn->ne0bridges = 0;
04634 mconn->nelbridges = 0;
04635 }
04636 }
04637
04638 //-----
04639 void MGraph::bisearchME(int nd, int pd, int ned, int col,
04640 MCOpi *mopi, MCBlock *mblk, ULong *momset, int *next, int *nart)
04641 {
04642 // Search biconnected component
04643 // visit : pd --> nd --> td
04644 // ned : the number of edges between pd and nd.
04645 // Output
04646 //
04647 //          |
04648 //          x-----+
04649 //          |       |
04650 //      ----x pd |
04651 //          |       |
04652 //      x----x nd | n art. pints.
04653 //      |  // \  |
04654 //      | /  \  |
04655 // m art p. x-x | x-x
04656 //          n1 x n3
04657 //          n2
04658 //
04659 // output of bisearchME(n1, nd, ...) ==> npart = m
04660 // output of bisearchME(n2, nd, ...) ==> npart = 1
04661 // output of bisearchME(n3, nd, ...) ==> npart = n
04662 //
04663 // output of bisearchME(nd, pd, ...) ==> npart = n+1
04664 //      n1 => block of (m+1) articulation points
04665 //      n2 => block of 2 articulation points
04666 //      n3 => no block
04667 //
04668 Bool newv;
04669 int td, td0, lp, conn, next1, nart1, nart2, nvisit, ndart;
04670 int opit, opit0, blk, l;
04671 MCOpi mopi1;
04672 MCBlock mblk1;
04673 ULong momset1, momset2, momset3;
04674
04675 mopi->init();
04676 mblk->init();
04677
04678 *momset = 0; // # the set of momenta
04679 *next = 0; // # external below 'nd' inclusive
04680 *nart = 0; // # articulation points 'nd' inclusive
04681 nart1 = 0; // # articulation points below 'nd'
04682 nart2 = 0; // # 'nd' is articulation point then 1
04683 ndart = 0; // # the number of blocks attached to 'nd'
04684
04685 newv = (bidef[nd] < 0);
04686 if (!newv) {
04687 grcc_fprintf(GRCC_Stdout, "*** nd=%d, pd=%d, bidef[nd]=%d\n", nd, pd, bidef[nd]);
04688 erEnd("bisearchME: illegal connection");
04689 }
04690
04691 bidef[nd] = bicount;
04692 bilow[nd] = bicount;
04693 bicol[nd] = col;
04694 bicount++;
04695
04696 opit0 = mconn->opistkpctr;
04697
04698 if (pd < 0) {
04699 mconn->pushNode(nd);
04700 }
04701
04702 // external node and not the root
04703 if (isExternal(nd)) {
04704 if (pd >= 0) {
04705 // not the root
04706 *momset = MASK(nd);
04707 mopi->next = 1;
04708 mopi->nlegs = 1;
04709 mopi->mom0lg = 0;
04710 mblk->nartps = 1;
04711 *next = 1;
04712 *nart = 1;
04713 mconn->addCEdge(nd, pd, *momset);

```



```

04714         return;
04715     }
04716 }
04717
04718 nvisit = 0;
04719 td0 = -1;
04720 for (td = 0; td < nNodes; td++) {
04721     conn = adjMat[nd][td];
04722
04723     // td is not adjacent to nd
04724     if (conn < 1) {
04725         continue;
04726     }
04727     nvisit++;
04728     td0 = td;
04729
04730     momset1 = 0;
04731
04732     // external node
04733     if (isExternal(td) && bidef[td] < 0) {
04734         bicol[td] = - col;
04735         (*next)++;
04736         ndart++;
04737         mconn->addArtic(nd, 1);
04738         mblk->nartps++;
04739         nart2 = 1;
04740         momset1 = MASK(td);
04741         mopi->nlegs++;
04742         mopi->next++;
04743         mconn->addCEdge(td, nd, momset1);
04744         *momset |= momset1;
04745         if (newv) {
04746             mconn->pushNode(td);
04747         }
04748
04749         // self-loop : pd --> nd --> nd
04750     } else if (td == nd) {
04751         lp = conn/2;
04752         mopi->loop += lp;
04753         mopi->nedges += lp;
04754         ndart += lp;
04755         bipart = False;
04756         mconn->addArtic(nd, lp);
04757         mconn->addBlockSelf(nd, lp);
04758         mblk->nartps++;
04759         nart2 = 1;
04760         for (l = 0; l < lp; l++) {
04761             int m = nExtern + mconn->nbacked;
04762             mconn->nbacked++;
04763             momset1 = MASK(m);
04764             mconn->addCEdge(td, nd, momset1);
04765         }
04766
04767         // back to the parent : pd --> nd --> pd, pd is a vertex
04768     } else if (td == pd) {
04769         if (ned > 1) {
04770             bilow[nd] = Min(bilow[nd], bidef[pd]);
04771             mopi->loop += ned - 1;
04772             mblk->loop += ned - 1;
04773         } else {
04774             // cancel this visit
04775             nvisit--;
04776         }
04777
04778         // td is already visited
04779     } else if (bidef[td] >= 0) {
04780         bipart &= (bicol[td] == - col);
04781         bilow[nd] = Min(bilow[nd], bidef[td]);
04782         if (bidef[nd] >= bidef[td]) {
04783             // new back-edge is found
04784             mopi->loop += conn;
04785             mopi->nedges += conn;
04786             mblk->loop += conn;
04787             mconn->pushEdge(td, nd);
04788             for (l = 0; l < conn; l++) {
04789                 int m = nExtern + mconn->nbacked;
04790                 mconn->nbacked++;
04791                 momset1 = MASK(m);
04792                 *momset |= momset1;
04793                 mconn->addCEdge(td, nd, momset1);
04794             }
04795         } else {
04796             // reverse direction of back-edge
04797             int cn = 0;
04798
04799             for (int ed = 0; ed < mconn->sedges; ed++) {
04800                 if (mconn->cedges[ed].nodes[0] == nd &&

```

```

04801         mconn->cedges[ed].nodes[1] == td &&
04802         mconn->cedges[ed].momdir > 0) {
04803             cn++;
04804             *momset &= ~ (mconn->cedges[ed].momset);
04805         } else if (mconn->cedges[ed].nodes[1] == nd &&
04806                 mconn->cedges[ed].nodes[0] == td &&
04807                 mconn->cedges[ed].momdir < 0) {
04808             cn++;
04809             *momset &= ~ (mconn->cedges[ed].momset);
04810         }
04811     }
04812     if (cn != conn) {
04813         gcc_fprintf(GRCC_Stdout, "*** revisit back-ed (%d --> %d) cn=%d != conn=%d\n",
04814                 nd, td, cn, conn);
04815         gcc_fprintf(GRCC_Stdout, "    sedges = %d\n", mconn->sedges);
04816         for (int ed = 0; ed < mconn->sedges; ed++) {
04817             gcc_fprintf(GRCC_Stdout, "    %d : (%d --> %d) : [%2d*%21d]\n", ed,
04818                     mconn->cedges[ed].nodes[0],
04819                     mconn->cedges[ed].nodes[1],
04820                     mconn->cedges[ed].momdir,
04821                     mconn->cedges[ed].momset);
04822         }
04823         print();
04824         egraph->print();
04825         mconn->print();
04826         erEnd("bisearchME: illegal connection");
04827     }
04828 }
04829
04830 // new node
04831 } else if (bidef[td] < 0) {
04832     // stack pointer
04833     if (pd < 0 && isExternal(nd)) {
04834         opit = opit0;
04835     } else {
04836         opit = mconn->opistkpctr;
04837     }
04838     blk1 = mconn->blkstkptr;
04839     mblk1.init();
04840     mconn->pushEdge(nd, td);
04841
04842     // visit child
04843     bisearchME(td, nd, adjMat[td][nd], - col,
04844             &mop1, &mblk1, &momset1, &next1, &nart1);
04845
04846     // momset
04847     momset2 = 0;
04848     for (l = 0; l < conn - 1; l++) {
04849         int m = nExtern + mconn->nbacked;
04850         mconn->nbacked++;
04851         momset3 = MASK(m);
04852         mconn->addCEdge(nd, td, momset3);
04853         momset2 |= momset3;
04854     }
04855     mconn->addCEdge(td, nd, momset1 | momset2);
04856     *momset |= momset1;
04857     // articulation point of bridge
04858     if (bilow[td] > bidef[nd]) {
04859         // new OPI component (not including 'td')
04860         int mom0lg = (momset1 == 0) ? 1 : 0;
04861         mop1.nlegs++;
04862         mop1.mom0lg += mom0lg;
04863
04864         mconn->addOPIC(&mop1, opit);
04865
04866         mop1.init();
04867         mop1.nlegs = 1;
04868         mop1.mom0lg = mom0lg;
04869         if (pd >= 0 || !isExternal(nd)) {
04870             // opi
04871             mconn->addBridge(nd, td, next1, nExtern);
04872
04873             // block
04874             ndart++;
04875             mconn->addArtic(nd, 1);
04876             // discard nparts from nodes below td
04877             mblk1.nartps = 2;
04878             mconn->addBlock(&mblk1, blk1);
04879             mblk1.init();
04880             mblk1.nartps = 1;
04881
04882             nart1 = 0;
04883             nart2 = 1; // nd is an articulation point
04884         }
04885     }
04886 }

```

```

04888         } else if (bilow[td] == bidef[nd]) {
04889             // articulation point of a block
04890             mopi1.nedges += conn;
04891             if (pd >= 0 || !isExternal(nd)) {
04892                 ndart++;
04893                 mconn->addArtic(nd, 1);
04894                 mblk1.nartps++;
04895                 mconn->addBlock(&mblk1, blk1);
04896                 mblk1.init();
04897                 mblk1.nartps = 1;
04898                 nart1 = 0;
04899                 nart2 = 1; // nd is an articulation point
04900             }
04901         } else {
04902             // inside a block
04903             mopi1.nedges += conn;
04904         }
04905     }
04906
04907     bilow[nd] = Min(bilow[nd], bilow[td]);
04908     mopi->nlegs += mopi1.nlegs;
04909     mopi->next += mopi1.next;
04910     mopi->loop += mopi1.loop;
04911     mopi->nedges += mopi1.nedges;
04912     mopi->ctloop += mopi1.ctloop;
04913     mopi->mom0lg += mopi1.mom0lg;
04914
04915     mblk->nartps += mblk1.nartps;
04916     mblk->loop += mblk1.loop;
04917
04918     *next += next1;
04919     *nart += nart1;
04920 } // of "if (bidef[td] < 0)"
04921 } // end of for
04922
04923 // no connection except for 'pd'==>'nd'. Then 'nd' is an art. point
04924 if (nvisit == 0) {
04925     nart2 = 1;
04926 }
04927 *nart += nart2;
04928
04929 // add # loop inside counter terms
04930 if (newv) {
04931     if (pd >= 0) {
04932         mconn->pushNode(nd);
04933     }
04934     mopi->ctloop += Max(0, nodes[nd]->extloop);
04935 }
04936
04937 // 'nd' is the root node
04938 if (pd < 0) {
04939     if (isExternal(nd)) {
04940         if (td0 >= 0) {
04941             mconn->addArtic(td0, 1);
04942         }
04943         if (mconn->nopic > 0) {
04944             (mconn->opics[mconn->nopic-1].next)++;
04945         }
04946     } else {
04947         mconn->addOPic(mopi, opit0);
04948         if (ndart == 1) {
04949             // 'nd' has only 1 child
04950             // the root is not really an art. point
04951             (*nart)--;
04952             mconn->addArtic(nd, -1);
04953             (mconn->blocks[mconn->nblocks-1].nartps)--;
04954         }
04955     }
04956 }
04957 }
04958
04959 //-----
04960 BigInt MGraph::generate(void)
04961 {
04962     // Generate graphs
04963     // the generation process starts from 'connectClass'.
04964
04965     MNodeClass *cl;
04966
04967     // Initial classification of nodes.
04968     cl = new MNodeClass(nNodes, nClasses);
04969     cl->init(clist, maxdeg, adjMat);
04970     cl->reorder(this);
04971     connectClass(cl);
04972
04973     delete cl;
04974 }

```

```

04975     return cDiag;
04976 }
04977
04978 //-----
04979 void MGraph::connectClass(MNodeClass *cl)
04980 {
04981     // Connect nodes in a class to others
04982
04983     int sc, sn;
04984     MNodeClass *xcl;
04985
04986     xcl = refineClass(cl);
04987
04988     if (xcl == NULL) {
04989 #ifdef MONITOR
04990         discardRefine++;
04991 #endif
04992     } else {
04993         if (xcl->flg0 >= xcl->nClasses) {
04994             newGraph(xcl);
04995         } else {
04996             sc = xcl->clord[xcl->flg0];
04997             sn = xcl->flist[sc];
04998             connectNode(xcl->flg0, sn, xcl);
04999         }
05000     }
05001     if (xcl != cl && xcl != NULL) {
05002         delete xcl;
05003     }
05004 }
05005
05006 //-----
05007 void MGraph::connectNode(int so, int ss, MNodeClass *cl)
05008 {
05009     int sc, sn;
05010
05011     sc = cl->clord[so];
05012     if (ss >= cl->flist[sc+1]) {
05013         connectClass(cl);
05014         return;
05015     }
05016
05017     for (sn = ss; sn < cl->flist[sc+1]; sn++) {
05018         connectLeg(so, sn, so, sn, cl);
05019         return;
05020     }
05021 }
05022
05023 //-----
05024 void MGraph::connectLeg(int so, int sn, int to, int ts, MNodeClass *cl)
05025 {
05026     // Add one connection between two legs.
05027     // 1. select another node to connect
05028     // 2. determine multiplicity of the connection
05029
05030     int sc, tc, tn, maxself, nc2, nc, maxcon, tsl, ncm, tol;
05031
05032     sc = cl->clord[so];
05033
05034 #ifdef CHECK
05035     if (sn >= cl->flist[sc+1]) {
05036         erEnd("connectLeg : illegal control");
05037         return;
05038     }
05039 #endif
05040
05041     // There remains no free legs in the node 'sn' : move to next node.
05042     if (nodes[sn]->freelg < 1) {
05043         if (!isConnected()) {
05044 #ifdef MONITOR
05045             discardDisc++;
05046 #endif
05047     } else {
05048         // next node in the current class.
05049         connectNode(so, sn+1, cl);
05050     }
05051     }
05052     return;
05053 }
05054
05055 // connect a free leg of the current node 'sn'.
05056 for (tol = to; tol < cl->nClasses; tol++) {
05057     tc = cl->clord[tol];
05058     if (tol == to) {
05059         tsl = ts;
05060     } else {
05061         tsl = cl->flist[tc];

```

```

05062     }
05063 #ifdef MINMAXLEG
05064     if (ts1 >= nNodes) {
05065         continue;
05066     }
05067     if ((nodes[sn]->cmindeg > 0 && nodes[ts1]->deg < nodes[sn]->cmindeg)
05068         || (nodes[sn]->cmaxdeg > 0 && nodes[ts1]->deg > nodes[sn]->cmaxdeg)
05069         || (nodes[ts1]->cmindeg > 0 && nodes[sn]->deg < nodes[ts1]->cmindeg)
05070         || (nodes[ts1]->cmaxdeg > 0 && nodes[sn]->deg > nodes[ts1]->cmaxdeg)) {
05071         continue;
05072     }
05073 #endif
05074     for (tn = ts1; tn < cl->flist[tc+1]; tn++) {
05075         if (sc == tc && sn > tn) {
05076             continue;
05077         } else if (nodes[tn]->freelg < 1) {
05078             continue;
05079         }
05080
05081         // self-loop
05082         if (sn == tn) {
05083             if (opt->values[GRCC_OPT_NoSelfLoop] > 0) {
05084                 continue;
05085             }
05086             if (nNodes > 1) {
05087                 // there are two or more nodes in the graph :
05088                 // avoid disconnected graph
05089                 maxself = Min((nodes[sn]->freelg)/2, (nodes[sn]->deg-1)/2);
05090             } else {
05091                 // there is only one node the graph.
05092                 maxself = nodes[sn]->freelg/2;
05093             }
05094
05095             if ((nExtern > 1) && (opt->values[GRCC_OPT_1PI] > 0
05096                 || opt->values[GRCC_OPT_NoTadpole] > 0)
05097                 && nNodes > 1) {
05098                 // there are two or more nodes in the graph :
05099                 // avoid to generate tadpole
05100                 maxself = Min((nodes[sn]->deg-2)/2, maxself);
05101             }
05102
05103             // for the number of connection.
05104             for (nc2 = maxself; nc2 > 0; nc2--) {
05105                 nc = 2*nc2;
05106                 ncm = CLWIGHTD(nc);
05107                 adjMat[sn][sn] = nc;
05108                 nodes[sn]->freelg -= nc;
05109                 cl->incMat(sn, tn, ncm);
05110
05111                 // next connection
05112                 connectLeg(so, sn, tol, tn+1, cl);
05113
05114                 // restore the configuration
05115                 cl->incMat(tn, sn, -ncm);
05116                 adjMat[sn][sn] = 0;
05117                 nodes[sn]->freelg += nc;
05118             }
05119
05120             // connections between different nodes.
05121         } else {
05122             // maximum possible connection number
05123             maxcon = Min(nodes[sn]->freelg, nodes[tn]->freelg);
05124
05125             // avoid disconnected graphs.
05126             if (nNodes > 2 && nodes[sn]->deg == nodes[tn]->deg) {
05127                 maxcon = Min(maxcon, nodes[sn]->deg-1);
05128             }
05129
05130             if (opt->values[GRCC_OPT_NoMultiEdge] > 0) {
05131                 maxcon = Min(maxcon, 1);
05132             }
05133
05134 #ifdef CHECK
05135             if ((adjMat[sn][tn] != 0) ||
05136                 (adjMat[sn][tn] != adjMat[tn][sn])) {
05137                 grcc_fprintf(GRCC_Stdout, "*** inconsistent connection: sn=%d, tn=%d",
05138                     sn, tn);
05139                 printAdjMat(cl);
05140                 erEnd("inconsistent connection ");
05141             }
05142 #endif
05143
05144             // vary number of connections
05145             for (nc = maxcon; nc > 0; nc--) {
05146                 ncm = CLWIGHTO(nc);
05147                 adjMat[sn][tn] = nc;
05148                 adjMat[tn][sn] = nc;

```

```

05149         nodes[sn]->freelg -= nc;
05150         nodes[tn]->freelg -= nc;
05151         cl->incMat(sn, tn, ncm);
05152         cl->incMat(tn, sn, ncm);
05153
05154         // next connection
05155         connectLeg(so, sn, tol, tn+1, cl);
05156
05157         // restore configuration
05158         cl->incMat(sn, tn, - ncm);
05159         cl->incMat(tn, sn, - ncm);
05160         adjMat[sn][tn] = 0;
05161         adjMat[tn][sn] = 0;
05162         nodes[sn]->freelg += nc;
05163         nodes[tn]->freelg += nc;
05164     }
05165 }
05166 }
05167 }
05168 }
05169
05170 //-----
05171 Bool MGraph::isOptM(void)
05172 {
05173     Bool ok = True;
05174
05175     opi         = (mconn->nopic == 1);
05176     opiloop     = (mconn->nlpopic <= 1);
05177     tadpole     = (mconn->ne0bridges >= ((nExtern == 1) ? 0 : 1));
05178     selfloop    = (mconn->nselfloops > 0);
05179     multiedge   = (mconn->nmultiedges > 0);
05180     tadblock    = (mconn->nalblocks > 0);
05181     block       = (mconn->neblocks <= 1);
05182     extself     = (mconn->nelbridges > 0);
05183
05184     if (opt->values[GRCC_OPT_1PI] > 0) {
05185         ok = ok && opi;
05186     } else if (opt->values[GRCC_OPT_1PI] < 0) {
05187         ok = ok && !opi;
05188     }
05189     if (opt->values[GRCC_OPT_NoExtSelf] > 0) {
05190         ok = ok && !extself;
05191     } else if (opt->values[GRCC_OPT_NoExtSelf] < 0) {
05192         ok = ok && extself;
05193     }
05194     if (opt->values[GRCC_OPT_NoTadpole] > 0) {
05195 #ifdef OLDOPT
05196         ok = ok && !tadpole;
05197 #else
05198         if (nExtern > 1) {
05199             ok = ok && !tadpole;
05200         }
05201 #endif
05202     } else if (opt->values[GRCC_OPT_NoTadpole] < 0) {
05203 #ifdef OLDOPT
05204         ok = ok && !tadpole;
05205 #else
05206         if (nExtern > 1) {
05207             ok = ok && tadpole;
05208         }
05209 #endif
05210     }
05211     if (opt->values[GRCC_OPT_NoSelfLoop] > 0) {
05212     } else if (opt->values[GRCC_OPT_NoSelfLoop] < 0) {
05213         ok = ok && selfloop;
05214     }
05215     if (opt->values[GRCC_OPT_NoMultiEdge] > 0) {
05216     } else if (opt->values[GRCC_OPT_NoMultiEdge] < 0) {
05217         ok = ok && multiedge;
05218     }
05219     if (opt->values[GRCC_OPT_No1PtBlock] > 0) {
05220         if (nExtern > 1) {
05221             ok = ok && !tadblock;
05222         }
05223     } else if (opt->values[GRCC_OPT_No1PtBlock] < 0) {
05224         if (nExtern > 1) {
05225             ok = ok && tadblock;
05226         }
05227     }
05228     if (opt->values[GRCC_OPT_Block] > 0) {
05229         ok = ok && block;
05230     } else if (opt->values[GRCC_OPT_Block] < 0) {
05231         ok = ok && !block;
05232     }
05233     return ok;
05234 }
05235

```

```

05236 //-----
05237 void MGraph::newGraph(MNodeClass *cl)
05238 {
05239     // A new candidate diagram is obtained.
05240     // It may be a disconnected graph
05241     //
05242     // Things to be done in the steps followed by this function.
05243     // 1. convert data format which treat the edges as objects.
05244     // 2. definition of loop momenta to the edges.
05245     // 3. analysis of loop structure in a graph.
05246     // 4. assignment of particles to edges and vertices to nodes
05247     // 5. recalculate symmetry factor
05248
05249     int connected;
05250     MNodeClass *xcl;
05251     Bool ok;
05252
05253     ngen++;
05254
05255     // refine class and check ordering condition
05256     xcl = refineClass(cl);
05257     if (xcl == NULL) {
05258 #ifdef MONITOR
05259         discardRefine++;
05260 #endif
05261
05262     // check whether this is a connected graph or not.
05263     } else {
05264         connected = isConnected();
05265         if (!connected) {
05266 #ifdef MONITOR
05267             discardDisc++;
05268 #endif
05269
05270         // connected graph : check isomorphism of the graph
05271     } else {
05272         ngconn++;
05273         if (!isIsomorphic(xcl)) {
05274 #ifdef MONITOR
05275             discardIso++;
05276 #endif
05277
05278         // We got a new connected and a unique representation of a class.
05279     } else {
05280         biconnME();
05281         if (isOptM()) {
05282             egraph->fromMGraph(this);
05283             if (egraph->isOptE()) {
05284                 curcl->copy(xcl);
05285 #ifdef ORBITS
05286                 orbits->toOrbits();
05287 #endif
05288                 ok = True;
05289                 if (opt->outmg != NULL) {
05290                     if (opt->proc != NULL) {
05291                         egraph->mId = opt->proc->mgrcount + 1;
05292                         egraph->sId = opt->proc->agrcount + 1;
05293                     } else if (opt->sproc != NULL) {
05294                         egraph->mId = opt->sproc->mgrcount + 1;
05295                         egraph->sId = opt->sproc->agrcount + 1;
05296                     } else {
05297                         egraph->mId = cDiag;
05298                     }
05299                     ok = (*(opt->outmg))(egraph, opt->argmg);
05300                 }
05301
05302                 if (ok) {
05303                     // # generated graphs
05304                     cDiag++;
05305                     wscon.add(1, nsym*esym);
05306
05307                     // # generated lPI graphs
05308                     if (opi) {
05309                         wsopi.add(1, nsym*esym);
05310                         clPI++;
05311                     }
05312
05313                     // # no tadpoles
05314                     if (!tadpole) {
05315                         cNoTadpole++;
05316                     }
05317                     // # no tadblocks
05318                     if (!tadblock) {
05319                         cNoTadBlock++;
05320                         if (opi) {
05321                             clPINoTadBlock++;
05322                         }

```

```

05323                                     }
05324
05325 #ifndef MONITOR
05326                                     grcc_fprintf(GRCC_Stdout, "\n");
05327                                     grcc_fprintf(GRCC_Stdout, "Graph : %ld (%ld)", cDiag, ngen);
05328                                     grcc_fprintf(GRCC_Stdout, " sym. factor = (%ld*%ld)\n", nsym, esym);
05329                                     printAdjMat(c1);
05330                                     // c1->printMat();
05331 #   ifdef ORBITS
05332                                     orbits->print();
05333 #   endif
05334 #endif
05335                                     // go to next step
05336                                     opt->newMGraph(this);
05337                                     } else {
05338                                     }
05339                                     } else {
05340                                     }
05341                                     } else {
05342                                     ;
05343                                     }
05344                                     }
05345                                     }
05346                                     }
05347                                     if (xcl != c1 && xcl != NULL) {
05348                                         delete xcl;
05349                                     }
05350                                     }
05351
05352 //=====
05353 // class MOrbits: orbits of nodes
05354 //-----
05355 MOrbits::MOrbits(void)
05356 {
05357     nOrbits = 0;
05358     nNodes  = 0;
05359 }
05360
05361 //-----
05362 MOrbits::~MOrbits(void)
05363 {
05364 }
05365
05366 //-----
05367 void MOrbits::print(void)
05368 {
05369     int c;
05370
05371     grcc_fprintf(GRCC_Stdout, "Orbits : nOrbits=%d: nd2or=", nOrbits);
05372     prIntArray(nNodes, nd2or, "\n");
05373     for (c = 0; c < nOrbits; c++) {
05374         grcc_fprintf(GRCC_Stdout, "      %2d: (%2d -- %2d) :", c, flist[c], flist[c+1]-1);
05375         prIntArray(flist[c+1]-flist[c], or2nd+flist[c], "\n");
05376     }
05377 }
05378
05379 //-----
05380 void MOrbits::initPerm(int nnodes)
05381 {
05382     int j;
05383
05384     nNodes = nnodes;
05385     for (j = 0; j < nNodes; j++) {
05386         nd2or[j] = j;
05387     }
05388 }
05389
05390 //-----
05391 void MOrbits::fromPerm(int *perm)
05392 {
05393     // update the orbits of nodes by the permutation
05394
05395     int j, j1, k, l;
05396
05397     for (j = 0; j < nNodes; j++) {
05398         if (nd2or[j] != j) {
05399             // not the first element of an old orbit
05400             continue;
05401         }
05402         // merge orbits to orbit 'j'
05403         k = j;
05404         for (j1 = 0; j1 < nNodes && (k = perm[k]) != j; j1++) {
05405             if (nd2or[k] == j) {
05406                 // 'k' is already in 'j'
05407                 ;
05408             } else {
05409                 // old orbit 'k' had several elements: merge to orbit 'j'

```



```

05410         for (l = 0; l < nNodes; l++) {
05411             if (nd2or[l] == k) {
05412                 nd2or[l] = j;
05413             }
05414         }
05415     }
05416 }
05417 #ifndef CHECK
05418     if (j1 >= nNodes) {
05419         grcc_fprintf(GRCC_Stdout, "*** fromPerm: illegal control: j=%d, k=%d\n", j, k);
05420         grcc_fprintf(GRCC_Stdout, "perm=");
05421         prIntArray(nNodes, perm, " nd2or=");
05422         prIntArray(nNodes, nd2or, "\n");
05423         erEnd("fromPerm: illegal control");
05424     }
05425 #endif
05426 }
05427 }
05428
05429 //-----
05430 void MORbits::toOrbits(void)
05431 {
05432     // fill elements from 'nd2or'
05433     int nd, nel, k, lor;
05434
05435     lor = -1;
05436     nel = 0;
05437     nOrbits = 0;
05438     for (nd = 0; nd < nNodes; nd++) {
05439         if (nd2or[nd] > lor) {
05440             // lowest element of a new orbit
05441             lor = nd2or[nd];
05442             flist[nOrbits++] = nel;
05443             for (k = nd; k < nNodes; k++) {
05444                 if (nd2or[k] == lor) {
05445                     or2nd[nel++] = k;
05446                 }
05447             }
05448         }
05449     }
05450     flist[nOrbits] = nNodes;
05451 }
05452
05453 //*****
05454 // n-edge connected components
05455 //=====
05456 // class MCEdge
05457 //-----
05458 MCEdge::MCEdge(void)
05459 {
05460     ;
05461 }
05462
05463 //-----
05464 MCEdge::~MCEdge(void)
05465 {
05466     ;
05467 }
05468
05469 //=====
05470 // class MCOpi: one n-edge connected component
05471 //-----
05472 MCOpi::MCOpi(void)
05473 {
05474     init();
05475 }
05476
05477 //-----
05478 MCOpi::~MCOpi(void)
05479 {
05480     nodes = NULL;
05481 }
05482
05483 //-----
05484 void MCOpi::init(void)
05485 {
05486     nodes = NULL;
05487     nnodes = 0;
05488     nlegs = 0;
05489     next = 0;
05490     nedges = 0;
05491     loop = 0;
05492     ctloop = 0;
05493     mom0lg = 0;
05494 }
05495
05496 //=====

```

```

05497 // class MCBridge
05498 //-----
05499 MCBridge::MCBridge(void)
05500 {
05501 }
05502
05503 //-----
05504 MCBridge::~MCBridge(void)
05505 {
05506 }
05507
05508 //=====
05509 // class MCBlock
05510 //-----
05511 MCBlock::MCBlock(void)
05512 {
05513     init();
05514 }
05515
05516 //-----
05517 MCBlock::~MCBlock(void)
05518 {
05519 }
05520
05521 //-----
05522 void MCBlock::init(void)
05523 {
05524     edges    = NULL;
05525     nedges   = 0;
05526     nartps   = 0;
05527     loop     = 0;
05528 }
05529
05530 //=====
05531 // class MConn: table of n-edge connected components
05532 //-----
05533 MConn::MConn(int nnod, int nedg)
05534 {
05535     // nnod : the number of nodes
05536     // nedg : the number of edges
05537
05538     snodes = nnod;
05539     sedges = nedg;
05540
05541     cedges   = new MCEdge[sedges];
05542     opics    = new MCOpi[snodes];
05543     bridges  = new MCBridge[sedges];
05544     blocks   = new MCBlock[sedges];
05545     articuls = new int[snodes];
05546
05547     // workspace
05548     opisp    = new int[snodes];
05549     opistk   = new int[snodes];
05550     blksp    = new Edge2n[sedges];
05551     blkstk   = new Edge2n[sedges];
05552     init();
05553 }
05554
05555 //-----
05556 MConn::~MConn(void)
05557 {
05558     delete[] blkstk;
05559     delete[] blksp;
05560     delete[] opistk;
05561     delete[] opisp;
05562
05563     delete[] articuls;
05564     delete[] blocks;
05565     delete[] bridges;
05566     delete[] opics;
05567     delete[] cedges;
05568 }
05569
05570 //-----
05571 void MConn::init(void)
05572 {
05573     int j;
05574
05575     nopic    = 0;
05576     nlpopic  = 0;
05577     nbacked  = 0;
05578     nctopic  = 0;
05579     nbridges = 0;
05580     ne0bridges = 0;
05581     ne1bridges = 0;
05582     nselfloops = 0;
05583     nmultiedges = 0;

```

```

05584     nbacked      = 0;
05585
05586     nblocks      = 0;
05587     nalblocks     = 0;
05588     narticuls     = 0;
05589     neblocks      = 0;
05590
05591     opistkptr     = 0;
05592     nopisp        = 0;
05593     blkstkptr     = 0;
05594     nblksp        = 0;
05595
05596     for (j = 0; j < snodes; j++) {
05597         articuls[j] = 0;
05598     }
05599 }
05600
05601 //-----
05602 void MConn::pushNode(int nd)
05603 {
05604     if (opistkptr >= snodes) {
05605         erEnd("MConn::pushNode: opistkptr >= snodes");
05606     }
05607     opistk[opistkptr++] = nd;
05608 }
05609
05610 //-----
05611 void MConn::pushEdge(int n0, int n1)
05612 {
05613     if (blkstkptr >= sedges) {
05614         erEnd("MConn::pushEdge: blkstkptr >= sedges");
05615     }
05616     if (n0 <= n1) {
05617         blkstk[blkstkptr][0] = n0;
05618         blkstk[blkstkptr][1] = n1;
05619     } else {
05620         blkstk[blkstkptr][0] = n1;
05621         blkstk[blkstkptr][1] = n0;
05622     }
05623     blkstkptr++;
05624 }
05625
05626 //-----
05627 void MConn::initCEdges(MGraph *mg)
05628 {
05629     int ed, n0, n1, e;
05630
05631     ed = 0;
05632     for (n0 = 0; n0 < mg->nNodes; n0++) {
05633         for (e = 0; e < mg->adjMat[n0][n0]/2; e++, ed++) {
05634             cedges[ed].nodes[0] = n0;
05635             cedges[ed].nodes[1] = n0;
05636             cedges[ed].momdir = 0;
05637         }
05638         for (n1 = n0+1; n1 < mg->nNodes; n1++) {
05639             for (e = 0; e < mg->adjMat[n0][n1]; e++, ed++) {
05640                 cedges[ed].nodes[0] = n0;
05641                 cedges[ed].nodes[1] = n1;
05642                 cedges[ed].momdir = 0;
05643             }
05644         }
05645     }
05646     if (ed != sedges) {
05647         grcc_fprintf(GRCC_Stdout, "*** ed=%d != sedges=%d\n", ed, sedges);
05648         erEnd("MConn::initCEdge: table overflow");
05649     }
05650 }
05651
05652 //-----
05653 void MConn::addCEdge(int n0, int n1, ULong momset)
05654 {
05655     int m0, m1, dir, ed;
05656
05657     if (n0 <= n1) {
05658         m0 = n0;
05659         m1 = n1;
05660         dir = 1;
05661     } else {
05662         m0 = n1;
05663         m1 = n0;
05664         dir = -1;
05665     }
05666     for (ed = 0; ed < sedges; ed++) {
05667         if (cedges[ed].nodes[0] == m0 && cedges[ed].nodes[1] == m1 &&
05668             cedges[ed].momdir == 0) {
05669             cedges[ed].momdir = dir;
05670         }
05671     }

```

```

05671         cedges[ed].momset = momset;
05672         return;
05673     }
05674 }
05675 }
05676
05677 //-----
05678 void MConn::addOPic(MCOpI *mopi, int stp)
05679 {
05680     int j, nn;
05681
05682     if (nopic >= snodes) {
05683         erEnd("MConn::addOPic: table overflow");
05684     }
05685
05686     nn = opistkptr - stp;
05687     for (j = 0; j < nn; j++) {
05688         opisp[nopisp+j] = opistk[stp+j];
05689     }
05690     opics[nopic].nnodes = nn;
05691     opics[nopic].nlegs = mopi->nlegs;
05692     opics[nopic].nedges = mopi->nedges;
05693     opics[nopic].next = mopi->next;
05694     opics[nopic].loop = mopi->loop;
05695     opics[nopic].ctloop = mopi->ctloop;
05696     opics[nopic].mom0lg = mopi->mom0lg;
05697     opics[nopic].nodes = opisp + nopisp;
05698     nopisp += nn;
05699     opistkptr = stp;
05700
05701     nopic++;
05702 }
05703
05704 //-----
05705 void MConn::addBridge(int n0, int n1, int nex, int nextot)
05706 {
05707     if (nbridges >= sedges) {
05708         erEnd("MConn::addBridge: table overflow");
05709     }
05710     if (n0 <= n1) {
05711         bridges[nbridges].nodes[0] = n0;
05712         bridges[nbridges].nodes[1] = n1;
05713     } else {
05714         bridges[nbridges].nodes[0] = n1;
05715         bridges[nbridges].nodes[1] = n0;
05716     }
05717     bridges[nbridges].next = Min(nex, nextot-nex);
05718     nbridges++;
05719 }
05720
05721 //-----
05722 void MConn::addArtic(int nd, int mul)
05723 {
05724     articuls[nd] += mul;
05725     if (articuls[nd] == mul) {
05726         narticuls++;
05727     } else if (articuls[nd] == 0) {
05728         narticuls--;
05729     }
05730 }
05731
05732 //-----
05733 void MConn::addBlock(MCBlock *mblk, int stp)
05734 {
05735     int j, nn;
05736
05737     if (nopic >= snodes) {
05738         erEnd("MConn::addOPic: table overflow");
05739     }
05740
05741     nn = blkstkpctr - stp;
05742     for (j = 0; j < nn; j++) {
05743         blksp[nblksp+j][0] = blkstk[stp+j][0];
05744         blksp[nblksp+j][1] = blkstk[stp+j][1];
05745     }
05746     blocks[nblocks].edges = blksp + nblksp;
05747     blocks[nblocks].nmedges = nn;
05748     blocks[nblocks].nartps = mblk->nartps;
05749     blocks[nblocks].loop = mblk->loop;
05750     nblksp += nn;
05751     blkstkpctr = stp;
05752
05753     nblocks++;
05754 }
05755
05756 //-----
05757 void MConn::addBlockSelf(int nd, int mult)

```

```

05758 {
05759     MCBLOCK mblk;
05760     int j, stp;
05761
05762     mblk.edges = NULL;
05763     mblk.nmedges = 0;
05764     mblk.nartps = 1;
05765     mblk.loop = 1;
05766
05767     for (j = 0; j < mult; j++) {
05768         stp = blkstkptr;
05769         pushEdge(nd, nd);
05770         addBlock(&mblk, stp);
05771     }
05772 }
05773
05774 //-----
05775 void MConn::print(void)
05776 {
05777     int j, k;
05778
05779     grcc_fprintf(GRCC_Stdout, "+++ MConn object: snodes=%d, sedges=%d\n", snodes, sedges);
05780     grcc_fprintf(GRCC_Stdout, "    nopic=%d, nlpopic=%d, nbacked=%d, nctopic=%d, "
05781         "nbridges=%d, ne0bridges=%d, nelbridges=%d\n",
05782         nopic, nlpopic, nbacked, nctopic,
05783         nbridges, ne0bridges, nelbridges);
05784     grcc_fprintf(GRCC_Stdout, "    nblocks=%d, neblocks=%d, nalblocks=%d, narticuls=%d,
nselfloop=%d\n",
05785         nblocks, neblocks, nalblocks, narticuls, nselfloops);
05786     grcc_fprintf(GRCC_Stdout, "    nmultiedges=%d\n", nmultiedges);
05787
05788     grcc_fprintf(GRCC_Stdout, "    cEdges (%d)\n", sedges);
05789     if (sedges > 0) {
05790         for (j = 0; j < sedges; j++) {
05791             grcc_fprintf(GRCC_Stdout, "        %2d: (%d,%d)[%2d*%2ld] \n", j,
05792                 cedges[j].nodes[0], cedges[j].nodes[1],
05793                 cedges[j].momdir, cedges[j].momset);
05794         }
05795     }
05796     grcc_fprintf(GRCC_Stdout, "\n");
05797
05798     grcc_fprintf(GRCC_Stdout, "    lPI components (%d)\n", nopic);
05799     for (j = 0; j < nopic; j++) {
05800         grcc_fprintf(GRCC_Stdout, "        %d: nleg=%d, nnodes=%d, nedge=%d, ",
05801             j, opics[j].nlegs, opics[j].nnodes, opics[j].nedges);
05802         grcc_fprintf(GRCC_Stdout, "next=%d, loop=%d, ctlp=%d: m0lg=%d [",
05803             opics[j].next, opics[j].loop, opics[j].ctloop,
05804             opics[j].mom0lg);
05805         for (k = 0; k < opics[j].nnodes; k++) {
05806             grcc_fprintf(GRCC_Stdout, " %d", opics[j].nodes[k]);
05807         }
05808         grcc_fprintf(GRCC_Stdout, "]\n");
05809     }
05810
05811     grcc_fprintf(GRCC_Stdout, "    bridges (%d)\n", nbridges);
05812     if (nbridges > 0) {
05813         grcc_fprintf(GRCC_Stdout, "        ");
05814         for (j = 0; j < nbridges; j++) {
05815             grcc_fprintf(GRCC_Stdout, " (%d,%d) [mom=%d] ",
05816                 bridges[j].nodes[0], bridges[j].nodes[1], bridges[j].next);
05817         }
05818         grcc_fprintf(GRCC_Stdout, "\n");
05819     }
05820
05821     grcc_fprintf(GRCC_Stdout, "    blocks (%d)\n", nblocks);
05822     for (j = 0; j < nblocks; j++) {
05823         grcc_fprintf(GRCC_Stdout, "        %d: nmedges=%d, nartps=%d, loop=%d: [",
05824             j, blocks[j].nmedges, blocks[j].nartps, blocks[j].loop);
05825         for (k = 0; k < blocks[j].nmedges; k++) {
05826             grcc_fprintf(GRCC_Stdout, " (%d,%d)", blocks[j].edges[k][0], blocks[j].edges[k][1]);
05827         }
05828         grcc_fprintf(GRCC_Stdout, "]\n");
05829     }
05830
05831     grcc_fprintf(GRCC_Stdout, "    articulation points (%d)\n", narticuls);
05832     if (narticuls > 0) {
05833         grcc_fprintf(GRCC_Stdout, "        ");
05834         for (j = 0; j < snodes; j++) {
05835             if (articuls[j] != 0) {
05836 #if 0
05837                 grcc_fprintf(GRCC_Stdout, "%d(%d) ", j, articuls[j]);
05838 #else
05839                 grcc_fprintf(GRCC_Stdout, "%d ", j);
05840 #endif
05841             }
05842         }
05843         grcc_fprintf(GRCC_Stdout, "\n");

```

```

05844     }
05845 }
05846
05847 //-----
05848 void MConn::prEdges(void)
05849 {
05850     int j;
05851
05852     grcc_fprintf(GRCC_Stdout, "  cEdges (%d)", sedges);
05853     if (sedges > 0) {
05854         for (j = 0; j < sedges; j++) {
05855             if (j % 5 == 0) {
05856                 grcc_fprintf(GRCC_Stdout, "\n    ");
05857             }
05858             grcc_fprintf(GRCC_Stdout, "(%d,%d) [%2d*%3ld] ",
05859                 cedges[j].nodes[0], cedges[j].nodes[1],
05860                 cedges[j].momdir, cedges[j].momset);
05861         }
05862         grcc_fprintf(GRCC_Stdout, "\n");
05863     }
05864 }
05865
05866 //*****
05867 // sgroup.cc
05868 //=====
05869 // compile options
05870 //=====
05871 // table of group elements
05872 //-----
05873 SGroup::SGroup(void)
05874 {
05875     // should be called before graph generation
05876     nnodes = -1;
05877     nelem = 0;
05878     elem = NULL;
05879 }
05880
05881 //-----
05882 SGroup::~SGroup(void)
05883 {
05884     int j;
05885
05886     if (elem != NULL) {
05887         for (j = GRCC_MAXGROUP-1; j >= 0; j--) {
05888             if (elem[j] != NULL) {
05889                 delete[] elem[j];
05890                 elem[j] = NULL;
05891             }
05892         }
05893         delete[] elem;
05894         elem = NULL;
05895     }
05896 }
05897
05898 //-----
05899 void SGroup::print(void)
05900 {
05901     int j;
05902
05903     grcc_fprintf(GRCC_Stdout, "SGroup : nnodes = %d, nelem=%ld, cgen=%d, csav=%d\n",
05904         nnodes, nelem, cgen, csav);
05905     grcc_fprintf(GRCC_Stdout, "  eclass =");
05906     prIntArray(necclass, eclass, "\n");
05907     for (j = 0; j < nelem; j++) {
05908         grcc_fprintf(GRCC_Stdout, "    %4d: ", j);
05909         prIntArray(nnodes, elem[j], "\n");
05910     }
05911 }
05912
05913 //-----
05914 void SGroup::newGroup(int nnd, int nclss, int *clss)
05915 {
05916     int j;
05917
05918     clearGroup();
05919     nnodes = nnd;
05920     necclass = nclss;
05921
05922     if (necclass > GRCC_MAXNODES) {
05923         erEnd("newGroup : too many classes (GRCC_MAXNODES)");
05924     }
05925     for (j = 0; j < necclass; j++) {
05926         eclass[j] = clss[j];
05927     }
05928
05929     nelem = 0;
05930     if (elem == NULL) {

```

```

05931         elem = new int*[GRCC_MAXGROUP];
05932         for (j = 0; j < GRCC_MAXGROUP; j++) {
05933             elem[j] = NULL;
05934         }
05935     }
05936 }
05937
05938 //-----
05939 void SGroup::clearGroup(void)
05940 {
05941     nelelem = 0;
05942     csav = -1;
05943     cgen = -1;
05944 }
05945
05946 //-----
05947 void SGroup::delGroup(void)
05948 {
05949     int j;
05950
05951     for (j = 0; j < GRCC_MAXGROUP; j++) {
05952         if (elem[j] != NULL) {
05953             delete[] elem[j];
05954             elem[j] = NULL;
05955         }
05956     }
05957     delete[] elem;
05958     elem = NULL;
05959     clearGroup();
05960 }
05961
05962 //-----
05963 int *SGroup::genNext(void)
05964 {
05965     cgen = nextPerm(nnodes, necclass, eclass, pgr, pgq, permg, cgen);
05966     if (cgen < 0) {
05967         return NULL;
05968     }
05969     return permg;
05970 }
05971
05972 //-----
05973 void SGroup::addGroup(int *p)
05974 {
05975     int j;
05976
05977     if (nelem >= GRCC_MAXGROUP) {
05978         erEnd("addGroup : too many elements (GRCC_MAXGROUP)");
05979     }
05980     if (elem[nelem] == NULL) {
05981         elem[nelem] = new int[GRCC_MAXNODES];
05982     }
05983     for (j = 0; j < nnodes; j++) {
05984         elem[nelem][j] = p[j];
05985     }
05986     nelelem++;
05987 }
05988
05989 //-----
05990 BigInt SGroup::nElem(void)
05991 {
05992     return nelelem;
05993 }
05994
05995 //-----
05996 int *SGroup::nextElem(void)
05997 {
05998     if (nelem < 0) {
05999         cgen = nextPerm(nelem, necclass, eclass,
06000             psr, psq, perms, cgen);
06001         return perms;
06002     }
06003     if (cgen < 0) {
06004         cgen = 0;
06005     }
06006     if (cgen >= nelelem) {
06007         return NULL;
06008     } else {
06009         return elem[cgen++];
06010     }
06011 }
06012
06013 //*****
06014 // egraph.cc
06015 // Generate scalar connected Feynman graphs.
06016 //=====
06017 static const char *GRCC_ND_names[] = GRCC_ND_NAMES;

```

```

06018 static const char *GRCC_ED_names[] = GRCC_ED_NAMES;
06019
06020 //=====
06021 // class ENode
06022 //-----
06023 ENode::ENode(void)
06024 {
06025     id      = -1;
06026     egraph  = NULL;
06027     edges   = NULL;
06028     maxdeg  = 0;
06029     deg     = 0;
06030
06031     klow    = NULL;
06032     extloop = 0;
06033     ndtype  = GRCC_ND_Undef;
06034
06035     intrct  = -1;
06036 }
06037
06038 //-----
06039 ENode::ENode(EGraph *egrph, int loops, int sdeg)
06040 {
06041     id      = -1;
06042     egraph  = egrph;
06043     edges   = NULL;
06044     maxdeg  = sdeg;
06045     deg     = 0;
06046
06047     klow    = NULL;
06048     extloop = 0;
06049     ndtype  = GRCC_ND_Undef;
06050
06051     intrct  = -1;
06052
06053     edges = new int[sdeg];
06054     klow  = new int[loops+1];
06055 }
06056
06057 //-----
06058 ENode::~ENode(void)
06059 {
06060     if (klow != NULL) {
06061         delete[] klow;
06062         delete[] edges;
06063         klow = NULL;
06064         edges = NULL;
06065     }
06066 }
06067
06068 //-----
06069 void ENode::copy(ENode *en)
06070 {
06071     int d;
06072
06073     deg = en->deg;
06074     extloop = en->extloop;
06075     ndtype = en->ndtype;
06076     for (d = 0; d < deg; d++) {
06077         edges[d] = en->edges[d];
06078     }
06079 }
06080
06081 //-----
06082 void ENode::initAss(EGraph *egrph, int nid, int sdg)
06083 {
06084     id      = nid;
06085     egraph  = egrph;
06086     maxdeg  = sdg;
06087 }
06088
06089 //-----
06090 void ENode::setId(EGraph *egrph, const int nid)
06091 {
06092     id      = nid;
06093     egraph  = egrph;
06094 }
06095
06096 //-----
06097 void ENode::setExtern(int typ, int pt)
06098 {
06099     intrct = pt;
06100     if (typ == GRCC_AT_Initial || typ == GRCC_AT_Final) {
06101         extloop = typ;
06102     } else {
06103         grcc_fprintf(GRCC_Stderr, "*** ENode::setExternal:: illegal typ=%d\n",
06104 
```



```

06105         typ);
06106         erEnd("ENode::setExternal:: illegal typ");
06107     }
06108 }
06109
06110 //-----
06111 void ENode::setType(int typ)
06112 {
06113     #ifndef CHECK
06114         if (ndtype != GRCC_ND_Undef && ndtype != typ) {
06115             grcc_fprintf(GRCC_Stderr, "*** ndtype is already defined : old=%d, new = %d\n",
06116                 ndtype, typ);
06117             erEnd("ndtype is already defined");
06118         }
06119     #endif
06120     ndtype = typ;
06121 }
06122
06123 //-----
06124 void ENode::print(void)
06125 {
06126     int j;
06127
06128     grcc_fprintf(GRCC_Stdout, "Enode %d deg=%d, extl=%2d, intr=%d ",
06129         id, deg, extloop, intrct);
06130     if (egraph->bicount > 0) {
06131         grcc_fprintf(GRCC_Stdout, " (%-8s) ", GRCC_ND_names[ndtype]);
06132     }
06133     grcc_fprintf(GRCC_Stdout, "edge=[");
06134     for (j = 0; j < deg; j++) {
06135         grcc_fprintf(GRCC_Stdout, " %2d", edges[j]);
06136     }
06137     grcc_fprintf(GRCC_Stdout, "]\n");
06138 }
06139
06140 //=====
06141 // class EEdge
06142 //-----
06143 EEdge::EEdge(void)
06144 {
06145     id      = -1;
06146     egraph  = NULL;
06147     nodes[0] = -1;
06148     nodes[1] = -1;
06149     // nlegs[0] = -1;
06150     // nlegs[1] = -1;
06151     ptcl    = GRCC_PT_Undef;
06152     deleted  = False;
06153
06154     edtype  = GRCC_ED_Undef;
06155     cut      = False;
06156
06157     emom = NULL;
06158     lmom = NULL;
06159
06160     momset = 0;
06161     momdir = 0;
06162 }
06163
06164 //-----
06165 EEdge::EEdge(EGraph *egrph, int nedges, int nloops)
06166 {
06167     id      = -1;
06168     egraph  = egrph;
06169     nodes[0] = -1;
06170     nodes[1] = -1;
06171     nlegs[0] = -1;
06172     nlegs[1] = -1;
06173     ptcl    = GRCC_PT_Undef;
06174     deleted  = False;
06175
06176     edtype  = GRCC_ED_Undef;
06177     cut      = False;
06178
06179     emom = new int[nedges];
06180     lmom = new int[nloops];
06181 }
06182
06183 //-----
06184 EEdge::~EEdge(void)
06185 {
06186     if (lmom != NULL) {
06187         delete[] lmom;
06188     }
06189     if (emom != NULL) {
06190         delete[] emom;
06191     }

```

```

06192 }
06193
06194 //-----
06195 void EEdge::copy(EEdge *ee)
06196 {
06197     nodes[0] = ee->nodes[0];
06198     nodes[1] = ee->nodes[1];
06199     id       = ee->id;
06200     ext      = ee->ext;
06201     dir      = ee->dir;
06202     edtype   = ee->edtype;
06203     cut      = ee->cut;
06204 }
06205
06206 //-----
06207 void EEdge::setId(EGraph *egrph, const int eid)
06208 {
06209     id       = eid;
06210     egraph   = egrph;
06211 }
06212
06213 //-----
06214 void EEdge::setType(int typ)
06215 {
06216     #ifdef CHECK
06217         if (edtype != GRCC_ED_Undef) {
06218             erEnd("edtype is already defined");
06219         }
06220     #endif
06221     edtype = typ;
06222 }
06223
06224 //-----
06225 void EEdge::setEMom(int nedges, int *em, int dir)
06226 {
06227     int e;
06228
06229     for (e = 0; e < nedges; e++) {
06230         if (em[e] != 0) {
06231             emom[e] = dir;
06232         }
06233     }
06234 }
06235
06236 //-----
06237 void EEdge::setLMom(int k, int dir)
06238 {
06239     lmom[k] = dir;
06240 }
06241
06242 //-----
06243 void EEdge::print(void)
06244 {
06245     int zero, n, k;
06246
06247     grcc_fprintf(GRCC_Stdout, "Edge %2d ext=%d ptcl=%2d ", id, ext, ptcl);
06248     if (egrph == NULL || !egrph->assigned) {
06249         grcc_fprintf(GRCC_Stdout, "[%d, %d]", nodes[0], nodes[1]);
06250     } else {
06251         grcc_fprintf(GRCC_Stdout, "[(%d,%d), (%d,%d)]", nodes[0], nlegs[0], nodes[1], nlegs[1]);
06252     }
06253     if (momdir != 0) {
06254         grcc_fprintf(GRCC_Stdout, "[%2d*%2lu]", momdir, momset);
06255     }
06256
06257     if (egrph != NULL && egrph->bicount > 0) {
06258         if (cut) {
06259             grcc_fprintf(GRCC_Stdout, " %-9s ", "Cut");
06260         } else {
06261             grcc_fprintf(GRCC_Stdout, " %-9s ", GRCC_ED_names[edtype]);
06262         }
06263         grcc_fprintf(GRCC_Stdout, "c%2d: ", opicomp);
06264         grcc_fprintf(GRCC_Stdout, "%s%d =", (ext)?"Q":"p", id);
06265         zero = True;
06266         for (n = 0; n < egrph->nEdges; n++) {
06267             if (emom[n] != 0) {
06268                 prMomStr(emom[n], "Q", n);
06269                 zero = False;
06270             }
06271         }
06272         for (k = 0; k < egrph->nLoops; k++) {
06273             if (lmom[k] != 0) {
06274                 prMomStr(lmom[k], "k", k);
06275                 zero = False;
06276             }
06277         }
06278         if (zero) {

```

```

06279         grcc_fprintf(GRCC_Stdout, " 0");
06280     }
06281 } else {
06282     if (egraph == NULL) {
06283         grcc_fprintf(GRCC_Stdout, " egraph=NULL");
06284     } else {
06285         grcc_fprintf(GRCC_Stdout, " egraph->bicount=%d", egraph->bicount);
06286     }
06287 }
06288 grcc_fprintf(GRCC_Stdout, "\n");
06289 }
06290
06291 //=====
06292 // class EFLine
06293 //-----
06294 EFLine::EFLine(void)
06295 {
06296     ftype = FL_Open;
06297     fkind = 0;
06298     nlist = 0;
06299 }
06300
06301 //-----
06302 void EFLine::print(const char *msg)
06303 {
06304     if (ftype == FL_Open) {
06305         grcc_fprintf(GRCC_Stdout, " Open");
06306     } else if (ftype == FL_Closed) {
06307         grcc_fprintf(GRCC_Stdout, " Loop");
06308     } else {
06309         grcc_fprintf(GRCC_Stdout, " ?%d", ftype);
06310     }
06311     if (fkind == GRCC_PT_Dirac) {
06312         grcc_fprintf(GRCC_Stdout, " Dirac");
06313     } else if (fkind == GRCC_PT_Majorana) {
06314         grcc_fprintf(GRCC_Stdout, " Major");
06315     } else if (fkind == GRCC_PT_Ghost) {
06316         grcc_fprintf(GRCC_Stdout, " Ghost");
06317     } else {
06318         grcc_fprintf(GRCC_Stdout, " ?%d", fkind);
06319     }
06320     grcc_fprintf(GRCC_Stdout, " len=%d ", nlist);
06321     prIntArray(nlist, elist, msg);
06322 }
06323
06324 //=====
06325 // class EGraph
06326 //-----
06327 EGraph::EGraph(int nnodes, int nedges, int mxdeg)
06328 {
06329     // Arguments
06330     //  nnodes : the number of nodes
06331     //  nedges : the number of edges
06332     //  mxdeg  : the maximum number of degrees of nodes
06333
06334     int j;
06335
06336     opt      = NULL;
06337     mgraph   = NULL;
06338     econn    = NULL;
06339
06340     sNodes   = nnodes;
06341     sEdges   = nedges;
06342     sMaxdeg  = mxdeg;
06343     sLoops   = sEdges - sNodes + 1; // assume connected graph
06344
06345     pId      = -1;
06346     mId      = -1;
06347     aId      = -1;
06348     sId      = 0;
06349     gSubId   = -1;
06350     assigned = False;
06351
06352     nNodes   = sNodes;
06353     nEdges   = sEdges;
06354     nLoops   = 0;
06355     nExtern  = 0;
06356
06357     nsym     = 1;
06358     esym     = 1;
06359     extperm  = 1;
06360     nsym1    = 1;
06361     multp    = 1;
06362     totalc   = 0;
06363
06364     nodes = new ENode*[sNodes];
06365     for (j = 0; j < sNodes; j++) {

```

```

06366         nodes[j] = new ENode(this, sNodes, sMaxdeg);
06367     }
06368     edges = new EEdge*[sEdges];
06369     for (j = 0; j < sEdges; j++) {
06370         edges[j] = new EEdge(this, sEdges, sLoops);
06371     }
06372
06373     bidef = new int[sNodes];
06374     bilow = new int[sNodes];
06375     bicount = -1;
06376
06377     extMom = new int[sEdges+1];
06378
06379     fsign = 1;
06380     nFlines = -1;
06381     for (j = 0; j < GRCC_MAXFLINES; j++) {
06382         flines[j] = NULL;
06383     }
06384 }
06385
06386 //-----
06387 EGraph::~EGraph(void)
06388 {
06389     int j;
06390
06391     for (j = GRCC_MAXFLINES-1; j >= 0; j--) {
06392         if (flines[j] != NULL) {
06393             delete flines[j];
06394             flines[j] = NULL;
06395         }
06396     }
06397
06398     delete[] extMom;
06399
06400     delete[] bilow;
06401     delete[] bidef;
06402
06403     for (j = 0; j < sEdges; j++) {
06404         delete edges[j];
06405         edges[j] = NULL;
06406     }
06407     delete[] edges;
06408
06409     for (j = 0; j < sNodes; j++) {
06410         delete nodes[j];
06411         nodes[j] = NULL;
06412     }
06413     delete[] nodes;
06414
06415 }
06416
06417 //-----
06418 void EGraph::copy(EGraph *eg)
06419 {
06420     int nd, ed;
06421
06422     #ifdef CHECK
06423         if (sNodes < eg->nNodes || sEdges < eg->nEdges ||
06424             sMaxdeg < eg->sMaxdeg || sLoops < eg->nLoops) {
06425             erEnd("EGraph::copy: sizes are too small");
06426         }
06427     #endif
06428     model = eg->model;
06429     proc = eg->proc;
06430     sproc = eg->sproc;
06431     mgraph = eg->mgraph;
06432     opt = mgraph->opt;
06433     econn = mgraph->mconn;
06434
06435     pId = eg->pId;
06436     mId = eg->mId;
06437     gSubId = eg->gSubId;
06438     nNodes = eg->nNodes;
06439     nEdges = eg->nEdges;
06440     nExtern = eg->nExtern;
06441     nLoops = eg->nLoops;
06442
06443     for (nd = 0; nd < nNodes; nd++) {
06444         nodes[nd]->copy(eg->nodes[nd]);
06445     }
06446     for (ed = 0; ed < nEdges; ed++) {
06447         edges[ed]->copy(eg->edges[ed]);
06448     }
06449
06450     nsym = eg->nsym;
06451     esym = eg->esym;
06452     extperm = eg->extperm;

```

```

06453     nsym1    = eg->nsym1;
06454     multp    = eg->multp;
06455
06456     bicount  = -1;
06457 }
06458
06459 //-----
06460 void EGraph::setExtLoop(int nd, int val)
06461 {
06462     // set the node 'nd' being an external node (-1) or a looped vertex
06463     nodes[nd]->extloop = val;
06464 }
06465
06466 //-----
06467 void EGraph::endSetExtLoop(void)
06468 {
06469     // end of calling 'setExtLoop'.
06470
06471     int n;
06472
06473     nExtern = 0;
06474     for (n = 0; n < nNodes; n++) {
06475         if (isExternal(n)) {
06476             nExtern++;
06477         }
06478     }
06479 }
06480
06481 //-----
06482 void EGraph::fromDGraph(DGraph *dg)
06483 {
06484     int deg[GRCC_MAXNODES];
06485     int maxdeg, nextern, nconn, tc;
06486     int n0, n1, e;
06487
06488     if (dg->nnodes > GRCC_MAXNODES) {
06489         grcc_fprintf(GRCC_Stderr, "*** too many nodes (GRCC_MAXNODES)\n");
06490         GRCC_ABORT();
06491     }
06492     if (dg->nedges > GRCC_MAXEDGES) {
06493         grcc_fprintf(GRCC_Stderr, "*** too many edges (GRCC_MAXEDGES)\n");
06494         GRCC_ABORT();
06495     }
06496
06497     for (e = 0; e < dg->nedges; e++) {
06498         if (dg->edges[e][0] >= dg->nnodes || dg->edges[e][0] >= dg->nnodes) {
06499             grcc_fprintf(GRCC_Stderr, "*** undefined node:");
06500             grcc_fprintf(GRCC_Stderr, "edge[%d] = {%d, %d}\n",
06501                 e, dg->edges[e][0], dg->edges[e][0]);
06502             GRCC_ABORT();
06503         }
06504     }
06505
06506     for (n0 = 0; n0 < dg->nnodes; n0++) {
06507         deg[n0] = 0;
06508     }
06509     for (e = 0; e < dg->nedges; e++) {
06510         deg[dg->edges[e][0]]++;
06511         deg[dg->edges[e][1]]++;
06512     }
06513     maxdeg = 0;
06514     nextern = 0;
06515     for (n0 = 0; n0 < dg->nnodes; n0++) {
06516         maxdeg = Max(maxdeg, deg[n0]);
06517         if (deg[n0] == 1) {
06518             nextern++;
06519         } if (deg[n0] < 0) {
06520             grcc_fprintf(GRCC_Stderr, "+++ node %d is isolated\n", n0);
06521         }
06522     }
06523
06524     mgraph = NULL;
06525     model   = NULL;
06526     proc    = NULL;
06527     sproc   = NULL;
06528     opt     = NULL;
06529
06530     nNodes  = dg->nnodes;
06531     nEdges  = dg->nedges;
06532     nLoops  = 0;
06533     nExtern = nextern;
06534
06535     pId     = 0;
06536     mId     = 0;
06537     aId     = 0;
06538     gSubId  = 0;
06539     nsym    = 1;

```

```

06540     nsym1    = 1;
06541     esym     = 1;
06542     extperm  = 1;
06543     multp    = 1;
06544     maxdeg   = maxdeg;
06545
06546     for (n0 = 0; n0 < dg->nnodes; n0++) {
06547         nodes[n0]->deg = 0;
06548         nodes[n0]->extloop = dg->nodes[n0].extloop;
06549     }
06550     for (e = 0; e < dg->nedges; e++) {
06551         n0 = dg->edges[e][0];
06552         n1 = dg->edges[e][1];
06553         edges[e]->nodes[0] = n0;
06554         edges[e]->nodes[1] = n1;
06555         edges[e]->ext = (isExternal(n0) || isExternal(n1));
06556         edges[e]->momdir = 0;
06557
06558         nodes[n0]->edges[nodes[n0]->deg++] = I2Vedge(e, -1);
06559         nodes[n1]->edges[nodes[n1]->deg++] = I2Vedge(e, +1);
06560     }
06561     for (n0 = 0; n0 < dg->nnodes; n0++) {
06562         nodes[n0]->setId(this, n0);
06563         nodes[n0]->intrct = dg->nodes[n0].intrct;
06564     }
06565
06566     for (e = 0; e < nEdges; e++) {
06567         edges[e]->setId(this, e);
06568     }
06569     tc = 0;
06570     for (n0 = 0; n0 < nNodes; n0++) {
06571         if (isExternal(n0)) {
06572             setExtern(n0, 1, GRCC_AT_Initial);
06573         } else {
06574             tc += 2*nodes[n0]->extloop + nodes[n0]->deg - 2;
06575         }
06576     }
06577     totalc = tc;
06578
06579     // get the number of connected components
06580     nconn = connComp();
06581     nLoops = nEdges - nNodes + nconn;
06582
06583     bicount = -1;
06584 }
06585
06586 //-----
06587 void EGraph::fromMGraph(MGraph *mg)
06588 {
06589     // construct EGraph from adjacency matrix.
06590     // This function should be called after
06591     // EGraph(), setExtLoop() and endSetExtLoop().
06592
06593     int n0, n1, m0, m1, ed, e;
06594     int j, ni, nf;
06595     PNodeClass *pnc;
06596     int k;
06597
06598 #ifdef CHECK
06599     if (sNodes < mg->nNodes || sEdges < mg->nEdges || sMaxdeg < mg->maxdeg) {
06600         erEnd("EGraph::fromMGraph: sizes are too small");
06601     }
06602     if (sLoops < mg->nLoops) {
06603         grcc_fprintf(GRCC_Stdout, "too small sLoops : %d < %d\n", sLoops, mg->nLoops);
06604         erEnd("too small sLoops");
06605     }
06606 #endif
06607
06608     mgraph = mg;
06609     opt = mgraph->opt;
06610     econn = mgraph->mconn;
06611     sproc = opt->sproc;
06612     if (sproc == NULL) {
06613         proc = NULL;
06614         model = NULL;
06615     } else {
06616         proc = sproc->proc;
06617         model = sproc->model;
06618     }
06619
06620     nNodes = mg->nNodes;
06621     nEdges = mg->nEdges;
06622     nLoops = mg->nLoops;
06623     nExtern = mg->nExtern;
06624     pId = mg->pId;
06625     mId = mg->cDiag;
06626     aId = -1;

```

```

06627     if (opt->proc != NULL) {
06628         mId = mg->opt->proc->mgrcount;
06629         sId = mg->opt->proc->agrcount;
06630     } else if (mg->opt->sproc != NULL) {
06631         mId = mg->opt->sproc->mgrcount;
06632         sId = mg->opt->sproc->agrcount;
06633     } else {
06634         mId = mg->cDiag;
06635         sId = mg->cDiag;
06636     }
06637     gSubId = 0;
06638     nsym = mg->nsym;
06639     esym = mg->esym;
06640     extperm = 1;
06641     nsym1 = nsym;
06642     multp = 1;
06643     maxdeg = mg->maxdeg;
06644
06645     totalc = 0;
06646     if (proc != NULL) {
06647         for (j = 0; j < proc->model->ncouple; j++) {
06648             totalc += proc->clist[j];
06649         }
06650     }
06651     for (ed = 0; ed < nEdges; ed++) {
06652         edges[ed]->edtype = GRCC_ED_Undef;
06653     }
06654
06655     ed = 0;
06656     for (n0 = 0; n0 < nNodes; n0++) {
06657         nodes[n0]->deg = 0;
06658     }
06659     for (n0 = 0; n0 < nNodes; n0++) {
06660         for (e = 0; e < mg->adjMat[n0][n0]/2; e++, ed++) {
06661             edges[ed]->nodes[0] = n0;
06662             edges[ed]->nodes[1] = n0;
06663             nodes[n0]->edges[nodes[n0]->deg++] = I2Vedge(ed, -1);
06664             nodes[n0]->edges[nodes[n0]->deg++] = I2Vedge(ed, +1);
06665             edges[ed]->ext = isExternal(n0);
06666         }
06667         for (n1 = n0+1; n1 < nNodes; n1++) {
06668             for (e = 0; e < mg->adjMat[n0][n1]; e++, ed++) {
06669                 edges[ed]->nodes[0] = n0;
06670                 edges[ed]->nodes[1] = n1;
06671                 nodes[n0]->edges[nodes[n0]->deg++] = I2Vedge(ed, -1);
06672                 nodes[n1]->edges[nodes[n1]->deg++] = I2Vedge(ed, +1);
06673                 edges[ed]->ext = (isExternal(n0) || isExternal(n1));
06674             }
06675         }
06676     }
06677     #ifdef CHECK
06678     if (ed != nEdges) {
06679         grcc_fprintf(GRCC_Stdout, "*** EGraph::init: ed=%d != nEdges=%d\n",
06680             ed, nEdges+1);
06681         erEnd("EGraph::init: illegal connection");
06682     }
06683     #endif
06684
06685     for (j = 0; j < nNodes; j++) {
06686         nodes[j]->setId(this, j);
06687         nodes[j]->intrct = mg->nodes[j]->extloop;
06688     }
06689
06690     for (j = 0; j < nEdges; j++) {
06691         edges[j]->setId(this, j);
06692     }
06693
06694     ni = 0;
06695     nf = 0;
06696     if (proc != NULL) {
06697         ni = proc->ninitl;
06698         for (j = 0; j < ni; j++) {
06699             setExtern(j, proc->initlPart[j], GRCC_AT_Initial);
06700         }
06701
06702         nf = proc->nfinal;
06703         for (j = 0; j < nf; j++) {
06704             setExtern(j+ni, proc->finalPart[j], GRCC_AT_Final);
06705         }
06706         nExtern = ni + nf;
06707     } else if (sproc != NULL) {
06708         pnc = sproc->pnclass;
06709         for (j = 0; j < sproc->pnclass->nclass; j++) {
06710             if (pnc->type[j] == GRCC_AT_Initial
06711                 || pnc->type[j] == GRCC_AT_External) {
06712                 for (k = pnc->cl2nd[j]; k < pnc->cl2nd[j+1]; k++) {
06713                     setExtern(j, k, GRCC_AT_Initial);

```

```

06714         ni++;
06715     }
06716     } else if (sproc->pnclass->type[j] == GRCC_AT_Final) {
06717         for (k = pnc->cl2nd[j]; k < pnc->cl2nd[j+1]; k++) {
06718             setExtern(j, k, GRCC_AT_Final);
06719             nf++;
06720         }
06721     }
06722 }
06723 }
06724 nExtern = ni + nf;
06725 }
06726 totalc = 0;
06727 for (j = 0; j < nNodes; j++) {
06728     if (!isExternal(j)) {
06729         totalc += 2*nodes[j]->extloop + nodes[j]->deg - 2;
06730     }
06731 }
06732
06733 for (ed = 0; ed < nEdges; ed++) {
06734     edges[ed]->momdir = 0;
06735 }
06736 for (e = 0; e < econn->sedges; e++) {
06737     m0 = econn->cedges[e].nodes[0];
06738     m1 = econn->cedges[e].nodes[1];
06739
06740     bool found = False;
06741     for (ed = 0; ed < nEdges; ed++) {
06742         if (edges[ed]->momdir != 0) {
06743             continue;
06744         }
06745         n0 = edges[ed]->nodes[0];
06746         n1 = edges[ed]->nodes[1];
06747         if (m0 == n0 && m1 == n1) {
06748             edges[ed]->momdir = econn->cedges[e].momdir;
06749             edges[ed]->momset = econn->cedges[e].momset;
06750             found = True;
06751             break;
06752         } else if (m0 == n1 && m1 == n0) {
06753             edges[ed]->momdir = - econn->cedges[e].momdir;
06754             edges[ed]->momset = econn->cedges[e].momset;
06755             found = True;
06756             break;
06757         }
06758     }
06759
06760     if (!found) {
06761         grcc_fprintf(GRCC_Stdout, "*** EGraph::fromMGraph:edge (%d->%d) is not found\n",
06762             m0, m1);
06763     }
06764 }
06765
06766 bicount = -1;
06767 }
06768
06769 //-----
06770 ENode *EGraph::setExtern(int n0, int pt, int ndtyp)
06771 {
06772     ENode *nd;
06773     int npt, ept, eg;
06774
06775     nd = nodes[n0];
06776     if (nd->deg != 1) {
06777         if (prlevel > 0) {
06778             grcc_fprintf(GRCC_Stderr, "*** illegal external particle %d, %d, %d\n",
06779                 n0, pt, ndtyp);
06780         }
06781         erEnd("illegal external particle");
06782     }
06783
06784     // particle comes into the node
06785     if (ndtyp == GRCC_AT_Initial) {
06786         npt = pt;
06787     } else if (ndtyp == GRCC_AT_Final) {
06788         npt = -pt;
06789     } else {
06790         npt = 0;
06791         if (prlevel > 0) {
06792             grcc_fprintf(GRCC_Stderr, "*** illegal type of an external particle : "
06793                 "node = %d, particle = %d, type = %d", n0, pt, ndtyp);
06794         }
06795         erEnd("illegal type of an external particle");
06796     }
06797
06798     // edge attached to the external node
06799     eg = n0;
06800

```



```

06801 // particle flows on e from leg=0 to 1.
06802 if (model != NULL) {
06803     npt = model->normalParticle(npt);
06804     ept = model->normalParticle(npt);
06805 } else {
06806     npt = 0;
06807     ept = 0;
06808 }
06809
06810 nd->setExtern(ndtyp, npt);
06811 edges[eg]->ptcl = ept;
06812
06813 return nd;
06814 }
06815
06816 //-----
06817 void EGraph::print(void)
06818 {
06819     // print the EGraph
06820
06821     int nd, ed, nlp;
06822
06823     nlp = nEdges - nNodes + 1;
06824     grcc_fprintf(GRCC_Stdout, "\nEGraph\n");
06825     grcc_fprintf(GRCC_Stdout, "    pId=%d, gSubId=%d, mId=%d, aId=%d, sId=%d\n",
06826                 pId, gSubId, mId, aId, sId);
06827     grcc_fprintf(GRCC_Stdout, "    sNodes=%d, sEdges=%d, sMaxdeg=%d, sLoops=%d\n",
06828                 sNodes, sEdges, sMaxdeg, sLoops);
06829     grcc_fprintf(GRCC_Stdout, "    nNodes=%d, nEdges=%d, nExtern=%d, nLoops=%d, totalc=%d\n",
06830                 nNodes, nEdges, nExtern, nlp, totalc);
06831     grcc_fprintf(GRCC_Stdout, "    ");
06832     if (model == NULL) {
06833         grcc_fprintf(GRCC_Stdout, "model=NULL,");
06834     } else {
06835         grcc_fprintf(GRCC_Stdout, "model=\"%s\",", model->name);
06836     }
06837     if (proc == NULL) {
06838         grcc_fprintf(GRCC_Stdout, "proc=NULL,");
06839     } else {
06840         grcc_fprintf(GRCC_Stdout, "proc=%d,", proc->id);
06841     }
06842     if (sproc == NULL) {
06843         grcc_fprintf(GRCC_Stdout, "sproc=NULL,");
06844     } else {
06845         grcc_fprintf(GRCC_Stdout, "sproc=%d,", sproc->id);
06846     }
06847     grcc_fprintf(GRCC_Stdout, "\n");
06848     grcc_fprintf(GRCC_Stdout, "    assigned=%d, sym = (%d * %d) ",
06849                 assigned, nsym, esym);
06850     grcc_fprintf(GRCC_Stdout, "extperm=%d, nsym1=%d, multp=%d\n",
06851                 extperm, nsym1, multp);
06852
06853     grcc_fprintf(GRCC_Stdout, " Nodes\n");
06854     for (nd = 0; nd < nNodes; nd++) {
06855         if (isExternal(nd)) {
06856             grcc_fprintf(GRCC_Stdout, "    %2d Extern ", nd);
06857         } else {
06858             grcc_fprintf(GRCC_Stdout, "    %2d Vertex ", nd);
06859         }
06860         nodes[nd]->print();
06861     }
06862     grcc_fprintf(GRCC_Stdout, " Edges\n");
06863     for (ed = 0; ed < nEdges; ed++) {
06864         if (edges[ed]->ext) {
06865             grcc_fprintf(GRCC_Stdout, "    %2d Extern ", ed);
06866         } else {
06867             grcc_fprintf(GRCC_Stdout, "    %2d Intern ", ed);
06868         }
06869         edges[ed]->print();
06870     }
06871     if (bicount > 0) {
06872         grcc_fprintf(GRCC_Stdout, " Biconn: nopicomp=%d, nopi2p=%d, opi2plp=%d, nadj2ptv=%d\n",
06873                     nopicomp, nopi2p, opi2plp, nadj2ptv);
06874     }
06875     grcc_fprintf(GRCC_Stdout, "\n");
06876
06877     if (nFlines > 0) {
06878         prFlines();
06879     }
06880 }
06881
06882 //-----
06883 void EGraph::printPy(FILE *fp, long mId)
06884 {
06885     if (!assigned) {
06886         mgraph->printPy(fp, mId);
06887         return;
06888     }

```

```

06888     }
06889
06890     fprintf(fp, "#AGraph : (gseq, {spid, gid, ...}, nodes, node-group)\n");
06891     fprintf(fp, " (%ld,\n", mId);
06892
06893     // {'proc':0, 'subproc':0, 'tgraph':1, ... }
06894     fprintf(fp, " {'proc':%d, 'subproc':%d, ", proc->id, sproc->id);
06895     fprintf(fp, "'tgraph':%ld, ", mId);
06896     fprintf(fp, "'OPI':%s, ", BOOLSTR(opt->values[GRCC_OPT_1PI]));
06897     fprintf(fp, "'agraph':%ld, \n", aId);
06898     fprintf(fp, " 'nnodes':%d, 'nnsym':%ld, 'nesym':%ld, },\n",
06899             nNodes, nsym, esym);
06900     // #nodes
06901     fprintf(fp, " [ #nodes\n");
06902     for (int nd = 0; nd < nNodes; nd++) {
06903         ENode *end = nodes[nd];
06904
06905         // #node 0: external=phi
06906         // #node 4: phi3=0
06907         fprintf(fp, " [ #node %d: ", nd);
06908         fprintf(fp, "\n");
06909
06910         // {'deg':1, 'cid':-3, ..., }
06911         fprintf(fp, " {'deg':%d, 'cid':%d, 'extern':%s, ",
06912                 end->deg, 0, BOOLSTR(isExternal(nd)));
06913         fprintf(fp, "'intr':%d, 'loop':%d, },\n",
06914                 end->intrct, end->extloop);
06915
06916         // legs : (edge, leg, particle)
06917         // [ (0, 1, 0), ... ],
06918         fprintf(fp, " [ ");
06919         for (int lg = 0; lg < end->deg; lg++) {
06920             int ed = V2Iedge(end->edges[lg]);
06921             int ptcl = edges[ed]->ptcl;
06922             int edlg = (edges[ed]->nodes[0] == nd) ? 1 : 0;
06923             int nnd = edges[ed]->nodes[edlg];
06924             int nlg = edges[ed]->nlegs[edlg];
06925             fprintf(fp, "%d, %d, %d), ", nnd, ptcl, nlg);
06926         }
06927         fprintf(fp, "],\n");
06928         fprintf(fp, " ],\n"); // end #node
06929     }
06930     fprintf(fp, " ], #node\n");
06931     fprintf(fp, " [ # group of 1 elements\n");
06932     fprintf(fp, " [], #\n");
06933     fprintf(fp, " ], # group\n");
06934     fprintf(fp, ")\n");
06935 }
06936
06937 //-----
06938 void EGraph::prFLines(void)
06939 {
06940     int j;
06941
06942     grcc_fprintf(GRCC_Stdout, " Fermion lines %d, sign=%d (mId=%ld, aId=%ld)\n",
06943                 nFlines, fsign, mId, aId);
06944     for (j = 0; j < nFlines; j++) {
06945         grcc_fprintf(GRCC_Stdout, "%4d ", j);
06946         flines[j]->print("\n");
06947     }
06948 }
06949
06950 //-----
06951 int EGraph::dirEdge(int n, int e)
06952 {
06953     if (nodes[n]->edges[e] > 0) {
06954         return 1;
06955     } else {
06956         return -1;
06957     }
06958 }
06959
06960 //-----
06961 int EGraph::cmpMom(int *lm0, int *em0, int *lm1, int *em1)
06962 {
06963     int lp, ed, cmp, sgn = 0;
06964
06965     for (lp = 0; lp < nLoops; lp++) {
06966         if (lm0[lp] != 0) {
06967             if (lm1[lp] == 0) {
06968                 return 1;
06969             } else if (sgn == 0) {
06970                 sgn = lm0[lp]*lm1[lp]; // cancel first non-zero elements
06971 #ifdef CHECK
06972                 if (Abs(sgn) != 1) {
06973                     erEnd("cmpMom : illegal element of lmom");
06974                 }

```

```

06975 #endif
06976         } else {
06977             cmp = lm0[lp] - sgn*lm1[lp];
06978             if (cmp != 0) {
06979                 return cmp;
06980             }
06981         }
06982     } else if (lm1[lp] != 0) { // lm0[lp] == 0
06983         return -1;
06984     }
06985 }
06986 for (ed = 0; ed < nEdges; ed++) {
06987     if (em0[ed] != 0) {
06988         if (em1[ed] == 0) {
06989             return 1;
06990         } else if (sgn == 0) {
06991             sgn = em0[ed]*em1[ed]; // cancel first non-zero elements
06992 #ifdef CHECK
06993             if (Abs(sgn) != 1) {
06994                 erEnd("cmpMom : illegal element of emom");
06995             }
06996 #endif
06997         } else {
06998             cmp = em0[ed] - sgn*em1[ed];
06999             if (cmp != 0) {
07000                 return cmp;
07001             }
07002         }
07003     } else if (em1[ed] != 0) { // em0[ed] == 0
07004         return -1;
07005     }
07006 }
07007 return 0;
07008 }
07009
07010 //-----
07011 int EGraph::groupLMom(int *grp, int *ed2gr)
07012 {
07013     // grp[g] : the number of elements in the group g (in [0,...,(ngrp-1)])
07014     // ed2gr[ed] : group number of edge ed.
07015
07016     int ed0, ed1, cmp, ngrp = -1;
07017
07018     for (ed0 = 0; ed0 < nEdges; ed0++) {
07019         ed2gr[ed0] = -1;
07020         grp[ed0] = 0;
07021     }
07022
07023     for (ed0 = 0; ed0 < nEdges; ed0++) {
07024         if (ed2gr[ed0] < 0) {
07025             ngrp++;
07026             ed2gr[ed0] = ngrp;
07027             grp[ngrp]++;
07028             for (ed1 = ed0+1; ed1 < nEdges; ed1++) {
07029                 cmp = cmpMom(edges[ed0]->lmom, edges[ed0]->emom,
07030                             edges[ed1]->lmom, edges[ed1]->emom);
07031                 if (cmp == 0) {
07032                     ed2gr[ed1] = ngrp;
07033                     grp[ngrp]++;
07034                 }
07035             }
07036         }
07037     }
07038     ngrp++;
07039
07040     return ngrp;
07041 }
07042
07043 //-----
07044 Bool EGraph::optQGrafM(Options *opt)
07045 {
07046     int *qgopt = opt->qgopt;
07047     int nopis[GRCC_MAXEDGES];
07048     ULong mext = 0;
07049     mext = (~ mext) << nExtern;
07050
07051 #ifdef PRINT
07052     grcc_fprintf(GRCC_Stdout, "optQGrafM:");
07053     print();
07054 #endif
07055
07056 #ifdef PRINT
07057     grcc_fprintf(GRCC_Stdout, "optQGrafM: %8ld\n", mId);
07058     econn->print();
07059 #endif
07060
07061     // count the number of self-energy 1PI components.

```

```

07062     int maxlegs = 0;
07063     for (int j = 0; j < econn->nopic; j++) {
07064         maxlegs = Max(maxlegs, econn->opics[j].nlegs);
07065     }
07066     if (maxlegs >= GRCC_MAXEDGES) {
07067         grcc_fprintf(GRCC_Stdout, "*** table overflow\n");
07068         GRCC_ABORT();
07069     }
07070     for (int k = 0; k < GRCC_MAXEDGES; k++) {
07071         nopis[k] = 0;
07072     }
07073     for (int j = 0; j < econn->nopic; j++) {
07074         nopis[econn->opics[j].nlegs]++;
07075     }
07076
07077     //
07078     if (qgopt[GRCC_QGRAF_OPT_ONEPI] > 0) {
07079         if (econn->nopic != 1) {
07080             return False;
07081         }
07082     } else if (qgopt[GRCC_QGRAF_OPT_ONEPI] < 0) {
07083         if (econn->nopic < 2) {
07084             return False;
07085         }
07086     }
07087
07088     if (qgopt[GRCC_QGRAF_OPT_ONSHELL] != 0) {
07089         if (nExtern == 1) {
07090             if (qgopt[GRCC_QGRAF_OPT_ONSHELL] > 0) {
07091                 if (econn->nopic != 1) {
07092                     return False;
07093                 }
07094             } else if (qgopt[GRCC_QGRAF_OPT_ONSHELL] < 0) {
07095                 if (econn->nopic == 1) {
07096                     return False;
07097                 }
07098             }
07099         } else {
07100             if (qgopt[GRCC_QGRAF_OPT_ONSHELL] > 0) {
07101                 if (econn->nelbridges > 0) {
07102                     return False;
07103                 }
07104             } else if (qgopt[GRCC_QGRAF_OPT_ONSHELL] < 0) {
07105                 if (econn->nelbridges <= 0) {
07106                     return False;
07107                 }
07108             }
07109         }
07110     }
07111
07112     if (qgopt[GRCC_QGRAF_OPT_NOSNAIL] != 0) {
07113         if (qgopt[GRCC_QGRAF_OPT_NOSNAIL] > 0) {
07114             if (nExtern == 1) {
07115                 if (econn->nblocks != 1) {
07116                     return False;
07117                 }
07118             } else {
07119                 if (mgraph->mconn->ne0bridges >= 1 ||
07120                     mgraph->mconn->nalblocks >= 1) {
07121                     return False;
07122                 }
07123             }
07124         } else if (qgopt[GRCC_QGRAF_OPT_NOSNAIL] < 0) {
07125             if (nExtern == 1) {
07126                 if (econn->nblocks == 1) {
07127                     return False;
07128                 }
07129             } else {
07130                 if (mgraph->mconn->ne0bridges < 1 &&
07131                     mgraph->mconn->nalblocks < 1) {
07132                     return False;
07133                 }
07134             }
07135         }
07136     }
07137
07138     if (qgopt[GRCC_QGRAF_OPT_NOTADPOLE] != 0) {
07139         if (nExtern == 1) {
07140             if (qgopt[GRCC_QGRAF_OPT_NOTADPOLE] > 0) {
07141                 if (econn->nopic != 1) {
07142                     return False;
07143                 }
07144             } else if (qgopt[GRCC_QGRAF_OPT_NOTADPOLE] < 0) {
07145                 if (econn->nopic == 1) {
07146                     return False;
07147                 }
07148             }

```

```

07149     } else {
07150         if (qgopt[GRCC_QGRAF_OPT_NOTADPOLE] > 0) {
07151             if (mgraph->mconn->ne0bridges != 0) {
07152                 return False;
07153             }
07154         } else if (qgopt[GRCC_QGRAF_OPT_NOTADPOLE] < 0) {
07155             if (mgraph->mconn->ne0bridges == 0) {
07156                 return False;
07157             }
07158         }
07159     }
07160 }
07161
07162 if (qgopt[GRCC_QGRAF_OPT_NOSIGMA] != 0) {
07163     bool ok = True;
07164     if (nExtern != 2) {
07165         if (nopis[2] > 1) {
07166             ok = False;
07167         }
07168     }
07169     for (int j = 0; j < econn->nopic; j++) {
07170         if (econn->opics[j].nlegs >= 2 &&
07171             econn->opics[j].nlegs == econn->opics[j].mom0lg) {
07172             ok = False;
07173         }
07174     }
07175
07176     // loop momenta
07177     for (int j = 0; j < econn->sedges - 1; j++) {
07178         ULong momj = econn->cedges[j].momset;
07179         if (momj == 0) {
07180             continue;
07181         }
07182         int extj = 0;
07183         if (econn->cedges[j].nodes[0] < nExtern) {
07184             extj = 1;
07185         } else if (econn->cedges[j].nodes[1] < nExtern) {
07186             extj = 1;
07187         }
07188         for (int k = j+1; k < econn->sedges; k++) {
07189             ULong momk = econn->cedges[k].momset;
07190             if (momk == 0) {
07191                 continue;
07192             }
07193             int extk = 0;
07194             if (econn->cedges[k].nodes[0] < nExtern) {
07195                 extk = 1;
07196             } else if (econn->cedges[k].nodes[1] < nExtern) {
07197                 extk = 1;
07198             }
07199             if (momj == momk) {
07200                 if (nExtern == 2) {
07201                     if (extj + extk != 2) {
07202                         ok = False;
07203                     }
07204                 } else {
07205                     ok = False;
07206                 }
07207             }
07208         }
07209     }
07210     if (qgopt[GRCC_QGRAF_OPT_NOSIGMA] > 0) {
07211         if (! ok) {
07212             return False;
07213         }
07214     } else if (qgopt[GRCC_QGRAF_OPT_NOSIGMA] < 0) {
07215         if (ok) {
07216             return False;
07217         }
07218     }
07219 }
07220
07221 if (qgopt[GRCC_QGRAF_OPT_SIMPLE] > 0) {
07222     if (mgraph->selfloop || mgraph->multiedge) {
07223         return False;
07224     }
07225 } else if (qgopt[GRCC_QGRAF_OPT_SIMPLE] < 0) {
07226     if (!mgraph->selfloop && !mgraph->multiedge) {
07227         return False;
07228     }
07229 }
07230
07231 if (qgopt[GRCC_QGRAF_OPT_BIPART] > 0) {
07232     if (! mgraph->bipart) {
07233         return False;
07234     }
07235 } else if (qgopt[GRCC_QGRAF_OPT_BIPART] < 0) {

```

```

07236         if (mgraph->bipart) {
07237             return False;
07238         }
07239     }
07240     // GRCC_QGRAF_OPT_CYCLI
07241     if (qgopt[GRCC_QGRAF_OPT_CYCLI] != 0) {
07242         int nb = 0;
07243         for (int k = 0; k < econn->nblocks; k++) {
07244             if (econn->blocks[k].loop > 0) {
07245                 nb++;
07246             }
07247         }
07248         if (qgopt[GRCC_QGRAF_OPT_CYCLI] > 0) {
07249             if (nb > 1) {
07250                 return False;
07251             }
07252         } else if (qgopt[GRCC_QGRAF_OPT_CYCLI] < 0) {
07253             if (nb <= 1) {
07254                 return False;
07255             }
07256         }
07257     }
07258     return True;
07259 }
07260 }
07261
07262 //-----
07263 Bool EGraph::optQGrafA(Options *opt)
07264 {
07265     #ifdef PRINT
07266         grcc_fprintf(GRCC_Stdout, "optQGrafA: %8ld\n", mId);
07267         econn->print();
07268     #endif
07269     Bool retval = True;
07270     if (opt->qgopt[GRCC_QGRAF_OPT_FLOOP] != 0) {
07271         for (int fl=0; fl < nFlines; fl++) {
07272             if (flines[fl]->ftype == FL_Closed) {
07273                 if (flines[fl]->nlist % 2 != 0) {
07274                     retval = False;
07275                     break;
07276                 }
07277             }
07278         }
07279         // `notfloop_` is the dual of `floop_`, so flip the decision:
07280         if (opt->qgopt[GRCC_QGRAF_OPT_FLOOP] == -1) {
07281             retval = (retval == True ? False : True);
07282         }
07283     }
07284     return retval;
07285 }
07286
07287 //-----
07288 Bool EGraph::isOptE(void)
07289 {
07290     EGraph edupv = EGraph(sNodes, sEdges, sMaxdeg);
07291     EGraph *edup = &edupv;
07292     int grp[GRCC_MAXEDGES], ed2gr[GRCC_MAXEDGES];
07293     int g, ed, ngrp;
07294     Bool ok;
07295     int minopi2p;
07296
07297     // if (opt->values[GRCC_OPT_No2PtL1PI] == 0
07298     //     && opt->values[GRCC_OPT_NoAdj2PtV] == 0) {
07299     //     return True;
07300     // }
07301
07302     for (ed = 0; ed < nEdges; ed++) {
07303         edges[ed]->cut = False;
07304     }
07305
07306     biconnE();
07307
07308     // QGraf options
07309     if (! optQGrafM(opt)) {
07310         return False;
07311     }
07312
07313     if (opt->values[GRCC_OPT_NoAdj2PtV] > 0) {
07314         if (nadj2ptv > 0) {
07315             return False;
07316         }
07317     } else if (opt->values[GRCC_OPT_NoAdj2PtV] < 0) {
07318         if (nadj2ptv < 1) {
07319             return False;
07320         }
07321     }
07322 }

```

```

07323     if (opt->values[GRCC_OPT_No2PtL1PI] == 0) {
07324         return True;
07325     }
07326
07327     if (nExtern == 2 && nopicomp == 1) {
07328         minopi2p = 2;
07329     } else {
07330         minopi2p = 1;
07331     }
07332
07333     if (opi2plp > 0 && nopi2p >= minopi2p) {
07334         if (opt->values[GRCC_OPT_No2PtL1PI] > 0) {
07335             return False;
07336         } else if (opt->values[GRCC_OPT_No2PtL1PI] < 0) {
07337             return True;
07338         }
07339     }
07340
07341     edup->copy(this);
07342     ngrp = groupLMom(grp, ed2gr);
07343
07344     for (g = 0; g < ngrp; g++) {
07345         ok = True;
07346         if (grp[g] < 2) {
07347             continue;
07348         }
07349         for (ed = 0; ed < nEdges; ed++) {
07350             if (ed2gr[ed] == g) {
07351                 if (edges[ed]->ext) {
07352                     ok = False;
07353                     break;
07354                 }
07355                 edup->edges[ed]->cut = True;
07356             } else {
07357                 edup->edges[ed]->cut = False;
07358             }
07359         }
07360         if (ok) {
07361             edup->gSubId++;
07362             edup->biconnE();
07363
07364             if (edup->nExtern == 2 && edup->nopicomp == 1) {
07365                 minopi2p = 2;
07366             } else {
07367                 minopi2p = 1;
07368             }
07369
07370             if (edup->opi2plp > 0 && edup->nopi2p >= minopi2p) {
07371                 if (opt->values[GRCC_OPT_No2PtL1PI] > 0) {
07372                     return False;
07373                 } else if (opt->values[GRCC_OPT_No2PtL1PI] < 0) {
07374                     return True;
07375                 }
07376             }
07377         }
07378     }
07379
07380     if (opt->values[GRCC_OPT_No2PtL1PI] > 0) {
07381         return True;
07382     } else if (opt->values[GRCC_OPT_No2PtL1PI] < 0) {
07383         return False;
07384     } else {
07385         erEnd("isOptE: illegal control");
07386         return False;
07387     }
07388 }
07389
07390 //-----
07391 int EGraph::findRoot(void)
07392 {
07393     int root, vr, er, e, nd0, nd1;
07394
07395     root = -1;
07396     vr = -1;
07397     er = -1;
07398     for (e = 0; e < nEdges; e++) {
07399         if (edges[e]->cut || edges[e]->edtype == GRCC_ED_Deleted) {
07400             continue;
07401         }
07402         if (edges[e]->visited) {
07403             continue;
07404         }
07405         if (root < 0) {
07406             if (edges[e]->ext) {
07407                 nd0 = edges[e]->nTimer[0];
07408                 nd1 = edges[e]->nTimer[1];
07409                 if (isExternal(nd0) && !isExternal(nd1)) {

```

```

07410         root = nd1;
07411         break;
07412     } else if (!isExternal(nd0) && isExternal(nd1)) {
07413         root = nd0;
07414         break;
07415     }
07416 }
07417 }
07418 if (er < 0) {
07419     er = edges[e]->nodes[0];
07420 }
07421 if (vr < 0) {
07422     vr = edges[e]->nodes[0];
07423 }
07424 }
07425 // if root >= 0 : vertex adjacent to an external node
07426 if (root < 0) {
07427     // if vr >= 0 : a vertex
07428     if (vr >= 0) {
07429         root = vr;
07430     }
07431     // no vertex, only external nodes;
07432 } else {
07433     root = er;
07434 }
07435 }
07436 return root;
07437 }
07438
07439 //-----
07440 int EGraph::connComp(void)
07441 {
07442     // Count the number of connected component
07443     // If a connected component without free leg is not the whole graph then
07444     // return False, otherwise return True.
07445
07446     int j, ncc, nelelem;
07447
07448     for (j = 0; j < nNodes; j++) {
07449         nodes[j]->visited = -1;
07450     }
07451     ncc = 0; // # connected components
07452     nelelem = 0; // # visited nodes
07453     while (nelelem < nNodes) {
07454         for (j = 0; j < nNodes; j++) {
07455             if (nodes[j]->visited < 0) {
07456                 break;
07457             }
07458         }
07459         if (j >= nNodes) {
07460             break;
07461         }
07462         nelelem += connVisit(j, ncc);
07463         ncc++;
07464     }
07465     if (nelelem != nNodes) {
07466         erEnd("EGraph::connComp: illegal control");
07467     }
07468     return ncc;
07469 }
07470
07471 //-----
07472 int EGraph::connVisit(int nd, int ncc)
07473 {
07474     // Visiting connected node used for 'connComp'
07475
07476     int e, en, nn, j, nelelem;
07477
07478     nelelem = 1;
07479     nodes[nd]->visited = ncc;
07480     for (e = 0; e < nodes[nd]->deg; e++) {
07481         en = V2Iedge(nodes[nd]->edges[e]);
07482         for (j = 0; j < 2; j++) {
07483             nn = edges[en]->nodes[j];
07484             if (nodes[nn]->visited < 0) {
07485                 nelelem += connVisit(nn, ncc);
07486             }
07487         }
07488     }
07489     return nelelem;
07490 }
07491
07492 //-----
07493 void EGraph::biconnE(void)
07494 {
07495     int e, ie, root;
07496     int extlst[GRCC_MAXEDGES], intlst[GRCC_MAXEDGES];

```



```

07497     int opiext, opiloop;
07498
07499     nadj2ptv = 0;
07500     for (e = 0; e < nEdges; e++) {
07501         if (nodes[edges[e]->nodes[0]]->deg == 2 &&
07502             nodes[edges[e]->nodes[1]]->deg == 2 &&
07503             edges[e]->nodes[0] != edges[e]->nodes[1]) {
07504             nadj2ptv++;
07505         }
07506     }
07507
07508     biinitE();          // initialization
07509     bconn = 0;          // the number of connected components
07510
07511     opiext = 0;
07512     opiloop = 0;
07513
07514     for (e = 0; e < nEdges; e++) {
07515         // root of a spanning tree to be searched.
07516         root = findRoot();
07517         if (root < 0) {
07518             // no root : end of search
07519             break;
07520         }
07521
07522         // recursive search
07523         bisearchE(root, extlst, intlst, &opiext, &opiloop);
07524
07525         // opi2plp = Max(opi2plp, opiloop);      // ???
07526
07527         // adjust the #external for the root
07528         if (isExternal(root)) {
07529             opiext++;
07530         }
07531
07532         // 2 point function
07533         if (opiext == 2) {
07534             opi2plp = Max(opi2plp, opiloop);
07535             nopi2p++;
07536         }
07537         for (ie = 0; ie < nEdges; ie++) {
07538             if (intlst[ie]) {
07539                 edges[ie]->opicomp = nopi2p;
07540             }
07541         }
07542         nopi2p++;
07543
07544         bconn++;        // the number of connected component
07545     }
07546
07547     // momentum conservation of external particles
07548     extMomConsv();
07549
07550     #ifdef CHECK
07551     chkMomConsv();
07552     #endif
07553     return;
07554 }
07555
07556 //-----
07557 void EGraph::biinitE(void)
07558 {
07559     int n, j, e;
07560
07561     bicount = 0;          // counter of visiting node
07562     loopm = 0;           // # of independent loop momentum
07563
07564     nopi2p = 0;
07565     opi2plp = 0;
07566     nopi2p = 0;
07567
07568     for (n = 0; n < nNodes; n++) {
07569         bidef[n] = -1;
07570         bilow[n] = -1;
07571         nodes[n]->ndtype = GRCC_ND_Undef;
07572         for (j = 0; j < nLoops; j++) {
07573             nodes[n]->klow[j] = -1;
07574         }
07575     }
07576
07577     for (e = 0; e < nEdges; e++) {
07578         // edges[e]->cut is defined before
07579         edges[e]->visited = False;
07580         edges[e]->conid = -1;          // connected component
07581         edges[e]->edtype = GRCC_ED_Undef;
07582         edges[e]->opicomp = -1;
07583         for (j = 0; j < nEdges; j++) {

```

```

07584         edges[e]->emom[j] = 0;           // coefficients of ext. mom.
07585     }
07586     for (j = 0; j < nLoops; j++) {
07587         edges[e]->lmom[j] = 0;           // coefficients of loop mom.
07588     }
07589 }
07590 }
07591
07592 //-----
07593 void EGraph::bisearchE(int nd, int *extlst, int *intlst, int *opiext, int *opiloop)
07594 {
07595     // Search in the spanning tree below 'nd'.
07596     // Arguments:
07597     //   nd       : input   : node to be visited
07598     //   extlst   : output  : set of external lines in the current lPI comp.
07599     //   intlst   : output  : set of internal lines in the current lPI comp.
07600     //   opiext   : output  : no. external lines of the current lPI component
07601     //   opiloop  : output  : no. loops of the current lPI component
07602     //
07603     // EGraph variables
07604     //   bicount  : counter of visiting node
07605     //   loopm    : no. of independent loop momentum
07606     //   nopicom  : no. of OPI components
07607     //   opi2plp  : maximum number loops among 2-point looped OPI components
07608     //   nopi2p   : no. of 2-point OPI components
07609     //   edges[ie]->opicomp : OPI component no. of the edge
07610     //
07611     int k, e, ie, ed, td, dir, j;
07612     int opiextl, opiloopl;
07613     int extlst1[GRCC_MAXEDGES]; // set of external nodes below
07614     int intlst1[GRCC_MAXEDGES]; // set of vertices in the lPI component
07615
07616     (*opiext) = 0;
07617     (*opiloop) = 0;
07618
07619     // visit node 'nd'
07620     bidef[nd] = bicount;
07621     bilow[nd] = bicount;
07622     for (k = 0; k < nLoops; k++) {
07623         nodes[nd]->klow[k] = bicount;
07624     }
07625     bicount++;
07626     for (j = 0; j < nEdges; j++) {
07627         extlst[j] = False;
07628         intlst[j] = False;
07629     }
07630
07631     // go to children : nd --> ed --> td
07632     for (e = 0; e < nodes[nd]->deg; e++) {
07633         ed = V2Iedge(nodes[nd]->edges[e]);
07634
07635         // check this edge is already visited
07636         if (edges[ed]->edtype == GRCC_ED_Deleted) {
07637             // permanently deleted edge
07638             continue;
07639         } else if (edges[ed]->visited) {
07640             // already visited (backward visit)
07641             continue;
07642         } else if (edges[ed]->cut) {
07643             // cut edge : treat as nd is an external
07644             (*opiext)++;
07645             nodes[nd]->setType(GRCC_ND_CPoint);
07646             continue;
07647         } else if (!isExternal(nd) && edges[ed]->ext) {
07648             // external
07649             edges[ed]->setType(GRCC_ED_Extern);
07650             edges[ed]->visited = True;
07651             edges[ed]->emom[ed] = 1;
07652             extlst[ed] = True;
07653             (*opiext)++;
07654             nodes[nd]->setType(GRCC_ND_CPoint);
07655             continue;
07656         }
07657     }
07658
07659     // mark visited
07660     edges[ed]->visited = True;
07661
07662     // momentum is assigned on the backward move. (nd) 1 --> 0 (td)
07663     td = edges[ed]->nodes[0];
07664     dir = 1;
07665     if (td == nd) {
07666         td = edges[ed]->nodes[1];
07667         dir = -1;
07668     }
07669
07670     // biconnected component

```

```

07671         if (bidef[td] >= 0) {
07672             // a back edge is found. Already visited. Don't go further
07673             bilow[nd] = Min(bilow[nd], bilow[td]);
07674             edges[ed]→setType(GRCC_ED_Back);
07675             intlst[ed] = True;
07676             (*opiloop)++;
07677
07678             // create new loop momentum
07679             k = loopm++;
07680             nodes[nd]→klow[k] = Min(nodes[nd]→klow[k], nodes[td]→klow[k]);
07681             edges[ed]→setLMom(k, dir);
07682
07683             // self-loop
07684             if (td == nd) {
07685                 nodes[nd]→setType(GRCC_ND_CPoint);
07686             }
07687         } else {
07688             // not a back edge
07689             // go down further
07690             bisearchE(td, extlst1, intlst1, &opiext1, &opiloop1);
07691
07692             // the set of external particles
07693             for (j = 0; j < nEdges; j++) {
07694                 extlst[j] = extlst[j] || extlst1[j];
07695             }
07696
07697             // loop momenta
07698             for (k = 0; k < nLoops; k++) {
07699                 if (nodes[td]→klow[k] <= nodes[nd]→klow[k]) {
07700                     // inside loop 'k'
07701                     edges[ed]→setLMom(k, dir);
07702                 }
07703                 nodes[nd]→klow[k] = Min(nodes[nd]→klow[k], nodes[td]→klow[k]);
07704             }
07705
07706             // cut point ?
07707             if (bilow[td] >= bidef[nd]) {
07708                 // a cut point is found
07709                 if (nodes[nd]→ndtype == GRCC_ND_Undef) {
07710                     nodes[nd]→setType(GRCC_ND_CPoint);
07711                 }
07712 #ifdef CHECK
07713                 } else if (nodes[nd]→ndtype != GRCC_ND_CPoint) {
07714                     if (prlevel > 0) {
07715                         grcc_fprintf(GRCC_Stderr, "bisearch: node %d is a cut point ",
07716                                     nd);
07717                         grcc_fprintf(GRCC_Stderr, "(not undef %d)\n",
07718                                     nodes[nd]→ndtype);
07719                     }
07720 #endif
07721                 }
07722             } // cut point
07723
07724             // bridge ?
07725             if (bilow[td] > bidef[nd]) {
07726                 // a bridge is found
07727                 if (edges[ed]→edtype == GRCC_ED_Undef) {
07728                     edges[ed]→setType(GRCC_ED_Bridge);
07729                     nodes[td]→setType(GRCC_ND_CPoint);
07730                 }
07731 #ifdef CHECK
07732                 } else if (edges[ed]→edtype != GRCC_ED_Bridge) {
07733                     if (prlevel > 0) {
07734                         grcc_fprintf(GRCC_Stderr, "bisearch: edges %d is a bridge ", ed);
07735                         grcc_fprintf(GRCC_Stderr, "(not undef %d)\n", edges[ed]→edtype);
07736                     }
07737                 }
07738             }
07739
07740             if (!edges[ed]→ext) {
07741                 // internal bridge
07742                 opiext1++; // from bridge
07743                 for (ie = 0; ie < nEdges; ie++) {
07744                     if (intlst1[ie]) {
07745                         // OPI component No.
07746                         edges[ie]→opicomp = nopicomp;
07747                         intlst1[ie] = False;
07748                     }
07749                 }
07750                 // 2pt OPI component
07751                 if (opiext1 == 2) {
07752                     opi2plp = Max(opi2plp, opiloop1);
07753
07754                     // the number of 2pt looped OPI components
07755                     nopi2p++;
07756                 }
07757                 // the number of OPI components
07758                 nopicomp++;

```

```

07758         }
07759         opiext1 = 1;    // from bridge
07760         opiloop1 = 0;
07761
07762     } else {
07763         // not a bridge
07764
07765         bilow[nd] = Min(bilow[nd], bilow[td]);
07766
07767         if (edges[ed]->edtype == GRCC_ED_Undef) {
07768             edges[ed]->setType(GRCC_ED_Inloop);
07769 #ifdef CHECK
07770         } else if (edges[ed]->edtype != GRCC_ED_Inloop) {
07771             if (prlevel > 0) {
07772                 grcc_fprintf(GRCC_Stderr, "bisearch: ");
07773                 grcc_fprintf(GRCC_Stderr, "edges %d is not undef (%d)\n",
07774                     ed, edges[ed]->edtype);
07775             }
07776 #endif
07777         }
07778         intlst1[ed] = True;
07779     }
07780
07781     // merge to the current variables
07782     for (ie = 0; ie < nEdges; ie++) {
07783         intlst[ie] = intlst[ie] || intlst1[ie];
07784     }
07785     (*opiext) += opiext1;
07786     (*opiloop) += opiloop1;
07787
07788     // linear combination of external momenta
07789     edges[ed]->setEMom(nEdges, extlst1, dir);
07790
07791     } // end of a child
07792 } // for (ed)
07793
07794 return;
07795 }
07796
07797 //-----
07798 void EGraph::extMomConsv(void)
07799 {
07800     int e, ex, lex, rs;
07801
07802     lex = -1;
07803     for (e = 0; e < nEdges; e++) {
07804         if (edges[e]->ext) {
07805             lex = e;
07806             extMom[e] = edges[e]->emom[e];
07807         } else {
07808             extMom[e] = 0;
07809         }
07810     }
07811     // eliminate momentum of the last external particle
07812     if (lex < 0) {
07813         return;
07814     }
07815     for (e = 0; e < nEdges; e++) {
07816         rs = edges[e]->emom[lex];
07817         if (rs != 0) {
07818             for (ex = 0; ex < nEdges; ex++) {
07819                 edges[e]->emom[ex] -= rs*extMom[ex];
07820             }
07821         }
07822     }
07823 }
07824
07825 //-----
07826 void EGraph::chkMomConsv(void)
07827 {
07828     // check momentum conservation
07829
07830     int esum[GRCC_MAXEDGES];
07831     int lsum[GRCC_MAXNODES];
07832     Bool ok, okn;
07833     int n, ex, lk, ej, e, dir;
07834
07835     if (bicount < 1) {
07836         return;
07837     }
07838     ok = True;
07839     for (n = 0; n < nNodes; n++) {
07840         if (isExternal(n)) {
07841             continue;
07842         }
07843         for (ex = 0; ex < nEdges; ex++) {
07844             esum[ex] = 0;

```

```

07845     }
07846     for (lk = 0; lk < nLoops; lk++) {
07847         lsum[lk] = 0;
07848     }
07849     for (ej = 0; ej < nodes[n]->deg; ej++) {
07850         e = V2Iedge(nodes[n]->edges[ej]);
07851         if (edges[e]->nodes[0] == edges[e]->nodes[0]) {
07852             continue;
07853         }
07854         dir = dirEdge(n, ej);
07855         for (ex = 0; ex < nEdges; ex++) {
07856             if (edges[e]->ext) {
07857                 esum[ex] += dir*edges[e]->emom[ex];
07858             }
07859         }
07860         for (lk = 0; lk < nLoops; lk++) {
07861             lsum[lk] += dir*edges[e]->lmom[lk];
07862         }
07863     }
07864
07865     okn = True;
07866     for (ex = 0; ex < nEdges; ex++) {
07867         if (esum[ex] != 0) {
07868             okn = False;
07869             grcc_fprintf(GRCC_Stderr, "chkMomConsv:n=%d, esum[%d]=%d\n",
07870                 n, ex, esum[ex]);
07871         }
07872     }
07873
07874     for (lk = 0; lk < nLoops; lk++) {
07875         if (lsum[lk] != 0) {
07876             okn = False;
07877             grcc_fprintf(GRCC_Stderr, "chkMomConsv:n=%d, lsum[%d]=%d\n",
07878                 n, lk, lsum[lk]);
07879         }
07880     }
07881 }
07882
07883 if (!okn) {
07884     ok = False;
07885     grcc_fprintf(GRCC_Stderr, "*** Violation of momentum conservation ");
07886     grcc_fprintf(GRCC_Stderr, "at node =%d\n",n);
07887 }
07888 }
07889
07890 if (!ok) {
07891     print();
07892     erEnd("inconsistent momentum");
07893 }
07894 }
07895
07896 //-----
07897 int EGraph::legParticle(int ed, int el)
07898 {
07899     // outgoing particle from (edge, leg) = (ed, el)
07900     return model->normalParticle((2*el-1)*edges[ed]->ptcl);
07901 }
07902
07903 //-----
07904 int EGraph::isFermion(int ed)
07905 {
07906     // return if particle on the edge ed follows Fermi statistics or not.
07907
07908     int ptcl, ptype;
07909
07910     ptcl = edges[ed]->ptcl;
07911     ptype = model->particles[Abs(ptcl)]->ptype;
07912
07913     return (ptype == GRCC_PT_Dirac || ptype == GRCC_PT_Majorana
07914         || ptype == GRCC_PT_Ghost);
07915 }
07916
07917 //-----
07918 int EGraph::fltrace(int fk, int nd0, int *fl)
07919 {
07920     // fk : kind of fermion
07921     //      = (GRCC_PT_Dirac, GRCC_PT_Majorana or GRCC_PT_Ghost)
07922     // nd0 : the last node visited
07923     // fl : list of signed edge on the fermion line.
07924     //      fl[j] :
07925     //      (V2Iedge(fl[j]), V2Ileg(fl[j])) is the next node
07926     //      fl[0] should be already defined
07927
07928     int nfl, k, i, nd, nl, e, ed, el, fgcnt, fkind, lk;
07929
07930     nfl = 1;
07931

```

```

07932 // maximal possible length of a fline is nEdges
07933 for (k = 0; k < nEdges; k++) {
07934     if (k >= nfl) {
07935         grcc_fprintf(GRCC_Stdout, "*** fltrace:illegal contorl: k=%d, nEdges=%d\n", k, nEdges);
07936         break;
07937     }
07938
07939     // get next node : nd0 ---- nd (nl)
07940     e = fl[k];
07941     ed = V2Iedge(e);
07942     el = V2Ileg(e);
07943 #ifdef CHECK
07944     if (ed > nEdges) {
07945         grcc_fprintf(GRCC_Stderr, "*** fltrace: ed=%d > nEdges=%d, fl=",
07946             ed, nEdges);
07947         prIntArray(nNodes, fl, "\n");
07948         erEnd("fltrace: illegal fl");
07949     }
07950 #endif
07951     nd = edges[ed]->nodes[el];
07952     nl = edges[ed]->nlegs[el];
07953
07954     // end of the fline
07955     if (isExternal(nd) || nd == nd0) {
07956         fgcnt = 1;
07957         break;
07958     }
07959
07960     // find leg of nd for going next
07961     fgcnt = 0;
07962     ed = 0;
07963     lk = 0;
07964     for (i = 0; i < nodes[nd]->deg; i++) {
07965         e = nodes[nd]->edges[i];
07966         ed = V2Iedge(e);
07967         fkind = model->particles[Abs(edges[ed]->ptcl)]->ptype;
07968         // lk = 1 : (# of fkind particle) is even
07969         // lk = -1 : (# of fkind particle) is odd
07970         if (fkind == fk) {
07971             if (lk == 1) {
07972                 lk = -1;
07973             } else {
07974                 lk = 1;
07975             }
07976         } else {
07977             lk = 0;
07978         }
07979         if (!edges[ed]->visited) {
07980             if (!isFermion(ed)) {
07981                 edges[ed]->visited = True;
07982             } else {
07983                 if (fkind == fk) {
07984                     // i should be the neighbor of nl
07985                     // (nl, i) or (i, nl) should be a pair of fkind
07986                     // (nl, i) : lk = -1
07987                     // (i, nl) : lk = 1
07988                     if ((lk == 1 && nl == i+1) || (lk == -1 && nl == i-1)) {
07989                         edges[ed]->visited = True;
07990                         fgcnt++;
07991                     }
07992                     if (fgcnt == 1) {
07993                         // -e = (signed edge points the next node)
07994                         fl[nfl++] = -e;
07995                         break;
07996                     }
07997                 }
07998             }
07999         }
08000     } // not visited
08001 } // end of for k
08002 if (fgcnt == 0) {
08003     grcc_fprintf(GRCC_Stdout, "*** fline: Fermion number is not conserved\n");
08004     grcc_fprintf(GRCC_Stdout, "    nd=%d, e=%d, ed=%d, fgcnt=%d\n",
08005         nd, e, ed, fgcnt);
08006     // erEnd("fline: Fermion number is not conserved");
08007 } else if (fgcnt > 1) {
08008     grcc_fprintf(GRCC_Stdout, "+++ fline: more than two fermions: check fsign\n");
08009     grcc_fprintf(GRCC_Stdout, "    nd=%d, e=%d, ed=%d, fgcnt=%d\n",
08010         nd, e, ed, fgcnt);
08011 }
08012 if (k >= nEdges || k >= nfl) {
08013     grcc_fprintf(GRCC_Stdout, "*** fline: illegal control\n");
08014     grcc_fprintf(GRCC_Stdout, "    nEdges=%d, nfl=%d, k=%d, ", nEdges, nfl, k);
08015     prIntArray(nfl, fl, "\n");
08016     erEnd("fline: illegal control");
08017 }
08018 return nfl;

```

```

08019 }
08020
08021 //-----
08022 void EGraph::getFLines(void)
08023 {
08024     int fl[GRCC_MAXNODES];
08025     int nextn, exto[GRCC_MAXNODES];
08026     int e, ed, nd, floop, nswap, nfl, fkind, el, ptcl;
08027
08028     nFlines = 0;
08029     for (ed = 0; ed < nEdges; ed++) {
08030         edges[ed]->visited = False;
08031     }
08032
08033     nextn = 0;
08034     for (ed = 0; ed < nEdges; ed++) {
08035         if (edges[ed]->visited) {
08036             continue;
08037         }
08038         if (!edges[ed]->ext) {
08039             continue;
08040         }
08041         el = 0;
08042         nd = edges[ed]->nodes[el];
08043         if (!isExternal(nd)) {
08044             el = 1;
08045             nd = edges[ed]->nodes[el];
08046             if (!isExternal(nd)) {
08047                 erEnd("*** illegal control");
08048             }
08049         }
08050         if (!isFermion(ed)) {
08051             continue;
08052         }
08053         ptcl = legParticle(ed, 1-el);
08054         if (ptcl < 0) {
08055             continue;
08056         }
08057
08058         //      el   ed   ell
08059         //      x----<----x      e = (signed edge of (ed, ell))
08060         //      nd   ptcl(>= 0)
08061         //      external
08062
08063         fl[0] = - I2Vedge(ed, el);
08064         nfl = 1;
08065         fkind = model->particles[Abs(edges[ed]->ptcl)]->ptype;
08066         edges[ed]->visited = True;
08067
08068         nfl = fltrace(fkind, nd, fl);
08069         addFLine(FL_Open, fkind, nfl, fl);
08070         e = fl[nfl-1];
08071         exto[nextn++] = edges[V2Iedge(e)]->nodes[V2Ileg(e)];
08072     }
08073
08074     // closed Fermion line
08075     floop = 0;
08076     for (ed = 0; ed < nEdges; ed++) {
08077         if (edges[ed]->visited) {
08078             continue;
08079         }
08080         if (!isFermion(ed)) {
08081             continue;
08082         }
08083         el = 0;
08084         ptcl = legParticle(ed, 1-el);
08085         if (ptcl < 0) {
08086             el = 1;
08087         }
08088         nd = edges[ed]->nodes[el];
08089
08090         //      el   ed   ell
08091         //      x----<----x      e = (signed edge of (ed, ell))
08092         //      nd   ptcl
08093
08094         fl[0] = - I2Vedge(ed, el);
08095         nfl = 1;
08096         edges[ed]->visited = True;
08097
08098         fkind = model->particles[Abs(edges[ed]->ptcl)]->ptype;
08099         nfl = fltrace(fkind, nd, fl);
08100         addFLine(FL_Closed, fkind, nfl, fl);
08101         floop++;
08102     }
08103     if (nextn > 0) {
08104         nswap = sortb(nextn, exto);
08105     } else {

```

```

08106         nswap = 0;
08107     }
08108     if ((floop+nswap) % 2 == 0) {
08109         fsign = 1;
08110     } else {
08111         fsign = -1;
08112     }
08113 }
08114
08115 //-----
08116 void EGraph::addFLine(const FLType ft, int fk, int nfl, int *fl)
08117 {
08118     int j;
08119
08120     if (nFlines >= GRCC_MAXFLINES) {
08121         erEnd("too many Fermion lines (GRCC_MAXEDGES)");
08122     }
08123     if (flines[nFlines] == NULL) {
08124         flines[nFlines] = new EFLine();
08125     }
08126     flines[nFlines]->ftype = ft;
08127     flines[nFlines]->fkind = fk;
08128     flines[nFlines]->nlist = nfl;
08129     for (j = 0; j < nfl; j++) {
08130         flines[nFlines]->elist[j] = fl[j];
08131     }
08132     nFlines++;
08133 }
08134
08135
08136 //*****
08137 // assign.cc
08138 //=====
08139 // Assign particles to the edges and interactions to nodes.
08140 // Completed graph data is saved in the form of EGraph.
08141
08142 // method : selection of assignable node
08143
08144 //=====
08145 // class NCand
08146 NCand::NCand(const NCandSt sta, const int dega, const int nilst, int *ilst)
08147 {
08148     // candidate list of interactions attached to a vertex.
08149
08150     int j;
08151
08152     deg = dega;
08153     st = sta;
08154     nilist = nilst;
08155     for (j = 0; j < nilist; j++) {
08156         ilist[j] = ilst[j];
08157     }
08158
08159 #ifdef CHECK
08160     if (st == AS_Assigned) {
08161         if (nilist != 1) {
08162             erEnd("illegal assignment");
08163         }
08164     } else if (st == AS_AssExt) {
08165         if (nilist != 1) {
08166             erEnd("illegal assignment");
08167         }
08168     }
08169 #endif
08170 }
08171
08172 //-----
08173 NCand::~NCand(void)
08174 {
08175 }
08176
08177 //-----
08178 void NCand::prNCand(const char* msg)
08179 {
08180     grcc_fprintf(GRCC_Stdout, "%d %d ", st, deg);
08181     prIntArray(nilist, ilist, msg);
08182 }
08183
08184 //=====
08185 // class ECand
08186 ECand::ECand(int dt, int nplst, int *plst)
08187 {
08188     int j;
08189
08190     det = dt;
08191     nplist = nplst;
08192     for (j = 0; j < nplist; j++) {

```



```

08193     plist[j] = plst[j];
08194 }
08195
08196 if (det && nplist != 1) {
08197     if (prlevel > 0) {
08198         grcc_fprintf(GRCC_Stderr, "*** ECand : len(plist) != 1 : det=%d ", det);
08199         prIntArrayErr(nplist, plist, "\n");
08200     }
08201     erEnd("ECand : len(plist) != 1");
08202 }
08203 }
08204
08205 //-----
08206 ECand::~ECand(void)
08207 {
08208 }
08209
08210 //-----
08211 void ECand::prECand(const char *msg)
08212 {
08213     prIntArray(nplist, plist, "");
08214     grcc_fprintf(GRCC_Stdout, " (det=%d)%s", det, msg);
08215 }
08216
08217 //=====
08218 // class ANode
08219 ANode::ANode(int dg)
08220 {
08221     int j;
08222
08223     deg = dg;
08224     nlegs = 0;
08225     anodes = new int[deg];
08226     aedges = new int[deg];
08227     aelegs = new int[deg];
08228     cand = NULL;
08229     for (j = 0; j < deg; j++) {
08230         anodes[j] = -1;
08231         aedges[j] = -1;
08232         aelegs[j] = -1;
08233     }
08234 }
08235
08236 //-----
08237 ANode::~ANode(void)
08238 {
08239     if (cand != NULL) {
08240         delete cand;
08241         cand = NULL;
08242     }
08243     if (aelegs != NULL) {
08244         delete[] aelegs;
08245         delete[] aedges;
08246         delete[] anodes;
08247         aelegs = NULL;
08248         aedges = NULL;
08249         anodes = NULL;
08250     }
08251 }
08252
08253 //-----
08254 int ANode::newleg(void)
08255 {
08256     // add a slot for a leg
08257
08258     int lg = nlegs;
08259
08260     nlegs++;
08261 #ifdef CHECK
08262     if (nlegs > deg) {
08263         grcc_fprintf(GRCC_Stderr, "*** ANode::newleg : nlegs = %d > deg = %d\n",
08264             nlegs, deg);
08265         erEnd("ANode::newleg : nlegs > deg");
08266     }
08267 #endif
08268     return lg;
08269 }
08270 }
08271
08272 //=====
08273 // class AEdge
08274 AEdge::AEdge(int n0, int l0, int n1, int l1)
08275 {
08276     nodes[0] = n0; // node of 0 side of the edge
08277     nodes[1] = n1; // node of 1 side of the edge
08278     nlegs[0] = l0; // leg of the node of 0 side of the edge
08279     nlegs[1] = l1; // leg of the node of 1 side of the edge

```

```

08280     ptcl    = GRCC_PT_Undef;    // particle defined in the model
08281     cand    = NULL;            // candidate
08282 }
08283
08284 //-----
08285 AEdge::~AEdge(void)
08286 {
08287     if (cand != NULL) {
08288         delete cand;
08289     }
08290 }
08291
08292 //=====
08293 // class Assign
08294 Assign::Assign(SProcess *sprc, MGraph *mgr, PNodeClass *pnc)
08295 {
08296     Bool ok;
08297     int j;
08298
08299     if (sprc == NULL) {
08300         erEnd("Assign: sprc == NULL");
08301     }
08302
08303     // pointers to related objects
08304     sprc      = sprc;
08305     model     = sprc->model;
08306     opt       = sprc->opt;
08307     mgr       = mgr;
08308     pnclass   = pnc;
08309
08310     egraph = mgr->egraph;
08311     if (egraph == NULL) {
08312         erEnd("Assign: egraph == NULL");
08313     }
08314
08315     astack    = sprc->astack;
08316     if (astack == NULL) {
08317         erEnd("Assign: astack == NULL");
08318     }
08319
08320     nNodes    = mgr->nNodes;
08321     nEdges    = mgr->nEdges;
08322     nExtern   = sprc->nExtern;
08323     nETotal   = 0;            // total number of edges
08324
08325     // counters of graphs
08326     nAGraphs  = 0;
08327     wAGraphs  = Fraction(0,1);
08328     nAOPI     = 0;
08329     wAOPI     = Fraction(0,1);
08330
08331     nodes     = new ANode*[nNodes];
08332     edges     = new AEdge*[nEdges];
08333
08334     for (j = 0; j < nNodes; j++) {
08335         nodes[j] = NULL;
08336     }
08337     for (j = 0; j < nEdges; j++) {
08338         edges[j] = NULL;
08339     }
08340     for (j = 0; j < model->ncouple; j++) {
08341         cplleft[j] = sprc->clist[j];
08342     }
08343
08344     // initialize astack
08345     astack->setAGraph(this);
08346     astack->checkPoint(checkpoint0);
08347
08348     // import from mgr
08349     ok = fromMGraph();
08350 #ifdef CHECK
08351     checkAG("Assign::Assign");
08352 #endif
08353
08354     if (ok) {
08355         // start assignment
08356         assignAllVertices();
08357     } else {
08358         // cannot assign
08359     }
08360
08361     // restore from the stack
08362 #ifdef CHECK
08363     astack->restoreMsg(checkpoint0, "Assign");
08364 #else
08365     astack->restore(checkpoint0);
08366 #endif

```

```

08367 }
08368
08369 //=====
08370 Assign::~Assign(void)
08371 {
08372     int j;
08373
08374     for (j = 0; j < nEdges; j++) {
08375         delete edges[j];
08376         edges[j] = NULL;
08377     }
08378     delete[] edges;
08379     edges = NULL;
08380
08381     for (j = 0; j < nNodes; j++) {
08382         delete nodes[j];
08383         nodes[j] = NULL;
08384     }
08385     delete[] nodes;
08386     nodes = NULL;
08387 }
08388
08389 //-----
08390 void Assign::prCand(const char *msg)
08391 {
08392     // Print candidate table
08393     int n, e, ne;
08394     AEdge *ed;
08395
08396     grcc_fprintf(GRCC_Stdout, "\n");
08397     grcc_fprintf(GRCC_Stdout, "+++ Candidate list: %s\n", msg);
08398     grcc_fprintf(GRCC_Stdout, "Nodes %d\n", nNodes);
08399     for (n = 0; n < nNodes; n++) {
08400         grcc_fprintf(GRCC_Stdout, "%d: edges=", n);
08401         prIntArray(nodes[n]->deg, nodes[n]->aedges, ": cand=");
08402         if (nodes[n]->cand == NULL) {
08403             grcc_fprintf(GRCC_Stdout, "NULL\n");
08404         } else {
08405             nodes[n]->cand->prNCand("\n");
08406         }
08407     }
08408     ne = Min(nEdges, nETotal);
08409     grcc_fprintf(GRCC_Stdout, "Edges %d\n", ne);
08410     for (e = 0; e < ne; e++) {
08411         ed = edges[e];
08412         if (ed == NULL) {
08413             grcc_fprintf(GRCC_Stdout, "NULL_Edge\n");
08414         } else {
08415             grcc_fprintf(GRCC_Stdout, "%d: %d->%d: cand=", e, ed->nodes[0], ed->nodes[1]);
08416             if (edges[e]->cand == NULL) {
08417                 grcc_fprintf(GRCC_Stdout, "NULL\n");
08418             } else {
08419                 edges[e]->cand->prECand("\n");
08420             }
08421         }
08422     }
08423 }
08424
08425 //=====
08426 void Assign::checkAG(const char *msg)
08427 {
08428     int n, lg;
08429     Bool ok = True;
08430
08431     for (n = 0; n < nNodes; n++) {
08432         for (lg = 0; lg < nodes[n]->deg; lg++) {
08433             if (nodes[n]->aedges[lg] < 0) {
08434                 grcc_fprintf(GRCC_Stdout, "*** checkAG:%s: n=%d, lg=%d, aedges=%d\n",
08435                     msg, n, lg, nodes[n]->aedges[lg]);
08436                 ok = False;
08437             }
08438         }
08439     }
08440     if (!ok) {
08441         prCand("checkAG");
08442         erEnd("checkAG: failed");
08443     }
08444 }
08445
08446 //=====
08447 // control of the process of particle/interaction assignment
08448
08449 //-----
08450 Bool Assign::assignAllVertices(void)
08451 {
08452     // Entry point of the assignment procedure
08453     Bool ok;

```

```

08454
08455 #ifdef CHECK
08456     checkCand("assignAllVertices:1");
08457 #endif
08458
08459     // start main part
08460 #ifdef SIMPSEL
08461     ok = selectVertexSimp(-1);
08462 #else
08463     ok = selectVertex();
08464 #endif
08465
08466 #ifdef CHECK
08467     checkCand("assignAllVertices:2");
08468 #endif
08469
08470     return ok;
08471 }
08472
08473 //-----
08474 Bool Assign::selectVertexSimp(int lastv)
08475 {
08476     // Select a vertex for assignment of particles to legs
08477     //
08478     // Argument
08479     // lastv : previously assigned vertex; Nothing when lastv<0.
08480
08481     Bool ok;
08482     int v;
08483
08484 #ifdef CHECK
08485     checkCand("selectVertexSimp");
08486 #endif
08487
08488     // find a vertex to which interaction is assigned
08489     v = selUnAssVertexSimp(lastv);
08490
08491     // no more vertex
08492     if (v < 0) {
08493
08494         ok = allAssigned();
08495         if (ok) {
08496             if (!egraph->optQGrafA(opt)) {
08497                 return False;
08498             }
08499             // egraph->biconnE(); // necessary ???
08500             opt->newAGraph(egraph);
08501         }
08502
08503         return ok;
08504     }
08505
08506     // select a leg and assign a particle to it
08507     ok = selectLeg(v, -1);
08508
08509     return ok;
08510 }
08511
08512 //-----
08513 Bool Assign::selectVertex(void)
08514 {
08515     // Select a vertex for assignment of particles to legs
08516     //
08517     // Argument
08518     // lastv : previously assigned vertex; Nothing when lastv<0.
08519
08520     Bool ok;
08521     int v;
08522
08523 #ifdef CHECK
08524     checkCand("selectVertex");
08525 #endif
08526
08527     // find a vertex to which interaction is assigned
08528     v = selUnAssVertex();
08529
08530     // no more vertex
08531     if (v < 0) {
08532
08533         ok = allAssigned();
08534         if (ok) {
08535             if (!egraph->optQGrafA(opt)) {
08536                 return False;
08537             }
08538             egraph->biconnE();
08539             opt->newAGraph(egraph);
08540         }

```

```

08541
08542         return ok;
08543     }
08544
08545     // select a leg and assign a particle to it
08546     ok = selectLeg(v, -1);
08547
08548     return ok;
08549 }
08550
08551 //-----
08552 Bool Assign::selectLeg(int v, int lastlg)
08553 {
08554     // Select a leg of vertex 'v' for assignment of particle,
08555     // where the last assigned leg was 'lastlg'.
08556
08557     int      pt, ln, j;
08558     Bool      ok;
08559     CheckPt   sav, sav0;
08560     int      nplist, plist[GRCC_MAXMPARTICLES2];
08561
08562 #ifdef CHECK
08563     checkNode(v, "selectLeg:0");
08564     checkCand("selectLeg");
08565 #endif
08566
08567     // find a leg to be assigned
08568     ln = selUnAssLeg(v, lastlg);
08569
08570     // no more assignable leg for the current node
08571     if (ln < 0) {
08572         // move to next vertex
08573         ok = assignVertex(v);
08574
08575         return ok;
08576     }
08577
08578     // make the list of possible particles, incoming to the vertex
08579     astack->checkPoint(sav0);
08580     nplist = candPart(v, ln, plist, GRCC_MAXMPARTICLES2);
08581
08582     // no candidate
08583     if (nplist < 1) {
08584         return False;
08585     }
08586
08587     // assign each of possible particle to the leg 'lg'.
08588     astack->checkPoint(sav);
08589     for (j = 0; j < nplist; j++) {
08590         pt = plist[j];
08591
08592         // try to assign 'pt' to (v, ln)
08593         if(assignPLeg(v, ln, pt)) {
08594             // move to next leg
08595             selectLeg(v, ln);
08596         }
08597
08598 #ifdef CHECK
08599         astack->restoreMsg(sav, "selectLeg2");
08600 #else
08601         astack->restore(sav);
08602 #endif
08603     }
08604
08605 #ifdef CHECK
08606     astack->restoreMsg(sav0, "selectLeg2");
08607 #else
08608     astack->restore(sav0);
08609 #endif
08610     return True;
08611 }
08612
08613 //-----
08614 void Assign::saveCouple(int *sav)
08615 {
08616     int j;
08617
08618     if (model->ncouple > 1) {
08619         for (j = 0; j < model->ncouple; j++) {
08620             sav[j] = cplleft[j];
08621         }
08622     }
08623 }
08624
08625 //-----
08626 void Assign::restoreCouple(int *sav)
08627 {

```

```

08628     int j;
08629
08630     if (model->ncouple > 1) {
08631         for (j = 0; j < model->ncouple; j++) {
08632             cplleft[j] = sav[j];
08633         }
08634     }
08635 }
08636
08637 //-----
08638 Bool Assign::subCouple(int *cpl)
08639 {
08640     int j;
08641
08642     if (model->ncouple > 1) {
08643         for (j = 0; j < model->ncouple; j++) {
08644             cplleft[j] -= cpl[j];
08645             if (cplleft[j] < 0) {
08646                 return False;
08647             }
08648         }
08649     }
08650     return True;
08651 }
08652
08653 //-----
08654 Bool Assign::assignVertex(int v)
08655 {
08656     // Assign interactions to vertex 'v'
08657
08658     Bool    done, ok;
08659     CheckPt sav, sav1;
08660     NCand   *nc;
08661     int      ia, n, j, cl;
08662     NCandSt  vst;
08663     Bool     ok1;
08664     int      savc0[GRCC_MAXNCPLG], savc1[GRCC_MAXNCPLG];
08665
08666 #ifdef CHECK
08667     checkCand("assignVertex");
08668 #endif
08669
08670     // save the configuration
08671     astack->checkPoint(sav);
08672     saveCouple(savc0);
08673
08674     // assign to all vertices with only one candidate
08675     done = False;
08676     for (n = 0; n < nNodes; n++) {
08677         // check if vertex
08678         cl = pnclass->nd2cl[n];
08679         if (!isATEXternal(pnclass->type[cl])) {
08680             nc = nodes[n]->cand;
08681             if (nc->st == AS_UnAssLegs && nc->nilist == 1) {
08682                 // bind interaction to the vertex
08683                 ia = nc->i1list[0];
08684                 vst = assignIVertex(n, ia);
08685                 if (vst == AS_Impossible) {
08686                     // discard the current configuration
08687                     #ifdef CHECK
08688                     astack->restoreMsg(sav, "assignVertex");
08689                     #else
08690                     astack->restore(sav);
08691                     #endif
08692                     restoreCouple(savc0);
08693                     return False;
08694                 } else if (vst == AS_Assigned) {
08695                     if (!subCouple(model->interacts[ia]->c1list)) {
08696                         #ifdef CHECK
08697                         astack->restoreMsg(sav, "assignVertex");
08698                         #else
08699                         astack->restore(sav);
08700                         #endif
08701                         restoreCouple(savc0);
08702                         return False;
08703                     }
08704                 }
08705             }
08706             if (n == v) {
08707                 done = (vst == AS_Assigned);
08708             }
08709         }
08710     }
08711 }
08712
08713 }
08714

```

```

08715
08716     // uniquely determined
08717     ok = True;
08718     if (done) {
08719         // move to the next vertex
08720 #ifdef SIMPSEL
08721         ok = selectVertexSimp(v);
08722 #else
08723         ok = selectVertex();
08724 #endif
08725
08726 #ifdef CHECK
08727     astack->restoreMsg(sav, "assignVertex");
08728 #else
08729     astack->restore(sav);
08730 #endif
08731     restoreCouple(savc0);
08732
08733     return ok;
08734 }
08735
08736 // there are still several possibilities.
08737
08738 astack->checkPoint(sav1);
08739 saveCouple(savc1);
08740 for (j = 0; j < nodes[v]->cand->nilist; j++) {
08741     ia = nodes[v]->cand->ilist[j];
08742
08743     // bind interaction to the vertex
08744     ok1 = True;
08745     vst = assignIVertex(v, ia);
08746     if (vst == AS_Assigned) {
08747         ok1 = subCouple(model->interacts[ia]->clist);
08748     }
08749     if (ok1 && (vst == AS_Assigned || vst == AS_Assigned0)) {
08750         // move to the next vertex
08751 #ifdef SIMPSEL
08752         ok = selectVertexSimp(v);
08753 #else
08754         ok = selectVertex();
08755 #endif
08756     }
08757
08758 #ifdef CHECK
08759     astack->restoreMsg(sav1, "assignVertex");
08760 #else
08761     astack->restore(sav1);
08762 #endif
08763     restoreCouple(savc1);
08764 }
08765
08766 #ifdef CHECK
08767     astack->restoreMsg(sav, "assignVertex");
08768 #else
08769     astack->restore(sav);
08770 #endif
08771     restoreCouple(savc0);
08772
08773     return ok;
08774 }
08775
08776 //-----
08777 Bool Assign::allAssigned(void)
08778 {
08779     // Assignment of particles and interactions is finished.
08780     // Check isomorphism etc. and count symmetry factor
08781     // The sign from Fermi statistics is calculated in fillEGraph
08782
08783     BigInt nsym, esym, nsym1;
08784     MNodeClass *cl;
08785     Bool ok = True;
08786 #ifdef OPTEXTONLY
08787 #else
08788     Bool ext;
08789     int e, p;
08790 #endif
08791
08792 #ifdef CHECK
08793     checkCand("allAssigned");
08794 #endif
08795
08796     // check order of coupling constants
08797 #ifdef CHECK
08798     if (!checkOrderCpl()) {
08799         erEnd("allAssigned: checkOrderCpl = False");
08800     }
08801 #endif

```

```

08802
08803 // check duplication by violating ordering condition
08804 if (!isOrdLegs()) {
08805     return False;
08806 }
08807
08808 // classification of nodes
08809 cl = mgraph->curcl;
08810
08811 // check isomorphism and calculate sfactor
08812 nsym = 0;
08813 esym = 0;
08814
08815 ok = isIsomorphic(cl, &nsym, &esym, &nsym1);
08816 if (!ok || nsym < 1 || esym < 1) {
08817     return False;
08818 }
08819
08820 //-----
08821 // Now we got a new agraph.
08822
08823 // Update counters
08824 nAGraphs++;
08825 wAGraphs.add(1, nsym*esym);
08826 if (mgraph->opi) {
08827     nAOPI++;
08828     wAOPI.add(1, nsym*esym);
08829 }
08830
08831 #ifdef CHECK
08832     checkAG("allAssigned");
08833 #endif
08834 // fill information to resulting Egraph
08835 fillEGraph(nAGraphs, nsym, esym, nsym1);
08836
08837 #ifdef OPTEXTONLY
08838 #else
08839 // check extonly particles
08840 for (e = 0; e < nEdges; e++) {
08841     p = Abs(egraph->edges[e]->ptcl);
08842     ext = egraph->edges[e]->ext;
08843     if (!ext) {
08844         if (model->particles[p]->extonly) {
08845             return False;
08846         }
08847     }
08848 }
08849 #endif
08850
08851 return True;
08852 }
08853
08854 //=====
08855 // Interface to MGraph and EGraph
08856 //-----
08857 Bool Assign::fromMGraph(void)
08858 {
08859     // Import from MGraph
08860
08861     int npall, *pall;
08862     int np[1];
08863     Bool ok;
08864     int j, k, n, nn, nl, deg, nc, ptcl, cl, typ, mcl, cll, typ1;
08865     int lcn, ia;
08866     int nilist, ilist[GRCC_MAXMINTERACT];
08867     NCandSt st;
08868
08869 // create ANodes
08870 for (j = 0; j < nNodes; j++) {
08871     nodes[j] = new ANode(mgraph->nodes[j]->deg);
08872 }
08873
08874 // initialization of edge table
08875 nETotal = 0;
08876
08877 // List of all particles
08878 pall = model->allParticles(&npall);
08879
08880 // add nodes
08881 for (n = 0; n < mgraph->nNodes; n++) {
08882     cl = pnclass->nd2cl[n];
08883     typ = pnclass->type[cl];
08884
08885 // external particle
08886 if (isATEXternal(typ)) {
08887
08888 #ifdef CHECK

```



```

08889         if (mgraph->nodes[n]->deg != 1) {
08890             grcc_fprintf(GRCC_Stdout, "*** assign:fromMGraph : "
08891                 "external but deg[%d] = %d != 1, type=%d\n",
08892                 n, mgraph->nodes[n]->deg, typ);
08893             mgraph->print();
08894             erEnd("assign:fromMGraph : illegal external particle");
08895         }
08896     #endif
08897     np[0] = pnclass->particle[c1];
08898     nodes[n]->cand = new NCand(AS_AssExt, 1, 1, np);
08899     for (n1 = 0; n1 < mgraph->nNodes; n1++) {
08900         nc = mgraph->adjMat[n][n1];
08901         for (k = 0; k < nc; k++) {
08902             addEdge(n, n1, 1, np);
08903         }
08904     }
08905
08906     // vertex
08907     } else {
08908
08909         // candidates of vertices
08910         deg = mgraph->nodes[n]->deg;
08911         st = AS_UnAssLegs;
08912         c1 = sproc->pnclass->nd2cl[n]; // class in the process
08913         mcl = sproc->pnclass->cl2mcl[c1]; // class in the model
08914
08915         if (nodes[n]->cand != NULL) {
08916             delete nodes[n]->cand;
08917         }
08918         lcn = model->ncouple;
08919
08920         // multiple coupling constants are defined in the model.
08921         if (lcn > 1) {
08922             nilist = 0;
08923             for (j = 0; j < model->cplgnvl[mcl]; j++) {
08924                 ia = model->cplgv1[mcl][j];
08925
08926                 // select by coupling constants
08927                 if (leqArray(lcn, model->interacts[ia]->clist, cplleft)) {
08928                     ilist[nilist++] = ia;
08929                 }
08930             }
08931             nodes[n]->cand = new NCand(st, deg, nilist, ilist);
08932
08933             // there is only one coupling constant
08934         } else {
08935             nodes[n]->cand =
08936                 new NCand(st, deg, model->cplgnvl[mcl], model->cplgv1[mcl]);
08937         }
08938
08939         // vertices
08940         for (n1 = n; n1 < mgraph->nNodes; n1++) {
08941             c11 = pnclass->nd2cl[n1];
08942             typ1 = pnclass->type[c11];
08943             if (!isATEXternal(typ1)) {
08944                 nc = mgraph->adjMat[n][n1];
08945                 if (n == n1) {
08946                     nc = nc/2;
08947                 }
08948                 for (k = 0; k < nc; k++) {
08949                     addEdge(n, n1, npall, pall);
08950                 }
08951             }
08952         }
08953     }
08954 }
08955 #ifdef CHECK
08956     if (nETotal != nEdges) {
08957         grcc_fprintf(GRCC_Stdout, "*** Assign::fromMGraph nETotal=%d != nEdges=%d\n",
08958             nETotal, nEdges);
08959         erEnd("Assign::fromMGraph nETotal= != nEdges");
08960     }
08961 #endif
08962
08963     // assign particle of an edge next to an external particle
08964     for (n = 0; n < mgraph->nNodes; n++) {
08965         c1 = pnclass->nd2cl[n];
08966         typ = pnclass->type[c1];
08967         if (isATEXternal(typ)) {
08968
08969             c1 = pnclass->nd2cl[n];
08970             ptcl = pnclass->particle[c1];
08971
08972             // particle ptcl flows in to the node and
08973             // particle (- ptcl) flows in from the edge.
08974
08975         #ifdef CHECK

```

```

08976         if (ptcl == 0) {
08977             erEnd("fromMGraph: ptcl=0");
08978         }
08979 #endif
08980         ok = assignPLeg(n, 0, -ptcl);
08981         if (!ok) {
08982             // impossible config
08983             return False;
08984         }
08985         nn = nodes[n]->anodes[0];
08986         ok = updateCandNode(nn);
08987         if (!ok) {
08988             // impossible config
08989             return False;
08990         }
08991     }
08992 }
08993
08994 #ifdef CHECK
08995     checkCand("fromMGraph");
08996     checkAG("fromMGraph");
08997 #endif
08998     return True;
09000 }
09001
09002 //-----
09003 void Assign::addEdge(int n0, int n1, int nplist, int *plist)
09004 {
09005     // add an edge and set connection information
09006     // n0 -- (new edge) -- n1, with candidates 'plist'
09007
09008     int lg0, lg1, eid;
09009     AEdge *aed;
09010
09011 #ifdef CHECK
09012     if (n0 >= nNodes || n1 >= nNodes) {
09013         grcc_fprintf(GRCC_Stdout, "*** Assign::addEdge : undefined nodes %d: [%d, %d]",
09014             nETotal, n0, n1);
09015         erEnd("Assign::addEdge : undefined nodes");
09016     }
09017 #endif
09018
09019     // legs to be connected
09020     lg0 = nodes[n0]->newleg();
09021     lg1 = nodes[n1]->newleg();
09022
09023     // create edge
09024 #ifdef CHECK
09025     if (nETotal >= nEdges) {
09026         erEnd("too many edges");
09027     }
09028 #endif
09029
09030     aed = new AEdge(n0, lg0, n1, lg1);
09031     aed->cand = new ECand(False, nplist, plist);
09032
09033 #ifdef CHECK
09034     if (edges[nETotal] != NULL) {
09035         erEnd("edges[nETotal] != NULL");
09036     }
09037 #endif
09038
09039     // register to the edge table
09040     eid = nETotal;
09041     edges[nETotal++] = aed;
09042
09043     // connect edge and nodes
09044     connect(n0, lg0, eid, 0, n1, lg1);
09045 }
09046
09047 //-----
09048 void Assign::connect(int n0, int l0, int eg, int el, int n1, int l1)
09049 {
09050     // Connect (n0, l0) -- (eg, el) -- (n1)
09051
09052     ANode *nd0, *nd1;
09053     AEdge *ed;
09054     int eo;
09055
09056     nd0 = nodes[n0];
09057     nd1 = nodes[n1];
09058     ed = edges[eg];
09059     eo = 1 - el;
09060
09061     nd0->anodes[l0] = n1;
09062     nd0->aedges[l0] = eg;

```

```

09063     nd0->aelegs[10] = e1;
09064
09065     nd1->anodes[11] = n0;
09066     nd1->aedges[11] = eg;
09067     nd1->aelegs[11] = eo;
09068
09069     ed->nodes[e1] = n0;
09070     ed->nlegs[e1] = 10;
09071
09072     ed->nodes[eo] = n1;
09073     ed->nlegs[eo] = 11;
09074 }
09075
09076 //-----
09077 Bool Assign::fillEGraph(int aid, BigInt nsym, BigInt esym, BigInt nsym1)
09078 {
09079     // Set variables of egraph with the result of particle assignment
09080     // Adjust the ordering of legs of vertices in a consistent way
09081     // with the interaction defined in the model.
09082
09083     ANode *an;
09084     ENode *en;
09085     int work[3][GRCC_MAXLEGS];
09086     int n, lr, e;
09087     int *elist;
09088     int lg, ed, eg;
09089 #ifdef CHECK
09090     int cl, ok = True;
09091 #endif
09092
09093     egraph->assigned = True;
09094
09095     egraph->aId = aid;
09096     egraph->nsym = nsym;
09097     egraph->esym = esym;
09098     egraph->bicount = -1;
09099     if (sproc != NULL) {
09100         egraph->extperm = sproc->extperm;
09101     } else {
09102         egraph->extperm = 1;
09103     }
09104     egraph->nsym1 = nsym1;
09105     egraph->multp = (egraph->extperm * egraph->nsym1) / egraph->nsym;
09106
09107 #ifdef CHECK
09108     checkAG("fillEGraph:0");
09109 #endif
09110     // external node
09111     for (n = 0; n < nNodes; n++) {
09112         // vertices
09113 #ifdef CHECK
09114         cl = pnclass->nd2cl[n];
09115         if (isATEXternal(pnclass->type[cl])) {
09116             ;
09117         } else if (nodes[n]->cand->st != AS_Assigned) {
09118             grcc_fprintf(GRCC_Stdout, "*** fillEGraph : node %d is not assigned", n);
09119             prCand("fillEGraph: node");
09120             erEnd("fillEGraph : node is not assigned");
09121         }
09122 #endif
09123
09124         an = nodes[n];
09125         en = egraph->nodes[n];
09126
09127         en->initAss(egraph, n, an->deg);
09128
09129         en->intrct = an->cand->ilist[0];
09130         elist = reordLeg(n, work[0], work[1], work[2]);
09131         for (lr = 0; lr < nodes[n]->deg; lr++) {
09132             if (elist == NULL) {
09133                 lg = lr;
09134             } else {
09135                 lg = elist[lr];
09136             }
09137 #ifdef CHECK
09138             if (lg < 0 || lg >= nodes[n]->deg) {
09139                 grcc_fprintf(GRCC_Stdout, "*** fillEGraph: n=%d, lr=%d: 0 <= lg=%d < %d\n",
09140                     n, lr, lg, nodes[n]->deg);
09141                 erEnd("fillEGraph: illegal reordering");
09142             }
09143 #endif
09144             ed = an->aedges[lg];
09145             eg = an->aelegs[lg];
09146
09147             en->edges[lr] = I2Vedge(ed, 2*eg-1);
09148
09149             egraph->edges[ed]->nodes[eg] = n;

```

```

09150         egraph->edges[ed]->nlegs[eg] = lr;
09151     }
09152 }
09153 // edges
09154 for (e = 0; e < nEdges; e++) {
09155 #ifdef CHECK
09156     if (edges[e]->cand->nplist != 1) {
09157         grcc_fprintf(GRCC_Stdout, "*** fillEGraph : edge %d is not assigned", e);
09158         prCand("fillEGraph: edge");
09159         erEnd("fillEGraph : edge is not assigned");
09160     }
09161 #endif
09162     egraph->edges[e]->ptcl = edges[e]->cand->plist[0];
09163 }
09164 #ifdef EDGEORDER
09165 for (e = 0; e < nEdges; e++) {
09166     if (egraph->edges[e]->ptcl < 0) {
09167         egraph->edges[e]->ptcl = - edges[e]->cand->plist[0];
09168         n = egraph->edges[e]->nodes[0];
09169         lr = egraph->edges[e]->nlegs[0];
09170         egraph->edges[e]->nodes[0] = egraph->edges[e]->nodes[1];
09171         egraph->edges[e]->nlegs[0] = egraph->edges[e]->nlegs[1];
09172         egraph->edges[e]->nodes[1] = n;
09173         egraph->edges[e]->nlegs[1] = lr;
09174
09175         egraph->nodes[n]->edges[lr] = - egraph->nodes[n]->edges[lr];
09176         n = egraph->edges[e]->nodes[0];
09177         lr = egraph->edges[e]->nlegs[0];
09178         egraph->nodes[n]->edges[lr] = - egraph->nodes[n]->edges[lr];
09179     }
09180 }
09181 #endif
09182 #ifdef CHECK
09183 for (ed = 0; ed < nEdges; ed++) {
09184     n = egraph->edges[ed]->nodes[0];
09185     lr = egraph->edges[ed]->nlegs[0];
09186     if (egraph->nodes[n]->edges[lr] != -ed-1) {
09187         grcc_fprintf(GRCC_Stderr, "+++ node[%d][%d]=%d != - (edge[%d][0] + 1) = %d\n",
09188             n, lr, egraph->nodes[n]->edges[lr], e, -ed-1);
09189         ok = False;
09190     }
09191     n = egraph->edges[ed]->nodes[1];
09192     lr = egraph->edges[ed]->nlegs[1];
09193     if (egraph->nodes[n]->edges[lr] != ed+1) {
09194         grcc_fprintf(GRCC_Stderr, "+++ node[%d][%d]=%d != + (edge[%d][0] + 1) = %d\n",
09195             n, lr, egraph->nodes[n]->edges[lr], e, ed+1);
09196         ok = False;
09197     }
09198 }
09199 if (!ok) {
09200     egraph->print();
09201     erEnd("fillEGraph: illegal connection");
09202 }
09203 #endif
09204
09205 // analyse fermion line and determine Fermi statistical sign factor
09206 egraph->getFLines();
09207
09208 #ifdef CHECK
09209 checkAG("fillEGraph:0");
09210 #endif
09211 return True;
09212 }
09213
09214 }
09215
09216 //-----
09217 int *Assign::reordLeg(int n, int *reord, int *plist, int *used)
09218 {
09219     // Reorder legs of node 'n' in accordance with the definition
09220     // of the interaction in the model
09221     //
09222     // plist[reord[j]] = model->interacts[ia]->plist[j]
09223
09224     int lg, ia, lr, deg, pt, ed;
09225     int *ilegs;
09226 #ifdef CHECK
09227     Bool found;
09228 #endif
09229
09230     // external node
09231     if (nodes[n]->cand->st == AS_AssExt) {
09232         reord[0] = 0;
09233         return 0;
09234     }
09235
09236     // degree of the node

```

```

09237     deg = nodes[n]->deg;
09238
09239     if (deg <= 1) {
09240         reord[0] = 0;
09241         return 0;
09242     }
09243
09244     // list of particles at the legs of the node 'n'
09245     for (lg = 0; lg < deg; lg++) {
09246         ed = nodes[n]->aedges[lg];
09247         pt = edges[ed]->cand->plist[0];
09248         plist[lg] = legEdgeParticle(n, lg, pt);
09249         used[lg] = 0;
09250     }
09251
09252     // list of legs in the interaction
09253     ia = nodes[n]->cand->ilist[0];
09254     ilegs = model->interacts[ia]->plist;
09255
09256     // reorder
09257     #ifdef CHECK
09258     found = False;
09259     #endif
09260     for (lr = 0; lr < deg; lr++) {
09261         for (lg = 0; lg < deg; lg++) {
09262             if (used[lg] == 0 && plist[lg] == ilegs[lr]) {
09263                 reord[lr] = lg;
09264                 used[lg] = 1;
09265             }
09266             #ifdef CHECK
09267             found = True;
09268             #endif
09269             break;
09270         }
09271     }
09272     #ifdef CHECK
09273     if (!found) {
09274         grcc_fprintf(GRCC_Stdout, "*** reordLeg: illegal list of particles:"
09275                     "interaction %d ", ia);
09276         prIntArray(deg, ilegs, " ");
09277         grcc_fprintf(GRCC_Stdout, "vertex %d ", n);
09278         prIntArray(deg, plist, "\n");
09279         prCand("reordLeg");
09280         erEnd("reordLeg: illegal list of particles");
09281     }
09282     #endif
09283 }
09284
09285 return reord;
09286 }
09287
09288 //=====
09289 // Adjust the direction of a particle on an edge and on a leg of
09290 // node.
09291 //-----
09292 int Assign::getLegParticle(int n, int ln)
09293 {
09294     // Get particle code of (n, ln) when particle 'pt' runs
09295     // in the direction of the edge at (n, ln).
09296     //
09297     // Get particle code in the direction the edge at (n, ln)
09298     // when particle 'pt' is at leg (n, ln).
09299     //
09300     // These two cases are realized by the same function
09301
09302     ANode *nd;
09303     int elg, ed, pt;
09304
09305     nd = nodes[n];
09306     elg = nd->aelegs[ln];
09307     ed = nd->aedges[ln];
09308     pt = edges[ed]->cand->plist[0];
09309
09310     // normalization of sign for neutral particle
09311     if (elg == 0) {
09312         return model->normalParticle(-pt);
09313     } else {
09314         return model->normalParticle(pt);
09315     }
09316 }
09317
09318 //-----
09319 int Assign::legEdgeParticle(int n, int ln, int pt)
09320 {
09321     // Get particle code of (n, ln) when particle 'pt' runs
09322     // in the direction of the edge at (n, ln).
09323     //

```

```

09324 // Get particle code in the direction the edge at (n, nl)
09325 // when particle 'pt' is at leg (n, ln).
09326 //
09327 // These two cases are realized by the same function
09328
09329 // normalization of sign for neutral particle
09330
09331 if (nodes[n]->aelegs[ln] == 0) {
09332     return model->normalParticle(-pt);
09333 } else {
09334     return model->normalParticle(pt);
09335 }
09336 }
09337
09338 //-----
09339 int Assign::legPart(int v, int lg, int nplist, int *plist, int *rlist, const int size)
09340 {
09341     // Convert list of candidate particles in 'plist' at (v, lg) ==> edge
09342     // or edge ==> (v, lg)
09343
09344     int j, nrlist;
09345
09346     nrlist = 0;
09347     for (j = 0; j < nplist; j++) {
09348         nrlist = intSetAdd(nrlist, rlist,
09349                             legEdgeParticle(v, lg, plist[j]), size);
09350     }
09351     return nrlist;
09352 }
09353
09354 //-----
09355 int Assign::candPart(int v, int ln, int *plist, const int size)
09356 {
09357     // update candidate particles incoming to (v, ln)
09358
09359     int en;
09360     int ntplist;
09361     ECand *ec;
09362
09363     en = nodes[v]->aedges[ln];
09364
09365 #ifdef CHECK
09366     if (edges[en]->cand->det) {
09367         grcc_fprintf(GRCC_Stdout, "*** candPart : particle of leg (%d, %d) "
09368                     "is assigned to %d\n",
09369                     v, ln, edges[en]->cand->plist[0]);
09370         checkCand("candPart");
09371         mgraph->printAdjMat(mgraph->curcl);
09372         prCand("candPart");
09373         erEnd("candPart : particle of leg is assigned");
09374     }
09375 #endif
09376
09377     ec = edges[en]->cand;
09378
09379     ntplist = legPart(v, ln, ec->nplist, ec->plist, plist, size);
09380
09381     return ntplist;
09382 }
09383
09384 //=====
09385 // tools for assignment
09386
09387 //-----
09388 int Assign::selUnAssVertexSimp(int lastv)
09389 {
09390     // Select a vertex for assignment to its leg.
09391     // We take one with minimum number of candidates
09392     //
09393     // There may be better method.
09394
09395     int v0, v;
09396
09397     // select vertex one by one in the sequential order of node number
09398     v0 = Max(lastv+1, nExtern);
09399     for (v = v0; v < nNodes; v++) {
09400         if (nodes[v]->cand->st == AS_UnAssLegs) {
09401             return v;
09402         }
09403     }
09404     return -1;
09405 }
09406
09407 //-----
09408 int Assign::selUnAssVertex(void)
09409 {
09410     // Select a vertex for assignment to its leg.

```

```

09411 // We take one with minimum number of candidates
09412 //
09413 // There may be better method.
09414
09415 int v0, v, n1;
09416
09417 // select vertex one by one in the sequential order of node number
09418 v0 = -1;
09419 n1 = 0;
09420 for (v = 0; v < nNodes; v++) {
09421     if (nodes[v]->cand->st == AS_UnAssLegs) {
09422         if (nodes[v]->cand->nilist == 1) {
09423             return v;
09424         } else if (nodes[v]->cand->nilist > 1) {
09425             // select node with fewer candidates
09426             if (v0 < 0 || nodes[v]->cand->nilist < n1) {
09427                 v0 = v;
09428                 n1 = nodes[v]->cand->nilist;
09429             }
09430         }
09431     }
09432 }
09433 return v0;
09434 }
09435
09436 //-----
09437 int Assign::selUnAssLeg(int v, int lastlg)
09438 {
09439     // Select a leg of vertex 'v' for the assignment.
09440     // The lastly selected leg was 'lastlg'.
09441
09442     int lg0, lg, e;
09443 #ifdef CHECK
09444     int n0, n1;
09445 #endif
09446
09447     // lg0 = Max(lastlg, lastlg + 1);
09448     lg0 = lastlg + 1;
09449
09450     for (lg = lg0; lg < nodes[v]->deg; lg++) {
09451 #ifdef CHECK
09452         if (lastlg >= 0) {
09453             n0 = nodes[v]->anodes[lastlg];
09454         } else {
09455             n0 = -1;
09456         }
09457         n1 = nodes[v]->anodes[lg];
09458         if (n0 > n1) {
09459             grcc_fprintf(GRCC_Stdout, "*** selUnAssLeg: n0=%d > n1=%d\n", n0, n1);
09460             grcc_fprintf(GRCC_Stdout, "*** illegal connection\n");
09461             erEnd("selUnAssLeg: n0 > n1");
09462         }
09463 #endif
09464         e = nodes[v]->aedges[lg];
09465         if (!edges[e]->cand->det) {
09466             return lg;
09467         }
09468     }
09469     return -1;
09470 }
09471
09472 //-----
09473 NCandSt Assign::assignIVertex(int v, int ia)
09474 {
09475     // Try to assign interaction 'ia' to 'v'
09476
09477     int lplist[GRCC_MAXLEGS], iplist[GRCC_MAXLEGS];
09478     int lg, e, pt, deg;
09479
09480     if (nodes[v]->cand->st == AS_Assigned || nodes[v]->cand->st == AS_AssExt) {
09481         return AS_Assigned0;
09482     }
09483
09484     deg = nodes[v]->deg;
09485     if (deg != model->interacts[ia]->nlegs) {
09486         return AS_Impossible;
09487     }
09488
09489     // list of particles at the legs of the vertex
09490     for (lg = 0; lg < deg; lg++) {
09491         e = nodes[v]->aedges[lg];
09492         if (edges[e]->cand->nplist != 1) {
09493             return AS_UnAssLegs;
09494         }
09495     }
09496 }

```

```

09498         } else {
09499             pt = edges[e]->cand->plist[0];
09500             lplist[lg] = legEdgeParticle(v, lg, pt);
09501         }
09502         iplist[lg] = model->interacts[ia]->plist[lg];
09503     }
09504
09505     sorti(deg, lplist);
09506     sorti(deg, iplist);
09507
09508     // from interaction
09509     if (cmpArray(deg, lplist, deg, iplist) == 0) {
09510         // orders of coupling constants should be checked ???
09511         astack->pushNode(v);
09512         nodes[v]->cand->st = AS_Assigned;
09513         nodes[v]->cand->nilist = 1;
09514         nodes[v]->cand->ilist[0] = ia;
09515         return AS_Assigned;
09516     }
09517
09518     return AS_Impossible;
09519 }
09520
09521 //-----
09522 Bool Assign::assignPLeg(int n, int ln, int pt)
09523 {
09524     // A particle 'pt' is assigned to leg 'ln' of node 'n'
09525
09526     ANode *nd;
09527     int e, ept;
09528     Bool ok;
09529
09530 #ifdef CHECK
09531     checkNode(n, "assignPLeg0");
09532     checkCand("assignPLeg");
09533 #endif
09534
09535     // nodes and the edge
09536     nd = nodes[n];
09537     e = nd->aedges[ln];
09538
09539     // already determined
09540     if (edges[e]->cand->det) {
09541         return True;
09542     }
09543
09544     if (!isOrdPLeg(n, ln, pt)) {
09545         return False;
09546     }
09547
09548     // particle on the edge
09549     ept = legEdgeParticle(n, ln, pt);
09550
09551 #ifdef CHECK
09552     if (!isIn(edges[e]->cand->nplist, edges[e]->cand->plist, ept)) {
09553         grcc_fprintf(GRCC_Stdout, "*** assignPLeg: particle %d is not in the cand. of e=%d",
09554             ept, e);
09555         edges[e]->cand->prECand("\n");
09556         prCand("assignPLeg");
09557         erEnd("assignPLeg: particle is not in the cand.");
09558     }
09559 #endif
09560
09561     // save the current configuration
09562     astack->pushEdge(e);
09563
09564     // determine the particle on the edge
09565     edges[e]->cand->nplist = 1;
09566     edges[e]->cand->plist[0] = ept;
09567
09568     ok = detEdge(e);
09569
09570     return ok;
09571 }
09572
09573 //-----
09574 Bool Assign::detEdge(int e)
09575 {
09576     Bool ok0, ok1;
09577
09578     if (edges[e]->cand->nplist != 1) {
09579         return False;
09580     }
09581     edges[e]->cand->det = True;
09582
09583     // update candidate list
09584     ok0 = updateCandNode(edges[e]->nodes[0]);

```



```

09585     if (ok0) {
09586         ok1 = updateCandNode(edges[e]->nodes[l]);
09587     } else {
09588         ok1 = False;
09589     }
09590
09591     return (ok0 && ok1);
09592 }
09593
09594 //-----
09595 Bool Assign::isOrdPLeg(int n, int ln, int pt)
09596 {
09597     int e0, e1, ln0, ep0, pt0, nn, n0, n1, ln1, pt1, e1, ep1;
09598
09599     if (nodes[n]->deg < 2) {
09600         return True;
09601     }
09602     nn = nodes[n]->anodes[ln];
09603     if (n <= nn) {
09604         n0 = n;
09605         n1 = nn;
09606         ln0 = ln;
09607         pt0 = pt;
09608     } else {
09609         n0 = nn;
09610         n1 = n;
09611         e0 = nodes[n]->aedges[ln];
09612         e1 = nodes[n]->aelegs[ln];
09613         ln0 = edges[e0]->nlegs[l-e1];
09614         ep0 = legEdgeParticle(n, ln, pt);
09615         pt0 = legEdgeParticle(n0, ln0, ep0);
09616     }
09617
09618     for (ln1 = 0; ln1 < nodes[n0]->deg; ln1++) {
09619         if (ln1 == ln0 || n1 != nodes[n0]->anodes[ln1]) {
09620             continue;
09621         }
09622         e1 = nodes[n0]->aedges[ln1];
09623         if (!edges[e1]->cand->det) {
09624             continue;
09625         }
09626         ep1 = edges[e1]->cand->plist[0];
09627         pt1 = legEdgeParticle(n0, ln1, ep1);
09628         if ((ln0 < ln1 && pt0 > pt1) ||
09629             (ln0 > ln1 && pt0 < pt1)) {
09630             return False;
09631         }
09632     }
09633     return True;
09634 }
09635
09636 //=====
09637 // Operations of candidate
09638 //-----
09639 Bool Assign::candPartClassify(int v, int *npdass, int *pdass, int *npuass, int *puass, const int size)
09640 {
09641     // Construct the classified lists of particles possible to assign to
09642     // the legs of node v
09643     //
09644     // pdass = (duplicated set of particles where edges has a unique
09645     // candidate)
09646     // puass = (duplicated set of other candidate particles)
09647
09648     int lg, e, npl, ass, jd, ju, j, pt, pts;
09649
09650     jd = 0;
09651     ju = 0;
09652     for (lg = 0; lg < nodes[v]->deg; lg++) {
09653         e = nodes[v]->aedges[lg];
09654         npl = edges[e]->cand->nplist;
09655         if (npl < 1) {
09656             return False;
09657         }
09658         ass = (npl == 1);
09659         for (j = 0; j < edges[e]->cand->nplist; j++) {
09660             pt = edges[e]->cand->plist[j];
09661             pts = legEdgeParticle(v, lg, pt);
09662             if (ass) {
09663                 jd = intSListAdd(jd, pdass, pts, size);
09664             } else {
09665                 ju = intSListAdd(ju, puass, pts, size);
09666             }
09667         }
09668     }
09669     *npdass = jd;
09670     *npuass = ju;
09671

```

```

09672     return True;
09673 }
09674
09675 //-----
09676 Bool Assign::updateCandNode(int v)
09677 {
09678     // Update the configuration
09679     // - nodes[v]->cand->ilist
09680     // - edges[e]->cand->plist for the adjacent edge
09681     // - nodes[n]->cand->unass for the adjacent node 'n' of 'e'
09682
09683     int npdass, pdass[GRCC_MAXPSLIST];
09684     int npuass, puass[GRCC_MAXPSLIST];
09685     int nsub0, sub0[GRCC_MAXPSLIST];
09686     int niplist, iplist[GRCC_MAXPSLIST];
09687     int nd0, d0[GRCC_MAXMPARTICLES2];
09688     int nd1, d1[GRCC_MAXMPARTICLES2];
09689     int ns0, s0[GRCC_MAXMPARTICLES2];
09690     int neplist, eplist[GRCC_MAXMPARTICLES2];
09691     int nitlist, itlist[GRCC_MAXMINTERACT];
09692     int e, it, j, k, i, lg;
09693
09694     // external node
09695     if (nodes[v]->cand->st == AS_AssExt) {
09696 #ifdef CHECK
09697         e = nodes[v]->aedges[0];
09698         if (e < 0 || edges[e]->cand->nplist != 1) {
09699             grcc_fprintf(GRCC_Stdout, "*** illegal external node: v=%d e=%d :", v, e);
09700             prCand("updateCandNode");
09701             erEnd("illegal external node");
09702         }
09703 #endif
09704         return True;
09705     }
09706
09707     // impossible configuration.
09708     if (nodes[v]->cand->nlist < 1) {
09709         return False;
09710     }
09711
09712     // list of possible particles on the edges of the vertex.
09713     // pdass = (set of assigned particles)
09714     // puass = (list of particles of edges with two of more candidates)
09715
09716     niplist = 0;
09717     nitlist = 0;
09718     if (!candPartClassify(v, &npdass, pdass, &npuass, puass, GRCC_MAXPSLIST)) {
09719         return False;
09720     }
09721
09722     if (npdass < 1) {
09723         return False;
09724     }
09725
09726     // from interaction : slist
09727     // Conditions :
09728     // 1. pdass \subset slist
09729     // 2. slist-pdass \subset puass
09730     // from interaction : slist
09731     // conditions : pdass \subset slist \subset (pdass + puass)
09732     // sub0 = slist - pdass,
09733     nitlist = 0;
09734     for (j = 0; j < nodes[v]->cand->nlist; j++) {
09735         it = nodes[v]->cand->ilist[j];
09736         // conditions:
09737         // 1. pdass \subset slist <==> pdass-slist = \emptyset
09738         // 2. slist-pdass \subset puass <==> (slist-pdass)-puass = \emptyset
09739         if (isSubSList(npdass, pdass,
09740             model->interacts[it]->nslist,
09741             model->interacts[it]->slist) ) {
09742             nsub0 = subtrSet(model->interacts[it]->nslist,
09743                 model->interacts[it]->slist,
09744                 npdass, pdass, sub0, GRCC_MAXPSLIST);
09745             if (nsub0 < 1) {
09746                 // slist == pdass : no additional candidate particle
09747                 nitlist = intSetAdd(nitlist, itlist, it, GRCC_MAXMINTERACT);
09748             } else {
09749                 if (isSubSList(nsub0, sub0, npuass, puass)) {
09750                     nitlist = intSetAdd(nitlist, itlist, it, GRCC_MAXMINTERACT);
09751                     for (k = 0; k < nsub0; k++) {
09752                         niplist = intSetAdd(niplist, iplist, sub0[k], GRCC_MAXPSLIST);
09753                     }
09754                 }
09755             }
09756         }
09757     }
09758 }

```

```

09759     }
09760
09761     if (nitlist < 1) {
09762         return False;
09763     }
09764
09765     if (nitlist != nodes[v]->cand->nilist) {
09766         astack->pushNode(v);
09767         nodes[v]->cand->nilist = nitlist;
09768         for (i = 0; i < nitlist; i++) {
09769             nodes[v]->cand->ilist[i] = itlist[i];
09770         }
09771 #ifdef CHECK
09772     } else if (nitlist > nodes[v]->cand->nilist) {
09773         erEnd("larger ncand");
09774 #endif
09775     }
09776
09777     // edge
09778     for (lg = 0; lg < nodes[v]->deg; lg++) {
09779         e = nodes[v]->aedges[lg];
09780         if (edges[e]->cand->nplist > 1) {
09781             // elist = {candidate particles in the direction of the edge}
09782             // s0 = plist \cap elist
09783             neplist = legPart(v, lg, niplist, iplist, eplist, GRCC_MAXMPARTICLES2);
09784             listDiff(edges[e]->cand->nplist, edges[e]->cand->plist,
09785                     neplist, eplist,
09786                     &nd0, d0, &ns0, s0, &nd1, d1);
09787
09788             if (ns0 < 1) {
09789                 return False;
09790             }
09791             if (ns0 < edges[e]->cand->nplist) {
09792                 astack->pushEdge(e);
09793                 edges[e]->cand->nplist = ns0;
09794                 for (i = 0; i < ns0; i++) {
09795                     edges[e]->cand->plist[i] = s0[i];
09796                 }
09797             }
09798         }
09799     }
09800     for (lg = 0; lg < nodes[v]->deg; lg++) {
09801         e = nodes[v]->aedges[lg];
09802         if (edges[e]->cand->nplist == 1 && !edges[e]->cand->det) {
09803             if (!detEdge(e)) {
09804                 return False;
09805             }
09806         }
09807     }
09808
09809     return True;
09810 }
09811
09812 //=====
09813 // Check order of coupling constants, duplication and isomorphism
09814 //-----
09815 Bool Assign::checkOrderCpl(void)
09816 {
09817     // Check order of coupling constants
09818
09819     int ord[GRCC_MAXNCPLG];
09820     int lcn, n, ia, j, cl;
09821
09822     // list of coupling constants
09823     lcn = model->ncouple;
09824
09825     if (lcn == 1) {
09826         return True;
09827     }
09828
09829     // sum up for each coupling constants
09830     for (j = 0; j < GRCC_MAXNCPLG; j++) {
09831         ord[j] = 0;
09832     }
09833     for (n = 0; n < nNodes; n++) {
09834         cl = pnclass->nd2cl[n];
09835         if (!isATEExternal(pnclass->type[cl])) {
09836             ia = nodes[n]->cand->ilist[0];
09837             for (j = 0; j < lcn; j++) {
09838                 ord[j] += model->interacts[ia]->clist[j];
09839             }
09840         }
09841     }
09842
09843     // comparison
09844     for (j = 0; j < lcn; j++) {
09845         if (sproc->clist[j] != ord[j]) {

```

```

09846         return False;
09847     }
09848 }
09849 return True;
09850 }
09851
09852 //-----
09853 Bool Assign::isOrdLegs(void)
09854 {
09855     // Whether graph is duplicated because of the violation
09856     // of ordering in the case of multiple connections
09857     // v0 <-- v --> v1
09858
09859     int v, l0, v0, l1, v1, e0, e1, q0, q1, p0, p1, c1;
09860
09861     for (v = 0; v < nNodes; v++) {
09862         c1 = pnclass->nd2cl[v];
09863         if (!isATEXternal(pnclass->type[c1])) {
09864             for (l0 = 0; l0 < nodes[v]->deg; l0++) {
09865                 v0 = nodes[v]->anodes[l0];
09866                 for (l1 = l0+1; l1 < nodes[v]->deg; l1++) {
09867                     v1 = nodes[v]->anodes[l1];
09868
09869                     if (v0 == v1 && v <= v0) {
09870                         // multiple connections (v, l0) and (v, l1)
09871                         e0 = nodes[v]->aedges[l0];
09872                         e1 = nodes[v]->aedges[l1];
09873                         q0 = edges[e0]->cand->plist[0];
09874                         q1 = edges[e1]->cand->plist[0];
09875                         p0 = legEdgeParticle(v, l0, q0);
09876                         p1 = legEdgeParticle(v, l1, q1);
09877                         if (p0 > p1) {
09878                             // violation of ordering
09879                             return False;
09880                         }
09881                     }
09882                 }
09883             }
09884         }
09885     }
09886     return True;
09887 }
09888
09889 //-----
09890 Bool Assign::isIsomorphic(MNodeClass *cl, BigInt *nsym, BigInt *esym, BigInt *nsym1)
09891 {
09892     // Check whether the current graph is the representative of
09893     // a isomorphic class.
09894     // Returns (nsym, esym)
09895     // nsym = symmetry factor by the permutation of nodes.
09896     // esym = symmetry factor by the permutation of edge.
09897     // If this graph is not a representative, then returns (0,0).
09898
09899     int j, cmp, n, c1n;
09900     BigInt ngelem;
09901     int *p;
09902     Bool invext;
09903
09904     ngelem = mgraph->group->nElem();
09905 #ifdef CHECK
09906     if (mgraph->nsym > 1 && ngelem <= 1) {
09907         grcc_fprintf(GRCC_Stdout, "*** isIsomorphic: illegal group: "
09908             "ngelem=%ld, mgraph->sym=(%ld, %ld)\n",
09909             ngelem, mgraph->nsym, mgraph->esym);
09910         erEnd("Assign::isIsomorphic: illegal group");
09911     }
09912 #endif
09913     if (ngelem == 0) {
09914         *nsym = 1;
09915         *nsym1 = 1;
09916     } else {
09917         *nsym = 0;
09918         *nsym1 = 0;
09919         for (j = 0; j < ngelem; j++) {
09920             // check the graph is the representative and count 'nsym'.
09921
09922             //p = mgraph->group->nextElem();
09923             p = mgraph->group->elem[j];
09924
09925             cmp = cmpPermGraph(p, c1);
09926
09927             if (cmp < 0) { // duplicated graph
09928                 return False;
09929             } else if (cmp == 0) { // not duplicated
09930                 (*nsym)++;
09931             }
09932         }
09933     }
09934 }

```

```

09933         if (opt->values[GRCC_OPT_SymmInitial] || opt->values[GRCC_OPT_SymmFinal]) {
09934             invext = True;
09935             for (n = 0; n < nNodes; n++) {
09936                 cln = pnclass->nd2cl[n];
09937                 if (!isATEXternal(pnclass->type[cln])) {
09938                     continue;
09939                 }
09940                 if (p[n] != n) {
09941                     invext = False;
09942                 }
09943             }
09944             if (invext) {
09945                 (*nsym1)++;
09946             }
09947             } else {
09948                 (*nsym1)++;
09949             }
09950         }
09951     }
09952 }
09953 if (*nsym1 == 0) {
09954     *nsym1 = *nsym;
09955 }
09956
09957 // calculate permutations of edges
09958 *esym = edgeSym();
09959 if (*esym < 1) {
09960     return False;
09961 }
09962
09963 return True;
09964 }
09965
09966 //-----
09967 int Assign::cmpPermGraph(int *p, MNodeClass *cl)
09968 {
09969     // compare the graph with one permutated by 'p'.
09970
09971     int n, cmp, n1, n2, p1, p2, j;
09972     ANode *nd1, *np1;
09973     // ANode *nd2, *np2;
09974     int njn, jn[GRCC_MAXLEGS];
09975     int njp, jp[GRCC_MAXLEGS];
09976
09977 #ifdef CHECK
09978     if (p == NULL) {
09979         erEnd("Assign::cmpPermGraph: p=NULL");
09980     }
09981 #endif
09982     for (n = 0; n < nNodes; n++) {
09983         if (!isATEXternal(pnclass->type[pnclass->nd2cl[n]])) {
09984             cmp = cmpNodes(n, p[n], cl);
09985             if (cmp != 0) {
09986                 return cmp;
09987             }
09988         }
09989     }
09990
09991     njn = 0;
09992     njp = 0;
09993     for (n1 = 0; n1 < nNodes; n1++) {
09994         p1 = p[n1];
09995         nd1 = nodes[n1];
09996         np1 = nodes[p1];
09997         for (n2 = n1; n2 < nNodes; n2++) {
09998             if (mgraph->adjMat[n1][n2] == 0) {
09999                 continue;
10000             }
10001
10002             p2 = p[n2];
10003             if (p1 == n1 && p2 == n2) {
10004                 continue;
10005             }
10006             cmp = mgraph->adjMat[n1][n2] - mgraph->adjMat[p1][p2];
10007             if (cmp != 0) {
10008                 return cmp;
10009             }
10010
10011             njn = 0;
10012             njp = 0;
10013             for (j = 0; j < nd1->deg; j++) {
10014                 if (nd1->anodes[j] == n2) {
10015                     jn[njn++] = getLegParticle(n1, j);
10016                 }
10017             }
10018
10019             for (j = 0; j < np1->deg; j++) {

```

```

10020         if (npl->anodes[j] == p2) {
10021             jp[njp++] = getLegParticle(p1, j);
10022         }
10023     }
10024
10025     cmp = njn - njp;
10026     if (cmp != 0) {
10027         return cmp;
10028     }
10029
10030     // ignore ordering for multiple connections
10031     if (mgraph->adjMat[n1][n2] >= 2) {
10032         sorti(njn, jn);
10033         sorti(njn, jp);
10034     }
10035
10036     for (j = 0; j < njn; j++) {
10037         cmp = jn[j] - jp[j];
10038         if (cmp != 0) {
10039             return cmp;
10040         }
10041     }
10042 }
10043 }
10044
10045 return 0;
10046 }
10047
10048 //-----
10049 int Assign::cmpNodes(int nd0, int nd1, MNodeClass *cn)
10050 {
10051     // Comparison of two nodes 'nd0' and 'nd1'
10052     // Ordering is lexicographical (class, connection configuration)
10053     //
10054     int cmp;
10055
10056     // Wether two nodes are in a same class or not.
10057     cmp = cn->ndcl[nd0] - cn->ndcl[nd1];
10058     if (cmp != 0) {
10059         return cmp;
10060     }
10061
10062     // interaction
10063     cmp = nodes[nd0]->cand->ilist[0] - nodes[nd1]->cand->ilist[0];
10064     return cmp;
10065 }
10066 }
10067
10068 //-----
10069 BigInt Assign::edgeSym(void)
10070 {
10071     // calculate permutations of edges
10072
10073     int lg[GRCC_MAXLEGS], lt[GRCC_MAXLEGS], lc;
10074     ANode *nd1;
10075     int n1, n2, j, k, mult;
10076     BigInt esym;
10077
10078     esym = 1;
10079     for (n1 = 0; n1 < nNodes; n1++) {
10080         nd1 = nodes[n1];
10081         for (n2 = n1; n2 < nNodes; n2++) {
10082             if (mgraph->adjMat[n1][n2] > 1) {
10083                 lc = 0;
10084                 for (j = 0; j < nd1->deg; j++) {
10085                     if (nd1->anodes[j] == n2) {
10086                         lg[lc] = 1;
10087                         lt[lc] = getLegParticle(n1, j);
10088                         lc++;
10089                     }
10090                 }
10091                 for (j = 0; j < lc-1; j++) {
10092                     if (lg[j] > 0) {
10093                         for (k = lc-1; k > j; k--) {
10094                             if (lt[j] == lt[k]) {
10095                                 lg[j] += lg[k];
10096                                 lg[k] = 0;
10097                             }
10098                         }
10099                     }
10100                 }
10101             }
10102
10103             // calculate symmetry factor
10104             for (j = 0; j < lc; j++) {
10105                 if (lg[j] < 1) {
10106                     continue;

```

```

10107         mult = lg[j];
10108         if (mult > 1) {
10109             if (n1 == n2) {
10110                 // self-loop
10111                 esym *= ipow(2,mult/2)*factorial(mult/2);
10112             } else {
10113                 esym *= factorial(mult);
10114             }
10115         }
10116     }
10117 }
10118 }
10119 }
10120
10121 return esym;
10122 }
10123
10124 #ifdef CHECK
10125 //=====
10126 // check
10127 //-----
10128 Bool Assign::checkCand(const char *msg)
10129 {
10130     // Check the consistency of Candidate data
10131
10132     Bool ok;
10133     ANode *na;
10134     NCand *nc;
10135     ECand *ec;
10136     int n, lg, e;
10137
10138     // check 'nodes->cand'
10139     ok = True;
10140     for (n = 0; n < nNodes; n++) {
10141         na = nodes[n];
10142         nc = na->cand;
10143
10144         // check unassigned vertex
10145         if (nc->st == AS_UnAssLegs) {
10146
10147             // check assigned vertex
10148         } else if (nc->st == AS_Assigned) {
10149             if (nc->nplist < 1) {
10150                 grcc_fprintf(GRCC_Stdout, "*** checkCand:7:%s:status (%d) of node %d says"
10151                     " interaction is assigned to %d but ilist=",
10152                     msg, nc->st, n, nc->st);
10153                 prIntArray(nc->nplist, nc->ilist, "\n");
10154                 ok = False;
10155             }
10156
10157             for (lg = 0; lg < nc->deg; lg++) {
10158                 e = na->aedges[lg];
10159                 ec = edges[e]->cand;
10160                 if (ec->nplist != 1) {
10161                     grcc_fprintf(GRCC_Stdout, "*** checkCand:8:%s:status (%d) of node %d says"
10162                         " interaction is assigned "
10163                         " but unassigned edge %d is found\n",
10164                         msg, nc->st, n, e);
10165                     ok = False;
10166                 }
10167             }
10168
10169             // external particle
10170         } else if (nc->st == AS_AssExt) {
10171             // pt = nc->ilist[0];
10172             // *** pte = legEdgeParticle(n, 0, - pt);
10173         } else {
10174             grcc_fprintf(GRCC_Stdout, "*** checkCand:10:%s:illegal status of node %d : %d",
10175                 msg, n, nc->st);
10176             ok = False;
10177         }
10178     }
10179
10180     // check edges
10181
10182     for (e = 0; e < nEdges; e++) {
10183         ec = edges[e]->cand;
10184         if (ec != NULL && ec->nplist < 1) {
10185             grcc_fprintf(GRCC_Stdout, "*** checkCand:12:%s:illegal edge %d\n", msg, e);
10186             ok = False;
10187         }
10188     }
10189
10190     if (!ok) {
10191         grcc_fprintf(GRCC_Stdout, "*** checkCand:15:%s:illegal configuration\n", msg);
10192         prCand("checkCand");
10193         grcc_fprintf(GRCC_Stdout, "*** checkCand:16:illegal configuration\n");

```

```

10194         erEnd("checkCand:l6:illegal configuration");
10195     }
10196     return ok;
10197 }
10198
10199 //-----
10200 void Assign::checkNode(int n, const char *msg)
10201 {
10202     int j, it;
10203
10204     for (j = 0; j < nodes[n]->cand->nilist; j++) {
10205         it = nodes[n]->cand->ilist[j];
10206         if (Abs(it) >= GRCC_MAXMINTERACT) {
10207             grcc_fprintf(GRCC_Stdout, "*** %s: n=%d, j=%d, it=%d\n", msg, n, j, it);
10208             nodes[n]->cand->prNCand(msg);
10209             grcc_fprintf(GRCC_Stdout, "\n");
10210             erEnd("checkNode:illegal it");
10211         }
10212     }
10213 }
10214 #endif // CHECK
10215
10216 //*****
10217 // astack.cc
10218 //=====
10219 // Assign particles to the edges and interactions to nodes.
10220 // Completed graph data is saved in the form of EGraph.
10221
10222 //=====
10223 // stack operations
10224 //-----
10225 void NStack::print(const char *msg)
10226 {
10227     grcc_fprintf(GRCC_Stdout, " node=%d, deg=%d, st=%d, ilist=", noden, deg, st);
10228     prilist(nilist, ilist, msg);
10229 }
10230
10231 //-----
10232 void EStack::print(const char *msg)
10233 {
10234     grcc_fprintf(GRCC_Stdout, " edge=%d, det=%d, plist=", edgen, det);
10235     prilist(nplist, plist, msg);
10236 }
10237
10238 //-----
10239 AStack::AStack(int nsize, int esize)
10240 {
10241     int j;
10242
10243     agraph = NULL;
10244
10245     nSize = nsize;
10246     eSize = esize;
10247     if (nSize > 0) {
10248         nStack = new NStack*[nSize];
10249     } else {
10250         nStack = NULL;
10251     }
10252     if (eSize > 0) {
10253         eStack = new EStack*[eSize];
10254     } else {
10255         eStack = NULL;
10256     }
10257
10258     nStackP = 0;
10259     eStackP = 0;
10260
10261     for (j = 0; j < nSize; j++) {
10262         nStack[j] = new NStack();
10263     }
10264
10265     for (j = 0; j < eSize; j++) {
10266         eStack[j] = new EStack();
10267     }
10268 }
10269
10270 //-----
10271 AStack::~AStack(void)
10272 {
10273     int j;
10274
10275     if (eStack != NULL) {
10276         for (j = eSize-1; j >= 0; j--) {
10277             if (eStack[j] != NULL) {
10278                 delete eStack[j];
10279                 eStack[j] = NULL;
10280             }

```



```

10281     }
10282     delete[] eStack;
10283     eStack = NULL;
10284 }
10285
10286     if (nStack != NULL) {
10287         for (j = nSize-1; j >= 0; j--) {
10288             if (nStack[j] != NULL) {
10289                 delete nStack[j];
10290                 nStack[j] = NULL;
10291             }
10292         }
10293         delete[] nStack;
10294         nStack = NULL;
10295     }
10296 }
10297
10298 //-----
10299 void AStack::setAGraph(Assign *ag)
10300 {
10301     agraph = ag;
10302 }
10303
10304 //-----
10305 void AStack::pushNode(int n)
10306 {
10307     NCand *nc;
10308     NStack *ns;
10309     int j;
10310
10311 #ifdef CHECK
10312     if (agraph == NULL) {
10313         erEnd("pushNode: agraph is not defined");
10314     }
10315 #endif
10316     if (nStackP >= GRCC_MAXNSTACK) {
10317         erEnd("N-stack overflow (GRCC_MAXNSTACK)");
10318     }
10319     nc = agraph->nodes[n]->cand;
10320     if (nc->nilist >= GRCC_MAXMINTERACT) {
10321         erEnd("N-stack: too long list (GRCC_MAXMINTERACT)");
10322     }
10323     ns = nStack[nStackP];
10324     ns->noden = n;
10325     ns->deg = nc->deg;
10326     ns->st = nc->st;
10327     ns->nilist = nc->nilist;
10328     for (j = 0; j < nc->nilist; j++) {
10329         ns->ilist[j] = nc->ilist[j];
10330     }
10331     nStackP++;
10332 }
10333
10334 //-----
10335 void AStack::pushEdge(int e)
10336 {
10337     ECand *ec;
10338     EStack *es;
10339     int j;
10340
10341 #ifdef CHECK
10342     if (agraph == NULL) {
10343         erEnd("pushEdge: agraph is not defined");
10344     }
10345 #endif
10346     if (eStackP >= GRCC_MAXESTACK) {
10347         erEnd("E-stack overflow (GRCC_MAXESTACK)");
10348     }
10349     ec = agraph->edges[e]->cand;
10350     if (ec->nplist >= GRCC_MAXMPARTICLES2) {
10351         erEnd("E-stack: Too long list (GRCC_MAXMPARTICLES)");
10352     }
10353     es = eStack[eStackP];
10354     es->edgen = e;
10355     es->det = ec->det;
10356     es->nplist = ec->nplist;
10357     for (j = 0; j < ec->nplist; j++) {
10358         es->plist[j] = ec->plist[j];
10359     }
10360     eStackP++;
10361 }
10362
10363 //-----
10364 void AStack::checkPoint(CheckPt sav)
10365 {
10366     sav[0] = nStackP;
10367     sav[1] = eStackP;

```

```

10368 }
10369
10370 //-----
10371 void AStack::restoreNode(int spr)
10372 {
10373     NStack *stc;
10374     int     sp, n, j;
10375
10376     for (sp = nStackP-1; sp >= spr; sp--) {
10377         stc = nStack[sp];
10378         n = stc->noden;
10379         agraph->nodes[n]->cand->deg = stc->deg;
10380         agraph->nodes[n]->cand->st = stc->st;
10381         agraph->nodes[n]->cand->nilist = stc->nilist;
10382         for (j = 0; j < stc->nilist; j++) {
10383             agraph->nodes[stc->noden]->cand->ilist[j] = stc->ilist[j];
10384         }
10385     }
10386     nStackP = spr;
10387 }
10388
10389 //-----
10390 void AStack::restoreEdge(int spr)
10391 {
10392     EStack *stc;
10393     int     sp, e, j;
10394
10395     for (sp = eStackP-1; sp >= spr; sp--) {
10396         stc = eStack[sp];
10397         e = eStack[sp]->edgen;
10398         agraph->edges[e]->cand->det = stc->det;
10399         agraph->edges[e]->cand->nplist = stc->nplist;
10400         for (j = 0; j < stc->nplist; j++) {
10401             agraph->edges[e]->cand->plist[j] = stc->plist[j];
10402         }
10403     }
10404     eStackP = spr;
10405 }
10406
10407 //-----
10408 void AStack::restore(CheckPt sav)
10409 {
10410     restoreNode(sav[0]);
10411     restoreEdge(sav[1]);
10412 }
10413
10414 #ifdef CHECK
10415 //-----
10416 void AStack::restoreMsg(CheckPt sav, const char *msg)
10417 {
10418     if (agraph == NULL) {
10419         erEnd("restore: agraph is not defined");
10420     }
10421
10422     restore(sav);
10423
10424     if (!agraph->checkCand("restore")) {
10425         grcc_fprintf(GRCC_Stdout, "restore is called from %s\n", msg);
10426     }
10427 }
10428 #endif
10429
10430 //-----
10431 void AStack::prStack(void)
10432 {
10433     int j;
10434
10435     grcc_fprintf(GRCC_Stdout, "+++ prStack : (%d, %d)", nStackP, eStackP);
10436     for (j = 0; j < nStackP; j++) {
10437         grcc_fprintf(GRCC_Stdout, "N:%4d ", j);
10438         nStack[j]->print("\n");
10439     }
10440     for (j = 0; j < eStackP; j++) {
10441         grcc_fprintf(GRCC_Stdout, "E:%4d ", j);
10442         eStack[j]->print("\n");
10443     }
10444 }
10445
10446 //*****
10447 // class Fraction
10448 //=====
10449 Fraction::Fraction(BigInt n, BigInt d)
10450 {
10451     BigInt g;
10452
10453     g = gcd(n, d);
10454     num = n/g;

```

```

10455     den  = d/g;
10456     ratio = Real(n)/Real(d);
10457 }
10458
10459 //-----
10460 void Fraction::print(const char *msg)
10461 {
10462     double err = Abs(Real(num)/Real(den) - ratio);
10463
10464     if (err > GRCC_FRACERROR) {
10465         grcc_fprintf(GRCC_Stdout, "%ld/%ld(%g) (overflow)%s", num, den, ratio, msg);
10466     } else {
10467         grcc_fprintf(GRCC_Stdout, "%ld/%ld(%g)%s", num, den, ratio, msg);
10468     }
10469 }
10470
10471 //-----
10472 void Fraction::setValue(BigInt n, BigInt d)
10473 {
10474     num = n;
10475     den = d;
10476     ratio = Real(n)/Real(d);
10477 }
10478
10479 //-----
10480 void Fraction::setValue(Fraction &f)
10481 {
10482     num  = f.num;
10483     den  = f.den;
10484     ratio = f.ratio;
10485 }
10486
10487 //-----
10488 BigInt Fraction::gcd(BigInt n0, BigInt n1)
10489 {
10490     BigInt nn[2], r;
10491     int nc;
10492
10493     nn[0] = Abs(n0);
10494     nn[1] = Abs(n1);
10495     nc = 0;
10496
10497     while (1) {
10498         r = nn[nc] % nn[1-nc];
10499         if (r == 0) {
10500             return nn[1-nc];
10501         }
10502         nn[nc] = r;
10503         nc = 1 - nc;
10504     }
10505 }
10506
10507 //-----
10508 void Fraction::normal(void)
10509 {
10510     BigInt g;
10511
10512     if (den == 0) {
10513         erEnd("Fraction: den==0");
10514     }
10515     g = gcd(num, den);
10516     num /= g;
10517     den /= g;
10518     if (den < 0) {
10519         num = - num;
10520         den = - den;
10521     }
10522 }
10523
10524 //-----
10525 void Fraction::add(BigInt n, BigInt d)
10526 {
10527     BigInt g, dl;
10528
10529     if (d < 0) {
10530         n = -n;
10531         d = -d;
10532     }
10533     if (den < 0) {
10534         num = - num;
10535         den = - den;
10536     }
10537     g = gcd(den, d);
10538     dl = d/g;
10539     num = num * dl + n * (den/g);
10540     den = dl*den;
10541     g = gcd(num, den);

```

```

10542     num /= g;
10543     den /= g;
10544     ratio += Real(n)/Real(d);
10545 }
10546
10547 //-----
10548 void Fraction::add(Fraction f)
10549 {
10550     double r = ratio + f.ratio;
10551
10552     add(f.num, f.den);
10553     ratio = r;
10554 }
10555
10556 //-----
10557 void Fraction::sub(Fraction f)
10558 {
10559     double r = ratio - f.ratio;
10560
10561     add(-f.num, f.den);
10562     ratio = r;
10563 }
10564
10565 //-----
10566 Bool Fraction::isEq(Fraction f)
10567 {
10568     return (num == f.num && den == f.den);
10569 }
10570
10571 //*****
10572 // common.cc
10573 //=====
10574
10575 // Wrapper function for printing messages. This allows the use
10576 // of FORM MesPrint when compiled as part of FORM, and the
10577 // usual fprintf to GRCC_Stdout or GRCC_Stderr otherwise.
10578 static void grcc_fprintf(FILE* out, const char* fmt, ...)
10579 {
10580     va_list args;
10581     va_start(args, fmt);
10582
10583 #ifndef NOFORM
10584     DUMMYUSE(out);
10585     // the second call of vsnprintf requires a copy of args
10586     va_list args_copy;
10587     va_copy(args_copy, args);
10588     // determine the required buffer size for the formatted string:
10589     const int len = vsnprintf(NULL, 0, fmt, args);
10590     char *buffer = new char[len+1];
10591     vsnprintf(buffer, len+1, fmt, args_copy);
10592     MLOCK(ErrorMessageLock);
10593     MesPrint("%s", buffer);
10594     MUNLOCK(ErrorMessageLock);
10595     delete[] buffer;
10596     va_end(args_copy);
10597 #else
10598     vfprintf(out, fmt, args);
10599 #endif
10600
10601     va_end(args);
10602     return;
10603 }
10604
10605 static void erEnd(const char *msg)
10606 {
10607     if (erExit != NULL) {
10608         (*erExit)(msg, erExitArg);
10609     }
10610     grcc_fprintf(GRCC_Stderr, "*** Error : %s\n\n", msg);
10611     GRCC_ABORT();
10612 }
10613
10614 #define MAXPART 100
10615 //-----
10616 static Bool nextPart(int nelelem, int nclist, int *clist, int *nl, int *r)
10617 {
10618     // Generate configuration nl[] sequentially such that
10619     //     sum_j^{nclist} nl[j]*clist[j] = nelelem
10620     //     0 <= nl[j]      (j = 0, ..., nclis-1)
10621     // For control
10622     //     for the first call : *r < 0
10623     //     otherwise          : *r >= 0,  nl is the last configuration
10624     // Returns
10625     //     True  : succeeded
10626     //     False : no more configuration
10627
10628     int rem, pn, j, c;

```

```

10629
10630     if (*r < 0) {
10631         *r = 0;
10632         rem = nelelem;
10633         for (j = 0; j < nclist; j++) {
10634             nl[j] = rem/clist[j];
10635             rem -= nl[j]*clist[j];
10636         }
10637         if (rem == 0) {
10638             return True;
10639         }
10640     } else {
10641         rem = 0;
10642     }
10643     for (c = 0; c < MAXPART; c++) {
10644         rem += nl[nclist-1]*clist[nclist-1];
10645         for (pn = nclist-2; pn >= 0 && nl[pn] == 0; pn--) {
10646             ;
10647         }
10648         if (pn < 0) {
10649             return False;
10650         }
10651         rem += clist[pn];
10652         nl[pn]--;
10653         for (j = pn+1; j < nclist; j++) {
10654             nl[j] = rem/clist[j];
10655             rem -= nl[j]*clist[j];
10656         }
10657         if (rem == 0) {
10658             return True;
10659         }
10660     }
10661     erEnd("*** nextPart : illegal control : too many repetition");
10662     return False;
10663 }
10664
10665 //-----
10666 static int *intdup(int n, int *a)
10667 {
10668     int *r, j;
10669
10670     r = new int[n];
10671     for (j = 0; j < n; j++) {
10672         r[j] = a[j];
10673     }
10674     return r;
10675 }
10676
10677 //-----
10678 static int *delintdup(int *a)
10679 {
10680     if (a != NULL) {
10681         delete[] a;
10682     }
10683     return NULL;
10684 }
10685
10686 //-----
10687 static void prillist(int n, const int *a, const char *msg)
10688 {
10689     grcc_fprintf(GRCC_Stdout, "[");
10690     for (int j = 0; j < n; j++) {
10691         if (j!=0) grcc_fprintf(GRCC_Stdout, ", ");
10692         grcc_fprintf(GRCC_Stdout, "%d", a[j]);
10693     }
10694     grcc_fprintf(GRCC_Stdout, "]\n", msg);
10695 }
10696
10697 //-----
10698 static int nextPerm(int nelelem, int nclass, int *cl, int *r, int *q, int *p, int count)
10699 {
10700     // Sequential generation of all permutations.
10701
10702     int j, k, n, e, t;
10703     Bool b;
10704
10705     for (j = 0; j < nelelem; j++) {
10706         p[j] = j;
10707     }
10708     if (count < 1) {
10709         for (j = 0; j < nelelem; j++) {
10710             q[j] = 0;
10711             r[j] = 0;
10712         }
10713         j = 0;
10714         for (k = 0; k < nclass; k++) {
10715             n = cl[k];

```

```

10716         for (e = 0; e < n; e++) {
10717             r[j] = n - e - 1;
10718             j++;
10719         }
10720     }
10721 #ifndef CHECK
10722     if (j != nelem) {
10723         erEnd("inconsistent # elements");
10724     }
10725 #endif
10726     return 1;
10727 }
10728 b = False;
10729 for (j = nelem-1; j >= 0; j--) {
10730     if (q[j] < r[j]) {
10731         for (k = j+1; k < nelem; k++) {
10732             q[k] = 0;
10733         }
10734         q[j]++;
10735         b = True;
10736         break;
10737     }
10738 }
10739 if (!b) {
10740     return (-count);
10741 }
10742
10743 for (j = 0; j < nelem; j++) {
10744     k = j + q[j];
10745     t = p[j];
10746     p[j] = p[k];
10747     p[k] = t;
10748 }
10749 return count + 1;
10750 }
10751
10752 //-----
10753 static BigInt factorial(int n)
10754 {
10755     // returns 1 for n < 1
10756
10757     int r, j;
10758
10759     r = 1;
10760     for (j = 2; j <= n; j++) {
10761         r *= j;
10762     }
10763     return r;
10764 }
10765
10766 //-----
10767 static BigInt ipow(int n, int p)
10768 {
10769     int r, j;
10770
10771     r = 1;
10772     for (j = 0; j < p; j++) {
10773         r *= n;
10774     }
10775     return r;
10776 }
10777
10778 //-----
10779 static int *newArray(int size, int val)
10780 {
10781     // memory allocation of an array
10782
10783     int *a, j;
10784
10785     a = new int[size];
10786     for (j = 0; j < size; j++) {
10787         a[j] = val;
10788     }
10789     return a;
10790 }
10791
10792 //-----
10793 static int *deleteArray(int *a)
10794 {
10795     // memory allocation of an array
10796
10797     delete[] a;
10798     return NULL;
10799 }
10800
10801 //-----
10802 static int **newMat(int n0, int n1, int val)

```

```

10803 {
10804     int **m, j, k;
10805
10806     m = new int*[n0];
10807     for (j = 0; j < n0; j++) {
10808         m[j] = new int[n1];
10809         for (k = 0; k < n1; k++) {
10810             m[j][k] = val;
10811         }
10812     }
10813     return m;
10814 }
10815
10816 //-----
10817 static int **deleteMat(int **m, int n0)
10818 {
10819     int j;
10820
10821     for (j = n0-1; j >= 0; j--) {
10822         delete[] m[j];
10823     }
10824     delete[] m;
10825
10826     return NULL;
10827 }
10828
10829 //-----
10830 static void bsort(int n, int *ord, int *a)
10831 {
10832     int i, j, t;
10833
10834     for (j = n-1; j > 0; j--) {
10835         for (i = 0; i < j; i++) {
10836             if (a[ord[i+1]] < a[ord[i]]) {
10837                 t = ord[i];
10838                 ord[i] = ord[i+1];
10839                 ord[i+1] = t;
10840             }
10841         }
10842     }
10843 }
10844
10845 //-----
10846 static void prMomStr(int mom, const char *ms, int mn)
10847 {
10848     if (mom == 0) {
10849         return;
10850     } else if (mom == 1) {
10851         grcc_fprintf(GRCC_Stdout, " + %s%d", ms, mn);
10852     } else if (mom > 0) {
10853         grcc_fprintf(GRCC_Stdout, " + %d*%s%d", mom, ms, mn);
10854     } else if (mom == -1) {
10855         grcc_fprintf(GRCC_Stdout, " - %s%d", ms, mn);
10856     } else {
10857         grcc_fprintf(GRCC_Stdout, " - %d*%s%d", -mom, ms, mn);
10858     }
10859 }
10860
10861 //-----
10862 static void prIntArray(int n, int *p, const char *msg)
10863 {
10864     int j;
10865
10866     grcc_fprintf(GRCC_Stdout, "[");
10867     for (j = 0; j < n; j++) {
10868         if (j!=0) grcc_fprintf(GRCC_Stdout, ", ");
10869         grcc_fprintf(GRCC_Stdout, "%2d", p[j]);
10870     }
10871     grcc_fprintf(GRCC_Stdout, "]%s", msg);
10872 }
10873
10874 //-----
10875 static void prIntArrayErr(int n, int *p, const char *msg)
10876 {
10877     int j;
10878
10879     grcc_fprintf(GRCC_Stderr, "[");
10880     for (j = 0; j < n; j++) {
10881         if (j!=0) grcc_fprintf(GRCC_Stderr, ", ");
10882         grcc_fprintf(GRCC_Stderr, "%2d", p[j]);
10883     }
10884     grcc_fprintf(GRCC_Stderr, "]%s", msg);
10885 }
10886
10887 //-----

```

```

10890 static void sortrec(int l, int r, int *a)
10891 {
10892     // quick sort : taken from N. Wirth
10893     int i, j, x, w;
10894
10895     i = l;
10896     j = r;
10897     x = a[(l+r)/2];
10898     do {
10899         while(a[i] < x) i++;
10900         while(x < a[j]) j--;
10901         if (i <= j) {
10902             w = a[i]; a[i] = a[j]; a[j] = w;
10903             i++; j--;
10904         }
10905     } while (i < j);
10906     if (l < j) sortrec(l, j, a);
10907     if (i < r) sortrec(i, r, a);
10908 }
10909
10910 //-----
10911 static void sorti(int size, int *a)
10912 {
10913     sortrec(0, size-1, a);
10914 }
10915
10916 //-----
10917 static int sortb(int n, int *a)
10918 {
10919     int i, j, t, nswap;
10920
10921     nswap = 0;
10922     for (j = n-1; j > 0; j--) {
10923         for (i = 0; i < j; i++) {
10924             if(a[i+1] < a[i]) {
10925                 t = a[i];
10926                 a[i] = a[i+1];
10927                 a[i+1] = t;
10928                 nswap++;
10929             }
10930         }
10931     }
10932
10933     // the sign of the permutation is (-1)^(nswap)
10934     return nswap;
10935 }
10936
10937 #ifdef CHECK
10938 //-----
10939 static Bool isIn(int n, int *a, int v)
10940 {
10941     int j;
10942
10943     for (j = 0; j < n; j++) {
10944         if (a[j] == v) {
10945             return True;
10946         }
10947     }
10948     return False;
10949 }
10950 #endif
10951
10952 //-----
10953 static int intSetAdd(int n, int *a, int v, const int size)
10954 {
10955     // Add 'v' into 'a'.
10956     // 'n' is the current # of element.
10957     // 'a' is assumed sorted without duplicated elements.
10958     // Size of 'a' should be large enough.
10959
10960     int j, k;
10961
10962     if (n >= size) {
10963         grcc_fprintf(GRCC_Stderr, "*** intSetAdd : array out of range (>%d)\n", size);
10964         erEnd("intSetAdd : array out of range (GRCC_MAXPSLIST)");
10965     }
10966     for (j = 0; j < n; j++) {
10967         if (a[j] > v) {
10968             break;
10969         } else if (a[j] == v) {
10970             return n;
10971         }
10972     }
10973
10974     // 'j' can be equal to 'n'
10975     for (k = n-1; k >= j; k--) {
10976         a[k+1] = a[k];

```



```

10977     }
10978     a[j] = v;
10979     return n+1;
10980 }
10981
10982 //-----
10983 static int intSListAdd(int n, int *a, int v, const int size)
10984 {
10985     // Add 'v' into 'a'.
10986     // 'n' is the current # of element.
10987     // 'a' is assumed sorted without duplicated elements.
10988     // Size of 'a' should be large enough.
10989
10990     int j, k;
10991
10992     if (n >= size) {
10993         grcc_fprintf(GRCC_Stderr, "*** intSListAdd : array out of range (>%d)\n", size);
10994         erEnd("intSListAdd : array out of range");
10995     }
10996     for (j = 0; j < n; j++) {
10997         if (a[j] >= v) {
10998             break;
10999         }
11000     }
11001
11002     // 'j' can be equal to 'n'
11003     for (k = n-1; k >= j; k--) {
11004         a[k+1] = a[k];
11005     }
11006     a[j] = v;
11007     return n+1;
11008 }
11009
11010 //-----
11011 static int cmpArray(int na, int *a, int nb, int *b)
11012 {
11013     int j, cmp;
11014
11015     cmp = na - nb;
11016     if (cmp != 0) {
11017         return cmp;
11018     }
11019     for (j = 0; j < na; j++) {
11020         cmp = a[j] - b[j];
11021         if (cmp != 0) {
11022             return cmp;
11023         }
11024     }
11025     return 0;
11026 }
11027
11028 //-----
11029 static Bool leqArray(int n, int *a, int *b)
11030 {
11031     int j;
11032
11033     for (j = 0; j < n; j++) {
11034         if (a[j] > b[j]) {
11035             return False;
11036         }
11037     }
11038     return True;
11039 }
11040
11041 //-----
11042 // convert list to sorted list
11043 static int toSList(int n, int *a)
11044 {
11045     sorti(n, a);
11046     return n;
11047 }
11048
11049 //-----
11050 // as set operation assuming a and b are sorted with duplication
11051 // p := a \ b; q := a \cap b; r := b \ a;
11052 // p, q, r are sorted without duplication
11053 static void listDiff(int na, int *a, int nb, int *b, int *np, int *p, int *nq, int *q, int *nr, int
    *r)
11054 {
11055     int ja, jb;
11056
11057     *np = *nq = *nr = 0;
11058     ja = jb = 0;
11059     while (ja < na && jb < nb) {
11060         while (ja < na && a[ja] < b[jb]) {
11061             p[(*np)++] = a[ja++];
11062         }

```

```

11063         while (jb < nb && b[jb] < a[ja]) {
11064             r[(*nr)++] = b[jb++];
11065         }
11066         if (ja < na && jb < nb && a[ja] == b[jb]) {
11067             q[(*nq)++] = a[ja++];
11068             jb++;
11069         }
11070     }
11071     for (; ja < na; ja++) {
11072         p[(*np)++] = a[ja];
11073     }
11074     for (; jb < nb; jb++) {
11075         r[(*nr)++] = b[jb];
11076     }
11077 }
11078
11079 //-----
11080 static int isSubSList(int na, int *a, int nb, int *b)
11081 {
11082     int ja, jb;
11083
11084     ja = jb = 0;
11085     while (ja < na && jb < nb) {
11086         for (; jb < nb && b[jb] < a[ja]; jb++) {
11087             ;
11088         }
11089         if (jb >= nb || b[jb] > a[ja]) {
11090             return False;
11091         }
11092         for (; jb < nb && ja < na && b[jb] == a[ja]; jb++, ja++) {
11093             ;
11094         }
11095     }
11096     return (ja >= na);
11097 }
11098
11099 //-----
11100 static int subtrSet(int na, int *a, int nb, int *b, int *c, int size)
11101 {
11102     // set subtraction 'c' := 'a' - 'b'.
11103     // 'a' and 'b' are sorted lists (not always unique)
11104     // 'c' is the set (sorted and unique elements)
11105
11106     int ja, jb, jc;
11107
11108     jc = 0;
11109     ja = jb = 0;
11110     while (ja < na && jb < nb) {
11111         while (ja < na && a[ja] < b[jb]) {
11112             if (jc == 0 || c[jc-1] != a[ja]) {
11113                 if (jc >= size) {
11114                     erEnd("subsutSet: too small size of array (GRCC_MAXPSLIST)");
11115                 }
11116                 c[jc++] = a[ja];
11117             }
11118             ja++;
11119         }
11120         while (jb < nb && b[jb] < a[ja]) {
11121             ;
11122         }
11123         if (ja < na && jb < nb && a[ja] == b[jb]) {
11124             ja++;
11125             jb++;
11126         }
11127     }
11128     for (; ja < na; ja++) {
11129         if (jc == 0 || c[jc-1] != a[ja]) {
11130             c[jc++] = a[ja];
11131         }
11132     }
11133     return jc;
11134 }
11135
11136 // } } ]

```

4.58 grcc.h

```

00001 #pragma once
00002
00003 /* #[ License : */
00004 /*
00005  * Copyright (C) 2023-2026 T. Kaneko
00006  * When using this file you are requested to refer to the publication

```

```

00007 *   Comput.Phys.Comm. 92 (1995) 127-152
00008 *
00009 *   This file is part of FORM.
00010 *
00011 *   FORM is free software: you can redistribute it and/or modify it under the
00012 *   terms of the GNU General Public License as published by the Free Software
00013 *   Foundation, either version 3 of the License, or (at your option) any later
00014 *   version.
00015 *
00016 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00017 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00018 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00019 *   details.
00020 *
00021 *   You should have received a copy of the GNU General Public License along
00022 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00023 */
00024 /* #] License : */
00025
00026 #include <stdio.h>
00027 #include <stdlib.h>
00028 #include <string.h>
00029 #include <time.h>
00030
00031 #define CHECK
00032 //=====
00033 extern "C" {
00034 #include "grccparam.h"
00035 }
00036
00037 //=====
00038 // Name space of this library
00039 // See 'grccparam.h' for #define
00040 #ifndef GRCC_NAMESPACE
00041 namespace Grcc {
00042 #endif
00043
00044 //=====
00045 // Common types
00046 //
00047 #define True 1
00048 #define False 0
00049 typedef int Bool;
00050
00051 //=====
00052 // Big integer (may be replaced by suitable one)
00053
00054 #define BigInt long
00055 #define ToBigInt(x) ((BigInt) (x))
00056
00057 //=====
00058 // Macro functions
00059 #define Real(x) ((double) (x))
00060 #define Abs(x) (((x) >= 0) ? (x) : (-x))
00061 #define Sign(x) (((x) >= 0) ? (1) : (-1))
00062 #define Max(x, y) ((x) > (y) ? (x) : (y))
00063 #define Min(x, y) ((x) < (y) ? (x) : (y))
00064 #define Second(t) ((double) (*t)=(double) (clock() / ((double)CLOCKS_PER_SEC))))
00065 #define isATEExternal(x) ((x)==GRCC_AT_Initial || (x)==GRCC_AT_Final || (x)==GRCC_AT_External)
00066
00067 //=====
00068 // types and classes
00069 typedef int Edge2n[2]; // pair of nodes expressing an edge
00070 class Assign;
00071 class AStack;
00072 class EEdge;
00073 class EGraph;
00074 class ENode;
00075 class Interaction;
00076 class MGraph;
00077 class MNodeClass;
00078 class MCEdge;
00079 class MCOpi;
00080 class MCBridge;
00081 class MCBlock;
00082 class MConn;
00083 class Model;
00084 class MORbits;
00085 class Options;
00086 class Output;
00087 class Particle;
00088 class PNodeClass;
00089 class Process;
00090 class SGroup;
00091 class SProcess;
00092 class DCGraph;
00093

```

```

00094 //=====
00095 // type of output functions
00096 typedef Bool OutEGB(EGraph *, void *);
00097 typedef void OutEG(EGraph *, void *);
00098 typedef void ErExit(const char *msg, void *);
00099
00100 //=====
00101 class Options {
00102 public:
00103     Model      *model;
00104     Process     *proc;
00105     SProcess    *sproc;
00106     Output      *out;          // for output
00107     OutEGB      *outmg;        // call back function of mgraph
00108     OutEGB      *endmg;        // call back function of end of mgraph
00109     OutEG        *outag;        // call back function of agraph
00110     void        *argmg;        // additional argument to outmg
00111     void        *argemg;       // additional argument to endmg
00112     void        *argag;        // additional argument to outag
00113
00114     //switches for graph generation
00115     OptQGRef qqref[2*GRCC_QGRAF_OPT_Size]; // array of reference to QG-options
00116     int      values[GRCC_OPT_Size];        // array of options
00117     int      qqopt[GRCC_QGRAF_OPT_Size];    // array of QGRAF options
00118
00119     int      nqgopt;                      // effective length of qqref
00120
00121     int      DUMMYPADDING;
00122
00123     // measuring time
00124     double time0;
00125     double time1;
00126
00127     //-----
00128     Options(void);
00129     ~Options(void);
00130     void setDefaultValues(void);
00131     void setOldDefaultValues(void);
00132
00133     void setValue(int ind, int val);
00134     int  getValue(int ind);
00135
00136     void setQGrafOpt(int *qgopt);
00137
00138     void print(void);
00139
00140     void setOutputF(Bool outgrf, const char *fname);
00141     void setOutputP(Bool outgrp, const char *fname);
00142     void printLevel(int l);
00143
00144     void printModel(void);
00145     void setOutMG(OutEGB *omg, void *pt);
00146     void setOutAG(OutEG *oag, void *pt);
00147     void setEndMG(OutEGB *omg, void *pt);
00148     void setErExit(ErExit *ere, void *pt);
00149     const OptDef *getDef(void);
00150     const OptDef *getOldDef(void);
00151     const OptQGDef *getQGDef(void);
00152
00153     void begin(Model *mdl);
00154     void end(void);
00155     void beginProc(Process *prc);
00156     void endProc(void);
00157     void beginSubProc(SProcess *sprc);
00158     void endSubProc(void);
00159     void newMGraph(MGraph *mgr);
00160     void newAGraph(EGraph *egr);
00161     void outModel(void);
00162 };
00163
00164 //=====
00165 class Output {
00166 public:
00167     Options      *opt;
00168     Model        *model;
00169     Process       *proc;
00170     SProcess      *sproc;
00171     char          *outgrf;
00172     FILE          *outgrfp;
00173     char          *outgrp;
00174     FILE          *outgrpp;
00175     int           procId;
00176     Bool          outproc;
00177
00178     Output(Options *optn);

```

```

00181     ~Output(void);
00182
00183     void setOutgrf(const char *fname);
00184     void setOutgrp(const char *fname);
00185     Bool outBeginF(Model *mdl, Bool pr);
00186     Bool outBeginP(Model *mdl, Bool pr);
00187     void outEndF(void);
00188     void outEndP(void);
00189     void outProcBeginF(Process *prc);
00190     void outProcBeginP(Process *prc);
00191     void outProcBegin0(int next, int couple, int loop);
00192     void outSProcBeginF(SProcess *sprc);
00193     void outSProcBeginP(SProcess *sprc);
00194     void outProcEndF(void);
00195     void outProcEndP(void);
00196     void outEGraphF(EGraph *egraph);
00197     void outEGraphP(EGraph *egraph);
00198     void outModelF(void);
00199     void outModelP(void);
00200
00201 };
00202
00203 //=====
00204 class Fraction {
00205 public:
00206     BigInt num, den;
00207     double ratio;
00208
00209     Fraction() { num = 0; den = 1; };
00210     Fraction(BigInt n, BigInt d);
00211
00212     void print(const char *msg);
00213     void setValue(BigInt n, BigInt d);
00214     void setValue(Fraction &f);
00215     void add(BigInt n, BigInt d);
00216     void add(Fraction f);
00217     void sub(Fraction f);
00218     BigInt gcd(BigInt n0, BigInt n1);
00219     void normal(void);
00220     Bool isEq(Fraction f);
00221 };
00222
00223 //*****
00224 // Model
00225 //=====
00226 //-----
00227 class Particle {
00228 public:
00229     Model *mdl;           // the model
00230     char *name;           // the name of the particle
00231     char *aname;          // the name of the anti-particle
00232     int id;               // id of this particle
00233     int ptype;            // the type of particle : GRCC_PT_Scalar, etc.
00234     int neutral;          // True if particle==anti-particle
00235     int pcode;            // Particle code
00236     int acode;            // Anti-particle code
00237     int cmindeg;          // min(deg of connectable interactions)
00238     int cmaxdeg;          // max(deg of connectable interactions)
00239
00240     int extonly;          // can appear only as external particle
00241
00242     //-----
00243     Particle(Model *mdl, int pid, PInput *pinp);
00244     ~Particle(void);
00245
00246     char *particleName(int p);
00247     int particleCode(int p);
00248     char *interactionName(int p);
00249     char *aparticle(void);
00250     void prParticle(void);
00251     int isNeutral(void);
00252     const char *typeName(void);
00253     const char *typeGName(void);
00254
00255 };
00256
00257 //-----
00258 class Interaction {
00259 public:
00260     Model *mdl;           // the model
00261     char *name;           // the name of the interaction
00262     int *plist;            // the list of particle codes
00263     int *clist;            // the orders of coupling constants
00264     int *slist;           // the sorted list of particle codes
00265
00266     int id;               // id of this interaction
00267     int csum;             // the total orders of coupling constants

```

```

00268     int     nlegs;           // the number of legs
00269     int     loop;           // the number of loops
00270     int     icode;          // user defined interaction code
00271
00272     int     nplist;          // the list of particle codes
00273     int     nclist;          // the orders of coupling constants
00274     int     nslist;          // the sorted list of particle codes
00275
00276     //-----
00277     Interaction(Model *modl, int iid, const char* nam, int icode, int *cpl, int nlgs, int *plst, int
csm, int lp);
00278     ~Interaction(void);
00279
00280     void prInteraction(void);
00281 };
00282
00283 //-----
00284 class Model {
00285 public:
00286     char     *name;           // the name of this model
00287     char     **cnlist;         // the list of coupling constant names
00288     int       ncouple;         // the number of coupling consants.
00289
00290     int       nParticles;      // the number of particles
00291     Particle  **particles;     // the list of particles
00292     int       pdef;           // the def. of partcls is ended
00293
00294     // list of particles and anti-p. without Undef.
00295     int       nallPart;
00296     int       allPart[GRCC_MAXMPARTICLES2];
00297
00298     // list of particles and anti-p. without ones of extloop=True
00299     int       nintPart;
00300     int       intPart[GRCC_MAXMPARTICLES2];
00301
00302     int       nInteracts;      // the number of interactions.
00303     Interaction **interacts;    // the list of interactions
00304     int       vdef;           // the definition of interaction is ended
00305
00306     int       maxnlegs;        // maximum number of legs of an interaction
00307     int       maxcpl;          // maximum number of coupling const.
00308     int       maxloop;         // maximum number of loops inside an intr.
00309
00310     // classification of interactions
00311     int       *cplgcp;         // coupling constants
00312     int       *cplglg;         // degree
00313     int       *cplgnvl;        // number of vertices
00314     int       **cplgvl;        // list of vertices
00315     int       ncplgcp;         // the number of classes
00316
00317     int       defpart;         // GRCC_DEFBYNAME or GRCC_DEFBYCODE
00318
00319     // methods
00320     Model(MInput *minp);
00321     ~Model(void);
00322
00323     void prModel(void);
00324     void addParticle(PInput *pinp);
00325     void addParticleEnd(void);
00326     void addInteraction(IInput *iinp);
00327     void addInteractionEnd(void);
00328     int findParticleName(const char *name);
00329     int findParticleCode(int pcd);
00330     int findInteractionName(const char *name);
00331     int findInteractionCode(int icd);
00332     char *particleName(int p);
00333     int particleCode(int p);
00334     int normalParticle(int pt);
00335     int antiParticle(int pt);
00336     int *allParticles(int *len);
00337     int findMClass(const int cpl, const int dgr);
00338     void prParticleArray(int n, int *a, const char *msg);
00339
00340     static void printMInput (MInput *min);
00341     static void printPInput (PInput *pin);
00342     static void printIInput (IInput *iin);
00343 };
00344
00345 //*****
00346 // Process
00347 //=====
00348 class PNodeClass {
00349 public:
00350     SProcess *sproc;
00351     int       *deg;           // The degree of each node
00352     int       *type;          // The type of the class : GRCC_AT_xxx
00353     int       *particle;      // The particle code of external node.

```

```

00354     int      *couple;      // The orders of coupling constans
00355     int      *cmindeg;     // min(deg of connectable vertex)
00356     int      *cmaxdeg;     // max(deg of connectable vertex)
00357     int      *count;       // The number of nodes in the class
00358     int      *cl2nd;       // cl2nd[class] <= nd < cl2nd[class+1] = set of nodes
00359     int      *nd2cl;       // nd2cl[node] = class
00360     int      *cl2mcl;      // cl2mcl[class] = class defined in the model.
00361     int      nnodes;
00362     int      nclass;
00363
00364     PNodeClass(SProcess *sprc, int nnodes, int nclss, NCInput *cls);
00365     PNodeClass(SProcess *sprc, int nnodes, int nclss, int *dgs, int *typ, int *ptcl, int *cpl, int
*cnt, int *cmind, int *cmaxd);
00366     ~PNodeClass(void);
00367
00368     void prPNodeClass(void);
00369     void prElem(int e);
00370 };
00371
00372 //=====
00373 class SProcess {
00374     // In sprocess the number of nodes and edges are fixed.
00375     // A node is considered as
00376     // (degree, total order of coupling constants),
00377     // which pair of data defines a class of nodes.
00378
00379 public:
00380     Model      *model;      // model
00381     Process    *proc;       // mother process
00382     Options    *opt;        // options
00383     PNodeClass *pnclass;    // list of classes (cpl and deg is determined)
00384     AStack     *astack;
00385
00386     MGraph     *mgraph;
00387     EGraph     *egraph;
00388     Assign     *agraph;
00389
00390     int      *cl2nd;        // (code of nodes in the class[c])
00391                        // = [cl2nd[c], ..., cl2nd[c+1]-1]
00392     int      *nd2cl;       // node nd is in the class pnclass[nd2cl[nd]]
00393     int      nclass;       // the number of classes
00394
00395     int      ninitl;       // the number of initial particles
00396     int      nfinal;      // the number of final particles
00397     int      nvert;       // the number of vertices
00398
00399     int      clist[GRCC_MAXNCPLG]; // coupling constans of the process
00400     int      id;          // sprocess id
00401     int      loop;        // the number of loops
00402     int      nNodes;      // the number of nodes
00403     int      nEdges;      // the number of edges
00404     int      nExtern;     // the number of external particles
00405     int      ncouple;     // the number of coupling constants
00406     int      tCouple;     // the total coupling constants
00407
00408     int      DUMMYPADDING;
00409
00410     BigInt   mgrcount;    // count generated mgraph
00411     BigInt   agrcount;    // count generated agraph
00412     BigInt   extperm;     // count generated agraph
00413
00414     // the results of the graph generation
00415     BigInt   nMGraphs;    // the number of generated M-graphs
00416     BigInt   nMOPI;      // the number of 1PI M-graphs
00417     Fraction wMGraphs;    // the weighted sum of M-graphs
00418     Fraction wMOPI;      // the weighted sum of 1PI M-graphs
00419
00420     BigInt   nAGraphs;    // the number of generated A-graphs
00421     BigInt   nAOPI;      // the number of 1PI A-graphs
00422     Fraction wAGraphs;    // the weighted sum of A-graphs
00423     Fraction wAOPI;      // the weighted sum of 1PI A-graphs
00424
00425     // methods
00426     SProcess(Model *mdl, Process *prc, Options *opts, int sid, int *clst, int ncls, NCInput *cls);
00427     SProcess(Model *mdl, Process *prc, Options *opts, int sid, int *clst, int ncls, int *cdeg, int
*ctyp, int *ptcl, int *cpl, int *cnum, int *cmind, int *cmaxd);
00428
00429     ~SProcess(void);
00430
00431     void prSProcess(void);
00432     BigInt generate(void);
00433     void assign(MGraph *mgr);
00434
00435     int toMNodeClass(int *ctyp, int *cldeg, int *clnum, int *cmind, int *cmaxd);
00436     PNodeClass *match(MGraph *mgr);
00437
00438     void endMGraph(MGraph *mgr);

```

```

00439     void endAGraph(EGraph *egr);
00440
00441     void resultMGraph(BigInt nmgraphs, Fraction mwsum, BigInt nmopi, Fraction mwopi);
00442     void resultAGraph(BigInt nagraphs, Fraction awsum, BigInt naopi, Fraction awopi);
00443
00444 };
00445
00446 //=====
00447 class Process {
00448 public:
00449     Model      *model;           // model used for this process
00450     Options    *opt;             // options
00451     BigInt     mgrcount;         // count generated mgraph
00452     BigInt     agrcount;         // count generated agraph
00453     int        *initlPart;       // list of ids of initial particles
00454     int        *finalPart;       // list of ids of final particles
00455
00456     int        id;              // process id
00457     int        ninitl;          // the number of initial particles
00458     int        nfinal;          // the number of final particles
00459     int        ctotall;         // the total number of coupling constants
00460     int        nExtern;         // the number of external particles
00461     int        loop;            // the number of external particles
00462     int        maxnlegs;        // the maximum possible degree of node
00463     int        clist[GRCC_MAXNCLG]; // coupling constants of the process
00464
00465     // table of sprocesses
00466     int        nSubproc;        // the number of sprocesses
00467     SProcess   *sptbl[GRCC_MAXSUBPROCS]; // the table of sprocesses
00468     SProcess   *sproc;          // the current sprocess
00469
00470     // stack for the assignment;
00471     AStack     *astack;
00472
00473     // the number of graphs
00474     BigInt     ngraphs;
00475     BigInt     nopi;
00476     BigInt     wgraphs;
00477     BigInt     wopi;
00478
00479     // the results of the graph generation
00480     BigInt     nMGraphs;         // the number of generated M-graphs
00481     BigInt     nMOPI;           // the number of 1PI M-graphs
00482     Fraction   wMGraphs;        // the weighted sum of M-graphs
00483     Fraction   wMOPI;          // the weighted sum of 1PI M-graphs
00484
00485     BigInt     nAGraphs;        // the number of generated A-graphs
00486     BigInt     nAOPI;           // the number of 1PI A-graphs
00487     Fraction   wAGraphs;        // the weighted sum of A-graphs
00488     Fraction   wAOPI;          // the weighted sum of 1PI A-graphs
00489
00490     double     sec;
00491
00492     Process(int pid, Model *model, Options *opt, int nin, int *initlPart, int nfin, int *finalPart,
00493 int *coupling);
00494     Process(int pid, Model *model, Options *opt, FGInput *fgi);
00495     ~Process(void);
00496
00497     void prProcess(void);
00498     void outProcP(FILE *fp);
00499     void prProcessP(const char *fname);
00500     void mkSProcess(void);
00501 };
00502
00503 //*****
00504 // classes for EGraph
00505 //=====
00506 // Constants
00507
00508 #define GRCC_ED_Undef  0
00509 #define GRCC_ED_Deleted 1
00510 #define GRCC_ED_Extern 2
00511 #define GRCC_ED_Back  3
00512 #define GRCC_ED_Bridge 4
00513 #define GRCC_ED_Inloop 5
00514 #define GRCC_ED_Size  6
00515 #define GRCC_ED_NAMES {"Undef", "Deleted", "Extern", "Back_Edge", "Bridge", "In_Loop"}
00516
00517 #define GRCC_ND_Undef  0
00518 #define GRCC_ND_Deleted 1
00519 #define GRCC_ND_Initial 2
00520 #define GRCC_ND_Final  3
00521 #define GRCC_ND_CPoint 4
00522 #define GRCC_ND_VBlock 5
00523 #define GRCC_ND_Inblock 6
00524 #define GRCC_ND_Size  7

```



```

00525 #define GRCC_ND_NAMES    {"Undef", "Deleted", "Init", "Final", "CPoint", "VBlock", "In_Block"}
00526
00527 //-----
00528 class ENode {
00529 public:
00530     EGraph *egraph;
00531     int     *edges;           // list of edges
00532                               // the value is \pm [(edge index)+1]
00533
00534     int     id;              // id of the enode
00535     int     maxdeg;          // maximum degree
00536     int     deg;             // degree
00537     int     extloop;         // loop inside this node or AT_Initial etc.
00538     int     ndtype;          // type of the node
00539     int     intrct;          // assigned interaction/particle (ext.)
00540
00541     // used in EGraph::biconn()
00542     int     *klow;
00543     int     visited;
00544
00545     int     DUMMYPADDING;
00546
00547 //-----
00548 // functions
00549 ENode(void);
00550 ENode(EGraph *egrph, int loops, int sdeg);
00551 ~ENode(void);
00552 void initAss(EGraph *egrph, int nid, int sdg);
00553 void setId(EGraph *egrph, const int nid);
00554 void copy(ENode *en);
00555 void setExtern(int typ, int pt);
00556 void setType(int typ);
00557 void print(void);
00558
00559 };
00560
00561 //-----
00562 class EEdge {
00563 // momentum is printed like: ("%s%d", (enode.ext)?"Q":"p", enode.momn)
00564 public:
00565     EGraph *egraph;          // egraph
00566
00567     int     id;              // id
00568     int     ext;             // id
00569     int     ptcl;            // assigned particle (agraph)
00570     int     deleted;          // deleted edge, if true
00571     int     nodes[2];         // nodes of bothsides
00572     int     nlegs[2];         // nodes of both side (agraph)
00573
00574     // for biconn
00575     int     *emom;            // external momenta
00576     int     *lmom;            // loop momenta
00577     int     *extMom;          // set of external momenta.
00578
00579     // momentum obtain in searchME
00580     ULong   momset;           // set of momenta in bit string (leaf --> root)
00581     int     momdir;           // direction (leg=0 --> leg=1)
00582
00583     Bool    cut;
00584     int     visited;
00585     int     conid;            // connected component
00586     int     edtype;           // type
00587     int     opicomp;          // id of lPI component
00588     int     dir;              // direction of momentum
00589
00590     int     DUMMYPADDING;
00591
00592 //-----
00593 // functions
00594 EEdge(void);
00595 EEdge(EGraph *egrph, int nedges, int nloops);
00596 ~EEdge(void);
00597 void copy(EEdge *ee);
00598 void setId(EGraph *egrph, const int eid);
00599 void print(void);
00600
00601 void setType(int typ);
00602 void setLMom(int k, int dir);
00603 void setEMom(int nedges, int *extn, int dir);
00604 };
00605
00606 //-----
00607 // type of fermion lines
00608
00609 typedef enum {FL_Open, FL_Closed} FLType;
00610
00611 //-----

```

```

00612 class EFLine {
00613 public:
00614     int     elist[GRCC_MAXNODES];    // list of  (\pm [(edge index)+1])
00615     FLType  ftype;
00616     int     fkind;
00617     int     nlist;
00618
00619     EFLine(void);
00620     void print(const char *msg);
00621 };
00622
00623 //-----
00624 class EGraph {
00625 public:
00626     Options    *opt;           // table of options
00627     Model      *model;
00628     Process    *proc;
00629     SProcess    *sproc;
00630     MGraph     *mgraph;
00631     MConn      *econn;
00632
00633     ENode **nodes;
00634     EEdge **edges;           // edges[nEdges+1]: index starts from 1
00635
00636     BigInt mId;              // id of mgraph
00637     BigInt aId;              // id of agraph in the same mgraph
00638     BigInt sId;              // sequential no. of agraph
00639     BigInt gSubId;           // ???
00640     Bool    assigned;        // mgraph (False) or agraph (True)
00641
00642     int     fsign;
00643     BigInt nsym, esym;       // symmetry factor with symm. ext.
00644     BigInt nsym1;            // symmetry factor without symm. ext.
00645     BigInt extperm;          // the order of group of symm. ext.
00646     BigInt multp;            // multiplicity of graph in symm. ext.
00647
00648     int     pId;
00649
00650     int     sNodes;          // memory size
00651     int     sEdges;          // memory size
00652     int     sMaxdeg;         // memory size
00653     int     sLoops;          // memory size
00654
00655     int     nNodes;
00656     int     nEdges;
00657     int     nExtern;
00658     int     maxdeg;          // maximum value of degree of nodes
00659     int     nLoops;
00660     int     totalc;          // total order of coupling constants
00661
00662     // biconnect
00663     int     nopicomp;
00664     int     opi2plp;
00665     int     nopi2p;
00666     int     nadj2ptv;        // the no. of edges connecting 2point vertices
00667
00668     int     DUMMYPADDING;
00669
00670     int     *bidef;
00671     int     *biLow;
00672     int     *extMom;
00673     int     bconn;
00674     int     bicount;
00675     int     loopm;
00676     int     opiCount;
00677
00678     // Fermion lines
00679     EFLine *flines[GRCC_MAXFLINES];
00680     int     nFlines;
00681
00682     int     DUMMYPADDING1;
00683
00684     //-----
00685     // functions
00686     EGraph(int nnodes, int nedges, int mxdeg);
00687     ~EGraph(void);
00688
00689     void copy(EGraph *eg);
00690     void print(void);
00691     void printPy(FILE *fp, long mId);
00692     void fromDGraph(DGraph *dg);
00693     void fromMGraph(MGraph *mgraph);
00694
00695     Bool    optQGrafM(Options *opt);
00696     Bool    optQGrafA(Options *opt);
00697     Bool    isOptE(void);
00698

```

```

00699     ENode *setExtern(int n0, int pt, int ndtp);
00700     Bool  isExternal(int nd)  { return (nodes[nd]->extloop < 0); };
00701     Bool  isFermion(int nd);
00702
00703     void setExtLoop(int nd, int val);
00704     void endSetExtLoop(void);
00705
00706     int connComp(void);
00707     int connVisit(int nd, int ncc);
00708
00709     void biconnE(void);
00710     void biinitE(void);
00711     void bisearchE(int nd, int *extlst, int *intlst, int *opiext, int *opiloop);
00712
00713     int findRoot(void);
00714     int dirEdge(int n, int e);
00715     void extMomConsv(void);
00716     int cmpMom(int *lm0, int *em0, int *lml, int *eml);
00717     int groupLMom(int *grp, int *ed2gr);
00718
00719     void chkMomConsv(void);
00720
00721     void prFLines(void);
00722     void getFLines(void);
00723     int fltrace(int fk, int nd0, int *fl);
00724     void addFLine(const FLType ft, int fk, int nfl, int *fl);
00725
00726     int legParticle(int ed, int lg);
00727 };
00728
00729 //*****
00730 // Symmetry group of graphs
00731 //=====
00732 class SGroup {
00733 public:
00734     BigInt size;           // # of allocated elements
00735     BigInt nelemt;         // # of saved elements
00736     int **elem;            // saved elements
00737     int nnodes;            // # of nodes.
00738     int neclss;            // # of classes
00739     int eclass[GRCC_MAXNODES]; // table of classes
00740     int cgen;              // counter for the generation
00741     int csav;              // counter for the saved elements
00742     int permg[GRCC_MAXNODES]; // resulting permutation
00743     int perms[GRCC_MAXNODES]; // curr. elem of saved ones.
00744
00745     int pgr[GRCC_MAXNODES]; // work
00746     int pgq[GRCC_MAXNODES]; // work
00747     int psr[GRCC_MAXNODES]; // work
00748     int psq[GRCC_MAXNODES]; // work
00749
00750     //-----
00751     // functions
00752
00753     SGroup(void);
00754     ~SGroup(void);
00755
00756     void print(void);
00757     void newGroup(int nelmt, int nclss, int *clss);
00758     void clearGroup(void);
00759     void delGroup(void);
00760     int *genNext(void);
00761     void addGroup(int *p);
00762     BigInt nElem(void);
00763     int *nextElem(void);
00764 };
00765
00766 //*****
00767 // MGraph
00768 //=====
00769 // class of nodes for MGraph
00770
00771 class MNode {
00772 public:
00773     int id;           // node id
00774     int deg;          // degree(node) = the number of legs
00775     int clss;         // initial class number in which the node belongs
00776     int extloop;
00777     int cmindeg;
00778     int cmaxdeg;
00779
00780     int freelg;       // the number of free legs
00781     int visited;
00782
00783     MNode(int id, int clss, NCInput *mgi);
00784     MNode(int id, int deg, int extloop, int clss, int cmindeg, int cmaxdeg);
00785 };

```

```

00786
00787 //=====
00788 // class of scalar graph expressed by matrix form
00789 //
00790 // Input : the classified set of nodes.
00791 // Output : control passed to 'EGraph(self)'
00792
00793 class MGraph {
00794
00795 public:
00796
00797     // initial conditions
00798     Options *opt;          // options
00799
00800     // symmetry group
00801     SGroup *group;         // symmetry group
00802     MOrbits *orbits;       // Orbits of nodes with respect to symmetry group
00803
00804     MNode **nodes;         // table of MNode object
00805     BigInt mId;            // process/sprocess ID
00806     int *clist;            // list of initial classes
00807     int pId;               // process/sprocess ID
00808     int nNodes;            // the number of nodes
00809     int nEdges;            // the number of edges
00810     int nLoops;            // the number of loops
00811     int nExtern;           // the number of external nodes
00812     int nClasses;          // the number of initial classes
00813     int mindeg;            // minimum value of degree of nodes
00814     int maxdeg;            // maximum value of degree of nodes
00815
00816     // the current graph
00817     int **adjMat;          // adjacency matrix
00818     MNodeClass *curcl;     // the current 'MNodeClass' object
00819     EGraph *egraph;
00820     BigInt nsym;           // symmetry factor from nodes
00821     BigInt esym;           // symmetry factor from edges
00822
00823     // generated set of graphs
00824     BigInt cDiag;          // the total number of generated graphs
00825     BigInt cLPI;           // the total number of LPI graphs
00826     BigInt cNoTadpole;     // the total number of graphs without tadpoles
00827     BigInt cNoTadBlock;    // the total number of graphs without tad-blocks
00828     BigInt cLPINoTadBlock; // the total number of LPI graphs without tad-blocks
00829     Fraction wscon;        // weighted sum of graphs
00830     Fraction wsopi;        // weighted sum of LPI graphs
00831
00832     // measures of efficiency
00833     BigInt ngen;           // generated graph before check
00834     BigInt ngconn;         // generated connected graph before check
00835
00836     BigInt nCallRefine;
00837     BigInt discardRefine;
00838     BigInt discardDisc;
00839     BigInt discardIso;
00840
00841     // for options
00842     Bool opi;
00843     Bool opiloop;
00844     Bool extself;
00845     Bool selfloop;
00846     Bool multiedge;
00847     Bool tadpole;
00848     Bool tadblock;
00849     Bool block;
00850     Bool bipart;
00851
00852     int DUMMYPADDING1;
00853
00854     // table of n edge-connected components
00855     MConn *mconn;
00856
00857     // work space for isomorphism
00858     int **modmat;          // permuted adjacency matrix
00859
00860     // work space for biconnected component
00861     int *bidef;
00862     int *bilow;
00863     int *bicol;
00864     int bicount;
00865
00866     int DUMMYPADDING2;
00867
00868     //-----
00869     // functions
00870     MGraph(int pid, int ncl, NCInput *mgi, Options *opt);
00871     MGraph(int pid, int ncl, int *cldeg, int *clnum, int *clxl, int *cmind, int *cmaxd, Options
*opt);

```

```

00872     ~MGraph(void);
00873
00874     void    init(void);
00875     BigInt  generate(void);
00876     Bool    isExternal(int nd)  { return (nodes[nd]->extloop < 0); };
00877     void    printAdjMat(MNodeClass *cl);
00878     void    print(void);
00879     void    printPy(FILE *fp, long mId);
00880
00881     Bool    isConnected(void);
00882     Bool    visit(int nd);
00883     Bool    isIsomorphic(MNodeClass *cl);
00884     void    permMat(int size, int *perm, int **mat0, int **mat1);
00885     int     compMat(int size, int **mat0, int **mat1);
00886     MNodeClass *refineClass(MNodeClass *cl);
00887     void    bisearchME(int nd, int pd, int ned, int col, MCOpi *mopi, MCBLOCK *mblk, ULong *momset, int
*next, int *nart);
00888     void    biconnME(void);
00889
00890     Bool    isOptM(void);
00891     void    connectClass(MNodeClass *cl);
00892     void    connectNode(int sc, int ss, MNodeClass *cl);
00893     void    connectLeg(int sc, int sn, int tc, int ts, MNodeClass *cl);
00894     void    newGraph(MNodeClass *cl);
00895 };
00896
00897 //=====
00898 // class of node-classes for MGraph
00899 //-----
00900 class MNodeClass {
00901 public:
00902     int    clmat[GRCC_MAXNODES][GRCC_MAXNODES]; // matrix used for classification
00903     int    clist[GRCC_MAXNODES]; // the number of nodes in each class
00904     int    ndcl[GRCC_MAXNODES]; // node --> class
00905     int    flist[GRCC_MAXNODES+1]; // the first node in each class
00906     int    clord[GRCC_MAXNODES]; // ordering of classes
00907     int    cmindeg[GRCC_MAXNODES]; // min(deg of connectable node)
00908     int    cmaxdeg[GRCC_MAXNODES]; // max(deg of connectable node)
00909     int    nNodes; // the number of nodes
00910     int    nClasses; // the number of classes
00911     int    maxdeg; // maximal value of degree(node)
00912
00913     int    flg0;
00914     int    flg1;
00915     int    flg2;
00916
00917     MNodeClass(int nnodes, int nclasses);
00918     ~MNodeClass(void);
00919
00920     void    init(int *cl, int mxdeg, int **adjmat);
00921     void    copy(MNodeClass* mnc);
00922     int     clCmp(int nd0, int nd1, int cn);
00923     void    printMat(void);
00924
00925     void    mkFlist(void);
00926     void    mkNdCl(void);
00927     void    mkClMat(int **adjmat);
00928     void    incMat(int nd, int td, int val);
00929     int     cmpMNCArray(int *a0, int *a1, int ma);
00930     void    reorder(MGraph *mg);
00931 };
00932
00933 //=====
00934 // class of an edge
00935 //-----
00936 class MCEdge {
00937 public:
00938     Edge2n nodes; // nodes at the both size of the edge (leaf --> root)
00939     ULong    momset; // set of momenta in bit string
00940     int      momdir; // set of momenta in bit string
00941
00942     int      DUMMYPADDING;
00943
00944     MCEdge(void);
00945     ~MCEdge(void);
00946 };
00947
00948 //=====
00949 // class of lPI component
00950 //-----
00951 class MCOpi {
00952 public:
00953     int *nodes; // array of nodes in
00954     int  nnodes; // # nodes in the lPI component
00955     int  nlegs; // # leg (bridges) of the lPI component
00956     int  next; // # external particles of the lPI comp.
00957     int  nedges; // # edges in the lPI comp.

```

```

00958     int    loop;        // # loops in the lPI comp.
00959     int    ctloop;      // # loops in the counter terms in the OP comp.
00960     int    mom0lg;      // # leg (bridges) with 0 momentum
00961
00962     int          DUMMYPADDING;
00963
00964     MCOpi(void);
00965     ~MCOpi(void);
00966
00967     void init(void);
00968 };
00969
00970 //=====
00971 // class of bridge
00972 //-----
00973 class MCBridge {
00974 public:
00975     Edge2n nodes; // nodes at the both size of the bridge
00976     int    next;  // # momenta of ext. particles flowing on the bridge
00977
00978     MCBridge(void);
00979     ~MCBridge(void);
00980 };
00981
00982 //=====
00983 // class of block
00984 //-----
00985 class MCBlock {
00986 public:
00987     Edge2n *edges; // array of edges in the block
00988     int    nedges; // # edges in the block
00989     int    nartps; // # articulation points of the block
00990     int    loop;   // # loop in the block
00991
00992     int          DUMMYPADDING;
00993
00994     MCBlock(void);
00995     ~MCBlock(void);
00996     void init(void);
00997 };
00998
00999 //=====
01000 // class of table of MConn
01001 //-----
01002 class MConn {
01003 public:
01004     // 2-edge connected components
01005     MCEdge *cedges; // table of edges
01006     MCOpi  *opics;  // table of n-edge-connected components
01007     MCBridge *bridges; // table of bridges
01008     MCBlock  *blocks; // table of blocks
01009     int    *articuls; // buffer for nodes for articulation points
01010
01011     int    *opisp; // buffer for nodes in lPI components
01012     int    *opistk; // stack of nodes for lPI components.
01013     Edge2n *blksp; // buffer for edges in blocks
01014     Edge2n *blkstk; // stack of edges for blocks
01015     int    snodes; // # nodes
01016     int    sedges; // # edges
01017
01018     // opi components (edge-connected)
01019     int    nopic; // # lPI components (nlPIComps)
01020     int    nlpopic; // # looped lPI components (nlPIComps)
01021     int    nctopic; // # lPI components of one counter term.
01022
01023     // edges
01024     int    nbacked; // # back edges
01025
01026     // bridges
01027     int    nbridges; // # bridges
01028     int    ne0bridges; // # bridges whose next=0
01029     int    nelbridges; // # bridges whose next=1
01030     int    nselfloops;
01031     int    nmultiedges;
01032
01033     // blocks (node-connected)
01034     int    nblocks; // # blocks
01035     int    nalblocks; // # bridges whose next=0
01036     int    narticuls; // # articulation points
01037     int    neblocks; // # effective looped blocks
01038
01039     // indices to work spaces
01040     int    nopisp; // # used in opisp
01041     int    opistkptr; // # stack pointer of opistk
01042     int    nblksp; // # used in blksp
01043     int    blkstkptr; // # stack pointer of tlkstk
01044

```

```

01045     int      DUMMYPADDING;
01046
01047     MConn(int nnod, int nedg);
01048     ~MConn(void);
01049
01050     void init(void);
01051     void initCEdges(MGraph *);
01052     void pushNode(int nd);
01053     void pushEdge(int n0, int n1);
01054     void addCEdge(int n0, int n1, ULong momset);
01055     void addOPic(MCOpI *mopi, int stp);
01056     void addBridge(int n0, int n1, int nex, int nextot);
01057     void addArtic(int nd, int mul);
01058     void addBlock(MCBlock *eblk, int stp);
01059     void addBlockSelf(int nd, int mul);
01060     void print(void);
01061     void prEdges(void);
01062 };
01063
01064 //=====
01065 //  class of orbits of nodes
01066 //-----
01067 class MOrbits {
01068     // Usage
01069     // 1. node ==> orbit
01070     //      (class) = nd2or[(node)]
01071     //
01072     // 2. class ==> nodes
01073     //      for (c = 0; c < nClass; c++) {
01074     //          for (j = flist[c]; j < flist[c+1]; j++) {
01075     //              (node) = or2nd[c];
01076     //          }
01077     //      }
01078 public:
01079     int nOrbits;
01080     int nNodes;
01081     int nd2or[GRCC_MAXNODES];
01082     int or2nd[GRCC_MAXNODES];
01083     int flist[GRCC_MAXNODES+1];
01084
01085     MOrbits(void);
01086     ~MOrbits(void);
01087
01088     void print(void);
01089
01090     // construction of orbits
01091     void initPerm(int nnodes);
01092     void fromPerm(int *perm);
01093     void toOrbits(void);
01094 };
01095
01096 //*****
01097 // Particle assignment
01098 //=====
01099 typedef int CheckPt[2];
01100
01101 typedef enum {
01102     AS_UnAssLegs, AS_Assigned, AS_Assigned0, AS_AssExt, AS_Impossible
01103 } NCandSt;
01104
01105 //=====
01106 class NCand {
01107     // List of candidates for the assignment of interactions to a vertex
01108
01109 public:
01110     int      deg;                // degree of the node
01111     NCandSt  st;                // status
01112     int      nilst;             // length of the list
01113     int      ilist[GRCC_MAXMINTERACT]; // list of candidates
01114
01115     //=====
01116     NCand(const NCandSt sta, const int dega, const int nilst, int *ilst);
01117     ~NCand(void);
01118
01119     // print
01120     void prNCand(const char* msg);
01121 };
01122
01123 //=====
01124 class ECand {
01125     // List of candidates for the assignment of particles to an edge
01126
01127 public:
01128     Bool     det;               // determined or not
01129     int      nplist;            // size of the list of candidates
01130     int      plist[GRCC_MAXMPARTICLES2]; // list of candidates
01131

```

```

01132     ECand(int dt, int nplist, int *plst);
01133     ~ECand(void);
01134
01135     // print
01136     void prECand(const char *msg);
01137 };
01138
01139 //=====
01140 class ANode {
01141     // Connection information of a node to others
01142     //
01143     // (this node) -- (adjacent edge) -- (next node)
01144     //      (n0, j)           e           n1
01145     //
01146     // e = (aedges[j], aelegs[j])
01147     // n1 = anodes[j]
01148
01149 public:
01150     int deg;           // degree of the node
01151     int nlegs;         // the number of legs already assigned
01152     int *anodes;       // anodes[j] = (next node of leg j)
01153     int *aedges;       // aedges[j] = (next edge of leg j)
01154     int *aelegs;       // aelegs[j] = (leg of the next edge)
01155     NCand *cand;       // candidate list
01156
01157     ANode(int dg);
01158     ~ANode(void);
01159
01160     int newleg(void);   // get a new leg
01161 };
01162
01163 //=====
01164 class AEdge {
01165     // Connection information of an edge
01166     // (self) ==> nodes [(nodes[0], nlegs[0]), (nodes[1], nlegs[1])]
01167     // particle 'ptcl' flows from leg=0 to 1.
01168
01169 public:
01170
01171     ECand *cand;       // candidate
01172     int nodes[2];      // nodes of both sides of the edge
01173     int nlegs[2];      // nodes of both sides of the edge
01174     int ptcl;          // particle defined in the model
01175
01176     int DUMMYPADDING;
01177
01178     AEdge(int n0, int l0, int n1, int l1);
01179     ~AEdge(void);
01180 };
01181
01182 //=====
01183 class Assign {
01184     // Class for particle/interaction assignment.
01185
01186 public:
01187
01188     EGraph *egraph;    // EGraph object
01189     MGraph *mgraph;    // MGraph object
01190     SProcess *sproc;   // Sub-process object
01191     Process *proc;     // Process object
01192     Model *model;      // Model object
01193     Options *opt;      // Option object
01194     AStack *astack;    // Stack for saving candidates
01195     PNodeClass *pnclass; // class of nodes
01196     MOrbits *orbits;   // orbits of nodes by symmetry group
01197
01198     int nNodes;        // the number of nodes
01199     int nEdges;        // the number of edges
01200     int nExtern;       // the number of external particles.
01201     int nETotal;       // the total number of edges
01202
01203     BigInt nAGraphs;   // the number of assigned graphs
01204     Fraction wAGraphs; // the weighted sum of graphs
01205     BigInt nAOPI;     // the number of assigned 1PI
01206     Fraction wAOPI;    // the weighted sum of 1PI
01207
01208     ANode **nodes;     // table of nodes
01209     AEdge **edges;     // table of edges
01210
01211     CheckPt checkpoint0; // save stack pointers
01212
01213     int cplleft[GRCC_MAXNCPLG]; // coupling constants left
01214
01215     //=====
01216     Assign(SProcess *sprc, MGraph *mgr, PNodeClass *pnc);
01217     ~Assign(void);
01218

```



```

01219 //=====
01220 // print lists of candidates
01221 void prCand(const char *msg);
01222
01223 // check
01224 void checkAG(const char *msg);
01225
01226 //=====
01227 // control of assignment
01228
01229 // entry point of assignment
01230 Bool assignAllVertices(void);
01231
01232 // select a source node for assignment
01233 Bool selectVertex(void);
01234 Bool selectVertexSimp(int lastv);
01235
01236 // select a source leg of the node for assignment
01237 Bool selectLeg(int v, int lastlg);
01238
01239 // assign a vertex a interaction
01240 Bool assignVertex(int v);
01241
01242 // assignment procedure is finished. Needs check.
01243 Bool allAssigned(void);
01244
01245 //=====
01246 // Input and output
01247
01248 // Input from mgraph
01249 Bool fromMGraph(void);
01250
01251 // add an edge
01252 void addEdge(int n0, int n1, int nplist, int *plist);
01253
01254 // connect nodes and edges.
01255 void connect(int n0, int l0, int eg, int el, int n1, int l1);
01256
01257 // Output to egraph
01258 Bool fillEGraph(int aid, BigInt nsym, BigInt esym, BigInt nsym1);
01259
01260 // reorder legs in accordance with the interaction definition
01261 int *reordLeg(int n, int *reord, int *plist, int *used);
01262
01263 //=====
01264 // direction of particle on an edge and at (node, leg)
01265
01266 // convert particle code
01267 // at node-leg ('n', 'ln') <==> edge at ('n', 'ln')
01268 int getLegParticle(int n, int ln);
01269
01270 // convert particle 'pt' on the edge to ('n', 'ln')
01271 int legEdgeParticle(int n, int ln, int pt);
01272
01273 // convert candidate list at ('v', 'lg') into in-coming direction
01274 int legPart(int v, int lg, int nplst, int *plst, int *rlist, const int size);
01275
01276 // convert candidate list at ('v', 'lg') into in-coming direction
01277 int candPart(int v, int ln, int *plist, const int size);
01278
01279 //=====
01280 // assignment
01281
01282 // find a vertex to be assigned
01283 int selUnAssVertex(void);
01284 int selUnAssVertexSimp(int lastv);
01285
01286 // find a leg of the vertex to be assigned
01287 int selUnAssLeg(int v, int lastlg);
01288
01289 // assign the interaction 'ia' to the vertex 'v'.
01290 NCandSt assignIVertex(int v, int ia);
01291
01292 // assign particle 'pt' the the leg 'ln' of the node 'n'.
01293 Bool assignPLeg(int n, int ln, int pt);
01294 Bool isOrdPLeg(int n, int ln, int pt);
01295 Bool detEdge(int e);
01296
01297 //=====
01298 // candidates
01299
01300 // construct lists of assigned / unassigned particles at a node
01301 Bool candPartClassify(int v, int *npdass, int *pdass, int *npuass, int *puass, const int size);
01302
01303 // update candidate list for a vertex 'v'.
01304 Bool updateCandNode(int v);
01305

```

```

01306 //=====
01307 // filter
01308
01309 Bool    checkOrderCpl(void);
01310 Bool    isOrdLegs(void);
01311 Bool    isIsomorphic(MNodeClass *cl, BigInt *nsym, BigInt *esym, BigInt *nsym1);
01312 int     cmpPermGraph(int *p, MNodeClass *cl);
01313 int     cmpNodes(int nd0, int nd1, MNodeClass *cn);
01314 BigInt  edgeSym(void);
01315
01316 void     saveCouple(int *sav);
01317 void     restoreCouple(int *sav);
01318 Bool     subCouple(int *cpl);
01319
01320 #ifdef CHECK
01321 //=====
01322 // check
01323 Bool     checkCand(const char *msg);
01324
01325 void     checkNode(int n, const char *msg);
01326 #endif
01327 };
01328
01329 //*****
01330 // Stack of candidate lists
01331 //=====
01332 class NStack {
01333 // Stack for backtracking method for a node.
01334
01335 public:
01336     int     noden;
01337     int     deg;
01338     NCandSt st;
01339     int     nilist;
01340     int     ilist[GRCC_MAXMININTERACT];
01341
01342     NStack() { };
01343     ~NStack() { };
01344
01345     void print(const char *msg);
01346 };
01347
01348
01349 //=====
01350 class EStack {
01351 // Stack for backtracking method for an edge.
01352
01353 public:
01354     int     edgen;
01355     int     det;
01356     int     nplist;
01357     int     plist[GRCC_MAXMPARTICLES2];
01358
01359     EStack() { };
01360     ~EStack() { };
01361
01362     void print(const char *msg);
01363 };
01364
01365
01366 //=====
01367 class AStack {
01368 // Stack for backtracking method for an edge.
01369
01370 public:
01371     Assign   *agraph;
01372
01373     NStack   **nStack;    // stack for nodes
01374     int       nStackP;    // stack pointer for nodes
01375     int       nSize;      // stack size
01376
01377     EStack   **eStack;    // stack for edges
01378     int       eStackP;    // stack pointer for edges
01379     int       eSize;      // stack size
01380
01381     AStack(int nSize, int eSize);
01382     ~AStack(void);
01383
01384     void setAGraph(Assign *ag);
01385
01386     void checkPoint(CheckPt sav);
01387     void restore(CheckPt sav);
01388     void restoreMsg(CheckPt sav, const char *msg);
01389     void prStack(void);
01390
01391     void pushNode(int n);
01392     void pushEdge(int e);

```

```

01393
01394     private:
01395         void restoreNode(int spr);
01396         void restoreEdge(int spr);
01397 };
01398
01399 //=====
01400 // end of namespace Grcc
01401 #ifdef GRCC_NAMESPACE
01402 }
01403 #endif

```

4.59 grccparam.h

```

00001 #ifndef GRCC_PARAM_H
00002 #else
00003 #define GRCC_PARAM_H
00004
00005 /* # License : */
00006 /*
00007  * Copyright (C) 2023-2026 T. Kaneko
00008  * When using this file you are requested to refer to the publication
00009  * Comput.Phys.Commun. 92 (1995) 127-152
00010  *
00011  * This file is part of FORM.
00012  *
00013  * FORM is free software: you can redistribute it and/or modify it under the
00014  * terms of the GNU General Public License as published by the Free Software
00015  * Foundation, either version 3 of the License, or (at your option) any later
00016  * version.
00017  *
00018  * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00019  * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00020  * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00021  * details.
00022  *
00023  * You should have received a copy of the GNU General Public License along
00024  * with FORM. If not, see <http://www.gnu.org/licenses/>.
00025  */
00026 /* # License : */
00027
00028 /*=====
00029  * Parameters
00030  */
00031 /*
00032 #define GRCC_NAMESPACE
00033 */
00034
00035 #define GRCC_MAXNCPLG 4
00036 #define GRCC_MAXLEGS 10
00037 #define GRCC_MAXMPARTICLES 50
00038 #define GRCC_MAXMINTERACT 500
00039 #define GRCC_MAXSUBPROCS 500
00040 #define GRCC_MAXNODES 20
00041 #define GRCC_MAXEDGES 100
00042 #define GRCC_MAXNSTACK 50
00043 #define GRCC_MAXESTACK 50
00044 #define GRCC_MAXGROUP 1000
00045 #define GRCC_MAXPSLIST 500
00046
00047 #define GRCC_MAXMPARTICLES2 (2*GRCC_MAXMPARTICLES-1)
00048 #define GRCC_MAXFLINES GRCC_MAXEDGES
00049
00050 #define GRCC_FRACERROR (1.0e-10)
00051
00052 /* Standard and error message destination */
00053 #define GRCC_Stdout stdout
00054 #define GRCC_Stderr stdout
00055
00056 #ifndef NOFORM
00057     #define GRCC_ABORT() Terminate(-1);
00058 #else
00059     #define GRCC_ABORT() exit(1)
00060 #endif
00061
00062 /*-----
00063  * Constants
00064  */
00065
00066 /* model definition */
00067 #define GRCC_DEFBYNAME 1 /* define interactions by particle name */
00068 #define GRCC_DEFBYCODE 2 /* define interactions by particle code */
00069

```

```

00070 /* types of particles */
00071 #define GRCC_PT_Undef      0
00072 #define GRCC_PT_Scalar    1
00073 #define GRCC_PT_Dirac     2
00074 #define GRCC_PT_Majorana  3
00075 #define GRCC_PT_Vector    4
00076 #define GRCC_PT_Ghost     5
00077 #define GRCC_PT_Size      6
00078
00079 #define GRCC_PTS_Undef     "undef"
00080 #define GRCC_PTS_Scalar   "scalar"
00081 #define GRCC_PTS_Dirac    "dirac"
00082 #define GRCC_PTS_Majorana "majorana"
00083 #define GRCC_PTS_Vector   "vector"
00084 #define GRCC_PTS_Ghost    "ghost"
00085
00086 #define GRCC_PT_Table { GRCC_PTS_Undef, GRCC_PTS_Scalar, GRCC_PTS_Dirac, GRCC_PTS_Majorana,
GRCC_PTS_Vector, GRCC_PTS_Ghost}
00087
00088 #define GRCC_PT_GTable { "Undef", "Scalar", "Fermion", "Majorana", "Vector", "Ghost"}
00089
00090 /* for mgraph generation */
00091 #define GRCC_AT_Vertex      0
00092 #define GRCC_AT_External   -1
00093 #define GRCC_AT_Initial    -2
00094 #define GRCC_AT_Final      -3
00095
00096 #define GRCC_AT_NdStr(x)    ((x)>=GRCC_AT_Vertex ?"Vertex":      \
00097                             ((x)==GRCC_AT_External ?"External":  \
00098                             ((x)==GRCC_AT_Final ?"Final":        \
00099                             ((x)==GRCC_AT_Initial ?"Initial": "Undef"))))
00100
00101 /* graph generation */
00102 #define GRCC_MGraph      0      /* mgraph (topology) generation */
00103 #define GRCC_AGraph      1      /* agraph generation */
00104
00105 /* options for graph greeneration */
00106 #define GRCC_OPT_Step      0
00107 #define GRCC_OPT_Outgrf    1
00108 #define GRCC_OPT_Outgrp    2
00109 #define GRCC_OPT_lPI       3
00110 #define GRCC_OPT_NoSelfLoop 4
00111 #define GRCC_OPT_NoTadpole 5
00112 #define GRCC_OPT_NoIPtBlock 6
00113 #define GRCC_OPT_No2PtLlPI 7
00114 #define GRCC_OPT_NoExtSelf 8
00115 #define GRCC_OPT_NoAdj2PtV 9
00116 #define GRCC_OPT_Block     10
00117 #define GRCC_OPT_NoMultiEdge 11
00118 #define GRCC_OPT_SymmInitial 12
00119 #define GRCC_OPT_SymmFinal 13
00120 #define GRCC_OPT_Size      14
00121
00122 typedef unsigned long ULong;
00123
00124 typedef struct {
00125     const char *name;
00126     const char *mean;
00127     int         defaultv;
00128     int         DUMMYPADDING;
00129 } OptDef;
00130
00131 typedef struct {
00132     const char *name;
00133     const char *cname;
00134     const char *mean;
00135 } OptQGDef;
00136
00137 typedef struct {
00138     const char *name;
00139     int         index;
00140     int         sign;
00141 } OptQGRef;
00142
00143 /*-----
00144  * QGRAF options
00145  */
00146 #define GRCC_QGRAF_OPT_ONEPI      0
00147 #define GRCC_QGRAF_OPT_ONSHELL    1
00148 #define GRCC_QGRAF_OPT_NOSIGMA    2
00149 #define GRCC_QGRAF_OPT_NOSNAIL    3
00150 #define GRCC_QGRAF_OPT_NOTADPOLE  4
00151 #define GRCC_QGRAF_OPT_SIMPLE      5
00152 #define GRCC_QGRAF_OPT_BIPART      6
00153 #define GRCC_QGRAF_OPT_CYCLI       7
00154 #define GRCC_QGRAF_OPT_FLOOP       8
00155 //TODO what does this do? Does it work?

```

```

00156 //define GRCC_QGRAF_OPT_TOPOL          9
00157
00158 #ifdef GRCC_QGRAF_OPT_TOPOL
00159 #define GRCC_QGRAF_OPT_Size          10
00160 #else
00161 #define GRCC_QGRAF_OPT_Size          9
00162 #endif
00163
00164 /*-----
00165  * Conversion of
00166  *   eind : index (ed) of EGraph.edges[ed]
00167  *   sind : value (v) of EGraph.nodes[nd]->edges[j])
00168  */
00169 #define I2Vedge(eind, sign)  (((sign<=0)?(-1):(1))*((eind)+1))
00170 #define V2Iedge(sind)       (abs(sind)-1)
00171 #define V2Ileg(sind)        ((sind)<0 ? 0 : 1)
00172 /*-----
00173  * data types for model definition
00174  */
00175 typedef struct {
00176     const char *name;           /* name of the model */
00177     int defpart;                /* particle is defined by name or code */
00178                                /* GRCC_DEFBYNAME or GRCC_DEFBYCODE */
00179     int ncouple;                /* No. of coupling constants */
00180     const char *cnamlist[GRCC_MAXNCPLG]; /* Names of coupling constants */
00181 } MInput;
00182
00183 typedef struct {
00184     const char *name;           /* name of particle */
00185     const char *aname;          /* name of anti-particle */
00186     int pcode;                  /* code of particle */
00187     int acode;                  /* code of anti-particle */
00188     const char *ptypen;         /* type : 'GRCC_PTS_Scalar' etc */
00189     int ptypec;                 /* type : 'GRCC_PT_Scalar' etc */
00190     int extonly;                /* only as external particle */
00191 } PInput;
00192
00193 typedef struct {
00194     const char *name;           /* name of interaction */
00195     int icode;                  /* code of interaction */
00196     int nplistn;                /* the number of legs */
00197     const char *plistn[GRCC_MAXLEGS]; /* list of particle names */
00198     int plistc[GRCC_MAXLEGS]; /* list of particle codes */
00199     int cvallist[GRCC_MAXNCPLG]; /* coupling constants */
00200 } IInput;
00201
00202 /*-----
00203  * data type for node class input
00204  */
00205 typedef struct {
00206     /* used as input to MGraph() and SProcess() */
00207     int cldeg;
00208     int clnum;
00209     int cltyp;
00210     int cmind;
00211     int cmaxd;
00212
00213     /* used as input to SProcess() */
00214     int ptcl;
00215     int cple;
00216 } NCInput;
00217
00218 /*-----
00219  * data types for process definition
00220  */
00221 typedef struct {
00222     long ninitl;                /* the number of initial particles */
00223     const char *initln[GRCC_MAXNODES]; /* list of initial particles (name) */
00224     int initlc[GRCC_MAXNODES]; /* list of final particles (code) */
00225     long nfinal;                /* the number of final particles */
00226     const char *finaln[GRCC_MAXNODES]; /* list of final particles (name) */
00227     int finalc[GRCC_MAXNODES]; /* list of final particles (code) */
00228     int coupl[GRCC_MAXNCPLG]; /* list of orders of c. consts */
00229 } FGInput;
00230
00231 /* class of node : input for SProcess */
00232 typedef struct {
00233     int cdeg;                    /* degree of each node */
00234     int ctyp;                    /* type : GRCC_AT_xxx */
00235     int cnum;                    /* the number of nodes in the class */
00236     int cple;                    /* total order of c.c. of each node */
00237                                /* = 0 for external particle */
00238     const char *pname;           /* particle name defined in the model */
00239                                /* = NULL for vertex */
00240     long pcode;                 /* particle code defined in the model */
00241                                /* = 0 for vertex */
00242     /* long = int + padding */

```

```

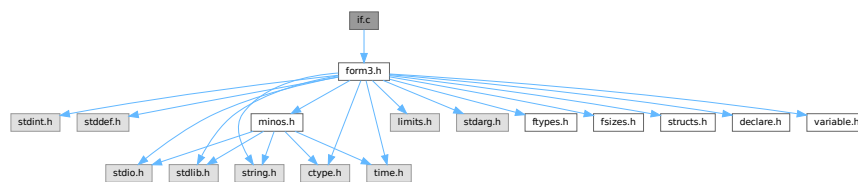
00243 } PGInput;
00244
00245 /*-----
00246  * minimum input data to EGraph
00247  */
00248 typedef struct {
00249     int     extloop;    /* external/loop */
00250     int     intrct;     /* particl/interaction */
00251 } DNode;
00252
00253 typedef int  DEdge[2]; /* {node0, node1} */
00254
00255 typedef struct {
00256     int     nnodes;
00257     int     nedges;
00258     DNode *nodes;
00259     DEdge *edges;
00260 } DGraph;
00261
00262 /*-----
00263  */
00264 #endif

```

4.60 if.c File Reference

#include "form3.h"

Include dependency graph for if.c:



Functions

- WORD [GetIfDollarNum](#) (WORD *ifp, WORD *ifstop)
- int [FindVar](#) (WORD *v, WORD *term)
- int [DolfStatement](#) (PHEAD WORD *ifcode, WORD *term)
- WORD [HowMany](#) (PHEAD WORD *ifcode, WORD *term)
- void [DoubleIfBuffers](#) (void)
- int [DoSwitch](#) (PHEAD WORD *term, WORD *lhs)
- int [DoEndSwitch](#) (PHEAD WORD *term, WORD *lhs)
- SWITCHTABLE * [FindCase](#) (WORD nswitch, WORD ncase)
- int [DoubleSwitchBuffers](#) (void)
- void [SwitchSplitMergeRec](#) (SWITCHTABLE *array, WORD num, SWITCHTABLE *auxarray)
- void [SwitchSplitMerge](#) (SWITCHTABLE *array, WORD num)

4.60.1 Detailed Description

Routines for the dealing with if statements.

Definition in file [if.c](#).

4.60.2 Function Documentation

4.60.2.1 GetIfDollarNum()

```
WORD GetIfDollarNum (
    WORD * ifp,
    WORD * ifstop )
```

Definition at line [106](#) of file [if.c](#).

4.60.2.2 FindVar()

```
int FindVar (
    WORD * v,
    WORD * term )
```

Definition at line [173](#) of file [if.c](#).

4.60.2.3 DolfStatement()

```
int DoIfStatement (
    PHEAD WORD * ifcode,
    WORD * term )
```

Definition at line [280](#) of file [if.c](#).

4.60.2.4 HowMany()

```
WORD HowMany (
    PHEAD WORD * ifcode,
    WORD * term )
```

Definition at line [840](#) of file [if.c](#).

4.60.2.5 DoubleIfBuffers()

```
void DoubleIfBuffers (
    void )
```

Definition at line [1033](#) of file [if.c](#).

4.60.2.6 DoSwitch()

```
int DoSwitch (
    PHEAD WORD * term,
    WORD * lhs )
```

Definition at line [1070](#) of file [if.c](#).

4.60.2.7 DoEndSwitch()

```
int DoEndSwitch (
    PHEAD WORD * term,
    WORD * lhs )
```

Definition at line 1086 of file [if.c](#).

4.60.2.8 FindCase()

```
SWITCHTABLE * FindCase (
    WORD nswitch,
    WORD ncase )
```

Definition at line 1097 of file [if.c](#).

4.60.2.9 DoubleSwitchBuffers()

```
int DoubleSwitchBuffers (
    void )
```

Definition at line 1135 of file [if.c](#).

4.60.2.10 SwitchSplitMergeRec()

```
void SwitchSplitMergeRec (
    SWITCHTABLE * array,
    WORD num,
    SWITCHTABLE * auxarray )
```

Definition at line 1184 of file [if.c](#).

4.60.2.11 SwitchSplitMerge()

```
void SwitchSplitMerge (
    SWITCHTABLE * array,
    WORD num )
```

Definition at line 1213 of file [if.c](#).

4.61 if.c

[Go to the documentation of this file.](#)

```

00001
00005 /* #[ License : */
00006 /*
00007 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008 *   When using this file you are requested to refer to the publication
00009 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010 *   This is considered a matter of courtesy as the development was paid
00011 *   for by FOM the Dutch physics granting agency and we would like to
00012 *   be able to track its scientific use to convince FOM of its value
00013 *   for the community.
00014 *
00015 *   This file is part of FORM.
00016 *
00017 *   FORM is free software: you can redistribute it and/or modify it under the
00018 *   terms of the GNU General Public License as published by the Free Software
00019 *   Foundation, either version 3 of the License, or (at your option) any later
00020 *   version.
00021 *
00022 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00025 *   details.
00026 *
00027 *   You should have received a copy of the GNU General Public License along
00028 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00029 */
00030 /* #[ License : */
00031 /*
00032     #[ Includes : if.c
00033 */
00034
00035 #include "form3.h"
00036
00037 /*
00038     #[ Includes :
00039     #[ If statement :
00040         #[ Syntax :
00041
00042         The `if' is a conglomerate of statements: if,else,endif
00043
00044         The if consists in principle of:
00045
00046         if ( number );
00047             statements
00048         else;
00049             statements
00050         endif;
00051
00052         The first set is taken when number != 0.
00053         The else is not mandatory.
00054         TRUE = 1 and FALSE = 0
00055
00056         The number can be built up via a logical expression:
00057
00058             expr1 condition expr2
00059
00060         each expression can be a subexpression again. It has to be
00061         enclosed in parentheses in that case.
00062         Conditions are:
00063             >, >=, <, <=, ==, !=, ||, &&
00064
00065         When Expressions are chained evaluation is from left to right,
00066         independent of whether this indicates nonsense.
00067         if ( a || b || c || d ); is a perfectly normal statement.
00068         if ( a >= b || c == d ); would be messed up. This should be:
00069         if ( ( a >= b ) || ( c == d ) );
00070
00071         The building blocks of the Expressions are:
00072
00073             Match(option,pattern)   The number of times pattern fits in term_
00074             Count(...)              The count value of term_
00075             Coeff[icient]           The coefficient of term_
00076             FindLoop(options)       Are there loops (as in ReplaceLoop).
00077
00078         Implementation for internal notation:
00079
00080         TYPEIF,length,gotolevel(if fail),EXPRTYPE,length,.....
00081
00082         EXPRTYPE can be:
00083             SHORTNUMBER             ->,4,sign,size
00084             LONGNUMBER              ->,{ncoef+2|,ncoef,numer,denom
00085             MATCH                   ->,patternsiz+3,keyword,pattern

```

```

00086      MULTIPLEOF      ->,3,thenumber
00087      COUNT           ->,countsize+2,countinfo
00088      TYPEFINDLOOP    ->,7 (findloop info)
00089      COEFFICIENT      ->,2
00090      IFDOLLAR         ->,3,dollarnumber
00091      SUBEXPR          ->,size,dummy,size1,EXPRTYPE,length,...
00092                      ,2,condition1,size2,...
00093                      This is like functions.
00094
00095      Note that there must be a restriction to the number of nestings
00096      of parentheses in an if statement. It has been set to 10.
00097
00098      The syntax of match corresponds to the syntax of the left side
00099      of an id statement. The only difference is the keyword
00100      MATCH vs TYPEIDNEW.
00101
00102      #] Syntax :
00103      #[ GetIfDollarNum :
00104  */
00105
00106  WORD GetIfDollarNum(WORD *ifp, WORD *ifstop)
00107  {
00108      DOLLARS d;
00109      WORD num, *w;
00110      if ( ifp[2] < 0 ) { return(-ifp[2]-1); }
00111      d = Dollars+ifp[2];
00112      if ( ifp+3 < ifstop && ifp[3] == IFDOLLAREXTRA ) {
00113          if ( d->nfactors == 0 ) {
00114              MLOCK(ErrorMessageLock);
00115              MesPrint("Attempt to use a factor of an unfactored $-variable");
00116              MUNLOCK(ErrorMessageLock);
00117              Terminate(-1);
00118          }
00119          num = GetIfDollarNum(ifp+3,ifstop);
00120          if ( num > d->nfactors ) {
00121              MLOCK(ErrorMessageLock);
00122              MesPrint("Dollar factor number %s out of range",num);
00123              MUNLOCK(ErrorMessageLock);
00124              Terminate(-1);
00125          }
00126          if ( num == 0 ) {
00127              return(d->nfactors);
00128          }
00129          w = d->factors[num-1].where;
00130          if ( w == 0 ) return(d->factors[num].value);
00131      getnumber:;
00132          if ( *w == 0 ) return(0);
00133          if ( *w == 4 && w[3] == 3 && w[2] == 1 && w[1] < MAXPOSITIVE && w[4] == 0 ) {
00134              return(w[1]);
00135          }
00136          if ( ( w[w[0]] != 0 ) || ( ABS(w[w[0]-1]) != w[0]-1 ) ) {
00137              MLOCK(ErrorMessageLock);
00138              MesPrint("Dollar factor number expected but found expression");
00139              MUNLOCK(ErrorMessageLock);
00140              Terminate(-1);
00141          }
00142          else {
00143              MLOCK(ErrorMessageLock);
00144              MesPrint("Dollar factor number out of range");
00145              MUNLOCK(ErrorMessageLock);
00146              Terminate(-1);
00147          }
00148          return(0);
00149      }
00150  /*
00151      Now we have just a dollar and should evaluate that into a short number
00152  */
00153      if ( d->type == DOLZERO ) {
00154          return(0);
00155      }
00156      else if ( d->type == DOLNUMBER || d->type == DOLTERMS ) {
00157          w = d->where; goto getnumber;
00158      }
00159      else {
00160          MLOCK(ErrorMessageLock);
00161          MesPrint("Dollar factor number is wrong type");
00162          MUNLOCK(ErrorMessageLock);
00163          Terminate(-1);
00164          return(0);
00165      }
00166  }
00167
00168  /*
00169      #] GetIfDollarNum :
00170      #[ FindVar :
00171  */
00172

```

```

00173 int FindVar(WORD *v, WORD *term)
00174 {
00175     WORD *t, *tstop, *m, *mstop, *f, *fstop, *a, *astop;
00176     GETSTOP(term,tstop);
00177     t = term+1;
00178     while ( t < tstop ) {
00179         if ( *v == *t && *v < FUNCTION ) { /* VECTOR, INDEX, SYMBOL, DOTPRODUCT */
00180             switch ( *v ) {
00181                 case SYMBOL:
00182                     m = t+2; mstop = t+t[1];
00183                     while ( m < mstop ) {
00184                         if ( *m == v[1] ) return(1);
00185                         m += 2;
00186                     }
00187                     break;
00188                 case INDEX:
00189                 case VECTOR:
00190 InVe:
00191                     m = t+2; mstop = t+t[1];
00192                     while ( m < mstop ) {
00193                         if ( *m == v[1] ) return(1);
00194                         m++;
00195                     }
00196                     break;
00197                 case DOTPRODUCT:
00198                     m = t+2; mstop = t+t[1];
00199                     while ( m < mstop ) {
00200                         if ( *m == v[1] && m[1] == v[2] ) return(1);
00201                         if ( *m == v[2] && m[1] == v[1] ) return(1);
00202                         m += 3;
00203                     }
00204                     break;
00205             }
00206         }
00207         else if ( *v == VECTOR && *t == INDEX ) goto InVe;
00208         else if ( *v == INDEX && *t == VECTOR ) goto InVe;
00209         else if ( ( *v == VECTOR || *v == INDEX ) && *t == DOTPRODUCT ) {
00210             m = t+2; mstop = t+t[1];
00211             while ( m < mstop ) {
00212                 if ( v[1] == m[0] || v[1] == m[1] ) return(1);
00213                 m += 3;
00214             }
00215         }
00216         else if ( *t >= FUNCTION ) {
00217             if ( *v == FUNCTION && v[1] == *t ) return(1);
00218             if ( functions[*t-FUNCTION].spec > 0 ) {
00219                 if ( *v == VECTOR || *v == INDEX ) { /* we need to check arguments */
00220                     int i;
00221                     for ( i = FUNHEAD; i < t[1]; i++ ) {
00222                         if ( v[1] == t[i] ) return(1);
00223                     }
00224                 }
00225             }
00226         }
00227         else {
00228             fstop = t + t[1]; f = t + FUNHEAD;
00229             while ( f < fstop ) { /* Do the arguments one by one */
00230                 if ( *f <= 0 ) {
00231                     switch ( *f ) {
00232                         case -SYMBOL:
00233                             if ( *v == SYMBOL && v[1] == f[1] ) return(1);
00234                             f += 2;
00235                             break;
00236                         case -VECTOR:
00237                         case -MINVECTOR:
00238                         case -INDEX:
00239                             if ( ( *v == VECTOR || *v == INDEX )
00240                                 && ( v[1] == f[1] ) ) return(1);
00241                             f += 2;
00242                             break;
00243                         case -SNUMBER:
00244                             f += 2;
00245                             break;
00246                         default:
00247                             if ( *v == FUNCTION && v[1] == -*f && *f <= -FUNCTION ) return(1);
00248                             if ( *f <= -FUNCTION ) f++;
00249                             else f += 2;
00250                             break;
00251                     }
00252                 }
00253                 else {
00254                     a = f + ARGHEAD; astop = f + *f;
00255                     while ( a < astop ) {
00256                         if ( FindVar(v,a) == 1 ) return(1);
00257                         a += *a;
00258                     }
00259                     f = astop;
00260                 }
00261             }
00262         }
00263     }
00264 }

```

```

00260     }
00261     }
00262     }
00263     t += t[1];
00264 }
00265 return(0);
00266 }
00267
00268 /*
00269     #] FindVar :
00270     #[ DoIfStatement :          WORD DoIfStatement(PHEAD ifcode,term)
00271
00272     The execution time part of the if-statement.
00273     The arguments are a pointer to the TYPEIF and a pointer to the term.
00274     The answer is either 1 (success) or 0 (fail).
00275     The calling routine can figure out where to go in case of failure
00276     by picking up gotolevel.
00277     Note that the whole setup asks for recursions.
00278 */
00279
00280 int DoIfStatement(PHEAD WORD *ifcode, WORD *term)
00281 {
00282     GETBIDENTITY
00283     WORD *ifstop, *ifp;
00284     UWORD *coef1 = 0, *coef2, *coef3, *cc;
00285     WORD ncoef1, ncoef2, ncoef3, i = 0, first, *r, acoef, ismull, ismul2, j;
00286     UWORD *Spac1, *Spac2;
00287     ifstop = ifcode + ifcode[1];
00288     ifp = ifcode + 3;
00289     if ( ifp >= ifstop ) return(1);
00290     if ( ( ifp + ifp[1] ) >= ifstop ) {
00291         switch ( *ifp ) {
00292             case LONGNUMBER:
00293                 if ( ifp[2] ) return(1);
00294                 else return(0);
00295             case MATCH:
00296             case TYPEIF:
00297                 if ( HowMany(BHEAD ifp,term) ) return(1);
00298                 else return(0);
00299             case TYPEFINDLOOP:
00300                 if ( Lus(term,ifp[3],ifp[4],ifp[5],ifp[6],ifp[2]) ) return(1);
00301                 else return(0);
00302             case TYPECOUNT:
00303                 if ( CountDo(term,ifp) ) return(1);
00304                 else return(0);
00305             case COEFFI:
00306             case MULTIPLEOF:
00307                 return(1);
00308             case IFDOLLAR:
00309                 {
00310                     DOLLARS d = Dollars + ifp[2];
00311 #ifdef WITHPTHREADS
00312                     int nummodopt, dtype = -1;
00313                     if ( AS.MultiThreaded ) {
00314                         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
00315                             if ( ifp[2] == ModOptdollars[nummodopt].number ) break;
00316                         }
00317                         if ( nummodopt < NumModOptdollars ) {
00318                             dtype = ModOptdollars[nummodopt].type;
00319                             if ( dtype == MODLOCAL ) {
00320                                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
00321                             }
00322                         }
00323                     }
00324                     dtype = d->type;
00325 #else
00326                     int dtype = d->type; /* We use dtype to make the operation atomic */
00327 #endif
00328                     if ( dtype == DOLZERO ) return(0);
00329                     if ( dtype == DOLUNDEFINED ) {
00330                         if ( AC.UnsureDollarMode == 0 ) {
00331                             MesPrint("$%s is undefined",AC.dollarnames->namebuffer+d->name);
00332                             Terminate(-1);
00333                         }
00334                     }
00335                 }
00336                 return(1);
00337             case IFEXPRESSION:
00338                 r = ifp+2; j = ifp[1] - 2;
00339                 while ( --j >= 0 ) {
00340                     if ( *r == AR.CurExpr ) return(1);
00341                     r++;
00342                 }
00343                 return(0);
00344             case IFISFACTORIZED:
00345                 r = ifp+2; j = ifp[1] - 2;
00346                 if ( j == 0 ) {

```

```

00347         if ( ( Expressions[AR.CurExpr].vflags & ISFACTORIZED ) != 0 )
00348             return(1);
00349         else
00350             return(0);
00351     }
00352     while ( --j >= 0 ) {
00353         if ( ( Expressions[*r].vflags & ISFACTORIZED ) == 0 ) return(0);
00354         r++;
00355     }
00356     return(1);
00357 case IFOCCURS:
00358     {
00359         WORD *OccStop = ifp + ifp[1];
00360         ifp += 2;
00361         while ( ifp < OccStop ) {
00362             if ( FindVar(ifp,term) == 1 ) return(1);
00363             if ( *ifp == DOTPRODUCT ) ifp += 3;
00364             else ifp += 2;
00365         }
00366     }
00367     return(0);
00368 case IFUSERFLAG:
00369     if ( ( Expressions[AR.CurExpr].uflags & (1 << ifp[2]) ) != 0 )
00370         return(1);
00371     return(0);
00372 default:
00373     /*
00374         Now we have a subexpression. Test first for one with a single item.
00375     */
00376     if ( ifp[3] == ( ifp[1] + 3 ) ) return(DoIfStatement(BHEAD ifp,term));
00377     ifstop = ifp + ifp[1];
00378     ifp += 3;
00379     break;
00380 }
00381 }
00382 /*
00383 Here is the composite condition.
00384 */
00385 coef3 = NumberMalloc("DoIfStatement");
00386 Spac1 = NumberMalloc("DoIfStatement");
00387 Spac2 = (UWORD *) (TermMalloc("DoIfStatement"));
00388 ncoef1 = 0; first = 1; ismull = 0;
00389 do {
00390     if ( !first ) {
00391         ifp += 2;
00392         if ( ifp[-2] == ORCOND && ncoef1 ) {
00393             coef1 = Spac1;
00394             ncoef1 = 1; coef1[0] = coef1[1] = 1;
00395             goto SkipCond;
00396         }
00397         if ( ifp[-2] == ANDCOND && !ncoef1 ) goto SkipCond;
00398     }
00399     coef2 = Spac2;
00400     ncoef2 = 1;
00401     ismul2 = 0;
00402     switch ( *ifp ) {
00403     case LONGNUMBER:
00404         ncoef2 = ifp[2];
00405         j = 2*(ABS(ncoef2));
00406         cc = (UWORD *) (ifp + 3);
00407         for ( i = 0; i < j; i++ ) coef2[i] = cc[i];
00408         break;
00409 #ifdef WITHFLOAT
00410     case IFFLOATNUMBER:
00411         /*
00412             The sloppy solution is: Convert to rational.
00413             This way we can write it over coef2,ncoef2
00414         */
00415         ncoef2 = FloatFunToRat(BHEAD coef2,ifp);
00416         break;
00417 #endif
00418     case MATCH:
00419     case TYPEIF:
00420         coef2[0] = HowMany(BHEAD ifp,term);
00421         coef2[1] = 1;
00422         if ( coef2[0] == 0 ) ncoef2 = 0;
00423         break;
00424     case TYPECOUNT:
00425         acoef = CountDo(term,ifp);
00426         coef2[0] = ABS(acoef);
00427         coef2[1] = 1;
00428         if ( acoef == 0 ) ncoef2 = 0;
00429         else if ( acoef < 0 ) ncoef2 = -1;
00430         break;
00431     case TYPEFINDLOOP:
00432         acoef = Lus(term,ifp[3],ifp[4],ifp[5],ifp[6],ifp[2]);
00433         coef2[0] = ABS(acoef);

```

```

00434         coef2[1] = 1;
00435         if ( acoef == 0 ) ncoef2 = 0;
00436         else if ( acoef < 0 ) ncoef2 = -1;
00437         break;
00438     case COEFFI:
00439         r = term + *term;
00440         ncoef2 = r[-1];
00441         i = ABS(ncoef2);
00442         cc = (UWORD *) (r - i);
00443         if ( ncoef2 < 0 ) ncoef2 = (ncoef2+1)»1;
00444         else ncoef2 = (ncoef2-1)»1;
00445         i--; for ( j = 0; j < i; j++ ) coef2[j] = cc[j];
00446         break;
00447     case SUBEXPR:
00448         ncoef2 = coef2[0] = DoIfStatement(BHEAD ifp,term);
00449         coef2[1] = 1;
00450         break;
00451     case MULTIPLEOF:
00452         ncoef2 = 1;
00453         coef2[0] = ifp[2];
00454         coef2[1] = 1;
00455         ismul2 = 1;
00456         break;
00457     case IFDOLLAREXTRA:
00458         break;
00459     case IFDOLLAR:
00460         {
00461         /*
00462             We need to abstract a long rational in coef2
00463             with length ncoef2. What if that cannot be done?
00464         */
00465             DOLLARS d = Dollars + ifp[2];
00466 #ifdef WITHPTHREADS
00467             int nummodopt, dtype = -1;
00468             if ( AS.MultiThreaded ) {
00469                 for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
00470                     if ( ifp[2] == ModOptdollars[nummodopt].number ) break;
00471                 }
00472                 if ( nummodopt < NumModOptdollars ) {
00473                     dtype = ModOptdollars[nummodopt].type;
00474                     if ( dtype == MODLOCAL ) {
00475                         d = ModOptdollars[nummodopt].dstruct+AT.identity;
00476                     }
00477                     else {
00478                         LOCK(d->pthreadlockread);
00479                     }
00480                 }
00481             }
00482 #endif
00483         /*
00484             We have to pick up the IFDOLLAREXTRA pieces for [1], [$y] etc.
00485         */
00486         if ( ifp+3 < ifstop && ifp[3] == IFDOLLAREXTRA ) {
00487             if ( d->nfactors == 0 ) {
00488                 MLOCK(ErrorMessageLock);
00489                 MesPrint("Attempt to use a factor of an unfactored $-variable");
00490                 MUNLOCK(ErrorMessageLock);
00491                 Terminate(-1);
00492             } {
00493                 WORD num = GetIfDollarNum(ifp+3,ifstop);
00494                 WORD *w;
00495                 while ( ifp+3 < ifstop && ifp[3] == IFDOLLAREXTRA ) ifp += 3;
00496                 if ( num > d->nfactors ) {
00497                     MLOCK(ErrorMessageLock);
00498                     MesPrint("Dollar factor number %s out of range",num);
00499                     MUNLOCK(ErrorMessageLock);
00500                     Terminate(-1);
00501                 }
00502                 if ( num == 0 ) {
00503                     ncoef2 = 1; coef2[0] = d->nfactors; coef2[1] = 1;
00504                     break;
00505                 }
00506                 w = d->factors[num-1].where;
00507                 if ( w == 0 ) {
00508                     if ( d->factors[num-1].value < 0 ) {
00509                         ncoef2 = -1; coef2[0] = -d->factors[num-1].value; coef2[1] = 1;
00510                     }
00511                     else {
00512                         ncoef2 = 1; coef2[0] = d->factors[num-1].value; coef2[1] = 1;
00513                     }
00514                     break;
00515                 }
00516                 if ( w[*w] == 0 ) {
00517                     r = w + *w - 1;
00518                     i = ABS(*r);
00519                     if ( i == ( *w-1 ) ) {
00520                         ncoef2 = (i-1)/2;

```

```

00521             if ( *r < 0 ) ncoef2 = -ncoef2;
00522             i--; cc = coef2; r = w + 1;
00523             while ( --i >= 0 ) *cc++ = (UWORD) (*r++);
00524             break;
00525         }
00526     }
00527     goto generic;
00528 }
00529 }
00530 else {
00531     switch ( d->type ) {
00532     case DOLUNDEFINED:
00533         if ( AC.UnsureDollarMode == 0 ) {
00534             #ifdef WITHPTHREADS
00535                 if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
00536             #endif
00537             MLOCK(ErrorMessageLock);
00538             MesPrint("%s is undefined", AC.dollarnames->namebuffer+d->name);
00539             MUNLOCK(ErrorMessageLock);
00540             Terminate(-1);
00541         }
00542         ncoef2 = 0; coef2[0] = 0; coef2[1] = 1;
00543         break;
00544     case DOLZERO:
00545         ncoef2 = coef2[0] = 0; coef2[1] = 1;
00546         break;
00547     case DOLSUBTERM:
00548         if ( d->where[0] != INDEX || d->where[1] != 3
00549             || d->where[2] < 0 || d->where[2] >= AM.OffsetIndex ) {
00550             if ( AC.UnsureDollarMode == 0 ) {
00551                 #ifdef WITHPTHREADS
00552                     if ( dtype > 0 && dtype != MODLOCAL ) {
00553                         UNLOCK(d->pthreadlockread); }
00554                 #endif
00555                 MLOCK(ErrorMessageLock);
00556                 MesPrint("%s is of wrong
type", AC.dollarnames->namebuffer+d->name);
00557                 MUNLOCK(ErrorMessageLock);
00558                 Terminate(-1);
00559             }
00560             ncoef2 = 0; coef2[0] = 0; coef2[1] = 1;
00561             break;
00562         }
00563         d->index = d->where[2];
00564         /* fall through */
00565     case DOLINDEX:
00566         if ( d->index == 0 ) {
00567             ncoef2 = coef2[0] = 0; coef2[1] = 1;
00568         }
00569         else if ( d->index > 0 && d->index < AM.OffsetIndex ) {
00570             ncoef2 = 1; coef2[0] = d->index; coef2[1] = 1;
00571         }
00572         else if ( AC.UnsureDollarMode == 0 ) {
00573             #ifdef WITHPTHREADS
00574                 if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
00575             #endif
00576             MLOCK(ErrorMessageLock);
00577             MesPrint("%s is of wrong type", AC.dollarnames->namebuffer+d->name);
00578             MUNLOCK(ErrorMessageLock);
00579             Terminate(-1);
00580         }
00581         ncoef2 = coef2[0] = 0; coef2[1] = 1;
00582         break;
00583     case DOLWILDARGS:
00584         if ( d->where[0] <= -FUNCTION ||
00585             ( d->where[0] < 0 && d->where[2] != 0 )
00586             || ( d->where[0] > 0 && d->where[d->where[0]] != 0 )
00587         ) {
00588             if ( AC.UnsureDollarMode == 0 ) {
00589                 #ifdef WITHPTHREADS
00590                     if ( dtype > 0 && dtype != MODLOCAL ) {
00591                         UNLOCK(d->pthreadlockread); }
00592                 #endif
00593                 MLOCK(ErrorMessageLock);
00594                 MesPrint("%s is of wrong
type", AC.dollarnames->namebuffer+d->name);
00595                 MUNLOCK(ErrorMessageLock);
00596                 Terminate(-1);
00597             }
00598             ncoef2 = coef2[0] = 0; coef2[1] = 1;
00599             break;
00600         }
00601         /* fall through */
00602     case DOLARGUMENT:
00603         if ( d->where[0] == -SNUMBER ) {
00604             if ( d->where[1] == 0 ) {
00605                 ncoef2 = coef2[0] = 0;

```

```

00604         }
00605         else if ( d->where[1] < 0 ) {
00606             ncoef2 = -1;
00607             coef2[0] = -d->where[1];
00608         }
00609         else {
00610             ncoef2 = 1;
00611             coef2[0] = d->where[1];
00612         }
00613         coef2[1] = 1;
00614     }
00615     else if ( d->where[0] == -INDEX
00616     && d->where[1] >= 0 && d->where[1] < AM.OffsetIndex ) {
00617         if ( d->where[1] == 0 ) {
00618             ncoef2 = coef2[0] = 0; coef2[1] = 1;
00619         }
00620         else {
00621             ncoef2 = 1; coef2[0] = d->where[1];
00622             coef2[1] = 1;
00623         }
00624     }
00625     else if ( d->where[0] > 0
00626     && d->where[ARGHEAD] == (d->where[0]-ARGHEAD)
00627     && ABS(d->where[d->where[0]-1]) ==
00628         (d->where[0] - ARGHEAD-1) ) {
00629         i = d->where[d->where[0]-1];
00630         ncoef2 = (ABS(i)-1)/2;
00631         if ( i < 0 ) { ncoef2 = -ncoef2; i = -i; }
00632         i--; cc = coef2; r = d->where + ARGHEAD+1;
00633         while ( --i >= 0 ) *cc++ = (UWORD)(*r++);
00634     }
00635     else {
00636         if ( AC.UnsureDollarMode == 0 ) {
00637             #ifdef WITHPTHREADS
00638                 if ( dtype > 0 && dtype != MODLOCAL ) {
00639                     UNLOCK(d->pthreadlockread); }
00640             #endif
00641             MLOCK(ErrorMessageLock);
00642             MesPrint("%s is of wrong
type",AC.dollarnames->namebuffer+d->name);
00643             MUNLOCK(ErrorMessageLock);
00644             Terminate(-1);
00645             ncoef2 = 0; coef2[0] = 0; coef2[1] = 1;
00646         }
00647         break;
00648     case DOLNUMBER:
00649     case DOLTERMS:
00650         if ( d->where[d->where[0]] == 0 ) {
00651             r = d->where + d->where[0]-1;
00652             i = ABS(*r);
00653             if ( i == ( d->where[0]-1 ) ) {
00654                 ncoef2 = (i-1)/2;
00655                 if ( *r < 0 ) ncoef2 = -ncoef2;
00656                 i--; cc = coef2; r = d->where + 1;
00657                 while ( --i >= 0 ) *cc++ = (UWORD)(*r++);
00658                 break;
00659             }
00660         }
00661         generic;;
00662         if ( AC.UnsureDollarMode == 0 ) {
00663             #ifdef WITHPTHREADS
00664                 if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
00665             #endif
00666             MLOCK(ErrorMessageLock);
00667             MesPrint("%s is of wrong type",AC.dollarnames->namebuffer+d->name);
00668             MUNLOCK(ErrorMessageLock);
00669             Terminate(-1);
00670             ncoef2 = 0; coef2[0] = 0; coef2[1] = 1;
00671             break;
00672         }
00673     }
00674     }
00675     #ifdef WITHPTHREADS
00676     if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
00677     #endif
00678     }
00679     break;
00680 case IFEXPRESSION:
00681     r = ifp+2; j = ifp[1] - 2; ncoef2 = 0;
00682     while ( --j >= 0 ) {
00683         if ( *r == AR.CurExpr ) { ncoef2 = 1; break; }
00684         r++;
00685     }
00686     coef2[0] = ncoef2;
00687     coef2[1] = 1;
00688     break;

```



```

00689         case IFISFACTORIZED:
00690             r = ifp+2; j = ifp[1] - 2;
00691             if ( j == 0 ) {
00692                 ncoef2 = 0;
00693                 if ( ( Expressions[AR.CurExpr].vflags & ISFACTORIZED ) != 0 ) {
00694                     ncoef2 = 1;
00695                 }
00696             }
00697             else {
00698                 ncoef2 = 1;
00699                 while ( --j >= 0 ) {
00700                     if ( ( Expressions[*r].vflags & ISFACTORIZED ) == 0 ) {
00701                         ncoef2 = 0;
00702                         break;
00703                     }
00704                     r++;
00705                 }
00706             }
00707             coef2[0] = ncoef2;
00708             coef2[1] = 1;
00709             break;
00710         case IFOCCURS:
00711             {
00712                 WORD *OccStop = ifp + ifp[1], *ifpp = ifp+2;
00713                 ncoef2 = 0;
00714                 while ( ifpp < OccStop ) {
00715                     if ( FindVar(ifpp,term) == 1 ) {
00716                         ncoef2 = 1; break;
00717                     }
00718                     if ( *ifpp == DOTPRODUCT ) ifp += 3;
00719                     else ifpp += 2;
00720                 }
00721                 coef2[0] = ncoef2;
00722                 coef2[1] = 1;
00723             }
00724             break;
00725         case IFUSERFLAG:
00726             {
00727                 ncoef2 = 0;
00728                 if ( ( Expressions[AR.CurExpr].uflags & (1 « ifp[2]) ) != 0 )
00729                     ncoef2 = 1;
00730                 coef2[0] = ncoef2;
00731                 coef2[1] = 1;
00732             }
00733             break;
00734         default:
00735             break;
00736     }
00737     if ( !first ) {
00738         if ( ifp[-2] != ORCOND && ifp[-2] != ANDCOND ) {
00739             if ( ( ifp[-2] == EQUAL || ifp[-2] == NOTEQUAL ) &&
00740                 ( ismul2 || ismul1 ) ) {
00741                 if ( ismul1 && ismul2 ) {
00742                     if ( coef1[0] == coef2[0] ) i = 1;
00743                     else i = 0;
00744                 }
00745                 else {
00746                     if ( ismul1 ) {
00747                         if ( ncoef2 )
00748                             Divvy(BHEAD coef2,&ncoef2,coef1,ncoef1);
00749                         cc = coef2; ncoef3 = ncoef2;
00750                     }
00751                     else {
00752                         if ( ncoef1 )
00753                             Divvy(BHEAD coef1,&ncoef1,coef2,ncoef2);
00754                         cc = coef1; ncoef3 = ncoef1;
00755                     }
00756                     if ( ncoef3 < 0 ) ncoef3 = -ncoef3;
00757                     if ( ncoef3 == 0 ) {
00758                         if ( ifp[-2] == EQUAL ) i = 1;
00759                         else i = 0;
00760                     }
00761                     else if ( cc[ncoef3] != 1 ) {
00762                         if ( ifp[-2] == EQUAL ) i = 0;
00763                         else i = 1;
00764                     }
00765                     else {
00766                         for ( j = 1; j < ncoef3; j++ ) {
00767                             if ( cc[ncoef3+j] != 0 ) break;
00768                         }
00769                         if ( j < ncoef3 ) {
00770                             if ( ifp[-2] == EQUAL ) i = 0;
00771                             else i = 1;
00772                         }
00773                         else if ( ifp[-2] == EQUAL ) i = 1;
00774                         else i = 0;
00775                     }
00776                 }
00777             }

```

```

00776         }
00777         goto donemul;
00778     }
00779     else if ( AddRat (BHEAD coef1,ncoef1,coef2,-ncoef2,coef3,&ncoef3) ) {
00780         NumberFree(coef3,"DoIfStatement"); NumberFree(Spac1,"DoIfStatement");
TermFree(Spac2,"DoIfStatement");
00781         MesCall("DoIfStatement"); return(-1);
00782     }
00783     switch ( ifp[-2] ) {
00784     case GREATER:
00785         if ( ncoef3 > 0 ) i = 1;
00786         else i = 0;
00787         break;
00788     case GREATEREQUAL:
00789         if ( ncoef3 >= 0 ) i = 1;
00790         else i = 0;
00791         break;
00792     case LESS:
00793         if ( ncoef3 < 0 ) i = 1;
00794         else i = 0;
00795         break;
00796     case LESSEQUAL:
00797         if ( ncoef3 <= 0 ) i = 1;
00798         else i = 0;
00799         break;
00800     case EQUAL:
00801         if ( ncoef3 == 0 ) i = 1;
00802         else i = 0;
00803         break;
00804     case NOTEQUAL:
00805         if ( ncoef3 != 0 ) i = 1;
00806         else i = 0;
00807         break;
00808     }
00809 donemul:    if ( i ) { ncoef2 = 1; coef2 = Spac2; coef2[0] = coef2[1] = 1; }
00810             else ncoef2 = 0;
00811             ismul1 = ismul2 = 0;
00812         }
00813     }
00814     else {
00815         first = 0;
00816     }
00817     coef1 = Spac1;
00818     i = 2*ABS(ncoef2);
00819     for ( j = 0; j < i; j++ ) coef1[j] = coef2[j];
00820     ncoef1 = ncoef2;
00821 SkipCond:
00822     ifp += ifp[1];
00823     } while ( ifp < ifstop );
00824
00825     NumberFree(coef3,"DoIfStatement"); NumberFree(Spac1,"DoIfStatement");
TermFree(Spac2,"DoIfStatement");
00826     if ( ncoef1 ) return(1);
00827     else return(0);
00828 }
00829
00830 /*
00831     #] DoIfStatement :
00832     #[ HowMany :                WORD HowMany(ifcode,term)
00833
00834     Returns the number of times that the pattern in ifcode
00835     can be taken out from term. There is a subkey in ifcode[2];
00836     The notation is identical to the lhs of an id statement.
00837     Most of the code comes from TestMatch.
00838 */
00839
00840 WORD HowMany(PHEAD WORD *ifcode, WORD *term)
00841 {
00842     GETBIDENTITY
00843     WORD *m, *t, *r, *w, power, RetVal, i, topje, *newterm;
00844     WORD *OldWork, *ww, *mm;
00845     int *RepSto, RepVal;
00846     int numdollars = 0;
00847     m = ifcode + IDHEAD;
00848     AN.FullProto = m;
00849     AN.WildValue = w = m + SUBEXPSIZE;
00850     m += m[1];
00851     AN.WildStop = m;
00852     OldWork = AT.WorkPointer;
00853     if ( ( ifcode[4] & 1 ) != 0 ) { /* We have at least one dollar in the pattern */
00854         AR.Eside = LHSIDEX;
00855         ww = AT.WorkPointer; i = m[0]; mm = m;
00856         NCOPY(ww,mm,i);
00857         *OldWork += 3;
00858         *ww++ = 1; *ww++ = 1; *ww++ = 3;
00859         AT.WorkPointer = ww;
00860         RepSto = AN.RepPoint;

```

```

00861     RepVal = *RepSto;
00862     NewSort(BHEAD0);
00863     if ( Generator(BHEAD OldWork,AR.Cnumlhs) ) {
00864         LowerSortLevel();
00865         *RepSto = RepVal;
00866         AN.RepPoint = RepSto;
00867         AT.WorkPointer = OldWork;
00868         return(-1);
00869     }
00870     AT.WorkPointer = ww;
00871     if ( EndSort(BHEAD ww,0) < 0 ) {}
00872     *RepSto = RepVal;
00873     AN.RepPoint = RepSto;
00874     if ( *ww == 0 || *(ww+ww) != 0 ) {
00875         if ( AP.lhdollarerror == 0 ) {
00876             MLOCK(ErrorMessageLock);
00877             MesPrint("&LHS must be one term");
00878             MUNLOCK(ErrorMessageLock);
00879             AP.lhdollarerror = 1;
00880         }
00881         AT.WorkPointer = OldWork;
00882         return(-1);
00883     }
00884     m = ww; AT.WorkPointer = ww = m + *m;
00885     if ( m[*m-1] < 0 ) { m[*m-1] = -m[*m-1]; }
00886     *m -= m[*m-1];
00887     AR.Eside = RHSIDE;
00888 }
00889 else {
00890     ww = term + *term;
00891     if ( AT.WorkPointer < ww ) AT.WorkPointer = ww;
00892 }
00893 ClearWild(BHEAD0);
00894 while ( w < AN.WildStop ) {
00895     if ( *w == LOADDOLLAR ) numdollars++;
00896     w += w[1];
00897 }
00898 AN.RepFunNum = 0;
00899 AN.RepFunList = AT.WorkPointer;
00900 AT.WorkPointer = (WORD *)((UBYTE *) (AT.WorkPointer)) + AM.MaxTer);
00901 topje = cbuf[AT.ebufnum].numrhs;
00902 if ( AT.WorkPointer >= AT.WorkTop ) {
00903     MLOCK(ErrorMessageLock);
00904     MesWork();
00905     MUNLOCK(ErrorMessageLock);
00906     return(-1);
00907 }
00908 AN.DisOrderFlag = ifcode[2] & SUBDISORDER;
00909 switch ( ifcode[2] & (~SUBDISORDER) ) {
00910     case SUBONLY :
00911         /* Must be an exact match */
00912         AN.UseFindOnly = 1; AN.ForFindOnly = 0;
00913     /*
00914     Copy the term first to scratchterm. This is needed
00915     because of the Substitute.
00916     */
00917     i = *term;
00918     t = term; newterm = r = AT.WorkPointer;
00919     NCOPY(r,t,i); AT.WorkPointer = r;
00920     RetVal = 0;
00921     if ( FindRest(BHEAD newterm,m) && ( AN.UsedOtherFind ||
00922         FindOnly(BHEAD newterm,m) ) ) {
00923         Substitute(BHEAD newterm,m,1);
00924         if ( numdollars ) {
00925             WildDollars(BHEAD (WORD *)0);
00926             numdollars = 0;
00927         }
00928         ClearWild(BHEAD0);
00929         RetVal = 1;
00930     }
00931     else RetVal = 0;
00932     break;
00933     case SUBMANY :
00934     /*
00935     Copy the term first to scratchterm. This is needed
00936     because of the Substitute.
00937     */
00938     i = *term;
00939     t = term; newterm = r = AT.WorkPointer;
00940     NCOPY(r,t,i); AT.WorkPointer = r;
00941     RetVal = 0;
00942     AN.UseFindOnly = 0;
00943     if ( ( power = FindRest(BHEAD newterm,m) ) > 0 ) {
00944         if ( ( power = FindOnce(BHEAD newterm,m) ) > 0 ) {
00945             AN.UseFindOnly = 0;
00946             do {
00947                 Substitute(BHEAD newterm,m,1);

```

```

00948             if ( numdollars ) {
00949                 WildDollars(BHEAD (WORD *)0);
00950                 numdollars = 0;
00951             }
00952             ClearWild(BHEAD0);
00953             RetVal++;
00954         } while ( FindRest(BHEAD newterm,m) && (
00955             AN.UsedOtherFind || FindOnce(BHEAD newterm,m) ) );
00956     }
00957     else if ( power < 0 ) {
00958         do {
00959             Substitute(BHEAD newterm,m,1);
00960             if ( numdollars ) {
00961                 WildDollars(BHEAD (WORD *)0);
00962                 numdollars = 0;
00963             }
00964             ClearWild(BHEAD0);
00965             RetVal++;
00966         } while ( FindRest(BHEAD newterm,m) );
00967     }
00968 }
00969 else if ( power < 0 ) {
00970     if ( FindOnce(BHEAD newterm,m) ) {
00971         do {
00972             Substitute(BHEAD newterm,m,1);
00973             if ( numdollars ) {
00974                 WildDollars(BHEAD (WORD *)0);
00975                 numdollars = 0;
00976             }
00977             ClearWild(BHEAD0);
00978         } while ( FindOnce(BHEAD newterm,m) );
00979         RetVal = 1;
00980     }
00981 }
00982 break;
00983 case SUBONCE :
00984 /*
00985     Copy the term first to scratchterm. This is needed
00986     because of the Substitute.
00987 */
00988     i = *term;
00989     t = term; newterm = r = AT.WorkPointer;
00990     NCOPY(r,t,i); AT.WorkPointer = r;
00991     RetVal = 0;
00992     AN.UseFindOnly = 0;
00993     if ( FindRest(BHEAD newterm,m) && ( AN.UsedOtherFind || FindOnce(BHEAD newterm,m) ) ) {
00994         Substitute(BHEAD newterm,m,1);
00995         if ( numdollars ) {
00996             WildDollars(BHEAD (WORD *)0);
00997             numdollars = 0;
00998         }
00999         ClearWild(BHEAD0);
01000         RetVal = 1;
01001     }
01002     else RetVal = 0;
01003     break;
01004 case SUBMULTI :
01005     RetVal = FindMulti(BHEAD term,m);
01006     break;
01007 case SUBVECTOR :
01008     RetVal = 0;
01009     for ( i = 0; i < *term; i++ ) ww[i] = term[i];
01010     while ( ( power = FindAll(BHEAD ww,m,AR.Cnumlhs,ifcode) ) != 0 ) { RetVal += power; }
01011     break;
01012 case SUBSELECT :
01013     ifcode += IDHEAD; ifcode += ifcode[1]; ifcode += *ifcode;
01014     AN.UseFindOnly = 1; AN.ForFindOnly = ifcode;
01015     if ( FindRest(BHEAD term,m) && ( AN.UsedOtherFind ||
01016         FindOnly(BHEAD term,m) ) ) RetVal = 1;
01017     else RetVal = 0;
01018     break;
01019 default :
01020     RetVal = 0;
01021     break;
01022 }
01023 AT.WorkPointer = AN.RepFunList;
01024 cbuf[AT.ebufnum].numrhs = topje;
01025 return(RetVal);
01026 }
01027
01028 /*
01029     #] HowMany :
01030     #[ DoubleIfBuffers :
01031 */
01032
01033 void DoubleIfBuffers(void)
01034 {

```

```

01035     int newmax, i;
01036     WORD *newsumcheck;
01037     LONG *newheap, *newifcount;
01038     if ( AC.MaxIf == 0 ) newmax = 10;
01039     else newmax = 2*AC.MaxIf;
01040     newheap = (LONG *)Malloc1(sizeof(LONG)*(newmax+1),"IfHeap");
01041     newsumcheck = (WORD *)Malloc1(sizeof(WORD)*(newmax+1),"IfSumCheck");
01042     newifcount = (LONG *)Malloc1(sizeof(LONG)*(newmax+1),"IfCount");
01043     if ( AC.MaxIf ) {
01044         for ( i = 0; i < AC.MaxIf; i++ ) {
01045             newheap[i] = AC.IfHeap[i];
01046             newsumcheck[i] = AC.IfSumCheck[i];
01047             newifcount[i] = AC.IfCount[i];
01048         }
01049         AC.IfStack = (AC.IfStack-AC.IfHeap) + newheap;
01050         M_free(AC.IfHeap,"AC.IfHeap");
01051         M_free(AC.IfCount,"AC.IfCount");
01052         M_free(AC.IfSumCheck,"AC.IfSumCheck");
01053     }
01054     else {
01055         AC.IfStack = newheap;
01056     }
01057     AC.IfHeap = newheap;
01058     AC.IfSumCheck = newsumcheck;
01059     AC.IfCount = newifcount;
01060     AC.MaxIf = newmax;
01061 }
01062
01063 /*
01064     #] DoubleIfBuffers :
01065     #] If statement :
01066     #[ Switch statement :
01067     #[ DoSwitch :
01068 */
01069 int DoSwitch(PHEAD WORD *term, WORD *lhs)
01070 {
01071     /*
01072     For the moment we ignore the compiler buffer problems.
01073     */
01074     WORD numdollar = lhs[2];
01075     WORD ncase = DolToNumber(BHEAD numdollar);
01076     SWITCHTABLE *swtab = FindCase(lhs[3],ncase);
01077     return(Generator(BHEAD term,swtab->value));
01078 }
01079
01080 /*
01081     #] DoSwitch :
01082     #[ DoEndSwitch :
01083 */
01084 int DoEndSwitch(PHEAD WORD *term, WORD *lhs)
01085 {
01086     SWITCH *sw = AC.SwitchArray+lhs[2];
01087     return(Generator(BHEAD term,sw->endswitch.value+1));
01088 }
01089
01090 /*
01091     #] DoEndSwitch :
01092     #[ FindCase :
01093 */
01094 SWITCHTABLE *FindCase(WORD nswitch, WORD ncase)
01095 {
01096     /*
01097     First find the switch table and determine how we have to search.
01098     */
01099     SWITCH *sw = AC.SwitchArray+nswitch;
01100     WORD hi, lo, med;
01101     if ( sw->typetable == DENSETABLE ) {
01102         med = ncase - sw->caseoffset;
01103         if ( med >= sw->numcases || med < 0 ) return(&sw->defaultcase);
01104     }
01105     else {
01106         /*
01107         We need a binary search in the table.
01108         */
01109         if ( ncase > sw->maxcase || ncase < sw->mincase ) return(&sw->defaultcase);
01110         hi = sw->numcases-1; lo = 0;
01111         for(;;) {
01112             med = (hi+lo)/2;
01113             if ( ncase == sw->table[med].ncase ) break;
01114             else if ( ncase > sw->table[med].ncase ) {
01115                 lo = med+1;
01116             }
01117             else if ( lo > hi ) return(&sw->defaultcase);
01118         }
01119         else {

```

```

01122             hi = med-1;
01123             if ( hi < lo ) return(&sw->defaultcase);
01124         }
01125     }
01126 }
01127 return(&sw->table[med]);
01128 }
01129
01130 /*
01131     #] FindCase :
01132     #[ DoubleSwitchBuffers :
01133 */
01134
01135 int DoubleSwitchBuffers(void)
01136 {
01137     int newmax, i;
01138     SWITCH *newarray;
01139     WORD *newheap;
01140     if ( AC.MaxSwitch == 0 ) newmax = 10;
01141     else newmax = 2*AC.MaxSwitch;
01142     newarray = (SWITCH *)Malloc1(sizeof(SWITCH)*(newmax+1),"SwitchArray");
01143     newheap = (WORD *)Malloc1(sizeof(WORD)*(newmax+1),"SwitchHeap");
01144     if ( AC.MaxSwitch ) {
01145         for ( i = 0; i < AC.MaxSwitch; i++ ) {
01146             newarray[i] = AC.SwitchArray[i];
01147             newheap[i] = AC.SwitchHeap[i];
01148         }
01149         M_free(AC.SwitchHeap,"AC.SwitchHeap");
01150         M_free(AC.SwitchArray,"AC.SwitchArray");
01151     }
01152     for ( i = AC.MaxSwitch; i <= newmax; i++ ) {
01153         newarray[i].table = 0;
01154         newarray[i].tablesize = 0;
01155         newarray[i].defaultcase.ncase = 0;
01156         newarray[i].defaultcase.value = 0;
01157         newarray[i].defaultcase.compbuffer = 0;
01158         newarray[i].endswitch.ncase = 0;
01159         newarray[i].endswitch.value = 0;
01160         newarray[i].endswitch.compbuffer = 0;
01161         newarray[i].typetable = 0;
01162         newarray[i].mincase = 0;
01163         newarray[i].maxcase = 0;
01164         newarray[i].numcases = 0;
01165         newarray[i].caseoffset = 0;
01166         newarray[i].iflevel = 0;
01167         newarray[i].whilelevel = 0;
01168         newarray[i].nestingsum = 0;
01169         newheap[i] = 0;
01170     }
01171     AC.SwitchArray = newarray;
01172     AC.SwitchHeap = newheap;
01173     AC.MaxSwitch = newmax;
01174     return(0);
01175 }
01176
01177 /*
01178     #] DoubleSwitchBuffers :
01179     #[ SwitchSplitMerge :
01180
01181     Sorts an array of WORDs. No adding of equal objects.
01182 */
01183
01184 void SwitchSplitMergeRec(SWITCHTABLE *array,WORD num,SWITCHTABLE *auxarray)
01185 {
01186     WORD n1,n2,i,j,k;
01187     SWITCHTABLE *t1,*t2, t;
01188     if ( num < 2 ) return;
01189     if ( num == 2 ) {
01190         if ( array[0].ncase > array[1].ncase ) {
01191             t = array[0]; array[0] = array[1]; array[1] = t;
01192         }
01193         return;
01194     }
01195     n1 = num/2;
01196     n2 = num - n1;
01197     SwitchSplitMergeRec(array,n1,auxarray);
01198     SwitchSplitMergeRec(array+n1,n2,auxarray);
01199     if ( array[n1-1].ncase <= array[n1].ncase ) return;
01200
01201     t1 = array; t2 = auxarray; i = n1; NCOPY(t2,t1,i);
01202     i = 0; j = n1; k = 0;
01203     while ( i < n1 && j < num ) {
01204         if ( auxarray[i].ncase <= array[j].ncase ) { array[k++] = auxarray[i++]; }
01205         else { array[k++] = array[j++]; }
01206     }
01207     while ( i < n1 ) array[k++] = auxarray[i++];
01208 /*

```

```

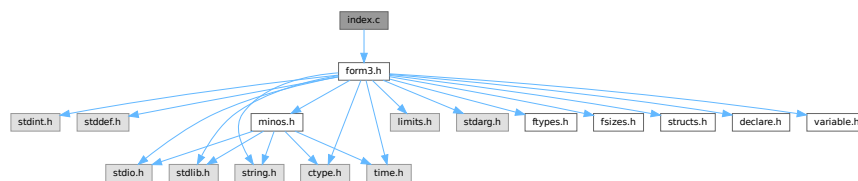
01209     Remember: remnants of j are still in place!
01210 */
01211 }
01212
01213 void SwitchSplitMerge(SWITCHTABLE *array,WORD num)
01214 {
01215     SWITCHTABLE *auxarray = (SWITCHTABLE *)Malloc1(sizeof(SWITCHTABLE)*num/2,"SwitchSplitMerge");
01216     SwitchSplitMergeRec(array,num,auxarray);
01217     M_free(auxarray,"SwitchSplitMerge");
01218 }
01219
01220 /*
01221     #] SwitchSplitMerge :
01222     #] Switch statement :
01223 */

```

4.62 index.c File Reference

```
#include "form3.h"
```

Include dependency graph for index.c:



Functions

- [POSITION](#) * [FindBracket](#) (WORD nexp, WORD *bracket)
- void [PutBracketInIndex](#) (PHEAD WORD *term, [POSITION](#) *newpos)
- void [ClearBracketIndex](#) (WORD numexp)
- void [OpenBracketIndex](#) (WORD nexpr)
- int [PutInside](#) (PHEAD WORD *term, WORD *code)

4.62.1 Detailed Description

The routines that deal with bracket indexing It creates the bracket index and it can find the brackets using the index.

Definition in file [index.c](#).

4.62.2 Function Documentation

4.62.2.1 FindBracket()

```

POSITION * FindBracket (
    WORD nexp,
    WORD * bracket )

```

Definition at line 65 of file [index.c](#).

4.62.2.2 PutBracketInIndex()

```
void PutBracketInIndex (
    PHEAD WORD * term,
    POSITION * newpos )
```

Definition at line 331 of file [index.c](#).

4.62.2.3 ClearBracketIndex()

```
void ClearBracketIndex (
    WORD numexp )
```

Definition at line 546 of file [index.c](#).

4.62.2.4 OpenBracketIndex()

```
void OpenBracketIndex (
    WORD nexpr )
```

Definition at line 578 of file [index.c](#).

4.62.2.5 PutInside()

```
int PutInside (
    PHEAD WORD * term,
    WORD * code )
```

Definition at line 609 of file [index.c](#).

4.63 index.c

[Go to the documentation of this file.](#)

```
00001
00008 /* # [ License : */
00009 /*
00010 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00011 * When using this file you are requested to refer to the publication
00012 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013 * This is considered a matter of courtesy as the development was paid
00014 * for by FOM the Dutch physics granting agency and we would like to
00015 * be able to track its scientific use to convince FOM of its value
00016 * for the community.
00017 *
00018 * This file is part of FORM.
00019 *
00020 * FORM is free software: you can redistribute it and/or modify it under the
00021 * terms of the GNU General Public License as published by the Free Software
00022 * Foundation, either version 3 of the License, or (at your option) any later
00023 * version.
00024 *
00025 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00028 * details.
00029 *
00030 * You should have received a copy of the GNU General Public License along
00031 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00032 */
```



```

00033 /* #] License : */
00034
00035 /*
00036  #[ Includes : index.c
00037 */
00038
00039 #include "form3.h"
00040
00041 /*
00042  #] Includes :
00043  #[ syntax and use :
00044
00045  The indexing of brackets is not automatic! It should only be used
00046  when one intends to use the contents of individual brackets.
00047  This is done with the addition of a + sign to the bracket statement:
00048  B+ a,b,c; or AB+ a,b,c;
00049  It does require resources! The index is kept in memory and is removed
00050  once the expression is treated and passed on to the output with different
00051  or no brackets.
00052  The index is limited to a given amount of space. Hence if there are too
00053  many brackets we will skip some in the index. Skipping goes by space
00054  occupied by the contents. We take the two adjacent bracket(s) with the
00055  least space together and represent them by the first one only. This gives
00056  a new spot.
00057  The expression struct has two pointers:
00058  bracketinfo      for using.
00059  newbracketinfo   for making new index.
00060
00061  #] syntax and use :
00062  #[ FindBracket :
00063 */
00064
00065 POSITION *FindBracket(WORD nexp, WORD *bracket)
00066 {
00067     GETIDENTITY
00068     BRACKETINDEX *bi;
00069     BRACKETINFO *bracketinfo;
00070     EXPRESSIONS e = &(Expressions[nexp]);
00071     LONG hi, low, med;
00072     int i;
00073     WORD oldsorttype = AR.SortType, *t1, *t2, j, bsize, *term, *p, *pstop, *pp;
00074     WORD *tstop, *cp, a[4], *bracketh;
00075     FILEHANDLE *fi;
00076     POSITION auxpos, toppos;
00077     switch ( e->status ) {
00078         case UNHIDELEXPRESSION:
00079         case UNHIDEGEXPRESSION:
00080         case DROPHLEXPRESSION:
00081         case DROPHGEXPRESSION:
00082         case HIDDENLEXPRESSION:
00083         case HIDDENGEXPRESSION:
00084             fi = AR.hidefile;
00085             break;
00086         default:
00087             fi = AR.infile;
00088             break;
00089     }
00090     if ( AT.bracketinfo ) bracketinfo = AT.bracketinfo;
00091     else bracketinfo = e->bracketinfo;
00092     hi = bracketinfo->indexfill; low = 0;
00093     if ( hi <= 0 ) return(0);
00094 /*
00095  The next code is needed for a problem with sorting when there are
00096  only functions outside the bracket. This gives ordinarily the wrong
00097  sorting. We solve that by taking HAAKJE along with the outside while
00098  running the Compare. But this means that we need to copy bracket.
00099 */
00100     bracketh = TermMalloc("FindBracket");
00101     i = *bracket; p = bracket; pp = bracketh; NCOPY(pp,p,i)
00102     pp -= 3; *pp++ = HAAKJE; *pp++ = 3; *pp++ = 0; *pp++ = 1; *pp++ = 1; *pp++ = 3;
00103     *bracketh += 3;
00104
00105     AT.fromindex = 1;
00106     AR.SortType = bracketinfo->SortType;
00107     bi = bracketinfo->indexbuffer + hi - 1;
00108     if ( *bracketh == 7 ) {
00109         if ( bracketinfo->bracketbuffer[bi->bracket] == 7 ) i = 0;
00110         else i = -1;
00111     }
00112     else if ( bracketinfo->bracketbuffer[bi->bracket] == 7 ) i = 1;
00113     else i = CompareTerms(BHEAD bracketh,bracketinfo->bracketbuffer+bi->bracket,0);
00114     if ( i < 0 ) {
00115         AR.SortType = oldsorttype;
00116         AT.fromindex = 0;
00117         TermFree(bracketh,"FindBracket");
00118         return(0);
00119     }

```

```

00120     else if ( i == 0 ) med = hi-1;
00121     else {
00122         for (;;) {
00123             med = (hi+low)/2;
00124             bi = bracketinfo->indexbuffer + med;
00125             if ( *bracketh == 7 ) {
00126                 if ( bracketinfo->bracketbuffer[bi->bracket] == 7 ) i = 0;
00127                 else i = -1;
00128             }
00129             else if ( bracketinfo->bracketbuffer[bi->bracket] == 7 ) i = 1;
00130             else i = CompareTerms(BHEAD bracketh,bracketinfo->bracketbuffer+bi->bracket,0);
00131             if ( i == 0 ) { break; }
00132             if ( i > 0 ) {
00133                 if ( low == med ) { /* no occurrence */
00134                     AR.SortType = oldsorttype;
00135                     AT.fromindex = 0;
00136                     TermFree(bracketh,"FindBracket");
00137                     return(0);
00138                 }
00139                 hi = med;
00140             }
00141             else if ( i < 0 ) {
00142                 if ( low == med ) break;
00143                 low = med;
00144             }
00145         }}
00146     /*
00147     The bracket is now either bi itself or between bi and the next one
00148     or it is not present at all.
00149     */
00150     AN.theposition = AS.OldOnFile[nexp];
00151     ADD2POS(AN.theposition,bi->start);
00152     /*
00153     The seek will have to move closer to the actual read so that we
00154     can place a lock around the two.
00155     if ( fi->handle >= 0 ) SeekFile(fi->handle,&AN.theposition,SEEK_SET);
00156     else SetScratch(fi,&AN.theposition);
00157     */
00158     /*
00159     Put the bracket in the compress buffer as if it were the last term read.
00160     Have a look at AR.CompressPointer. (set it right)
00161     */
00162     term = AT.WorkPointer;
00163     t1 = bracketinfo->bracketbuffer+bi->bracket;
00164     j = *t1;
00165     /*
00166     Note that in the bracketbuffer, the bracket sits with HAAKJE.
00167
00168     The next is (hopefully) a bug fix. Originally the code read bsize = j
00169     but that overcounts one. We have the part outside the bracket and the
00170     coefficient which is 1,1,3. But we also have the length indicator.
00171     Where we use the variable bsize we do not include the length indicator,
00172     and we have the part outside plus 7,3,0 which is also three words.
00173     */
00174     bsize = j-1;
00175     t2 = AR.CompressPointer;
00176     NCOPY(t2,t1,j)
00177     if ( i == 0 ) { /* We found the proper bracket already */
00178         AR.SortType = oldsorttype;
00179         AT.fromindex = 0;
00180         TermFree(bracketh,"FindBracket");
00181         return(&AN.theposition);
00182     }
00183     /*
00184     Here we have to skip to the bracket if it exists (!)
00185     Let us first look whether the expression is in memory.
00186     If not we have to make a buffer to increase speed..
00187     */
00188     if ( fi->handle < 0 ) {
00189         p = (WORD *) ((UBYTE *) (fi->PObuffer)
00190             + BASEPOSITION(AS.OldOnFile[nexp])
00191             + BASEPOSITION(bi->start));
00192         pstop = (WORD *) ((UBYTE *) (fi->PObuffer)
00193             + BASEPOSITION(AS.OldOnFile[nexp])
00194             + BASEPOSITION(bi->next));
00195         while ( p < pstop ) {
00196             /*
00197             Check now: if size or part from previous term < size of bracket
00198             we have to setup the bracket again and test.
00199             Otherwise, skip immediately to the next term.
00200             */
00201             if ( *p <= -bsize ) { /* no change of bracket */
00202                 p++; p += *p + 1;
00203             }
00204             else if ( *p < 0 ) { /* changes bracket */
00205                 pp = p;
00206                 t2 = AR.CompressPointer;

```

```

00207         t1 = t2 - *p++ + 1;
00208         j = *p++;
00209         NCOPY(t1,p,j)
00210         t2++; while ( *t2 != HAAKJE ) t2 += t2[1];
00211         t2 += t2[1];
00212         a[1] = t2[0]; a[2] = t2[1]; a[3] = t2[2];
00213         *t2++ = 1; *t2++ = 1; *t2++ = 3;
00214         *AR.CompressPointer = t2 - AR.CompressPointer;
00215         bsize = *AR.CompressPointer - 1;
00216         if ( *bracketh == 7 ) {
00217             if ( AR.CompressPointer[0] == 7 ) i = 0;
00218             else i = -1;
00219         }
00220         else if ( AR.CompressPointer[0] == 7 ) i = 1;
00221         else i = CompareTerms(BHEAD bracketh,AR.CompressPointer,0);
00222         t2[-3] = a[1]; t2[-2] = a[2]; t2[-1] = a[3];
00223         if ( i == 0 ) {
00224             SETBASEPOSITION(AN.theposition,(pp-fi->PObuffer)*sizeof(WORD));
00225             fi->POfill = pp;
00226             goto found;
00227         }
00228         if ( i > 0 ) break; /* passed what was possible */
00229     }
00230     else { /* no compression. We have to check! */
00231         WORD *oldworkpointer = AT.WorkPointer, *t3, *t4;
00232         t2 = p + 1; while ( *t2 != HAAKJE ) t2 += t2[1];
00233         t2 += t2[1];
00234         /*
00235          Here we need to copy the term. Modifying has proven to
00236          be NOT threadsafe.
00237          */
00238         t3 = oldworkpointer; t4 = p;
00239         while ( t4 < t2 ) *t3++ = *t4++;
00240         *t3++ = 1; *t3++ = 1; *t3++ = 3;
00241         *oldworkpointer = t3 - oldworkpointer;
00242         bsize = *oldworkpointer - 1;
00243         AT.WorkPointer = t3;
00244         t3 = oldworkpointer;
00245         if ( *bracketh == 7 ) {
00246             if ( t3[0] == 7 ) i = 0;
00247             else i = -1;
00248         }
00249         else if ( t3[0] == 7 ) i = 1;
00250         else {
00251             i = CompareTerms(BHEAD bracketh,t3,0);
00252         }
00253         AT.WorkPointer = oldworkpointer;
00254         if ( i == 0 ) {
00255             SETBASEPOSITION(AN.theposition,(p-fi->PObuffer)*sizeof(WORD));
00256             fi->POfill = p;
00257             goto found;
00258         }
00259         if ( i > 0 ) break; /* passed what was possible */
00260         p += *p;
00261     }
00262 }
00263 AR.SortType = oldsorttype;
00264 AT.fromindex = 0;
00265 TermFree(bracketh,"FindBracket");
00266 return(0); /* Bracket does not exist */
00267 }
00268 else {
00269     /*
00270     In this case we can work with the old representation without HAAKJE.
00271     We stop searching when we reach toppos and we do not call Compare.
00272     */
00273     toppos = AS.OldOnFile[nexp];
00274     ADD2POS(toppos,bi->next);
00275     cp = AR.CompressPointer;
00276     for(;;) {
00277         auxpos = AN.theposition;
00278         GetOneTerm(BHEAD term,fi,&auxpos,0);
00279         if ( *term == 0 ) {
00280             AR.SortType = oldsorttype;
00281             AT.fromindex = 0;
00282             return(0); /* Bracket does not exist */
00283         }
00284         tstop = term + *term;
00285         tstop -= ABS(tstop[-1]);
00286         t1 = term + 1;
00287         while ( *t1 != HAAKJE && t1 < tstop ) t1 += t1[1];
00288         i = *bracket-4;
00289         if ( t1-term == *bracket-3 ) {
00290             t1 = term + 1; t2 = bracket+1;
00291             while ( i > 0 && *t1 == *t2 ) { t1++; t2++; i--; }
00292             if ( i <= 0 ) {
00293                 AR.CompressPointer = cp;

```

```

00294             goto found;
00295         }
00296     }
00297     AR.CompressPointer = cp;
00298     AN.theposition = auxpos;
00299 /*
00300     Now check whether we passed the 'point'
00301 */
00302     if ( ISGEPOS(AN.theposition,toppos) ) {
00303         AR.SortType = oldsorttype;
00304         AR.CompressPointer = cp;
00305         AT.fromindex = 0;
00306         TermFree(bracketh,"FindBracket");
00307         return(0); /* Bracket does not exist */
00308     }
00309 }
00310 }
00311 found:
00312     AR.SortType = oldsorttype;
00313     AT.fromindex = 0;
00314     TermFree(bracketh,"FindBracket");
00315     return(&AN.theposition);
00316 }
00317
00318 /*
00319     #] FindBracket :
00320     #[ PutBracketInIndex :
00321
00322     Call via
00323     if ( AR.BracketOn ) PutBracketInIndex(BHEAD term);
00324
00325     This means that there should be a bracket somewhere
00326     Note that the brackets come in in proper order.
00327
00328     DON'T forget AR.SortType to be put into e->bracketinfo->SortType
00329 */
00330
00331 void PutBracketInIndex(PHEAD WORD *term, POSITION *newpos)
00332 {
00333     GETBIDENTITY
00334     BRACKETINDEX *bi, *b1, *b2, *b3;
00335     BRACKETINFO *b;
00336     POSITION thepos;
00337     EXPRESSIONS e = Expressions + AR.CurExpr;
00338     LONG hi, i, average;
00339     WORD *t, *tstop, *t1, *t2, *oldt, oldsize, a[4];
00340     if ( ( b = e->newbracketinfo ) == 0 ) return;
00341     DIFPOS(thepos,*newpos,e->onfile);
00342     tstop = term + *term;
00343     tstop -= ABS(tstop[-1]);
00344     t = term+1;
00345     while ( *t != HAAKJE && t < tstop ) t += t[1];
00346     if ( t >= tstop ) return; /* no ticket, no laundry */
00347     t += t[1]; /* include HAAKJE for the sorting */
00348     a[0] = t[0]; a[1] = t[1]; a[2] = t[2];
00349     oldt = t; oldsize = *term; *t++ = 1; *t++ = 1; *t++ = 3;
00350     *term = t - term;
00351     AT.fromindex = 1;
00352 /*
00353     Check now with the last bracket in the buffer.
00354     If it is the same we can abort.
00355 */
00356     hi = b->indexfill;
00357     if ( hi > 0 ) {
00358         bi = b->indexbuffer + hi - 1;
00359         bi->next = thepos;
00360         if ( *term == 7 ) {
00361             if ( b->bracketbuffer[bi->bracket] == 7 ) i = 0;
00362             else i = -1;
00363         }
00364         else if ( b->bracketbuffer[bi->bracket] == 7 ) i = 1;
00365         else i = CompareTerms(BHEAD term,b->bracketbuffer+bi->bracket,0);
00366         if ( i == 0 ) { /* still the same bracket */
00367             bi->termsinbracket++;
00368             goto bracketdone;
00369         }
00370         if ( i > 0 ) { /* We have a problem */
00371 /*
00372         There is a special case in which we have only functions and
00373         term is contained completely in the bracket
00374 */
00375         /*
00376         t = term + 1;
00377         tstop = term + *term - 3;
00378         while ( t < tstop && *t > HAAKJE ) t += t[1];
00379         if ( t < tstop ) goto problems;
00380 */

```

```

00381         for ( i = 1; i < *term - 3; i++ ) {
00382             if ( term[i] != b->bracketbuffer[bi->bracket+i] ) break;
00383         }
00384         if ( i < *term - 3 ) {
00385             /*
00386             problems;;
00387             */
00388             *term = oldsize; oldt[0] = a[0]; oldt[1] = a[1]; oldt[2] = a[2];
00389             MLOCK(ErrorMessageLock);
00390             MesPrint("Error!!!! Illegal bracket sequence detected in PutBracketInIndex");
00391             #ifdef WITHPTHREADS
00392             MesPrint("Worker = %w");
00393             #endif
00394             PrintTerm(term,"term into index");
00395             PrintTerm(b->bracketbuffer+bi->bracket,"Last in index");
00396             MUNLOCK(ErrorMessageLock);
00397             AT.fromindex = 0;
00398             Terminate(-1);
00399         }
00400         i = -1;
00401     }
00402 }
00403 /*
00404 If there is room for more brackets, we add this one.
00405 */
00406 if ( b->bracketfill+*term >= b->bracketbuffersize
00407     && ( b->bracketbuffersize < AM.MaxBracketBufferSize
00408         || ( e->vflags & ISFACTORIZED ) != 0 ) ) {
00409     /*
00410     Enlarge bracket buffer
00411     */
00412     WORD *oldbracketbuffer = b->bracketbuffer;
00413     i = Max(b->bracketbuffersize * 2, b->bracketfill+*term+1);
00414     if ( i > AM.MaxBracketBufferSize && ( e->vflags & ISFACTORIZED ) == 0 )
00415         i = AM.MaxBracketBufferSize;
00416     if ( i > b->bracketfill+*term ) {
00417         b->bracketbuffersize = i;
00418         b->bracketbuffer = (WORD *)Malloca1(b->bracketbuffersize*sizeof(WORD),
00419             "new bracket buffer");
00420         t1 = b->bracketbuffer; t2 = oldbracketbuffer;
00421         i = b->bracketfill;
00422         NCOPY(t1,t2,i)
00423         if ( oldbracketbuffer ) M_free(oldbracketbuffer,"old bracket buffer");
00424     }
00425 }
00426 if ( b->bracketfill+*term < b->bracketbuffersize ) {
00427     if ( b->indexfill >= b->indexbuffersize ) {
00428         /*
00429         Enlarge index
00430         */
00431         BRACKETINDEX *oldindexbuffer = b->indexbuffer;
00432         b->indexbuffersize *= 2;
00433         b->indexbuffer = (BRACKETINDEX *)
00434             Malloca1(b->indexbuffersize*sizeof(BRACKETINDEX),"new bracket index");
00435         b1 = b->indexbuffer; b2 = oldindexbuffer;
00436         i = b->indexfill;
00437         NCOPY(b1,b2,i)
00438         if ( oldindexbuffer ) M_free(oldindexbuffer,"old bracket index");
00439     }
00440 }
00441 else {
00442     /*
00443     We have too many brackets in the buffer. Try to improve.
00444     This is the interesting algorithm. We try to eliminate about 1/4 to
00445     1/2 of the brackets from the index. This should be done by size of
00446     the bracket contents to make the searching as fast as possible.
00447     But! Do not touch the last bracket.
00448     Note that we are always filling from the back.
00449     Algorithm: Throw away every second bracket, unless b1+b2 is much longer
00450     than average. How much is something we can tune.
00451     */
00452     average = DIVPOS(thepos,b->indexfill+1);
00453     if ( average <= 0 ) {
00454         MLOCK(ErrorMessageLock);
00455         MesPrint("Problems with bracket buffer. Increase MaxBracketBufferSize in form.set");
00456         MesPrint("Current size is %l",AM.MaxBracketBufferSize*sizeof(WORD));
00457         MUNLOCK(ErrorMessageLock);
00458         Terminate(-1);
00459     }
00460     average *= 4; /* 2*2: one 2 for much longer, one 2 because we have pairs */
00461     t2 = b->bracketbuffer;
00462     b3 = b1 = b->indexbuffer;
00463     bi = b->indexbuffer + b->indexfill;
00464     b2 = b1+1;
00465     while ( b2+2 < bi ) {
00466         if ( DIFBASE(b2->next,b1->start) > average ) {
00467             t1 = b->bracketbuffer + b1->bracket;

```

```

00468         b1->bracket = t2 - b->bracketbuffer;
00469         i = *t1; NCOPY(t2,t1,i)
00470         *b3++ = *b1;
00471         t1 = b->bracketbuffer + b2->bracket;
00472         b2->bracket = t2 - b->bracketbuffer;
00473         i = *t1; NCOPY(t2,t1,i)
00474         *b3++ = *b2;
00475         if ( b3 <= b1 ) {
00476             PUTZERO(b1->start);
00477             PUTZERO(b1->next);
00478             b1->bracket = 0;
00479             b1->termsinbracket = 0;
00480         }
00481         if ( b3 <= b2 ) {
00482             PUTZERO(b2->start);
00483             PUTZERO(b2->next);
00484             b2->bracket = 0;
00485             b2->termsinbracket = 0;
00486         }
00487     }
00488     else {
00489         t1 = b->bracketbuffer + b1->bracket;
00490         b1->bracket = t2 - b->bracketbuffer;
00491         i = *t1; NCOPY(t2,t1,i)
00492         b1->next = b2->next;
00493         b1->termsinbracket += b2->termsinbracket;
00494         *b3++ = *b1;
00495         if ( b3 <= b1 ) {
00496             PUTZERO(b1->start);
00497             PUTZERO(b1->next);
00498             b1->bracket = 0;
00499             b1->termsinbracket = 0;
00500         }
00501         PUTZERO(b2->start);
00502         PUTZERO(b2->next);
00503         b2->bracket = 0;
00504         b2->termsinbracket = 0;
00505     }
00506     b1 += 2; b2 += 2;
00507 }
00508 while ( b1 < bi ) {
00509     t1 = b->bracketbuffer + b1->bracket;
00510     b1->bracket = t2 - b->bracketbuffer;
00511     i = *t1; NCOPY(t2,t1,i)
00512     *b3++ = *b1;
00513     if ( b3 <= b1 ) {
00514         PUTZERO(b1->start);
00515         PUTZERO(b1->next);
00516         b1->bracket = 0;
00517         b1->termsinbracket = 0;
00518     }
00519     b1++;
00520 }
00521 b->indexfill = b3 - b->indexbuffer;
00522 b->bracketfill = t2 - b->bracketbuffer;
00523 }
00524 bi = b->indexbuffer + b->indexfill;
00525 b->indexfill++;
00526 bi->bracket = b->bracketfill;
00527 bi->start = thepos;
00528 bi->next = thepos;
00529 bi->termsinbracket = 1;
00530 /*
00531 Copy the bracket into the buffer
00532 */
00533 t1 = term; t2 = b->bracketbuffer + bi->bracket; i = *t1;
00534 b->bracketfill += i;
00535 NCOPY(t2,t1,i)
00536 bracketdone:
00537 *term = oldsize; oldt[0] = a[0]; oldt[1] = a[1]; oldt[2] = a[2];
00538 AT.fromindex = 0;
00539 }
00540
00541 /*
00542 #] PutBracketInIndex :
00543 #[ ClearBracketIndex :
00544 */
00545
00546 void ClearBracketIndex(WORD numexp)
00547 {
00548     BRACKETINFO *b;
00549     if ( numexp >= 0 ) {
00550         b = Expressions[numexp].bracketinfo;
00551         Expressions[numexp].bracketinfo = 0;
00552     }
00553     else if ( numexp == -1 ) {
00554         GETIDENTITY

```

```

00555         b = AT.bracketinfo;
00556         AT.bracketinfo = 0;
00557     }
00558     else {
00559         numexp = -numexp-2;
00560         b = Expressions[numexp].newbracketinfo;
00561         Expressions[numexp].newbracketinfo = 0;
00562     }
00563     if ( b == 0 ) return;
00564     b->indexfill = b->indexbuffersize = 0;
00565     b->bracketfill = b->bracketbuffersize = 0;
00566     M_free(b->bracketbuffer, "ClearBracketBuffer");
00567     M_free(b->indexbuffer, "ClearIndexBuffer");
00568     M_free(b, "BracketInfo");
00569 }
00570
00571 /*
00572 #] ClearBracketIndex :
00573 #[ OpenBracketIndex :
00574
00575     Note: This routine is thread-safe
00576 */
00577 void OpenBracketIndex(WORD nexpr)
00578 {
00579     EXPRESSIONS e = Expressions + nexpr;
00580     BRACKETINFO *bi;
00581     LONG i;
00582     bi = (BRACKETINFO *)Malloc1(sizeof(BRACKETINFO), "BracketInfo");
00583     e->newbracketinfo = bi;
00584     /*
00585     i = 20*AM.MaxTer/sizeof(WORD);
00586     if ( i < 1000 ) i = 1000;
00587     */
00588     i = 2000;
00589     bi->bracketbuffer = (WORD *)Malloc1(i*sizeof(WORD), "Bracket Buffer");
00590     bi->bracketbuffersize = i;
00591     bi->bracketfill = 0;
00592     i = 50;
00593     bi->indexbuffer = (BRACKETINDEX *)Malloc1(i*sizeof(BRACKETINDEX), "Bracket Index");
00594     bi->indexbuffersize = i;
00595     bi->indexfill = 0;
00596     bi->SortType = AC.SortType;
00597 }
00598
00599
00600 /*
00601 #] OpenBracketIndex :
00602 #[ PutInside :
00603
00604     Puts a term, or a bracket determined part of a term inside a function.
00605
00606     AT.WorkPointer points at term+*term
00607 */
00608
00609 int PutInside(PHEAD WORD *term, WORD *code)
00610 {
00611     WORD *from, *to, *oldbuf, *tStop, *t, *tt, oldon, oldact, inc, argsize, *termout;
00612     int i, ii, error;
00613
00614     if ( code[1] == 4 && ( code[2] == 0 || code[2] == 1 ) ) {
00615         /*
00616         Put all inside. Move the term by 1+FUNHEAD+ARGHEAD
00617         */
00618         from = term+*term; to = from+1+ARGHEAD+FUNHEAD; i = ii = *term;
00619         to[0] = 1; to[1] = 1; to[2] = 3;
00620         while ( --i >= 0 ) *--to = *--from;
00621         to = term;
00622         *to++ = term[0]+4+ARGHEAD+FUNHEAD;
00623         *to++ = code[3];
00624         *to++ = ii+FUNHEAD+ARGHEAD;
00625         *to++ = 1; /* set dirty flags, because there could be a fast notation */
00626         FILLFUN3(to)
00627         *to++ = ii+ARGHEAD;
00628         *to++ = 1;
00629         FILLARG(to)
00630         return(0);
00631     }
00632     /*
00633     First we save the old bracket variables. Then we set variables to
00634     influence the PutBracket routine and call it.
00635     After that we set the values back and sort out the results by placing the
00636     inside of the bracket inside the function.
00637     */
00638     termout = AT.WorkPointer;
00639     oldbuf = AT.BrackBuf;
00640     oldon = AR.BracketOn;
00641     oldact = AT.PolyAct;

```

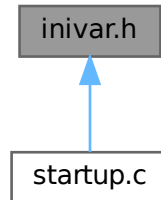
```

00642     AR.BracketOn = -code[2];
00643     AT.BrackBuf  = code+4;
00644     AT.PolyAct = 0;
00645     error = PutBracket(BHEAD term);
00646     AT.PolyAct = oldact;
00647     AT.BrackBuf = oldbuf;
00648     AR.BracketOn = oldon;
00649     if ( error ) return(error);
00650     i = *termout; from = termout; to = term;
00651     NCOPY(to,from,i);
00652     tStop = term +*term; tStop -= tStop[-1];
00653     t = term+1;
00654     while ( t < tStop && *t != HAAKJE ) t += t[1];
00655     from = term + *term;
00656     inc = FUNHEAD+ARGHEAD-t[1]+1;
00657     tt = t + t[1];
00658     argsize = from-tt+1;
00659     to = from + inc;
00660     to[0] = 1;
00661     to[1] = 1;
00662     to[2] = 3;
00663     while ( from > tt ) *--to = *--from;
00664     *--to = argsize;
00665     *t++ = code[3];
00666     *t++ = argsize+FUNHEAD+ARGHEAD;
00667     *t++ = 1;
00668     FILLFUN3(t);
00669     *t++ = argsize+ARGHEAD;
00670     *t++ = 1;
00671     FILLARG(t);
00672     *term += inc+3;
00673     AT.WorkPointer = term+*term;
00674     if ( Normalize(BHEAD term) ) error = 1;
00675     return(error);
00676 }
00677
00678 /*
00679 #] PutInside :
00680 */
00681
00682 /*
00683 The next routines are for indexing the local output files in a parallel
00684 sort. This indexing is needed to get a fast determination of the
00685 splitting terms needed to divide the terms evenly over the processors.
00686 Actually this method works well for ParFORM, but may not work well
00687 for TFORM.
00688
00689 #[ PutTermInIndex :
00690
00691 Puts a term in the term index.
00692 Action:
00693     if the index hasn't reached its full size
00694         if there is room, put the term
00695         if there is no room: extend the buffer, put the term
00696     else
00697         check if the last term has a number of the type skip*m+1
00698         if no, overwrite the last term
00699         if yes, check whether there is room for one more term
00700             yes: add the term
00701             no: drop all even terms, compress the list,
00702                 multiply skip by 2, and add this term.
00703
00704 int PutTermInIndex(WORD *term,POSITION *position)
00705 {
00706     return(0);
00707 }
00708
00709 #] PutTermInIndex :
00710 */

```


4.64 inivar.h File Reference

This graph shows which files directly or indirectly include this file:



Data Structures

- struct **fixedfun**

Variables

- [FIXEDGLOBALS](#) FG
- [ALLGLOBALS](#) A
- [FIXEDSET](#) fixedsets []
- UBYTE [BufferForOutput](#) [MAXLINELENGTH+14]
- char * [setupfilename](#) = "form.set"

4.64.1 Detailed Description

Contains the initialization of a number of structs at compile time This file should only be included in the file [startup.c](#) !!!

Definition in file [inivar.h](#).

4.64.2 Variable Documentation

4.64.2.1 FG

[FIXEDGLOBALS](#) FG

Definition at line 34 of file [inivar.h](#).

4.64.2.2 A

[ALLGLOBALS](#) A

Definition at line 133 of file [inivar.h](#).

4.64.2.3 fixedsets

```
FIXEDSET fixedsets[]
```

Initial value:

```
= {
  {"pos_", "integers > 0", CSYMBOL, 0}
, {"pos0_", "integers >= 0", CSYMBOL, 0}
, {"neg_", "integers < 0", CSYMBOL, 0}
, {"neg0_", "integers <= 0", CSYMBOL, 0}
, {"even_", "even integers", CSYMBOL, 0}
, {"odd_", "odd integers", CSYMBOL, 0}
, {"int_", "all integers", CSYMBOL, 0}
, {"symbol_", "only symbols", CSYMBOL, MAXPOSITIVE}
, {"fixed_", "fixed indices", CINDEX, 0}
, {"index_", "all indices", CINDEX, 0}
, {"number_", "all rationals", CSYMBOL, 0}
, {"dummyindices_", "dummy indices", CINDEX, 0}
, {"vector_", "only vectors", CVECTOR, 0}
}
```

Definition at line 259 of file [inivar.h](#).

4.64.2.4 BufferForOutput

```
UBYTE BufferForOutput[MAXLINELENGTH+14]
```

Definition at line 275 of file [inivar.h](#).

4.64.2.5 setupfilename

```
char* setupfilename = "form.set"
```

Definition at line 277 of file [inivar.h](#).

4.65 inivar.h

[Go to the documentation of this file.](#)

```
00001
00007 /* #[] License : */
00008 /*
00009 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00010 * When using this file you are requested to refer to the publication
00011 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00012 * This is considered a matter of courtesy as the development was paid
00013 * for by FOM the Dutch physics granting agency and we would like to
00014 * be able to track its scientific use to convince FOM of its value
00015 * for the community.
00016 *
00017 * This file is part of FORM.
00018 *
00019 * FORM is free software: you can redistribute it and/or modify it under the
00020 * terms of the GNU General Public License as published by the Free Software
00021 * Foundation, either version 3 of the License, or (at your option) any later
00022 * version.
00023 *
00024 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00025 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00026 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00027 * details.
00028 *
00029 * You should have received a copy of the GNU General Public License along
00030 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00031 */
00032 /* #[] License : */
00033
```

```

00034 FIXEDGLOBALS FG = {
00035     {Traces                                     /* Dummy as zeroeth argument */
00036     ,Traces
00037     ,EpfcCon
00038     ,RatioGen
00039     ,SymGen
00040     ,TenVec
00041     ,DoSumF1
00042     ,DoSumF2}
00043
00044     ,{TraceFind                                 /* Dummy as zeroeth argument */
00045     ,TraceFind
00046     ,EpffFind
00047     ,RatioFind
00048     ,SymFind
00049     ,TenVecFind}
00050
00051     ,{"symbol"
00052     , "index"
00053     , "vector"
00054     , "function"
00055     , "set"
00056     , "expression"
00057     , "dotproduct"
00058     , "number"
00059     , "subexp"
00060     , "delta"}
00061
00062     ,{"(local)"
00063     , "(skip/local)"
00064     , "(drop/local)"
00065     , "(dropped)"
00066     , "(global)"
00067     , "(skip/global)"
00068     , "(drop/global)"
00069     , "(dropped)"
00070     , "(stored)"
00071     , "(local-hidden)"
00072     , "(local-to be hidden)"
00073     , "(local-hidden-dropped)"
00074     , "(local-to be unhidden)"
00075     , "(global-hidden)"
00076     , "(global-to be hidden)"
00077     , "(global-hidden-dropped)"
00078     , "(global-to be unhidden)"
00079     , "(into-hide-local)"
00080     , "(into-hide-global)"
00081     , "(spectator)"
00082     , "(drop/spectator)"}
00083
00084     ,{" Functions"
00085     , " Commuting Functions"}
00086
00087     ,{"left      "
00088     , "active    "
00089     , "in output"}
00090
00091     ,(char *)0
00092     ,(char *)0
00093     ,(UBYTE *)"1"
00094     ,(WORD)0
00095     ,(WORD)0
00096     ,(WORD)0
00097     ,(WORD)0
00098
00099 /* ASCII table of character types. Note that on some computers this
00100 table may be different from the ASCII table.
00101
00102 -1 Illegal character
00103 0 Alphabetic character
00104 1 Digit
00105 2 . $ _ ? # or '
00106 3 [,]
00107 4 ( ) = ; or ,
00108 5 + - * % / ^ :
00109 6 blank, tab, linefeed
00110 7 {,|,}
00111 8 ! & < >
00112 9 "
00113 10 The ultimate end.
00114 */
00115 ,{
00116     10,255,255,255,255,255,255,255,255,255, 6, 6,255,255, 6,255,255,
00117     255,255,255,255,255,255,255,255,255,255,10,255,255,255,255,255,
00118     6, 8, 9, 2, 2, 5, 8, 2, 4, 4, 5, 5, 4, 5, 2, 5,
00119     1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 5, 4, 8, 4, 8, 2,
00120     255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
00121     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3,255, 3, 5, 2,

```

```

00121     255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
00122     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 7, 7, 255, 255,
00123     255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
00124     255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
00125     255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
00126     255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
00127     255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
00128     255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
00129     255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
00130     255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255 }
00131 };
00132
00133 ALLGLOBALS A;
00134 #ifdef WITHPTHREADS
00135 ALLPRIVATES **AB;
00136 #endif
00137
00138 static struct fixedfun {
00139     char *name;
00140     int      commu , tensor      , complx      , symmetric;
00141 } fixedfunctions[] = {
00142     {"exp_" , 1, 0, 0, 0} /* EXPONENT */
00143     , {"denom_" , 1, 0, 0, 0} /* DENOMINATOR */
00144     , {"setfun_" , 1, 0, 0, 0} /* SETFUNCTION */
00145     , {"g_" , 1, GAMMAFUNCTION, VARTYPEIMAGINARY, 0} /* GAMMA */
00146     , {"gi_" , 1, GAMMAFUNCTION, VARTYPEIMAGINARY, 0} /* GAMMAI */
00147     , {"g5_" , 1, GAMMAFUNCTION, VARTYPEIMAGINARY, 0} /* GAMMAFIVE */
00148     , {"g6_" , 1, GAMMAFUNCTION, VARTYPEIMAGINARY, 0} /* GAMMASIX */
00149     , {"g7_" , 1, GAMMAFUNCTION, VARTYPEIMAGINARY, 0} /* GAMMASEVEN */
00150     , {"sum_" , 1, 0, 0, 0} /* SUMF1 */
00151     , {"sump_" , 1, 0, 0, 0} /* SUMF2 */
00152     , {"dum_" , 1, 0, 0, 0} /* DUMFUN */
00153     , {"replace_" , 0, 0, 0, 0} /* REPLACEMENT */
00154     , {"reverse_" , 1, 0, 0, 0} /* REVERSEFUNCTION */
00155     , {"distrib_" , 1, 0, 0, 0} /* DISTRIBUTION */
00156     , {"dd_" , 0, TENSORFUNCTION, 0, 0} /* DELTA3 */
00157     , {"dummy_" , 0, 0, 0, 0} /* DUMMYFUN */
00158     , {"dummyten_" , 0, TENSORFUNCTION, 0, 0} /* DUMMYTEN */
00159     , {"e_" , 0, TENSORFUNCTION|1, VARTYPEIMAGINARY, ANTISYMMETRIC}
00160     /* LEVICIVITA */
00161     , {"fac_" , 0, 0, 0, 0} /* FACTORIAL */
00162     , {"invfac_" , 0, 0, 0, 0} /* INVERSEFACTORIAL */
00163     , {"binom_" , 0, 0, 0, 0} /* BINOMIAL */
00164     , {"nargs_" , 0, 0, 0, 0} /* NUMARGSFUN */
00165     , {"sign_" , 0, 0, 0, 0} /* SIGNFUN */
00166     , {"mod_" , 0, 0, 0, 0} /* MODFUNCTION */
00167     , {"mod2_" , 0, 0, 0, 0} /* MOD2FUNCTION */
00168     , {"min_" , 0, 0, 0, 0} /* MINFUNCTION */
00169     , {"max_" , 0, 0, 0, 0} /* MAXFUNCTION */
00170     , {"abs_" , 0, 0, 0, 0} /* ABSFUNCTION */
00171     , {"sig_" , 0, 0, 0, 0} /* SIGFUNCTION */
00172     , {"integer_" , 0, 0, 0, 0} /* INTFUNCTION */
00173     , {"theta_" , 0, 0, 0, 0} /* THETA */
00174     , {"thetap_" , 0, 0, 0, 0} /* THETA2 */
00175     , {"delta_" , 0, 0, 0, 0} /* DELTA2 */
00176     , {"deltap_" , 0, 0, 0, 0} /* DELTAP */
00177     , {"bernoulli_" , 0, 0, 0, 0} /* BERNOULLIFUNCTION */
00178     , {"count_" , 0, 0, 0, 0} /* COUNTFUNCTION */
00179     , {"match_" , 0, 0, 0, 0} /* MATCHFUNCTION */
00180     , {"pattern_" , 0, 0, VARTYPECOMPLEX, 0} /* PATTERNFUNCTION */
00181     , {"term_" , 1, 0, 0, 0} /* TERMFUNTION */
00182     , {"conjg_" , 1, 0, VARTYPECOMPLEX, 0} /* CONJUGATEFUNCTION */
00183     , {"root_" , 0, 0, 0, 0} /* ROOTFUNCTION */
00184     , {"table_" , 1, 0, 0, 0} /* TABLEFUNCTION */
00185     , {"firstbracket_" , 0, 0, 0, 0} /* FIRSTBRACKET */
00186     , {"termsin_" , 0, 0, 0, 0} /* TERMSINEXPR */
00187     , {"nterms_" , 0, 0, 0, 0} /* NUMTERMSFUN */
00188     , {"gcd_" , 0, 0, 0, 0} /* GCDFUNCTION */
00189     , {"div_" , 0, 0, 0, 0} /* DIVFUNCTION */
00190     , {"rem_" , 0, 0, 0, 0} /* REMFUNCTION */
00191     , {"maxpowerof_" , 0, 0, 0, 0} /* MAXPOWEROF */
00192     , {"minpowerof_" , 0, 0, 0, 0} /* MINPOWEROF */
00193     , {"tbl_" , 0, 0, 0, 0} /* TABLESTUB */
00194     , {"factorin_" , 0, 0, 0, 0} /* FACTORIN */
00195     , {"termsinbracket_" , 0, 0, 0, 0} /* TERMSINBRACKET */
00196     , {"farg_" , 0, 0, 0, 0} /* WILDARGFUN */
00197 /*
00198     The following names are reserved for the floating point library.
00199     As long as we have no floating point numbers they do not do anything.
00200 */
00201     , {"sqrt_" , 0, 0, 0, 0} /* SQRTFUNCTION */
00202     , {"ln_" , 0, 0, 0, 0} /* LNFUNTION */
00203     , {"sin_" , 0, 0, 0, 0} /* SINFUNTION */
00204     , {"cos_" , 0, 0, 0, 0} /* COSFUNCTION */
00205     , {"tan_" , 0, 0, 0, 0} /* TANFUNCTION */
00206     , {"asin_" , 0, 0, 0, 0} /* ASINFUNTION */
00207     , {"acos_" , 0, 0, 0, 0} /* ACOSFUNCTION */

```

```

00208     ,{"atan_"      ,0,0      ,0      ,0} /* ATANFUNCTION */
00209     ,{"atan2_"     ,0,0      ,0      ,0} /* ATAN2FUNCTION */
00210     ,{"sinh_"      ,0,0      ,0      ,0} /* SINHFUNCTION */
00211     ,{"cosh_"      ,0,0      ,0      ,0} /* COSHFUNCTION */
00212     ,{"tanh_"      ,0,0      ,0      ,0} /* TANHFUNCTION */
00213     ,{"asinh_"     ,0,0      ,0      ,0} /* ASINHFUNCTION */
00214     ,{"acosh_"     ,0,0      ,0      ,0} /* ACOSHFUNCTION */
00215     ,{"atanh_"     ,0,0      ,0      ,0} /* ATANHFUNCTION */
00216     ,{"li2_"       ,0,0      ,0      ,0} /* LI2FUNCTION */
00217     ,{"lin_"       ,0,0      ,0      ,0} /* LINFUNCTION */
00218 /*
00219     From here on we continue with new functions (26-sep-2010)
00220 */
00221     ,{"extrasymbol_" ,0,0      ,0      ,0} /* EXTRASYMFUN */
00222     ,{"random_"      ,0,0      ,0      ,0} /* RANDOMFUNCTION */
00223     ,{"ranperm_"     ,1,0      ,0      ,0} /* RANPERM */
00224     ,{"numfactors_"  ,0,0      ,0      ,0} /* NUMFACTORS */
00225     ,{"firstterm_"   ,0,0      ,0      ,0} /* FIRSTTERM */
00226     ,{"content_"     ,0,0      ,0      ,0} /* CONTENTTERM */
00227     ,{"prime_"       ,0,0      ,0      ,0} /* PRIMENUMBER */
00228     ,{"exteuclidean_" ,0,0      ,0      ,0} /* EXTEUCLIDEAN */
00229     ,{"makerational_" ,0,0      ,0      ,0} /* MAKERATIONAL */
00230     ,{"inverse_"     ,0,0      ,0      ,0} /* INVERSEFUNCTION */
00231     ,{"id_"          ,1,0      ,0      ,0} /* IDFUNCTION */
00232     ,{"putfirst_"    ,1,0      ,0      ,0} /* PUTFIRST */
00233     ,{"perm_"        ,1,0      ,0      ,0} /* PERMUTATIONS */
00234     ,{"partitions_"  ,1,0      ,0      ,0} /* PARTITIONS */
00235     ,{"mul_"         ,0,0      ,0      ,0} /* MULFUNCTION */
00236     ,{"moebius_"     ,0,0      ,0      ,0} /* MOEBIUS */
00237     ,{"topologies_"  ,0,0      ,0      ,0} /* TOPOLOGIES */
00238     ,{"diagrams_"    ,0,0      ,0      ,0} /* DIAGRAMS */
00239     ,{"topo_"        ,0,0      ,0      ,0} /* TOPO */
00240     ,{"node_"        ,0,0      ,0      ,0} /* VERTEX */
00241     ,{"edge_"        ,0,0      ,0      ,0} /* EDGE */
00242     ,{"sizeof_"      ,0,0      ,0      ,0} /* SIZEOFFUNCTION */
00243     ,{"block_"       ,0,0      ,0      ,0} /* BLOCK */
00244     ,{"onepi_"       ,0,0      ,0      ,0} /* ONEPI */
00245     ,{"phi_"         ,0, VERTEXFUNCTION,0} /* PHI */
00246     ,{"float_"       ,0,0      ,0      ,0} /* FLOATFUN */
00247     ,{"tofloat_"     ,0,0      ,0      ,0} /* TOFLOAT */
00248     ,{"torat_"       ,0,0      ,0      ,0} /* TORAT */
00249     ,{"mzv_"         ,0,0      ,0      ,0} /* MZV */
00250     ,{"euler_"       ,0,0      ,0      ,0} /* EULER */
00251     ,{"mzvhalf_"     ,0,0      ,0      ,0} /* MZVHALF */
00252     ,{"agm_"         ,0,0      ,0      ,0} /* AGMFUNCTION */
00253     ,{"gamma_"       ,0,0      ,0      ,0} /* GAMMAFUN */
00254     ,{"eexp_"        ,0,0      ,0      ,0} /* EXPFUNCTION */
00255     ,{"hpl_"         ,0,0      ,0      ,0} /* HPLFUNCTION */
00256     ,{"mpl_"         ,0,0      ,0      ,0} /* MPLFUNCTION */
00257 };
00258
00259 FIXEDSET fixedsets[] = {
00260     {"pos_"      , "integers > 0", CSYMBOL, 0} /* POS_ 0 */
00261     ,{"pos0_"    , "integers >= 0", CSYMBOL, 0} /* POS0_ 1 */
00262     ,{"neg_"     , "integers < 0", CSYMBOL, 0} /* NEG_ 2 */
00263     ,{"neg0_"    , "integers <= 0", CSYMBOL, 0} /* NEG0_ 3 */
00264     ,{"even_"    , "even integers", CSYMBOL, 0} /* EVEN_ 4 */
00265     ,{"odd_"     , "odd integers", CSYMBOL, 0} /* ODD_ 5 */
00266     ,{"int_"     , "all integers", CSYMBOL, 0} /* Z_ 6 */
00267     ,{"symbol_"  , "only symbols", CSYMBOL, MAXPOSITIVE} /* SYMBOL_ 7 */
00268     ,{"fixed_"   , "fixed indices", CINDEX, 0} /* FIXED_ 8 */
00269     ,{"index_"   , "all indices", CINDEX, 0} /* INDEX_ 9 */
00270     ,{"number_"  , "all rationals", CSYMBOL, 0} /* Q_ 10 */
00271     ,{"dummyindices_" , "dummy indices", CINDEX, 0} /* DUMMYINDEX_ 11 */
00272     ,{"vector_"  , "only vectors", CVECTOR, 0} /* VECTOR_ 12 */
00273 };
00274
00275 UBYTE BufferForOutput[MAXLINELENGTH+14];
00276
00277 char *setupfilename = "form.set";
00278
00279 #ifdef WITHPTHREADS
00280 INILOCK(ErrorMessageLock)
00281 INILOCK(FileReadLock)
00282 #endif

```


4.66.2.2 FindLus()

```
int FindLus (
    int from,
    int level,
    int openindex )
```

Definition at line 515 of file [lus.c](#).

4.66.2.3 SortTheList()

```
int SortTheList (
    int * slist,
    int num )
```

Definition at line 578 of file [lus.c](#).

4.66.2.4 AllLoops()

```
int AllLoops (
    PHEAD WORD * term,
    WORD level )
```

Definition at line 650 of file [lus.c](#).

4.66.2.5 StartLoops()

```
LONG StartLoops (
    PHEAD WORD * term,
    WORD level,
    LONG vert,
    WORD nvert,
    WORD * arglist,
    WORD nargs,
    WORD * loop,
    WORD nloop )
```

Definition at line 894 of file [lus.c](#).

4.66.2.6 GenLoops()

```
LONG GenLoops (
    PHEAD WORD * term,
    WORD level,
    LONG vert,
    WORD nvert,
    WORD * arglist,
    WORD nargs,
    WORD * loop,
    WORD nloop )
```

Definition at line 980 of file [lus.c](#).

4.66.2.7 LoopOutput()

```
void LoopOutput (
    PHEAD WORD * term,
    WORD level,
    WORD * loop,
    WORD nloop )
```

Definition at line [1059](#) of file [lus.c](#).

4.66.2.8 AllPaths()

```
int AllPaths (
    PHEAD WORD * term,
    WORD level )
```

Definition at line [1123](#) of file [lus.c](#).

4.66.2.9 GenPaths()

```
LONG GenPaths (
    PHEAD WORD * term,
    WORD level,
    LONG vert,
    WORD nvert,
    WORD * arglist,
    WORD nargs,
    WORD * path,
    WORD npath )
```

Definition at line [1413](#) of file [lus.c](#).

4.66.2.10 PathOutput()

```
void PathOutput (
    PHEAD WORD * term,
    WORD level,
    WORD * path,
    WORD npath )
```

Definition at line [1486](#) of file [lus.c](#).

4.66.2.11 AllOnePI()

```
WORD AllOnePI (
    WORD * term,
    WORD level )
```

Definition at line [1566](#) of file [lus.c](#).

4.66.2.12 RemoveBridges()

```
int RemoveBridges (
    void )
```

Definition at line 1586 of file [lus.c](#).

4.66.2.13 TakeOneLine()

```
int TakeOneLine (
    WORD * term,
    WORD level )
```

Definition at line 1596 of file [lus.c](#).

4.66.2.14 OutputOnePI()

```
int OutputOnePI (
    PHEAD WORD * term,
    WORD level )
```

Definition at line 1608 of file [lus.c](#).

4.67 lus.c

[Go to the documentation of this file.](#)

```
00001
00007 /* #[] License : */
00008 /*
00009 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00010 * When using this file you are requested to refer to the publication
00011 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00012 * This is considered a matter of courtesy as the development was paid
00013 * for by FOM the Dutch physics granting agency and we would like to
00014 * be able to track its scientific use to convince FOM of its value
00015 * for the community.
00016 *
00017 * This file is part of FORM.
00018 *
00019 * FORM is free software: you can redistribute it and/or modify it under the
00020 * terms of the GNU General Public License as published by the Free Software
00021 * Foundation, either version 3 of the License, or (at your option) any later
00022 * version.
00023 *
00024 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00025 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00026 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00027 * details.
00028 *
00029 * You should have received a copy of the GNU General Public License along
00030 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00031 */
00032 /* #[] License : */
00033 /*
00034 #[] Includes : lus.c
00035 */
00036
00037 #include "form3.h"
00038
00039 /*
00040 #[] Includes :
00041 #[] Lus :
00042
00043 Routine to find loops.
00044 Mode: 0: Just tell whether there is such a loop.
```

```

00045         1: Take out the functions and replace by outfun with the
00046             remaining arguments of the loop function
00047         > AM.OffsetIndex: This index must be included in the loop.
00048         < -AM.OffsetIndex: This index must be included in the loop. Replace.
00049     Return value: 0: no loop. 1: there is/was such a loop.
00050     funnum: the function(s) in which we look for a loop.
00051     numargs: the number of arguments admissible in the function.
00052     outfun: the output function in case of a substitution.
00053     loopsize: the size of the loop we are looking for.
00054             if < 0 we look for all loops.
00055 */
00056
00057 int Lus(WORD *term, WORD funnum, WORD loopsize, WORD numargs, WORD outfun, WORD mode)
00058 {
00059     GETIDENTITY
00060     WORD *w, *t, *tt, *m, *r, **loc, *tstop, minloopsize;
00061     int nfun, i, j, jj, k, n, sign = 0, action = 0, L, ten, ten2, totnum,
00062     sign2, *alist, *wi, mini, maxi, medi = 0;
00063     if ( numargs <= 1 ) return(0);
00064     GETSTOP(term,tstop);
00065 /*
00066     First count the number of functions with the proper number of arguments.
00067 */
00068     t = term+1; nfun = 0;
00069     if ( ( ten = functions[funnum-FUNCTION].spec ) >= TENSORFUNCTION ) {
00070         while ( t < tstop ) {
00071             if ( *t == funnum && t[1] == FUNHEAD+numargs ) { nfun++; }
00072             t += t[1];
00073         }
00074     }
00075     else {
00076         while ( t < tstop ) {
00077             if ( *t == funnum ) {
00078                 i = 0; m = t+FUNHEAD; t += t[1];
00079                 while ( m < t ) { i++; NEXTARG(m) }
00080                 if ( i == numargs ) nfun++;
00081             }
00082             else t += t[1];
00083         }
00084     }
00085     if ( loopsize < 0 ) minloopsize = 2;
00086     else minloopsize = loopsize;
00087     if ( funnum < minloopsize ) return(0); /* quick abort */
00088     if ( ((functions[funnum-FUNCTION].symmetric) & ~REVERSEORDER) == ANTISYMMETRIC ) sign = 1;
00089     if ( mode == 1 || mode < 0 ) {
00090         ten2 = functions[outfun-FUNCTION].spec >= TENSORFUNCTION;
00091     }
00092     else ten2 = -1;
00093 /*
00094     Allocations:
00095 */
00096     if ( AN.numflocs < funnum ) {
00097         if ( AN.funlocs ) M_free(AN.funlocs,"Lus: AN.funlocs");
00098         AN.numflocs = funnum;
00099         AN.funlocs = (WORD **)Malloc1(sizeof(WORD *)*AN.numflocs,"Lus: AN.funlocs");
00100     }
00101     if ( AN.numfargs < funnum*numargs ) {
00102         if ( AN.funargs ) M_free(AN.funargs,"Lus: AN.funargs");
00103         AN.numfargs = funnum*numargs;
00104         AN.funargs = (int *)Malloc1(sizeof(int *)*AN.numfargs,"Lus: AN.funargs");
00105     }
00106 /*
00107     Make a list of relevant indices
00108 */
00109     alist = AN.funargs; loc = AN.funlocs;
00110     t = term+1;
00111     if ( ten >= TENSORFUNCTION ) {
00112         while ( t < tstop ) {
00113             if ( *t == funnum && t[1] == FUNHEAD+numargs ) {
00114                 *loc++ = t;
00115                 t += FUNHEAD;
00116                 j = i = numargs; while ( --i >= 0 ) {
00117                     if ( *t >= AM.OffsetIndex &&
00118                         ( *t >= AM.OffsetIndex+WILDOFFSET ||
00119                           indices[*t-AM.OffsetIndex].dimension != 0 ) ) {
00119                         *alist++ = *t++; j--;
00120                     }
00121                     else t++;
00122                 }
00123                 while ( --j >= 0 ) *alist++ = -1;
00124             }
00125             else t += t[1];
00126         }
00127     }
00128     else {
00129         nfun = 0;
00130         while ( t < tstop ) {

```

```

00132         if ( *t == funnum ) {
00133             w = t;
00134             i = 0; m = t+FUNHEAD; t += t[1];
00135             while ( m < t ) { i++; NEXTARG(m) }
00136             if ( i == numargs ) {
00137                 m = w + FUNHEAD;
00138                 while ( m < t ) {
00139                     if ( *m == -INDEX && m[1] >= AM.OffsetIndex &&
00140                         ( m[1] >= AM.OffsetIndex+WILDOFFSET ||
00141                           indices[m[1]-AM.OffsetIndex].dimension != 0 ) ) {
00142                         *alist++ = m[1]; m += 2; i--;
00143                     }
00144                     else if ( ten2 >= TENSORFUNCTION && *m != -INDEX
00145                             && *m != -VECTOR && *m != -MINVECTOR &&
00146                             ( *m != -SNUMBER || *m < 0 || *m >= AM.OffsetIndex ) ) {
00147                         i = numargs; break;
00148                     }
00149                     else { NEXTARG(m) }
00150                 }
00151                 if ( i < numargs ) {
00152                     *loc++ = w;
00153                     nfun++;
00154                     while ( --i >= 0 ) *alist++ = -1;
00155                 }
00156             }
00157             else t += t[1];
00158         }
00159     }
00160     if ( nfun < minloopsize ) return(0);
00161 }
00162 /*
00163 We have now nfun objects. Not all indices may be usable though.
00164 If the list is not long, we use a quadratic algorithm to remove
00165 indices and vertices that cannot be used. If it becomes large we
00166 sort the list of available indices (and their multiplicity) and
00167 work with binary searches.
00168 */
00169 alist = AN.funargs; totnum = numargs*nfun;
00170 if ( nfun > 7 ) {
00171     if ( AN.funisize < totnum ) {
00172         if ( AN.funinds ) M_free(AN.funinds,"AN.funinds");
00173         AN.funisize = (totnum*3)/2;
00174         AN.funinds = (int *)Malloc1(AN.funisize*2*sizeof(int),"AN.funinds");
00175     }
00176     i = totnum; n = 0; wi = AN.funinds;
00177     while ( --i >= 0 ) {
00178         if ( *alist >= 0 ) { n++; *wi++ = *alist; *wi++ = 1; }
00179         alist++;
00180     }
00181     n = SortTheList(AN.funinds,n);
00182     do {
00183         action = 0;
00184         for ( i = 0; i < nfun; i++ ) {
00185             alist = AN.funargs + i*numargs;
00186             jj = numargs;
00187             for ( j = 0; j < jj; j++ ) {
00188                 if ( alist[j] < 0 ) break;
00189                 mini = 0; maxi = n-1;
00190                 while ( mini <= maxi ) {
00191                     medi = (mini + maxi) / 2; k = AN.funinds[2*medi];
00192                     if ( alist[j] > k ) mini = medi + 1;
00193                     else if ( alist[j] < k ) maxi = medi - 1;
00194                     else break;
00195                 }
00196                 if ( AN.funinds[2*medi+1] <= 1 ) {
00197                     (AN.funinds[2*medi+1])--;
00198                     jj--; k = j; while ( k < jj ) { alist[k] = alist[k+1]; k++; }
00199                     alist[jj] = -1; j--;
00200                 }
00201             }
00202             if ( jj < 2 ) {
00203                 if ( jj == 1 ) {
00204                     mini = 0; maxi = n-1;
00205                     while ( mini <= maxi ) {
00206                         medi = (mini + maxi) / 2; k = AN.funinds[2*medi];
00207                         if ( alist[0] > k ) mini = medi + 1;
00208                         else if ( alist[0] < k ) maxi = medi - 1;
00209                         else break;
00210                     }
00211                     (AN.funinds[2*medi+1])--;
00212                     if ( AN.funinds[2*medi+1] == 1 ) action++;
00213                 }
00214                 nfun--; totnum -= numargs; AN.funlocs[i] = AN.funlocs[nfun];
00215                 wi = AN.funargs + nfun*numargs;
00216                 for ( j = 0; j < numargs; j++ ) alist[j] = *wi++;
00217                 i--;
00218             }

```

```

00219         }
00220     } while ( action );
00221 }
00222 else {
00223     for ( i = 0; i < totnum; i++ ) {
00224         if ( alist[i] == -1 ) continue;
00225         for ( j = 0; j < totnum; j++ ) {
00226             if ( alist[j] == alist[i] && j != i ) break;
00227         }
00228         if ( j >= totnum ) alist[i] = -1;
00229     }
00230     do {
00231         action = 0;
00232         for ( i = 0; i < nfun; i++ ) {
00233             alist = AN.funargs + i*numargs;
00234             n = numargs;
00235             for ( k = 0; k < n; k++ ) {
00236                 if ( alist[k] < 0 ) { alist[k--] = alist[--n]; alist[n] = -1; }
00237             }
00238             if ( n <= 1 ) {
00239                 if ( n == 1 ) { j = alist[0]; }
00240                 else j = -1;
00241                 nfun--; totnum -= numargs; AN.funlocs[i] = AN.funlocs[nfun];
00242                 wi = AN.funargs + nfun * numargs;
00243                 for ( k = 0; k < numargs; k++ ) alist[k] = wi[k];
00244                 i--;
00245                 if ( j >= 0 ) {
00246                     for ( k = 0, jj = 0, wi = AN.funargs; k < totnum; k++, wi++ ) {
00247                         if ( *wi == j ) { jj++; if ( jj > 1 ) break; }
00248                     }
00249                     if ( jj <= 1 ) {
00250                         for ( k = 0, wi = AN.funargs; k < totnum; k++, wi++ ) {
00251                             if ( *wi == j ) { *wi = -1; action = 1; }
00252                         }
00253                     }
00254                 }
00255             }
00256         }
00257     } while ( action );
00258 }
00259 if ( nfun < minloopsz ) return(0);
00260 /*
00261 Now we have nfun objects, each with at least 2 indices, each of which
00262 occurs at least twice in our list. There will be a loop!
00263 */
00264 if ( mode != 0 && mode != 1 ) {
00265     if ( mode > 0 ) AN.tohunt = mode - 5;
00266     else AN.tohunt = -mode - 5;
00267     AN.nargs = numargs; AN.numoffuns = nfun;
00268     i = 0;
00269     if ( loopsize < 0 ) {
00270         if ( loopsize == -1 ) k = nfun;
00271         else { k = -loopsize-1; if ( k > nfun ) k = nfun; }
00272         for ( L = 2; L <= k; L++ ) {
00273             if ( FindLus(0,L,AN.tohunt) ) goto Success;
00274         }
00275     }
00276     else if ( FindLus(0,loopsize,AN.tohunt) ) { L = loopsize; goto Success; }
00277 }
00278 else {
00279     AN.nargs = numargs; AN.numoffuns = nfun;
00280     if ( loopsize < 0 ) {
00281         jj = 2; k = nfun;
00282         if ( loopsize < -1 ) { k = -loopsize-1; if ( k > nfun ) k = nfun; }
00283     }
00284     else { jj = k = loopsize; }
00285     for ( L = jj; L <= k; L++ ) {
00286         for ( i = 0; i <= nfun-L; i++ ) {
00287             alist = AN.funargs + i * numargs;
00288             for ( jj = 0; jj < numargs; jj++ ) {
00289                 if ( alist[jj] < 0 ) continue;
00290                 AN.tohunt = alist[jj];
00291                 for ( j = jj+1; j < numargs; j++ ) {
00292                     if ( alist[j] < 0 ) continue;
00293                     if ( FindLus(i+1,L-1,alist[j]) ) {
00294                         alist[0] = alist[jj];
00295                         alist[1] = alist[j];
00296                         goto Success;
00297                     }
00298                 }
00299             }
00300         }
00301     }
00302 }
00303 return(0);
00304 Success;
00305 if ( mode == 0 || mode > 1 ) return(1);

```

```

00306 /*
00307     Now we have to make the replacement and fix the potential sign
00308 */
00309     sign2 = 1;
00310     wi = AN.funargs + i*numargs; loc = AN.funlocs + i;
00311     for ( k = 0; k < L; k++ ) *(loc[k]) = -1;
00312     if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
00313     w = AT.WorkPointer + 1;
00314     m = t = term + 1;
00315     while ( t < tstop ) {
00316         if ( *t == -1 ) break;
00317         t += t[1];
00318     }
00319     while ( m < t ) *w++ = *m++;
00320     r = w;
00321     *w++ = outfun;
00322     w++;
00323     *w++ = DIRTYFLAG;
00324     FILLFUN3(w)
00325     if ( functions[outfun-FUNCTION].spec >= TENSORFUNCTION ) {
00326         if ( ten >= TENSORFUNCTION ) {
00327             for ( i = 0; i < L; i++ ) {
00328                 alist = wi + i*numargs;
00329                 m = loc[i] + FUNHEAD;
00330                 for ( k = 0; k < numargs; k++ ) {
00331                     if ( m[k] == alist[0] ) {
00332                         if ( k != 0 ) {
00333                             jj = m[k]; m[k] = m[0]; m[0] = jj;
00334                             sign = -sign;
00335                         }
00336                         break;
00337                     }
00338                 }
00339                 for ( k = 1; k < numargs; k++ ) {
00340                     if ( m[k] == alist[1] ) {
00341                         if ( k != 1 ) {
00342                             jj = m[k]; m[k] = m[1]; m[1] = jj;
00343                             sign = -sign;
00344                         }
00345                         break;
00346                     }
00347                 }
00348                 m += 2;
00349                 for ( k = 2; k < numargs; k++ ) *w++ = *m++;
00350             }
00351         }
00352         else {
00353             WORD *t1, *t2, *t3;
00354             for ( i = 0; i < L; i++ ) {
00355                 alist = wi + i*numargs;
00356                 tt = loc[i];
00357                 m = tt + FUNHEAD;
00358                 for ( k = 0; k < numargs; k++ ) {
00359                     if ( *m == -INDEX && m[1] == alist[0] ) {
00360                         if ( k != 0 ) {
00361                             if ( ( k & 1 ) != 0 ) sign = -sign;
00362                             /*
00363                                 now move to position 0
00364                             */
00365                             t2 = m+2; t1 = m; t3 = tt+FUNHEAD;
00366                             while ( t1 > t3 ) { *--t2 = *--t1; }
00367                             t3[0] = -INDEX; t3[1] = alist[0];
00368                         }
00369                         break;
00370                     }
00371                 }
00372                 NEXTARG(m)
00373                 m = tt + FUNHEAD + 2;
00374                 for ( k = 1; k < numargs; k++ ) {
00375                     if ( *m == -INDEX && m[1] == alist[1] ) {
00376                         if ( k != 1 ) {
00377                             if ( ( k & 1 ) == 0 ) sign = -sign;
00378                             /*
00379                                 now move to position 1
00380                             */
00381                             t2 = m+2; t1 = m; t3 = tt+FUNHEAD+2;
00382                             while ( t1 > t3 ) { *--t2 = *--t1; }
00383                             t3[0] = -INDEX; t3[1] = alist[1];
00384                         }
00385                         break;
00386                     }
00387                 }
00388                 NEXTARG(m)
00389             }
00390             /*
00391                 now copy the remaining arguments to w
00392                 keep in mind that the output function is a tensor!
00393             */

```

```

00393         t1 = tt + FUNHEAD + 4;
00394         t2 = tt + tt[1];
00395         while ( t1 < t2 ) {
00396             if ( *t1 == -INDEX || *t1 == -VECTOR ) {
00397                 *w++ = t1[1]; t1 += 2;
00398             }
00399             else if ( *t1 == -MINVECTOR ) {
00400                 *w++ = t1[1]; t1 += 2; sign2 = -sign2;
00401             }
00402             else if ( ( *t1 == -SNUMBER ) && ( t1[1] >= 0 ) && ( t1[1] < AM.OffsetIndex ) ) {
00403                 *w++ = t1[1]; t1 += 2; sign2 = -sign2;
00404             }
00405             else {
00406                 MLOCK(ErrorMessageLock);
00407                 MesPrint("Illegal attempt to use a non-index-like argument in a tensor in
ReplaceLoop statement");
00408                 MUNLOCK(ErrorMessageLock);
00409                 Terminate(-1);
00410             }
00411         }
00412     }
00413 }
00414 }
00415 else {
00416     if ( ten >= TENSORFUNCTION ) {
00417         for ( i = 0; i < L; i++ ) {
00418             alist = wi + i*numargs;
00419             m = loc[i] + FUNHEAD;
00420             for ( k = 0; k < numargs; k++ ) {
00421                 if ( m[k] == alist[0] ) {
00422                     if ( k != 0 ) {
00423                         jj = m[k]; m[k] = m[0]; m[0] = jj;
00424                         sign = -sign;
00425                         break;
00426                     }
00427                 }
00428             }
00429             for ( k = 1; k < numargs; k++ ) {
00430                 if ( m[k] == alist[1] ) {
00431                     if ( k != 1 ) {
00432                         jj = m[k]; m[k] = m[1]; m[1] = jj;
00433                         sign = -sign;
00434                         break;
00435                     }
00436                 }
00437             }
00438             m += 2;
00439             for ( k = 2; k < numargs; k++ ) {
00440                 if ( *m >= AM.OffsetIndex ) { *w++ = -INDEX; }
00441                 else if ( *m < 0 ) { *w++ = -VECTOR; }
00442                 else { *w = -SNUMBER; }
00443                 *w++ = *m++;
00444             }
00445         }
00446     }
00447     else {
00448         WORD *t1, *t2, *t3;
00449         for ( i = 0; i < L; i++ ) {
00450             alist = wi + i*numargs;
00451             tt = loc[i];
00452             m = tt + FUNHEAD;
00453             for ( k = 0; k < numargs; k++ ) {
00454                 if ( *m == -INDEX && m[1] == alist[0] ) {
00455                     if ( k != 0 ) {
00456                         if ( ( k & 1 ) != 0 ) sign = -sign;
00457                     /*
00458                         now move to position 0
00459                     */
00460                     t2 = m+2; t1 = m; t3 = tt+FUNHEAD;
00461                     while ( t1 > t3 ) { *--t2 = *--t1; }
00462                     t3[0] = -INDEX; t3[1] = alist[0];
00463                 }
00464                 break;
00465             }
00466             NEXTARG(m)
00467         }
00468         m = tt + FUNHEAD + 2;
00469         for ( k = 1; k < numargs; k++ ) {
00470             if ( *m == -INDEX && m[1] == alist[1] ) {
00471                 if ( k != 1 ) {
00472                     if ( ( k & 1 ) == 0 ) sign = -sign;
00473                 /*
00474                     now move to position 1
00475                 */
00476                 t2 = m+2; t1 = m; t3 = tt+FUNHEAD+2;
00477                 while ( t1 > t3 ) { *--t2 = *--t1; }
00478                 t3[0] = -INDEX; t3[1] = alist[1];

```

```

00479             }
00480             break;
00481         }
00482         NEXTARG(m)
00483     }
00484     /*
00485         now copy the remaining arguments to w
00486     */
00487     t1 = tt + FUNHEAD + 4;
00488     t2 = tt + tt[1];
00489     while ( t1 < t2 ) *w++ = *t1++;
00490 }
00491 }
00492 }
00493 r[1] = w-r;
00494 while ( t < tstop ) {
00495     if ( *t == -1 ) { t += t[1]; continue; }
00496     i = t[1];
00497     NCOPY(w,t,i)
00498 }
00499 tstop = term + *term;
00500 while ( t < tstop ) *w++ = *t++;
00501 if ( sign < 0 ) w[-1] = -w[-1];
00502 i = w - AT.WorkPointer;
00503 *AT.WorkPointer = i;
00504 t = term; w = AT.WorkPointer;
00505 NCOPY(t,w,i)
00506 *AN.RepPoint = 1; /* For Repeat */
00507 return(1);
00508 }
00509
00510 /*
00511 #] Lus :
00512 #[ FindLus :
00513 */
00514
00515 int FindLus(int from, int level, int openindex)
00516 {
00517     GETIDENTITY
00518     int i, j, k, jj, *alist, *blist, *w, *m, partner;
00519     WORD **loc = AN.funlocs, *wor;
00520     if ( level == 1 ) {
00521         for ( i = from; i < AN.numoffuns; i++ ) {
00522             alist = AN.funargs + i*AN.nargs;
00523             for ( j = 0; j < AN.nargs; j++ ) {
00524                 if ( alist[j] == openindex ) {
00525                     for ( k = 0; k < AN.nargs; k++ ) {
00526                         if ( k == j ) continue;
00527                         if ( alist[k] == AN.tohunt ) {
00528                             loc[from] = loc[i];
00529                             alist = AN.funargs + from*AN.nargs;
00530                             alist[0] = openindex; alist[1] = AN.tohunt;
00531                             return(1);
00532                         }
00533                     }
00534                 }
00535             }
00536         }
00537     }
00538     else {
00539         for ( i = from; i < AN.numoffuns; i++ ) {
00540             alist = AN.funargs + i*AN.nargs;
00541             for ( j = 0; j < AN.nargs; j++ ) {
00542                 if ( alist[j] == openindex ) {
00543                     if ( from != i ) {
00544                         wor = loc[i]; loc[i] = loc[from]; loc[from] = wor;
00545                         blist = w = AN.funargs + from*AN.nargs;
00546                         m = alist;
00547                         k = AN.nargs;
00548                         while ( --k >= 0 ) { jj = *m; *m++ = *w; *w++ = jj; }
00549                     }
00550                     else blist = alist;
00551                     for ( k = 0; k < AN.nargs; k++ ) {
00552                         if ( k == j || blist[k] < 0 ) continue;
00553                         partner = blist[k];
00554                         if ( FindLus(from+1, level-1, partner) ) {
00555                             blist[0] = openindex; blist[1] = partner;
00556                             return(1);
00557                         }
00558                     }
00559                     if ( from != i ) {
00560                         wor = loc[i]; loc[i] = loc[from]; loc[from] = wor;
00561                         w = AN.funargs + from*AN.nargs;
00562                         m = alist;
00563                         k = AN.nargs;
00564                         while ( --k >= 0 ) { jj = *m; *m++ = *w; *w++ = jj; }
00565                     }
00566                 }
00567             }
00568         }
00569     }
00570 }

```

```

00566     }
00567     }
00568     }
00569 }
00570 return(0);
00571 }
00572
00573 /*
00574 #] FindLus :
00575 #[ SortTheList :
00576 */
00577
00578 int SortTheList(int *slist, int num)
00579 {
00580     GETIDENTITY
00581     int i, nleft, nright, *t1, *t2, *t3, *rlist;
00582     if ( num <= 2 ) {
00583         if ( num <= 1 ) return(num);
00584         if ( slist[0] < slist[2] ) return(2);
00585         if ( slist[0] > slist[2] ) {
00586             i = slist[0]; slist[0] = slist[2]; slist[2] = i;
00587             i = slist[1]; slist[1] = slist[3]; slist[3] = i;
00588             return(2);
00589         }
00590         slist[1] += slist[3];
00591         return(1);
00592     }
00593     else {
00594         nleft = num/2; rlist = slist + 2*nleft;
00595         nright = SortTheList(rlist,num-nleft);
00596         nleft = SortTheList(slist,nleft);
00597         if ( AN.tlistsize < nleft ) {
00598             if ( AN.tlistbuf ) M_free(AN.tlistbuf,"AN.tlistbuf");
00599             AN.tlistsize = (nleft*3)/2;
00600             AN.tlistbuf = (int *)Malloc1(AN.tlistsize*2*sizeof(int),"AN.tlistbuf");
00601         }
00602         i = nleft; t1 = slist; t2 = AN.tlistbuf;
00603         while ( --i >= 0 ) { *t2++ = *t1++; *t2++ = *t1++; }
00604         i = nleft+nright; t1 = AN.tlistbuf; t2 = rlist; t3 = slist;
00605         while ( nleft > 0 && nright > 0 ) {
00606             if ( *t1 < *t2 ) {
00607                 *t3++ = *t1++; *t3++ = *t1++; nleft--;
00608             }
00609             else if ( *t1 > *t2 ) {
00610                 *t3++ = *t2++; *t3++ = *t2++; nright--;
00611             }
00612             else {
00613                 *t3++ = *t1++; t2++; *t3++ = (*t1++) + (*t2++); i--;
00614                 nleft--; nright--;
00615             }
00616         }
00617         while ( --nleft >= 0 ) { *t3++ = *t1++; *t3++ = *t1++; }
00618         while ( --nright >= 0 ) { *t3++ = *t2++; *t3++ = *t2++; }
00619         return(i);
00620     }
00621 }
00622
00623 /*
00624 #] SortTheList :
00625 #[ AllLoops :
00626
00627 Routine finds all possible loops that can be made in the
00628 arguments of the symmetric commuting vertex function v and creates
00629 for each loop a new term which is the original term times the loop.
00630 The occurrences of v can have different numbers of arguments.
00631 Each argument that occurs twice (and in different instances of v)
00632 in total will be considered.
00633
00634 The input parameters are in C->lhs[level] and are
00635 C->lhs[level][0] TYPEALLLOOPS
00636 C->lhs[level][1] 6
00637 C->lhs[level][2] Number of function v.
00638 C->lhs[level][3] Number of the output function loop.
00639 C->lhs[level][4] option1: the type of argument.
00640 C->lhs[level][5] option2: 0,1: what to do if no loop.
00641 Eligible arguments must be of the type SYMBOL, VECTOR, INDEX or SNUMBER.
00642 They must all be of the same type, indicated by option1.
00643 Extra restriction: In a loop, each v can be visited at most once.
00644 If there is no loop at all, option2 determines whether the term
00645 remains unmodified, or is replaced by zero.
00646 Function v can be either a regular function or a tensor.
00647 To facilitate this we copy the relevant arguments into the workspace.
00648 */
00649
00650 int AllLoops(PHEAD WORD *term,WORD level)
00651 {
00652     CBUF *C = cbuf+AM.rbufnum;

```



```

00653     WORD vcode = C->lhs[level][2];      /* The input function */
00654     WORD option1 = C->lhs[level][4];     /* type of argument */
00655     WORD option2 = C->lhs[level][5];     /* what to do when no loop */
00656     WORD *tstop, *t, *tend, *tstart, *tfrom;
00657     WORD i, j, jj;
00658     WORD *arglist, nargs, *loop, nloop;
00659     WORD *oldworkpointer = AT.WorkPointer;
00660     LONG oldpworkpointer = AT.pWorkPointer;
00661     LONG numgenerated = 0, vert;
00662     WORD *a, *a1, *a2, *a3, *v, *vv, nvert, *to, *from, *tos, action;
00663 /*
00664     Search for the first occurrence of vcode.
00665 */
00666     tstop = term+term; tstop -= ABS(tstop[-1]);
00667     t = term + 1;
00668     while ( t < tstop && *t != vcode ) t += t[1];
00669     if ( t == tstop ) {
00670         if ( option2 == 0 ) return(0);
00671         else return(Generator(BHEAD term,level));
00672     }
00673     tstart = t;
00674     nvert = 0;
00675     do {
00676         t += t[1]; nvert++;
00677     } while ( t < tstop && *t == vcode );
00678     tend = t;
00679 /*
00680     Make room for 2*nvert pointers
00681 */
00682     WantAddPointers(2*nvert);
00683     vert = AT.pWorkPointer;
00684     AT.pWorkPointer += 2*nvert;
00685     nvert = 0;
00686 /*
00687     Next we copy these functions into the workspace, but only the arguments
00688     that agree with option1. Because of the difference between tensors and
00689     regular functions in the copy we strip the type of the argument.
00690 */
00691     to = AT.WorkPointer;
00692     from = tstart;
00693     if ( functions[vcode-FUNCTION].spec == TENSORFUNCTION ) {
00694         from = tstart;
00695         while ( from < tend ) {
00696             tos = to++;
00697             tfrom = from+from[1];
00698             from += FUNHEAD;
00699             while ( from < tfrom ) {
00700                 if ( option1 == -INDEX && *from >= 0 ) {
00701                     *to++ = *from++;
00702                 }
00703                 else if ( option1 == -VECTOR && *from < 0 ) {
00704                     *to++ = *from++ - AM.OffsetVector;
00705                 }
00706                 else {
00707                     from++;
00708                 }
00709             }
00710             *tos = to-from;
00711             if ( *tos < 3 ) to = tos;
00712             else AT.pWorkspace[vert+nvert++] = tos;
00713         }
00714     }
00715     else if ( functions[vcode-FUNCTION].spec == 0 ) {
00716         from = tstart;
00717         while ( from < tend ) {
00718             tos = to++;
00719             tfrom = from + from[1];
00720             from += FUNHEAD;
00721             while ( from < tfrom ) {
00722                 if ( option1 == -VECTOR
00723                     && ( from[0] == -INDEX || from[0] == -VECTOR )
00724                     && from[1] < 0 ) {
00725                     from++;
00726                     *to++ = *from++ - AM.OffsetVector;
00727                 }
00728                 else if ( option1 == *from ) {
00729                     from++; *to++ = *from++;
00730                 }
00731                 else {
00732                     NEXTARG(from);
00733                 }
00734             }
00735             *tos = to-tos;
00736             if ( *tos < 3 ) to = tos;
00737             else AT.pWorkspace[vert+nvert++] = tos;
00738         }
00739     }

```

```

00740     else {
00741         AT.WorkPointer = oldworkpointer;
00742         AT.pWorkPointer = oldpworkpointer;
00743         if ( option2 == 0 ) return(0);
00744         return(Generator(BHEAD term,level));
00745     }
00746     AT.WorkPointer = to;
00747     /*
00748     Now make a list of all relevant arguments.
00749     */
00750     a = arglist = AT.WorkPointer;
00751     for ( i = 0; i < nvert; i++ ) {
00752         v = AT.pWorkSpace[vert+i];
00753         j = *v-1; v++;
00754         NCOPY(a,v,j);
00755     }
00756     nargs = a-arglist;
00757     AT.WorkPointer = a;
00758     /*
00759     Now sort the list. Bubble type sort.
00760     */
00761     a1 = arglist; a2 = a1+1;
00762     while ( a2 < a ) {
00763         a3 = a2;
00764         while ( a3 > a1 && a3[-1] > a3[0] ) {
00765             EXCH(*a3,a3[-1]);
00766             a3--;
00767         }
00768         a2++;
00769     }
00770     /*
00771     Now remove the elements from the list that do not occur exactly twice.
00772     */
00773     a1 = arglist; a2 = a1; a3 = a1+nargs;
00774     while ( a2 < a3 ) {
00775         if ( a2+1 == a3 ) { break; }
00776         else if ( a2+2 == a3 ) {
00777             if ( a2[0] == a2[1] ) { *a1++ = a2[0]; }
00778             break;
00779         }
00780         else {
00781             if ( a2[0] != a2[1] ) { a2++; }
00782             else if ( a2[0] != a2[2] ) {
00783                 *a1++ = a2[0]; a2 += 2;
00784             }
00785             else {
00786                 a2++; while ( a2 < a3 && a2[-1] == a2[0] ) a2++;
00787             }
00788         }
00789     }
00790     nargs = a1-arglist;
00791     /*
00792     Now we need to redo the list of vertices and remove the elements
00793     that are not in our arglist.
00794     */
00795     do {
00796         action = 0;
00797         for ( i = 0; i < nvert; i++ ) {
00798             vv = v = AT.pWorkSpace[vert+i];
00799             v++;
00800             for ( j = 1; j < vv[0]; j++ ) {
00801                 for ( jj = 0; jj < nargs; jj++ ) {
00802                     if ( *v == arglist[jj] ) break;
00803                 }
00804                 if ( jj >= nargs ) { /* was not in the list */
00805                     vv[0] = vv[0]-1;
00806                     for ( jj = j; jj < vv[0]; jj++ ) vv[jj] = vv[jj+1];
00807                 }
00808                 v++;
00809             }
00810         }
00811     } /*
00812     Next we need to remove vertices that have only one object remaining.
00813     Also, the object can be removed from arglist. After that we go back
00814     to clean up the list.
00815     */
00816     for ( i = 0; i < nvert; i++ ) {
00817         vv = AT.pWorkSpace[vert+i];
00818         if ( vv[0] == 1 ) {
00819             AT.pWorkSpace[vert+i] = AT.pWorkSpace[vert+nvert-1];
00820             nvert--; i--; continue;
00821         }
00822         else if ( vv[0] == 2 ) {
00823             for ( j = 0; j < nargs; j++ ) {
00824                 if ( arglist[j] == vv[1] ) break;
00825             }
00826             while ( j < nargs-1 ) arglist[j] = arglist[j+1];

```

```

00827         nargs--;
00828         AT.pWorkSpace[vert+i] = AT.pWorkSpace[vert+nvert-1];
00829         nvert--; i--;
00830         action = 1;
00831     }
00832 }
00833 } while ( action );
00834 /*
00835     Because the user can remove tadpoles rather easily as in
00836         id v(?a,i?,?b,i?,?c) = v(?a,?b,?c)
00837     there is no need to avoid them as potential loops.
00838
00839     At this point we are ready to look for loops.
00840     We have
00841         arglist,nargs    list of eligible objects.
00842         vert,nvert       position in pWorkSpace for vertices and their number.
00843                         The vertices themselves are in the WorkSpace.
00844         loop,nloop       The buildup of the loop.
00845 */
00846     loop = AT.WorkPointer;
00847     AT.WorkPointer += nargs;
00848     nloop = 0;
00849     numgenerated += StartLoops(BHEAD term,level,vert,nvert,arglist,nargs,loop,nloop);
00850     AT.WorkPointer = oldworkpointer;
00851     AT.pWorkPointer = oldpworkpointer;
00852
00853     if ( numgenerated == 0 && option2 != 0 ) return(Generator(BHEAD term,level));
00854     return(0);
00855 }
00856
00857 /*
00858     #] AllLoops :
00859     #[ StartLoops :
00860
00861     Algorithm.
00862     1: have a list of vertices and objects that can be part of the loops.
00863     2: starting with the first element of arglist, look for the two
00864         vertices that contain this object. The first is vstart.
00865     3: At the second, look at all possible objects to continue from here.
00866     4: For each, find the vertex with its partner.
00867     5: if the partner vertex is vstart, we have a loop. --> 9.
00868     6: The partner vertex is not allowed to be a vertex that we passed
00869         in the current build-up of the loop.
00870     7: Put the object in loop and increase nloop.
00871     8: Now sum over all allowed objects in the partner vertex. --> 4.
00872
00873     9: Create the function loop with the arguments from loop,nloop.
00874     10: If a reverse cyclic permutation gives a smaller result, drop this
00875         solution and continue with the last summing over objects at a vertex.
00876     11: If not, output the result by adding the function loop
00877         to the term and call Generator(term,level)
00878
00879     12: when all possibilities with the first element of arglist have been
00880         exhausted, raise arglist and lower nargs by one. --> 2
00881     13: when nargs == 1, we can stop.
00882
00883     The inclusion of a new object when we sum over the possibilities at a
00884     vertex can best be done in a recursion, because we have no idea how
00885     many nested loops we would otherwise need. Of course the recursion can
00886     be emulated, but that makes the algorithm rather messy.
00887
00888     We have two routines: StartLoops and GenLoops.
00889     StartLoops takes care of the first argument (and the summing over who
00890     is the first argument), while GenLoops treats an additional vertex until
00891     a loop is closed. GenLoops also sends completed loops off to Generator.
00892 */
00893
00894 LONG StartLoops(PHEAD WORD *term,WORD level,LONG vert,WORD nvert,
00895     WORD *arglist,WORD nargs,WORD *loop,WORD nloop)
00896 {
00897     LONG numgenerated = 0;
00898     WORD *v, *vv, *vstart, istart, *vpartner, ipartner, j;
00899     while ( nargs > 1 ) {
00900 /*
00901         Look for a vertex with arglist[0]. This is our starting vertex.
00902         Next we look for its partner and we put arglist[0] in loop.
00903 */
00904         nloop = 0;
00905         loop[nloop++] = arglist[0];
00906         for ( istart = 0; istart < nvert; istart++ ) {
00907             vstart = AT.pWorkSpace[vert+istart];
00908             v = vstart+1; vv = vstart + *vstart;
00909             do {
00910                 if ( *v == arglist[0] ) goto havestart;
00911                 v++;
00912             } while ( v < vv );
00913         }

```

```

00914 /*
00915     If we come here, we have a problem.
00916 */
00917     MesPrint("Internal error in StartLoops. Object not found.");
00918     Terminate(-1);
00919     return(-1);
00920 havestart:
00921     AT.pWorkSpace[vert+nvert] = vstart;
00922 /*
00923     Check for tadpole.
00924 */
00925     v++;
00926     while ( v < vv ) {
00927         if ( *v == arglist[0] ) { /* tadpole */
00928             LoopOutput(BHEAD term,level,loop,nloop);
00929             numgenerated++;
00930             goto nextarg;
00931         }
00932         v++;
00933     }
00934 /*
00935     Now the partner vertex.
00936 */
00937     for ( ipartner = istart+1; ipartner < nvert; ipartner++ ) {
00938         vpartner = AT.pWorkSpace[vert+ipartner];
00939         vv = vpartner+vpartner; v = vpartner+1;
00940         do {
00941             if ( *v == arglist[0] ) goto havepartner;
00942             v++;
00943         } while ( v < vv );
00944     }
00945     return(numgenerated);
00946 havepartner:
00947     AT.pWorkSpace[vert+nvert+1] = vpartner;
00948 /*
00949     Now we run through all other possibilities at vpartner.
00950     vv is still OK.
00951 */
00952     v = vpartner+1;
00953     while ( v < vv ) {
00954         if ( *v != arglist[0] ) {
00955             for ( j = 1; j < nargs; j++ ) {
00956                 if ( *v == arglist[j] ) {
00957                     loop[nloop++] = *v;
00958                     numgenerated += GenLoops(BHEAD term,level,vert,nvert,
00959                                             arglist,nargs,loop,nloop);
00960                     nloop--;
00961                     break;
00962                 }
00963             }
00964         }
00965         v++;
00966     }
00967 nextarg:
00968     arglist++; nargs--;
00969 }
00970 return(numgenerated);
00971 }
00972
00973 /*
00974 #] StartLoops :
00975 #[ GenLoops :
00976
00977 We enter with an open line in loop[nloop-1]
00978 */
00979
00980 LONG GenLoops(PHEAD WORD *term,WORD level,WORD vert,WORD nvert,
00981               WORD *arglist,WORD nargs,WORD *loop,WORD nloop)
00982 {
00983     LONG numgenerated = 0;
00984     WORD *vstart, *v, *vv, i, j, *vpartner;
00985 /*
00986     Start with checking whether the partner is in vstart (=vert[nvert])
00987 */
00988     vstart = AT.pWorkSpace[nvert];
00989     vv = vstart + *vstart; v = vstart+1;
00990     while ( v < vv ) {
00991         if ( *v == loop[nloop-1] ) {
00992 /*
00993             This closes the loop.
00994             Now we can output it.
00995 */
00996             LoopOutput(BHEAD term,level,loop,nloop);
00997             numgenerated++;
00998             return(numgenerated);
00999         }
01000         v++;

```

```

01001     }
01002  /*
01003     Start with finding the partner.
01004  */
01005     for ( i = 0; i < nvert; i++ ) {
01006         vpartner = AT.pWorkSpace[vert+i];
01007         if ( vpartner == vstart ) continue;
01008         for ( j = 0; j < nloop; j++ ) {
01009             if ( vpartner == AT.pWorkSpace[vert+nvert+j] ) break;
01010         }
01011         if ( j < nloop ) continue;
01012         v = vpartner+1; vv = vpartner + *vpartner;
01013         while ( v < vv ) {
01014             if ( *v == loop[nloop-1] ) {
01015  /*
01016                 Found the partner.
01017                 Now we can sum over the remaining permitted arguments of this vertex.
01018  */
01019                 v = vpartner + 1;
01020                 while ( v < vv ) {
01021  /*
01022                     *v should be in arglist.
01023  */
01024                     for ( j = 0; j < nargs; j++ ) {
01025                         if ( *v == arglist[j] ) break;
01026                     }
01027                     if ( j >= nargs ) { v++; continue; }
01028  /*
01029                     *v should not be in loops.
01030  */
01031                     for ( j = 0; j < nloop; j++ ) {
01032                         if ( *v == loop[j] ) break;
01033                     }
01034                     if ( j >= nloop ) {
01035                         AT.pWorkSpace[vert+nvert+nloop] = vpartner;
01036                         loop[nloop++] = *v;
01037                         numgenerated += GenLoops(BHEAD term,level,vert,nvert,
01038                                                 arglist,nargs,loop,nloop);
01039                         nloop--;
01040                     }
01041                     v++;
01042                 }
01043                 return(numgenerated);
01044             }
01045             v++;
01046         }
01047     }
01048  /*
01049     The partner is in a vertex we have finished before.
01050  */
01051     return(numgenerated);
01052 }
01053
01054  /*
01055     #] GenLoops :
01056     #[ LoopOutput :
01057  */
01058
01059 void LoopOutput(PHEAD WORD *term, WORD level, WORD *loop, WORD nloop)
01060 {
01061     CBUF *C = cbuf+AM.rbufnum;
01062     WORD loopfun = C->lhs[level][3]; /* The output function */
01063     WORD option1 = C->lhs[level][4]; /* type of argument */
01064     WORD *tstop, *tstop1, *t, *tt;
01065     WORD *outterm, *loop1;
01066     WORD i;
01067     tstop1 = term+*term; tstop = tstop1 - ABS(tstop1[-1]);
01068  /*
01069     Construct the rcycle symmetrized version of loop.
01070  */
01071     if ( nloop > 2 ) {
01072         loop1 = AT.WorkPointer;
01073         loop1[0] = loop[0];
01074         for ( i = 1; i < nloop; i++ ) { loop1[i] = loop[nloop-i]; }
01075         if ( loop1[1] < loop[1] ) {
01076             AT.WorkPointer += nloop;
01077             loop = loop1;
01078         }
01079     }
01080     outterm = AT.WorkPointer;
01081     tt = outterm; t = term;
01082     while ( t < tstop ) *tt++ = *t++;
01083     *tt++ = loopfun;
01084     if ( functions[loopfun-FUNCTION].spec == TENSORFUNCTION ) {
01085         *tt++ = FUNHEAD+nloop;
01086         FILLFUN(tt)
01087         if ( option1 == -VECTOR ) {

```

```

01088         for ( i = 0; i < nloop; i++ ) *tt++ = loop[i]+AM.OffsetVector;
01089     }
01090     else {
01091         for ( i = 0; i < nloop; i++ ) *tt++ = loop[i];
01092     }
01093 }
01094 else {
01095     *tt++ = FUNHEAD+nloop*2;
01096     FILLFUN(tt)
01097     for ( i = 0; i < nloop; i++ ) {
01098         *tt++ = option1;
01099         if ( option1 == -VECTOR ) *tt++ = loop[i] + AM.OffsetVector;
01100         else *tt++ = loop[i];
01101     }
01102 }
01103 while ( t < tstop1 ) *tt++ = *t++;
01104 *outterm = tt - outterm;
01105 AT.WorkPointer = tt;
01106 if ( Generator(BHEAD outterm,level) ) {
01107     MesCall("LoopOutput");
01108     Terminate(-1);
01109 }
01110 AT.WorkPointer = outterm;
01111 }
01112
01113 /*
01114 #] LoopOutput :
01115 #[ AllPaths :
01116
01117 This routine has many similarities with AllLoops.
01118 In AllLoops we have startingpoint and endpoint at the same vertex.
01119 In AllPaths the startingpoint and the endpoints are different and
01120 given in advance. This endfun can occur only twice.
01121 */
01122
01123 int AllPaths(PHEAD WORD *term,WORD level)
01124 {
01125     CBUF *C = cbuf+AM.rbufnum;
01126     WORD endcode = C->lhs[level][2]; /* The endpoint function */
01127     WORD vcode = C->lhs[level][3]; /* The intermediate function */
01128     WORD option1 = C->lhs[level][5]; /* type of argument */
01129     WORD option2 = C->lhs[level][6]; /* what to do when no loop */
01130     WORD *t, *tstop, *tendl, *tend2, *tstart, *tend, numend, nvert, npass;
01131     WORD *tfrom, *to, *tos, *from;
01132     WORD *arglist, nargs, *path, npath, *a, *a1, *a2, *a3;
01133     WORD i, j, jj, *v, *vv, action;
01134     LONG vert,vert1; /* ,vert2; */
01135     WORD numgenerated = 0;
01136     WORD *oldworkpointer = AT.WorkPointer;
01137     LONG oldpworkpointer = AT.pWorkPointer;
01138 /*
01139 Search for the two occurrences of endcode.
01140 */
01141 tstop = term+*term; tstop -= ABS(tstop[-1]);
01142 t = term + 1;
01143 while ( t < tstop && *t != endcode ) t += t[1];
01144 if ( t == tstop ) { /* no endpoints */
01145     if ( option2 == 0 ) return(0);
01146     else return(Generator(BHEAD term,level));
01147 }
01148 tendl = t;
01149 numend = 0;
01150 while ( t < tstop && *t == endcode ) { tend2 = t; t += t[1]; numend++; }
01151 if ( numend != 2 ) { /* not 2 endpoints -> no path */
01152     if ( option2 == 0 ) return(0);
01153     else return(Generator(BHEAD term,level));
01154 }
01155 /*
01156 Search for the intermediate functions.
01157 */
01158 t = term + 1;
01159 nvert = 0;
01160 while ( t < tstop && *t != vcode ) t += t[1];
01161 tstart = t;
01162 while ( t < tstop && *t == vcode ) {
01163     t += t[1]; nvert++;
01164 }
01165 tend = t;
01166 /*
01167 Next we copy the relevant content of the various functions into the
01168 Workspace. We keep pointers to the functions in pWorkspace.
01169 First the two endpoints and then the intermediate ones.
01170 */
01171 WantAddPointers((2*nvert+8));
01172 vert = AT.pWorkPointer+2;
01173 vert1 = AT.pWorkPointer;
01174 /* vert2 = AT.pWorkPointer+1; */

```

```

01175     AT.pWorkPointer += 2*nvert+8;
01176     nvert = 0;
01177  /*
01178     Copy the endpoints.
01179  */
01180     to = AT.WorkPointer;
01181
01182     if ( functions[endcode-FUNCTION].spec == TENSORFUNCTION ) {
01183         from = tend1; npass = 0;
01184     redo1:
01185         tos = to++;
01186         tfrom = from+from[1];
01187         from += FUNHEAD;
01188         while ( from < tfrom ) {
01189             if ( option1 == -INDEX && *from >= 0 ) {
01190                 *to++ = *from++;
01191             }
01192             else if ( option1 == -VECTOR && *from < 0 ) {
01193                 *to++ = *from++ - AM.OffsetVector;
01194             }
01195             else {
01196                 from++;
01197             }
01198         }
01199         *tos = to-tos;
01200         if ( *tos < 2 ) to = tos;
01201         else AT.pWorkSpace[vert1+npass] = tos;
01202         npass++;
01203         if ( from == tend2 ) goto redo1;
01204     }
01205     else if ( functions[endcode-FUNCTION].spec == 0 ) {
01206         from = tend1;
01207         npass = 0;
01208     redo2:
01209         tos = to++;
01210         tfrom = from + from[1];
01211         from += FUNHEAD;
01212         while ( from < tfrom ) {
01213             if ( option1 == -VECTOR
01214                 && ( from[0] == -INDEX || from[0] == -VECTOR )
01215                 && from[1] < 0 ) {
01216                 from++;
01217                 *to++ = *from++ - AM.OffsetVector;
01218             }
01219             else if ( option1 == *from ) {
01220                 from++; *to++ = *from++;
01221             }
01222             else {
01223                 NEXTARG(from);
01224             }
01225         }
01226         *tos = to-tos;
01227         if ( *tos < 2 ) to = tos;
01228         else AT.pWorkSpace[vert1+npass] = tos;
01229         npass++;
01230         if ( from == tend2 ) goto redo2;
01231     }
01232     else {
01233         AT.WorkPointer = oldworkpointer;
01234         AT.pWorkPointer = oldpworkpointer;
01235         if ( option2 == 0 ) return(0);
01236         return(Generator(BHEAD term,level));
01237     }
01238  /*
01239     And now the intermediate functions
01240  */
01241     from = tstart;
01242     if ( functions[vcode-FUNCTION].spec == TENSORFUNCTION ) {
01243         from = tstart;
01244         while ( from < tend ) {
01245             tos = to++;
01246             tfrom = from+from[1];
01247             from += FUNHEAD;
01248             while ( from < tfrom ) {
01249                 if ( option1 == -INDEX && *from >= 0 ) {
01250                     *to++ = *from++;
01251                 }
01252                 else if ( option1 == -VECTOR && *from < 0 ) {
01253                     *to++ = *from++ - AM.OffsetVector;
01254                 }
01255                 else {
01256                     from++;
01257                 }
01258             }
01259             *tos = to-tos;
01260             if ( *tos < 3 ) to = tos;
01261             else AT.pWorkSpace[vert+nvert++] = tos;

```

```

01262     }
01263 }
01264 else if ( functions[vcode-FUNCTION].spec == 0 ) {
01265     from = tstart;
01266     while ( from < tend ) {
01267         tos = to++;
01268         tfrom = from + from[1];
01269         from += FUNHEAD;
01270         while ( from < tfrom ) {
01271             if ( option1 == -VECTOR
01272                 && ( from[0] == -INDEX || from[0] == -VECTOR )
01273                 && from[1] < 0 ) {
01274                 from++;
01275                 *to++ = *from++ - AM.OffsetVector;
01276             }
01277             else if ( option1 == *from ) {
01278                 from++; *to++ = *from++;
01279             }
01280             else {
01281                 NEXTARG(from);
01282             }
01283         }
01284         *tos = to-tos;
01285         if ( *tos < 3 ) to = tos;
01286         else AT.pWorkSpace[vert+nvert++] = tos;
01287     }
01288 }
01289 else {
01290     AT.WorkPointer = oldworkpointer;
01291     AT.pWorkPointer = oldpworkpointer;
01292     if ( option2 == 0 ) return(0);
01293     return(Generator(BHEAD term,level));
01294 }
01295 AT.WorkPointer = to;
01296 /*
01297 Now make a list of all relevant arguments.
01298 */
01299 a = arglist = AT.WorkPointer;
01300 for ( i = -2; i < nvert; i++ ) {
01301     v = AT.pWorkSpace[vert+i];
01302     j = *v-1; v++;
01303     NCOPY(a,v,j);
01304 }
01305 nargs = a-arglist;
01306 AT.WorkPointer = a;
01307 /*
01308 Now sort the list. Bubble type sort.
01309 */
01310 a1 = arglist; a2 = a1+1;
01311 while ( a2 < a ) {
01312     a3 = a2;
01313     while ( a3 > a1 && a3[-1] > a3[0] ) {
01314         EXCH(*a3,a3[-1]);
01315         a3--;
01316     }
01317     a2++;
01318 }
01319 /*
01320 Now remove the elements from the list that do not occur exactly twice.
01321 */
01322 a1 = arglist; a2 = a1; a3 = a1+nargs;
01323 while ( a2 < a3 ) {
01324     if ( a2+1 == a3 ) { break; }
01325     else if ( a2+2 == a3 ) {
01326         if ( a2[0] == a2[1] ) { *a1++ = a2[0]; }
01327         break;
01328     }
01329     else {
01330         if ( a2[0] != a2[1] ) { a2++; }
01331         else if ( a2[0] != a2[2] ) {
01332             *a1++ = a2[0]; a2 += 2;
01333         }
01334         else {
01335             a2++; while ( a2 < a3 && a2[-1] == a2[0] ) a2++;
01336         }
01337     }
01338 }
01339 nargs = a1-arglist;
01340 /*
01341 Now we need to redo the list of vertices and remove the elements
01342 that are not in our arglist.
01343 */
01344 do {
01345     action = 0;
01346     for ( i = -2; i < nvert; i++ ) {
01347         vv = v = AT.pWorkSpace[vert+i];
01348         v++;

```



```

01349         for ( j = 1; j < vv[0]; j++ ) {
01350             for ( jj = 0; jj < nargs; jj++ ) {
01351                 if ( *v == arglist[jj] ) break;
01352             }
01353             if ( jj >= nargs ) { /* was not in the list */
01354                 vv[0] = vv[0]-1;
01355                 for ( jj = j; jj < vv[0]; jj++ ) vv[jj] = vv[jj+1];
01356             }
01357             v++;
01358         }
01359     }
01360 /*
01361     Next we need to remove vertices that have only one object remaining.
01362     Also, the object can be removed from arglist. After that we go back
01363     to clean up the list.
01364 */
01365     for ( i = -2; i < nvert; i++ ) {
01366         vv = AT.pWorkSpace[vert+i];
01367         if ( vv[0] == 1 ) {
01368             AT.pWorkSpace[vert+i] = AT.pWorkSpace[vert+nvert-1];
01369             nvert--; i--; continue;
01370         }
01371         else if ( vv[0] == 2 && i >= 0 ) {
01372             for ( j = 0; j < nargs; j++ ) {
01373                 if ( arglist[j] == vv[1] ) break;
01374             }
01375             while ( j < nargs-1 ) arglist[j] = arglist[j+1];
01376             nargs--;
01377             AT.pWorkSpace[vert+i] = AT.pWorkSpace[vert+nvert-1];
01378             nvert--; i--;
01379             action = 1;
01380         }
01381     }
01382     } while ( action );
01383 /*
01384     Now we have a clean list of connections and we can start building
01385     the path. We start with summing over all objects in vert1.
01386 */
01387     path = AT.WorkPointer; npath = 0;
01388     AT.WorkPointer += nvert+8;
01389
01390     t = AT.pWorkSpace[vert1];
01391     if ( *t >= 2 ) {
01392         for ( i = 1; i < *t; i++ ) {
01393             AT.pWorkSpace[vert+nvert] = t;
01394             path[npath++] = t[i];
01395             numgenerated += GenPaths(BHEAD term, level, vert, nvert, arglist, nargs, path, npath);
01396             npath--;
01397         }
01398     }
01399     AT.WorkPointer = oldworkpointer;
01400     AT.pWorkPointer = oldpworkpointer;
01401     if ( numgenerated == 0 && option2 != 0 ) return(Generator(BHEAD term, level));
01402     return(0);
01403 }
01404
01405 /*
01406     #] AllPaths :
01407     #[ GenPaths :
01408
01409     We enter with an open line in path[npath-1]
01410     We traverse though the vertices until we reach vert-1, the endpoint.
01411 */
01412
01413 LONG GenPaths(PHEAD WORD *term, WORD level, LONG vert, WORD nvert,
01414     WORD *arglist, WORD nargs, WORD *path, WORD npath)
01415 {
01416     LONG numgenerated = 0;
01417     WORD *t, *vpartner, *v, *vv;
01418     WORD i, j;
01419 /*
01420     Check whether path[npath-1] is part of the endpoint.
01421 */
01422     t = AT.pWorkSpace[vert-1];
01423     for ( i = 1; i < *t; i++ ) {
01424         if ( t[i] == path[npath-1] ) { /* Got a path! */
01425             PathOutput(BHEAD term, level, path, npath);
01426             numgenerated++;
01427             return(numgenerated);
01428         }
01429     }
01430 /*
01431     Start with finding the partner.
01432 */
01433     for ( i = 0; i < nvert; i++ ) {
01434         vpartner = AT.pWorkSpace[vert+i];
01435         for ( j = 0; j < npath; j++ ) {

```

```

01436         if ( vpartner == AT.pWorkSpace[vert+nvert+j] ) break;
01437     }
01438     if ( j < npath ) continue;
01439     v = vpartner+1; vv = vpartner + *vpartner;
01440     while ( v < vv ) {
01441         if ( *v == path[npath-1] ) {
01442             /*
01443                 Found the partner.
01444                 Now we can sum over the remaining permitted arguments of this vertex.
01445             */
01446             v = vpartner + 1;
01447             while ( v < vv ) {
01448                 /*
01449                     *v should be in arglist.
01450                 */
01451                 for ( j = 0; j < nargs; j++ ) {
01452                     if ( *v == arglist[j] ) break;
01453                 }
01454                 if ( j >= nargs ) { v++; continue; }
01455             /*
01456                 *v should not be in path.
01457             */
01458             for ( j = 0; j < npath; j++ ) {
01459                 if ( *v == path[j] ) break;
01460             }
01461             if ( j >= npath ) {
01462                 AT.pWorkSpace[vert+nvert+npath] = vpartner;
01463                 path[npath++] = *v;
01464                 numgenerated += GenPaths(BHEAD term, level, vert, nvert,
01465                     arglist, nargs, path, npath);
01466                 npath--;
01467             }
01468             v++;
01469         }
01470         return(numgenerated);
01471     }
01472     v++;
01473 }
01474 }
01475 /*
01476 The partner is in a vertex we have finished before.
01477 */
01478 return(numgenerated);
01479 }
01480
01481 /*
01482 #] GenPaths :
01483 #[ PathOutput :
01484 */
01485
01486 void PathOutput(PHEAD WORD *term, WORD level, WORD *path, WORD npath)
01487 {
01488     CBUF *C = cbuf+AM.rbufnum;
01489     WORD pathfun = C->lhs[level][4]; /* The output function */
01490     WORD option1 = C->lhs[level][5]; /* type of argument */
01491     WORD *tstop, *tstop1, *t, *tt;
01492     WORD *outterm;
01493     WORD i;
01494     tstop1 = term+*term; tstop = tstop1 - ABS(tstop1[-1]);
01495     outterm = AT.WorkPointer;
01496     tt = outterm; t = term;
01497     while ( t < tstop ) *tt++ = *t++;
01498     *tt++ = pathfun;
01499     if ( functions[pathfun-FUNCTION].spec == TENSORFUNCTION ) {
01500         *tt++ = FUNHEAD+npath;
01501         FILLFUN(tt)
01502         if ( option1 == -VECTOR ) {
01503             for ( i = 0; i < npath; i++ ) *tt++ = path[i]+AM.OffsetVector;
01504         }
01505         else {
01506             for ( i = 0; i < npath; i++ ) *tt++ = path[i];
01507         }
01508     }
01509     else {
01510         *tt++ = FUNHEAD+npath*2;
01511         FILLFUN(tt)
01512         for ( i = 0; i < npath; i++ ) {
01513             *tt++ = option1;
01514             if ( option1 == -VECTOR ) *tt++ = path[i] + AM.OffsetVector;
01515             else *tt++ = path[i];
01516         }
01517     }
01518     while ( t < tstop1 ) *tt++ = *t++;
01519     *outterm = tt - outterm;
01520     AT.WorkPointer = tt;
01521     if ( Generator(BHEAD outterm, level) ) {
01522         MesCall("PathOutput");

```

```

01523     Terminate(-1);
01524 }
01525 AT.WorkPointer = outterm;
01526 }
01527
01528
01529
01530 /*
01531 #] PathOutput :
01532 #[ AllOnePI :
01533
01534 We assume a graph that has loops and is OnePI.
01535 This routine initializes the recursion that creates all onePI subgraphs.
01536
01537 Algorithm:
01538   a: have a routine that removes all bridges: RemoveBridges
01539   b: output an empty diagram.
01540   c: output the diagram itself
01541   d: cut a line, followed by removing all bridges.
01542   e: if the diagram still has lines, go to c, else go to the next line in d.
01543 In order to avoid factorial blowup, we need to prune the tree as fast as possible.
01544
01545 1: list of lines to be cut.
01546 2: once we have tried a line and removed its bridges, we do not have to
01547    try this line deeper in the tree, neither its bridges.
01548
01549
01550 1 2 3
01551   cut 1.                x
01552       cut 2 -> bridge 3    x
01553   cut 2.                x
01554       cut 3 -> bridge 1    ??? mag niet
01555   cut 3.                x
01556 Hence 1,2,3,4,5,6,7
01557   1 still to go 2,3,4,5,6,7
01558   2 still to go 3,4,5,6,7
01559   3 still to go 4,5,6,7
01560   etc.
01561 bridges: if x is bridge, we take x from the list_to_go.
01562 If we have a bridge that is a number lower than the list_to_go we skip this possibility.
01563 We reach the end when the list_to_go is empty.
01564 */
01565
01566 WORD AllOnePI(WORD *term,WORD level)
01567 {
01568     CBUF *C = cbuf+AM.rbufnum;
01569     WORD vcode = C->lhs[level][2];    /* The vertex function */
01570     WORD option1 = C->lhs[level][4];    /* type of argument */
01571 /*
01572 First we have to collect all relevant information about the diagram.
01573 We should start with removing bridges to make the real starting point onePI.
01574 */
01575     DUMMYUSE(term)
01576     DUMMYUSE(vcode)
01577     DUMMYUSE(option1)
01578     return(0);
01579 }
01580
01581 /*
01582 #] AllOnePI :
01583 #[ RemoveBridges :
01584 */
01585
01586 int RemoveBridges(void)
01587 {
01588     return(0);
01589 }
01590
01591 /*
01592 #] RemoveBridges :
01593 #[ TakeOneLine :
01594 */
01595
01596 int TakeOneLine(WORD*term,WORD level)
01597 {
01598     DUMMYUSE(term)
01599     DUMMYUSE(level)
01600     return(0);
01601 }
01602
01603 /*
01604 #] TakeOneLine :
01605 #[ OutputOnePI :
01606 */
01607
01608 int OutputOnePI(PHEAD WORD *term,WORD level)
01609 {

```

```

01610     return(Generator(BHEAD term,level));
01611 }
01612
01613 /*
01614    #] OutputOnePI :
01615 */
01616

```

4.68 mallocprotect.h

```

00001 #ifndef MALLOCPROTECT_H
00002 #define MALLOCPROTECT_H
00003
00010 /* #] License : */
00011 /*
00012  * Copyright (C) 1984-2026 J.A.M. Vermaseren
00013  * When using this file you are requested to refer to the publication
00014  * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00015  * This is considered a matter of courtesy as the development was paid
00016  * for by FOM the Dutch physics granting agency and we would like to
00017  * be able to track its scientific use to convince FOM of its value
00018  * for the community.
00019  *
00020  * This file is part of FORM.
00021  *
00022  * FORM is free software: you can redistribute it and/or modify it under the
00023  * terms of the GNU General Public License as published by the Free Software
00024  * Foundation, either version 3 of the License, or (at your option) any later
00025  * version.
00026  *
00027  * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00028  * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00029  * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00030  * details.
00031  *
00032  * You should have received a copy of the GNU General Public License along
00033  * with FORM. If not, see <http://www.gnu.org/licenses/>.
00034 */
00035 /* #] License : */
00036
00037 /*
00038    #] Documentation :
00039
00040 The enhanced malloc debugger. It intercepts access (both write AND
00041 read) to the memory outside the allocated chunk in the following
00042 manner: it prints out (to the stderr) the information about the event
00043 and goes to the spillock. The user is able to attach the debugger to
00044 the running process, investigate the problem and continue evaluation.
00045
00046 In case of error, the debugger will print to the stderr the following
00047 information:
00048 "***** PID: ###, Addr: ###, signal ### (code ###)"
00049 "Attach gdb -p ### and set var loopForever=0 in a corresponding frame to continue"
00050 or for TFORM:
00051 "Attach gdb -p ### and set var loopForever=0 in a corresponding frame
00052   of a thread with LPW id ### to continue"
00053
00054 and go to the spillock. The user may attach gdb by the command
00055 gdb -p ### and type "where" (in case of TFORM, the user should first
00056 switch to the proper thread). Then investigation of the corresponding
00057 frames should clarify the situation. In order to continue the program,
00058 the user might set (in the corresponding frame of the corresponding
00059 thread) the variable loopForever to 0. The debugger will try to remove
00060 local problem lead to the exception.
00061
00062 To activate the debugger, the macro MALLOCPROTECT should be
00063 defined. There are three possible values:
00064 #define MALLOCPROTECT -1 (any integer <0)
00065 #define MALLOCPROTECT 0
00066 #define MALLOCPROTECT 1 (any integer>0)
00067
00068 If MALLOCPROTECT < 0, the debugger will intercept any access to the
00069 memory before the allocated chunk, even one byte.
00070
00071 If MALLOCPROTECT == 0, the debugger will intercept any access to the
00072 memory before the allocated chunk, even one byte, and access to the
00073 memory page next to the allocated one.
00074
00075 If MALLOCPROTECT > 0, the debugger will intercept any access to the
00076 memory after the allocated chunk, even one byte, and access to the
00077 memory page before the allocated one.
00078
00079 The original FORM malloc debugger is able to catch errors when the

```

```

00080 allocated memory is freed improper, or if some small portion OUT of
00081 the allocated chunk is written. This debugger is a complementary
00082 one. It permits to catch situation when some small portion of memory
00083 before or after the allocated chunk is written or even just read.
00084
00085 The idea is to protect the beginning or the end of the allocated chunk
00086 for any kind of access and install the SIGSEGV handler in order to
00087 intercept the access to this memory. Moreover, the allocated memory is
00088 always immediately returned to the system so if the user tries to
00089 access the memory after it is freed then the handler is triggered,
00090 also.
00091
00092 The problem here is that we are able to allocate / protect only the
00093 whole page. So the user has to run the debugger twice: one time with
00094 the left alignment (#define MALLOCPROTECT <0), and then with the right
00095 alignment (#define MALLOCPROTECT >0). During the first run the possible
00096 errors like x[-1] will be caught, and the second run will manifest
00097 reading over the allocated ares.
00098
00099 The leftmost extra page is always allocated and protected. The size
00100 of the allocated chunk is stored in the beginning of this page.
00101
00102     #] Documentation :
00103     #[ Includes :
00104 */
00105
00106 #include <stdio.h>
00107 #include <stdlib.h>
00108 #include <unistd.h>
00109 #include <signal.h>
00110 #include <sys/mman.h>
00111
00112 #ifdef WITHPTHREADS
00113 #ifdef LINUX
00114 #include <sys/syscall.h>
00115 #endif
00116 #endif
00117
00118 /*
00119     #] Includes :
00120 */
00121
00122 static int pageSize=4096; /*the default value*/
00123
00124
00125 /*
00126     #[ segv_handler :
00127 */
00128
00129 /*The handler will be invoked on some signal (usually SIGSEGV). It
00130 will print some diagnostics to stderr and hands up in infinite loop
00131 waiting the user for debugging:*/
00132 static void segv_handler(int sig, siginfo_t *sip, void *xxx) {
00133     char *vadr;
00134     char *errStr;
00135     int actionBeforeExit=0; /* < 0 - unprotect, > 0 - map */
00136     switch(sip->si_signo) { /* All POSIX signals for which the field siginfo_t.si_addr is defined:*/
00137         case SIGILL:
00138             switch(sip->si_code) {
00139                 case ILL_ILLOPC:
00140                     errStr="SIGILL: Illegal opcode";
00141                     break;
00142                 case ILL_ILLOPN:
00143                     errStr="SIGILL: Illegal operand";
00144                     break;
00145                 case ILL_ILLADR:
00146                     errStr="SIGILL: Illegal addressing mode";
00147                     break;
00148                 case ILL_ILLTRP:
00149                     errStr="SIGILL: Illegal trap";
00150                     break;
00151                 case ILL_PRVOPC:
00152                     errStr="SIGILL: Privileged opcode";
00153                     break;
00154                 case ILL_PRVREG:
00155                     errStr="SIGILL: Privileged register";
00156                     break;
00157                 case ILL_COPROC:
00158                     errStr="SIGILL: Coprocessor error";
00159                     break;
00160                 case ILL_BADSTK:
00161                     errStr="SIGILL: Internal stack error";
00162                     break;
00163                 default:
00164                     errStr="SIGILL: Unknown signal code";
00165             } /*case SIGILL:*/
00166             break;

```

```

00167         case SIGFPE:
00168             switch(sip->si_code){
00169                 case FPE_INTDIV:
00170                     errStr="SIGFPE: Integer divide-by-zero";
00171                     break;
00172                 case FPE_INTOVF:
00173                     errStr="SIGFPE: Integer overflow";
00174                     break;
00175                 case FPE_FLTDIV:
00176                     errStr="SIGFPE: Floating point divide-by-zero";
00177                     break;
00178                 case FPE_FLTOVF:
00179                     errStr="SIGFPE: Floating point overflow";
00180                     break;
00181                 case FPE_FLTUND:
00182                     errStr="SIGFPE: Floating point underflow";
00183                     break;
00184                 case FPE_FLTRES:
00185                     errStr="SIGFPE: Floating point inexact result";
00186                     break;
00187                 case FPE_FLTINV:
00188                     errStr="SIGFPE: Invalid floating point operation";
00189                     break;
00190                 case FPE_FLTSUB:
00191                     errStr="SIGFPE: Subscript out of range";
00192                     break;
00193                 default:
00194                     errStr="SIGFPE: Unknown signal code";
00195             }/*switch(sip->si_code)*/
00196             break;
00197         case SIGSEGV:
00198             switch(sip->si_code){
00199                 case SEGV_MAPERR:
00200                     errStr="SIGSEGV: Address not mapped";
00201                     actionBeforeExit = 1;
00202                     break;
00203                 case SEGV_ACCERR:
00204                     errStr="SIGSEGV: Invalid permissions";
00205                     actionBeforeExit = -1;
00206                     break;
00207                 default:
00208                     errStr="SIGSEGV: Unknown signal code";
00209             }/*switch(sip->si_code)*/
00210             break;
00211         case SIGBUS:
00212             switch(sip->si_code){
00213                 case BUS_ADRALN:
00214                     errStr="SIGBUS: Invalid address alignment";
00215                     break;
00216                 case BUS_ADRERR:
00217                     errStr="SIGBUS: Non-existent physical address";
00218                     break;
00219                 case BUS_OBJERR:
00220                     errStr="SIGBUS: Object-specific hardware error";
00221                     break;
00222                 default:
00223                     errStr="SIGBUS: Unknown signal code";
00224             }/*switch(sip->si_code)*/
00225             break;
00226         default:
00227             errStr="Unknown signal";
00228     }/*switch(sip->si_signo)*/
00229
00230     vadr = (caddr_t)sip->si_addr;
00231     fprintf(stderr, "\n***** PID: %ld, Addr: %p, signal %s (code %d)\n",
00232         (long) getpid(), vadr, errStr, sip->si_code);
00233
00234     /*Block*/
00235     /*The process hangs up at this block.
00236     Attach gdb to investigate and continue:
00237     */
00238     volatile int loopForever=1;
00239     size_t alignedAdr=((size_t)vadr) & (~ (pageSize-1));
00240     fprintf(stderr, "    Attach gdb -p %ld and set var loopForever=0 in a corresponding frame",
00241         (long) getpid());
00242     #ifdef CORRECTCODE
00243     #ifdef WITHPTHREADS
00244     #ifdef LINUX
00245         fprintf(stderr, "\n    of a thread with LPW id %ld", (long) syscall(SYS_gettid));
00246     #else
00247         /*If the compiler fails here, just remove the next line:*/
00248         fprintf(stderr, "\n    of thread with LPW id %ld", (long) getpid());
00249     #endif
00250     #endif
00251     #endif
00252     fprintf(stderr, " to continue\n");
00253

```

```

00254
00255     while(loopForever)
00256         sleep(1);
00257
00258     if(actionBeforeExit<0)/*After changing loopForever=0, unprotect the page to continue:*/
00259         mprotect((char*)alignedAdr, pageSize, PROT_READ | PROT_WRITE);
00260     if(actionBeforeExit>0)/*After changing loopForever=0, map the page to continue:*/
00261 /*         mmap((void*)alignedAdr,pageSize,PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 0, 0); */
00262         mmap((void*)alignedAdr,pageSize,PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, 0, 0);
00263     }/*Block*/
00264 }/*segv_handler*/
00265
00266 /*
00267 #] segv_handler :
00268 */
00269
00270 /*
00271 #[ mprotectInit :
00272 */
00273
00274 static FORM_INLINE int mprotectInit(void)
00275 {
00276     struct sigaction sa;
00277     pageSize = getpagesize();
00278     sa.sa_sigaction = &segv_handler;
00279
00280     sigemptyset(&sa.sa_mask);
00281     sigaddset(&sa.sa_mask, SIGIO);
00282     sigaddset(&sa.sa_mask, SIGALRM);
00283
00284     sa.sa_flags = SA_SIGINFO;
00285
00286     if (sigaction(SIGILL, &sa, NULL)) {
00287         fprintf(stderr,"Error on assigning %s.\n","SIGILL");
00288         return SIGILL;
00289     }
00290     if (sigaction(SIGFPE, &sa, NULL)) {
00291         fprintf(stderr,"Error on assigning %s.\n","SIGFPE");
00292         return SIGFPE;
00293     }
00294     if (sigaction(SIGSEGV, &sa, NULL)) {
00295         fprintf(stderr,"Error on assigning %s.\n","SIGSEGV");
00296         return SIGSEGV;
00297     }
00298     if (sigaction(SIGBUS, &sa, NULL)) {
00299         fprintf(stderr,"Error on assigning %s.\n","SIGBUS");
00300         return SIGBUS;
00301     }
00302     return 0;
00303 }/*mprotectInit*/
00304
00305 /*
00306 #] mprotectInit :
00307 */
00308
00309 /*
00310 #[ mprotectMalloc :
00311 */
00312
00313 static void *mprotectMalloc(size_t theSize)
00314 {
00315     #if MALLOCPROTECT < 0
00316         /*Only one side is protected*/
00317         size_t nPages=1;
00318     #else
00319         /*Both sides are protected*/
00320         size_t nPages=2;
00321     #if MALLOCPROTECT > 0
00322         /*will need the original theSize*/
00323         size_t requestedSize=theSize;
00324     #endif
00325     #endif
00326
00327     char *ret=NULL;
00328     size_t *ptr;
00329
00330
00331
00332
00333     /*Align required size to the pagesize:*/
00334     if(theSize % pageSize)
00335         nPages++;
00336     theSize= (theSize/pageSize+nPages)*pageSize;
00337
00338 /*     ret=(char*)mmap(0,theSize,PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 0, 0); */
00339     ret=(char*)mmap(0,theSize,PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, 0, 0);
00340     if(ret == MAP_FAILED)

```

```

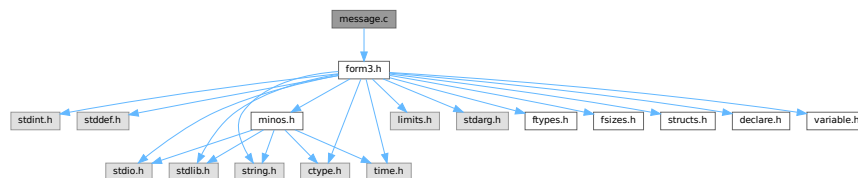
00341     return NULL;
00342     ptr=(size_t *)ret;
00343     *ptr=theSize;
00344     if(mprotect(ret, pageSize, PROT_NONE))return NULL;
00345     #if MALLOCPROTECT < 0
00346     return ret+pageSize;
00347     #else
00348     if(mprotect(ret+(theSize-pageSize), pageSize, PROT_NONE))return NULL;
00349     #if MALLOCPROTECT ==0
00350     return ret+pageSize;
00351     #endif
00352     #endif
00353     /*MALLOCPROTECT > 0, but we need conditional compilation
00354     since the variable requestedSize:*/
00355     #if MALLOCPROTECT > 0
00356
00357     /*Potential problems with alignment if the requested size is not
00358     a multiple of items. But no problems on x86-64:*/
00359     return ret+ (theSize-pageSize-requestedSize);
00360     #endif
00361 }/*mprotectMalloc*/
00362 /*
00363  #] mprotectMalloc :
00364  */
00365 /*
00366  #[ mprotectFree :
00367  */
00368 */
00369 static void mprotectFree(char *ptr)
00370 {
00371     size_t theSize;
00372     if(ptr==NULL) return;
00373     #if MALLOCPROTECT > 0
00374     /*The memory block was moved to the right, find the left page boundary:*/
00375     { /*Block*/
00376         size_t alignedPtr=((size_t)ptr) & ~(pageSize-1);
00377         ptr=(char*)alignedPtr;
00378     } /*Block*/
00379     #endif
00380     #endif
00381     ptr-=pageSize;
00382     mprotect(ptr, pageSize, PROT_READ);
00383     theSize=((size_t*)ptr);
00384     munmap(ptr,theSize);
00385 }/*mprotectFree*/
00386 /*
00387  #] mprotectFree :
00388  */
00389 */
00390 */
00391 */
00392 #endif

```

4.69 message.c File Reference

#include "form3.h"

Include dependency graph for message.c:



Functions

- NORETURN void [Error0](#) (char *s)
- NORETURN void [Error1](#) (char *s, UBYTE *t)

- NORETURN void [Error2](#) (char *s1, char *s2, UBYTE *t)
- NORETURN void [MesWork](#) (void)
- int [MesPrint](#) (const char *fmt,...)
- void [Warning](#) (char *s)
- void [HighWarning](#) (char *s)
- int [MesCall](#) (char *s)
- int [MesCerr](#) (char *s, UBYTE *t)
- int [MesComp](#) (char *s, UBYTE *p, UBYTE *q)
- void [PrintTerm](#) (WORD *term, char *where)
- void [PrintTermC](#) (WORD *term, char *where)
- void [PrintSubTerm](#) (WORD *term, char *where)
- void [PrintWords](#) (WORD *buffer, LONG number)
- void [PrintSeq](#) (WORD *a, char *text)

4.69.1 Detailed Description

Contains the routines that write messages. This includes the very important routine MesPrint which is the FORM equivalent of printf but then with escape sequences that are relevant for symbolic manipulation. The FORM statement Print "..." is passed almost literally to MesPrint.

Definition in file [message.c](#).

4.69.2 Function Documentation

4.69.2.1 Error0()

```
NORETURN void Error0 (  
    char * s )
```

Definition at line 56 of file [message.c](#).

4.69.2.2 Error1()

```
NORETURN void Error1 (  
    char * s,  
    UBYTE * t )
```

Definition at line 67 of file [message.c](#).

4.69.2.3 Error2()

```
NORETURN void Error2 (  
    char * s1,  
    char * s2,  
    UBYTE * t )
```

Definition at line 78 of file [message.c](#).

4.69.2.4 MesWork()

```
NORETURN void MesWork (
    void )
```

Definition at line [89](#) of file [message.c](#).

4.69.2.5 MesPrint()

```
int MesPrint (
    const char * fmt,
    ... )
```

Definition at line [135](#) of file [message.c](#).

4.69.2.6 Warning()

```
void Warning (
    char * s )
```

Definition at line [778](#) of file [message.c](#).

4.69.2.7 HighWarning()

```
void HighWarning (
    char * s )
```

Definition at line [790](#) of file [message.c](#).

4.69.2.8 MesCall()

```
int MesCall (
    char * s )
```

Definition at line [802](#) of file [message.c](#).

4.69.2.9 MesCerr()

```
int MesCerr (
    char * s,
    UBYTE * t )
```

Definition at line [812](#) of file [message.c](#).

4.69.2.10 MesComp()

```
int MesComp (
    char * s,
    UBYTE * p,
    UBYTE * q )
```

Definition at line 831 of file [message.c](#).

4.69.2.11 PrintTerm()

```
void PrintTerm (
    WORD * term,
    char * where )
```

Definition at line 845 of file [message.c](#).

4.69.2.12 PrintTermC()

```
void PrintTermC (
    WORD * term,
    char * where )
```

Definition at line 875 of file [message.c](#).

4.69.2.13 PrintSubTerm()

```
void PrintSubTerm (
    WORD * term,
    char * where )
```

Definition at line 909 of file [message.c](#).

4.69.2.14 PrintWords()

```
void PrintWords (
    WORD * buffer,
    LONG number )
```

Definition at line 931 of file [message.c](#).

4.69.2.15 PrintSeq()

```
void PrintSeq (
    WORD * a,
    char * text )
```

Definition at line 949 of file [message.c](#).

4.70 message.c

[Go to the documentation of this file.](#)

```

00001
00009 /* #[] License : */
00010 /*
00011 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00012 * When using this file you are requested to refer to the publication
00013 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014 * This is considered a matter of courtesy as the development was paid
00015 * for by FOM the Dutch physics granting agency and we would like to
00016 * be able to track its scientific use to convince FOM of its value
00017 * for the community.
00018 *
00019 * This file is part of FORM.
00020 *
00021 * FORM is free software: you can redistribute it and/or modify it under the
00022 * terms of the GNU General Public License as published by the Free Software
00023 * Foundation, either version 3 of the License, or (at your option) any later
00024 * version.
00025 *
00026 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00029 * details.
00030 *
00031 * You should have received a copy of the GNU General Public License along
00032 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00033 */
00034 /* #[] License : */
00035 /*
00036 #[] Includes :
00037
00038 The static variables for the messages can remain as such also for
00039 the parallel version as messages are to be locked to avoid problems
00040 with simultaneous messages.
00041 */
00042
00043 #include "form3.h"
00044
00045 static int iswarning = 0;
00046
00047 static char hex[] = {'0','1','2','3','4','5','6','7','8','9',
00048                     'A','B','C','D','E','F'};
00049
00050 /*
00051 #[] Includes :
00052 #[] exit :
00053 #[] Error0 :
00054 */
00055
00056 NORETURN void Error0(char *s)
00057 {
00058     MesPrint("=== %s",s);
00059     Terminate(-1);
00060 }
00061
00062 /*
00063 #[] Error0 :
00064 #[] Error1 :
00065 */
00066
00067 NORETURN void Error1(char *s, UBYTE *t)
00068 {
00069     MesPrint("@%s %s",s,t);
00070     Terminate(-1);
00071 }
00072
00073 /*
00074 #[] Error1 :
00075 #[] Error2 :
00076 */
00077
00078 NORETURN void Error2(char *s1, char *s2, UBYTE *t)
00079 {
00080     MesPrint("@%s%s %s",s1,s2,t);
00081     Terminate(-1);
00082 }
00083
00084 /*
00085 #[] Error2 :
00086 #[] MesWork :
00087 */
00088
00089 NORETURN void MesWork(void)

```

```

00090 {
00091     MesPrint("=== Workspace overflow. %l bytes is not enough.",AM.WorkSize);
00092     MesPrint("=== Change parameter WorkSpace in %s",setupfilename);
00093     Terminate(-1);
00094 }
00095
00096 /*
00097     #] MesWork :
00098     #[ MesPrint :
00099
00100     Kind of a printf function for simple messages.
00101     The main concern is getting the arguments in a portable way.
00102     Note: many compilers have errors when sizeof(WORD) < sizeof(int)
00103     %a array of size n WORDs (two parameters, first is int, second WORD *)
00104     %b array of size n UBYTES (two parameters, first is int, second UBYTE *)
00105     %C array of size n chars (two parameters, first is int, second char *)
00106     %d word;
00107     %l long;
00108     %L long long *;
00109     %s string;
00110     %#i unsigned word filled
00111     %#d word positioned
00112     %#l long word positioned.
00113     %#L long long word * positioned.
00114     %#s string positioned.
00115     %#p position in file.
00116     %r The current term in raw format (internal representation)
00117     %t The current term (AN.currentTerm)
00118     %T The current term (AN.currentTerm) with its sign
00119     %w Number of the thread(worker)
00120     %$ The next $ in AN.listinprint
00121     %x hexadecimal. Takes 8 places. Mainly for debugging.
00122     %% %
00123     %# #
00124     # " ==> "
00125     @ " ==> " Preprocessor error
00126     & ' --> ' Regular compiler error
00127     Each call is terminated with a new line.
00128     Put a % at the end of the string to suppress the new line.
00129
00130     New feature (7-dec-2011): The & will only work when we do not block it
00131     from the execution of the print statement because we need the & also for
00132     the tabulator in the print "" statement.
00133 */
00134
00135 int MesPrint(const char *fmt, ... )
00136 {
00137     GETIDENTITY
00138     char Out[MAXLINELENGTH+14], *stopper, *t, *s, *u, c, *carray;
00139     UBYTE extrabuffer[MAXLINELENGTH+14];
00140     int w, x, i, specialerror = 0;
00141     LONG num, y;
00142     WORD *array;
00143     UBYTE *oldoutfill = AO.OutputLine, *barray;
00144     /*[19apr2004 mt]:*/
00145     LONG (*OldWrite)(int handle, UBYTE *buffer, LONG size) = WriteFile;
00146     /*:[19apr2004 mt]*/
00147     va_list ap;
00148     va_start(ap,fmt);
00149     s = (char *)fmt;
00150 #ifdef WITHMPI
00151     /*
00152     * On slaves, if AS.printflag is
00153     *   = 0 : print nothing.
00154     *   > 0 : synchronized output. All text will be sent to the master
00155     *         in the next MUNLOCK().
00156     *   < 0 : normal output.
00157     */
00158     if ( PF.me != MASTER && AS.printflag == 0 ) return(0);
00159     if ( PF.me == MASTER || AS.printflag < 0 )
00160 #endif
00161     FLUSHCONSOLE;
00162     /*
00163     * MesPrints() never prints a message to an external channel even if
00164     * WriteFile is set to &WriteToExternalChannel.
00165     */
00166 #ifdef WITHMPI
00167     WriteFile = PF.me == MASTER || AS.printflag > 0 ? &PF_WriteFileToFile : &WriteFileToFile;
00168 #else
00169     WriteFile = &WriteFileToFile;
00170 #endif
00171     AO.OutputLine = extrabuffer;
00172     t = Out;
00173     stopper = Out + AC.LineLength;
00174     while ( *s ) {
00175         if ( ( ( *s == '&' && AO.ErrorBlock == 0 ) || *s == '@' || *s == '#' ) && AC.CurrentStream !=
00176             0 ) {

```

```

00176         u = (char *)AC.CurrentStream->name;
00177         while ( *u ) {
00178             *t++ = *u++;
00179             if ( t >= stopper ) {
00180                 num = t - Out;
00181                 WriteString(ERROROUT, (UBYTE *)Out, num);
00182                 num = 0; t = Out;
00183             }
00184         }
00185         *t++ = ' ';
00186         if ( t+20 >= stopper ) {
00187             num = t - Out;
00188             WriteString(ERROROUT, (UBYTE *)Out, num);
00189             num = 0; t = Out;
00190         }
00191         *t++ = 'L'; *t++ = 'i'; *t++ = 'n'; *t++ = 'e'; *t++ = ' ';
00192         if ( *s == '&' ) y = AC.CurrentStream->prevline;
00193         else y = AC.CurrentStream->linenumber;
00194         t = LongCopy(y, t);
00195         if ( !iswarning && ( *s == '&' || *s == '@' ) ) {
00196             for ( i = 0; i < NumDoLoops; i++ ) DoLoops[i].errorsinloop = 1;
00197         }
00198     }
00199     if ( ( *s == '&' && AO.ErrorBlock == 0 ) ) {
00200         *t++ = ' '; *t++ = '-'; *t++ = '-'; *t++ = '>'; *t++ = ' '; s++;
00201     }
00202     else if ( *s == '@' || *s == '#' ) {
00203         *t++ = ' '; *t++ = '='; *t++ = '='; *t++ = '>'; *t++ = ' '; s++;
00204     }
00205     /*
00206     else if ( *s == '&' && AO.ErrorBlock == 1 ) {
00207     }
00208     */
00209     else if ( *s != '%' ) {
00210         *t++ = *s++;
00211         if ( t >= stopper ) {
00212             num = t - Out;
00213             WriteString(ERROROUT, (UBYTE *)Out, num);
00214             num = 0; t = Out;
00215         }
00216     }
00217     else {
00218         s++;
00219         if ( *s == 'd' ) {
00220             if ( ( w = va_arg(ap, int) ) < 0 ) { *t++ = '-'; w = -w; }
00221             t = (char *)NumCopy(w, (UBYTE *)t);
00222         }
00223         else if ( *s == 'l' ) {
00224             if ( ( y = va_arg(ap, LONG) ) < 0 ) { *t++ = '-'; y = -y; }
00225             t = LongCopy(y, t);
00226         }
00227     }
00228     /* #ifdef __GLIBC_HAVE_LONG_LONG */
00229     else if ( *s == 'p' ) {
00230         POSITION *pp;
00231         off_t ly;
00232         pp = va_arg(ap, POSITION *);
00233         ly = BASEPOSITION(*pp);
00234         if ( ly < 0 ) { *t++ = '-'; ly = -ly; }
00235         /*----change 10-feb-2003 did not have & */
00236         t = LongLongCopy(&(ly), t);
00237     }
00238     /* #endif */
00239     else if ( *s == 'c' ) {
00240         c = (char)(va_arg(ap, int));
00241         *t++ = c; *t = 0;
00242     }
00243     else if ( *s == 'a' ) {
00244         w = va_arg(ap, int);
00245         array = va_arg(ap, WORD *);
00246         while ( w > 0 ) {
00247             t = (char *)NumCopy(*array, (UBYTE *)t);
00248             if ( t >= stopper ) {
00249                 num = t - Out;
00250                 WriteString(ERROROUT, (UBYTE *)Out, num);
00251                 t = Out;
00252                 *t++ = ' ';
00253             }
00254             *t++ = ' ';
00255             w--; array++;
00256         }
00257     }
00258     else if ( *s == 'b' ) {
00259         w = va_arg(ap, int);
00260         barray = va_arg(ap, UBYTE *);
00261         while ( w > 0 ) {
00262             *t++ = hex[ ((*barray) >> 4) & 0xF];

```

```

00263         *t++ = hex[( *barray)&0xF];
00264         *t = 0;
00265         if ( t >= stopper ) {
00266             num = t - Out;
00267             WriteString(ERROROUT, (UBYTE *)Out, num);
00268             t = Out;
00269             *t++ = ' ';
00270         }
00271         *t++ = ' ';
00272         w--; barray++;
00273     }
00274 }
00275 else if ( *s == 'C' ) {
00276     w = va_arg(ap, int);
00277     carray = va_arg(ap, char *);
00278     while ( w > 0 ) {
00279         if ( *carray < 32 ) *t++ = '^';
00280         else *t++ = *carray;
00281         *t = 0;
00282         if ( t >= stopper ) {
00283             num = t - Out;
00284             WriteString(ERROROUT, (UBYTE *)Out, num);
00285             t = Out;
00286             *t++ = ' ';
00287         }
00288         w--; carray++;
00289     }
00290 }
00291 else if ( *s == 'I' ) {
00292     int *iarray;
00293     w = va_arg(ap, int);
00294     iarray = va_arg(ap, int *);
00295     while ( w > 0 ) {
00296         t = (char *)LongCopy((LONG) (*iarray), (char *)t);
00297         if ( t >= stopper ) {
00298             num = t - Out;
00299             WriteString(ERROROUT, (UBYTE *)Out, num);
00300             t = Out;
00301             *t++ = ' ';
00302         }
00303         *t++ = ' ';
00304         w--; array++;
00305     }
00306 }
00307 else if ( *s == 'E' ) {
00308     LONG *larray;
00309     w = va_arg(ap, int);
00310     larray = va_arg(ap, LONG *);
00311     while ( w > 0 ) {
00312         t = (char *)LongCopy(*larray, (char *)t);
00313         if ( t >= stopper ) {
00314             num = t - Out;
00315             WriteString(ERROROUT, (UBYTE *)Out, num);
00316             t = Out;
00317             *t++ = ' ';
00318         }
00319         *t++ = ' ';
00320         w--; array++;
00321     }
00322 }
00323 else if ( *s == 's' ) {
00324     u = va_arg(ap, char *);
00325     while ( *u ) {
00326         if ( t >= stopper ) {
00327             num = t - Out;
00328             WriteString(ERROROUT, (UBYTE *)Out, num);
00329             t = Out;
00330         }
00331         *t++ = *u++;
00332     }
00333     *t = 0;
00334 }
00335 else if ( *s == 't' || *s == 'T' ) {
00336     WORD oldskip = AO.OutSkip, noleadsign;
00337     WORD oldmode = AC.OutputMode;
00338     WORD oldbracket = AO.IsBracket;
00339     WORD oldlength = AC.LineLength;
00340     UBYTE *oldStop = AO.OutStop;
00341     if ( AN.currentTerm ) {
00342         if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
00343         AO.IsBracket = 0;
00344         AO.OutSkip = 1;
00345         AC.OutputMode = 0;
00346         AO.OutFill = AO.OutputLine;
00347         AO.OutStop = AO.OutputLine + AC.LineLength;
00348         *t = 0;
00349         AddToLine((UBYTE *)Out);

```

```

00350         if ( *s == 'T' ) noleadsign = 1;
00351     else noleadsign = 0;
00352     if ( WriteInnerTerm(AN.currentTerm,noleadsign) ) Terminate(-1);
00353     t = Out;
00354     u = (char *)AO.OutputLine;
00355     *(AO.OutFill) = 0;
00356     while ( u < (char *) (AO.OutFill) ) *t++ = *u++;
00357     *t = 0;
00358     AO.OutSkip = oldskip;
00359     AC.OutputMode = oldmode;
00360     AO.IsBracket = oldbracket;
00361     AC.LineLength = oldlength;
00362     AO.OutStop = oldStop;
00363 }
00364 }
00365 else if ( *s == 'r' ) {
00366     WORD oldskip = AO.OutSkip;
00367     WORD oldmode = AC.OutputMode;
00368     WORD oldbracket = AO.IsBracket;
00369     WORD oldlength = AC.LineLength;
00370     UBYTE *oldStop = AO.OutStop;
00371     if ( AN.currentTerm ) {
00372         WORD *tt = AN.currentTerm;
00373         if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
00374         AO.IsBracket = 0;
00375         AO.OutSkip = 1;
00376         AC.OutputMode = 0;
00377         AO.OutFill = AO.OutputLine;
00378         AO.OutStop = AO.OutputLine + AC.LineLength;
00379         *t = 0;
00380         i = *tt;
00381         while ( --i >= 0 ) {
00382             t = (char *)NumCopy(*tt, (UBYTE *)t);
00383             tt++;
00384             if ( t >= stopper ) {
00385                 num = t - Out;
00386                 WriteString(ERROROUT, (UBYTE *)Out, num);
00387                 num = 0; t = Out;
00388             }
00389             *t++ = ' '; *t++ = ' ';
00390         }
00391         *t = 0;
00392         AO.OutSkip = oldskip;
00393         AC.OutputMode = oldmode;
00394         AO.IsBracket = oldbracket;
00395         AC.LineLength = oldlength;
00396         AO.OutStop = oldStop;
00397     }
00398 }
00399 else if ( *s == '$' ) {
00400 /*
00401     #[ dollars :
00402 */
00403     WORD oldskip = AO.OutSkip;
00404     WORD oldmode = AC.OutputMode;
00405     WORD oldbracket = AO.IsBracket;
00406     WORD oldlength = AC.LineLength;
00407     UBYTE *oldStop = AO.OutStop;
00408     WORD *term, indsubterm[3], *tt;
00409     WORD value[5], first, num;
00410     if ( *AN.listinprint != DOLLAREXPRESSION ) {
00411         specialerror = 1;
00412     }
00413     else {
00414         DOLLARS d = Dollars + AN.listinprint[1];
00415 #ifndef WITHPTHREADS
00416         int nummodopt, dtype;
00417         dtype = -1;
00418         if ( AS.MultiThreaded ) {
00419             for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
00420                 if ( AN.listinprint[1] == ModOptdollars[nummodopt].number ) break;
00421             }
00422             if ( nummodopt < NumModOptdollars ) {
00423                 dtype = ModOptdollars[nummodopt].type;
00424                 if ( dtype == MODLOCAL ) {
00425                     d = ModOptdollars[nummodopt].dstruct+AT.identity;
00426                 }
00427                 else {
00428                     LOCK(d->pthreadlockread);
00429                 }
00430             }
00431         }
00432 #endif
00433         AO.IsBracket = 0;
00434         AO.OutSkip = 0;
00435         AC.OutputMode = 0;
00436         AO.OutFill = AO.OutputLine;

```



```

00437         AO.OutStop = AO.OutputLine + AC.LineLength;
00438         *t = 0;
00439         AddToLine((UBYTE *)Out);
00440         if ( d->nfactors >= 1 && AN.listinprint[2] == DOLLAREXP2 ) {
00441             if ( d->type == 0 ||
00442                 ( d->factors == 0 && d->nfactors != 1 ) ) goto dollarzero;
00443             num = EvalDoLoopArg(BHEAD AN.listinprint+2,-1);
00444             if ( num == 0 ) {
00445                 value[0] = 4; value[1] = d->nfactors; value[2] = 1; value[3] = 3; value[4]
= 0;
00446                 term = value; goto printterms;
00447             }
00448             if ( num == 1 && d->nfactors == 1 ) {
00449                 term = d->where;
00450                 if ( *term == 0 ) goto dollarzero;
00451                 goto printterms;
00452             }
00453             if ( num > d->nfactors ) {
00454                 MesPrint("\nFactor number for dollar is too large.");
00455                 Terminate(-1);
00456             }
00457             term = d->factors[num-1].where;
00458             if ( term == 0 ) {
00459                 if ( d->factors[num-1].value < 0 ) {
00460                     value[0] = 4; value[1] = -d->factors[num-1].value; value[2] = 1;
value[3] = -3; value[4] = 0;
00461                 }
00462                 else {
00463                     value[0] = 4; value[1] = d->factors[num-1].value; value[2] = 1;
value[3] = 3; value[4] = 0;
00464                 }
00465                 term = value;
00466             }
00467             goto printterms;
00468         }
00469         if ( d->type == DOLTERMS || d->type == DOLNUMBER ) {
00470             term = d->where;
00471             first = 1;
00472             do {
00473                 if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
00474                 AO.IsBracket = 0;
00475                 AO.OutSkip = 1;
00476                 AC.OutputMode = 0;
00477                 AO.OutFill = AO.OutputLine;
00478                 AO.OutStop = AO.OutputLine + AC.LineLength;
00479                 *t = 0;
00480                 AddToLine((UBYTE *)Out);
00481                 if ( WriteInnerTerm(term,first) ) {
00482                     #ifdef WITHPTHREADS
00483                         if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
00484                     #endif
00485                     Terminate(-1);
00486                 }
00487                 first = 0;
00488                 t = Out;
00489                 u = (char *)AO.OutputLine;
00490                 *(AO.OutFill) = 0;
00491                 while ( u < (char *) (AO.OutFill) ) *t++ = *u++;
00492                 *t = 0;
00493                 AO.OutSkip = oldskip;
00494                 AC.OutputMode = oldmode;
00495                 AO.IsBracket = oldbracket;
00496                 AC.LineLength = oldlength;
00497                 AO.OutStop = oldStop;
00498                 term += *term;
00499             } while ( *term );
00500             AO.OutSkip = oldskip;
00501         }
00502         else if ( d->type == DOLSUBTERM ) {
00503             tt = d->where;
00504             dosubterm:
00505                 if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
00506                 AO.IsBracket = 0;
00507                 AO.OutSkip = 1;
00508                 AC.OutputMode = 0;
00509                 AO.OutFill = AO.OutputLine;
00510                 AO.OutStop = AO.OutputLine + AC.LineLength;
00511                 *t = 0;
00512                 AddToLine((UBYTE *)Out);
00513                 if ( WriteSubTerm(tt,1) ) {
00514                     #ifdef WITHPTHREADS
00515                         if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
00516                     #endif
00517                     Terminate(-1);
00518                 }
00519                 t = Out;
00520                 u = (char *)AO.OutputLine;
00521                 *(AO.OutFill) = 0;

```

```

00521         while ( u < (char *) (AO.OutFill) ) *t++ = *u++;
00522         *t = 0;
00523         AO.OutSkip = oldskip;
00524         AC.OutputMode = oldmode;
00525         AO.IsBracket = oldbracket;
00526         AC.LineLength = oldlength;
00527         AO.OutStop = oldStop;
00528     }
00529     else if ( d->type == DOLUNDEFINED ) {
00530         *t++ = '*'; *t++ = '*'; *t++ = '*'; *t = 0;
00531     }
00532     else if ( d->type == DOLZERO ) {
00533         *t++ = '0'; *t = 0;
00534     }
00535     else if ( d->type == DOLINDEX ) {
00536         tt = indsubterm; *tt = INDEX;
00537         tt[1] = 3; tt[2] = d->index;
00538         goto dosubterm;
00539     }
00540     else if ( d->type == DOLARGUMENT ) {
00541         if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
00542         AO.IsBracket = 0;
00543         AO.OutSkip = 1;
00544         AC.OutputMode = 0;
00545         AO.OutFill = AO.OutputLine;
00546         AO.OutStop = AO.OutputLine + AC.LineLength;
00547         *t = 0;
00548         AddToLine((UBYTE *)Out);
00549         WriteArgument(d->where);
00550         t = Out;
00551         u = (char *)AO.OutputLine;
00552         *(AO.OutFill) = 0;
00553         while ( u < (char *) (AO.OutFill) ) *t++ = *u++;
00554         *t = 0;
00555         AO.OutSkip = oldskip;
00556         AC.OutputMode = oldmode;
00557         AO.IsBracket = oldbracket;
00558         AC.LineLength = oldlength;
00559         AO.OutStop = oldStop;
00560     }
00561     else if ( d->type == DOLWILDARGS ) {
00562         tt = d->where;
00563         if ( *tt == 0 ) { tt++;
00564             while ( *tt ) {
00565                 if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
00566                 AO.IsBracket = 0;
00567                 AO.OutSkip = 1;
00568                 AC.OutputMode = 0;
00569                 AO.OutFill = AO.OutputLine;
00570                 AO.OutStop = AO.OutputLine + AC.LineLength;
00571                 *t = 0;
00572                 AddToLine((UBYTE *)Out);
00573                 WriteArgument(tt);
00574                 NEXTARG(tt);
00575                 if ( *tt ) TokenToLine((UBYTE *)",");
00576                 t = Out;
00577                 u = (char *)AO.OutputLine;
00578                 *(AO.OutFill) = 0;
00579                 while ( u < (char *) (AO.OutFill) ) *t++ = *u++;
00580                 *t = 0;
00581                 AO.OutSkip = oldskip;
00582                 AC.OutputMode = oldmode;
00583                 AO.IsBracket = oldbracket;
00584                 AC.LineLength = oldlength;
00585                 AO.OutStop = oldStop;
00586             }
00587         }
00588     }
00589     else if ( *tt > 0 ) { /* Tensor arguments */
00590         i = *tt++;
00591         while ( --i >= 0 ) {
00592             indsubterm[0] = INDEX;
00593             indsubterm[1] = 3;
00594             indsubterm[2] = *tt++;
00595             if ( AC.LineLength > MAXLINELENGTH ) AC.LineLength = MAXLINELENGTH;
00596             AO.IsBracket = 0;
00597             AO.OutSkip = 1;
00598             AC.OutputMode = 0;
00599             AO.OutFill = AO.OutputLine;
00600             AO.OutStop = AO.OutputLine + AC.LineLength;
00601             *t = 0;
00602             AddToLine((UBYTE *)Out);
00603             if ( WriteSubTerm(indsubterm,1) ) Terminate(-1);
00604             if ( i > 0 ) TokenToLine((UBYTE *)",");
00605             t = Out;
00606             u = (char *)AO.OutputLine;
00607             *(AO.OutFill) = 0;
00608             while ( u < (char *) (AO.OutFill) ) *t++ = *u++;

```

```

00608                                     *t = 0;
00609                                     AO.OutSkip = oldskip;
00610                                     AC.OutputMode = oldmode;
00611                                     AO.IsBracket = oldbracket;
00612                                     AC.LineLength = oldlength;
00613                                     AO.OutStop = oldStop;
00614                                     }
00615                                     }
00616                                     }
00617 #ifdef WITHPTHREADS
00618                                     if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
00619 #endif
00620                                     AN.listinprint += 2;
00621                                     while ( AN.listinprint[0] == DOLLAREXP2 ) AN.listinprint += 2;
00622                                     }
00623 /*
00624                                     #] dollars :
00625 */
00626                                     }
00627 #ifdef WITHPTHREADS
00628                                     else if ( *s == 'W' ) { /* number of the thread with time */
00629                                         LONG millitime;
00630                                         WORD timepart;
00631                                         t = (char *)NumCopy(identity, (UBYTE *)t);
00632                                         millitime = TimeCPU(1);
00633                                         timepart = (WORD)(millitime%1000);
00634                                         millitime /= 1000;
00635                                         timepart /= 10;
00636                                         *t++ = '('; *t = 0;
00637                                         t = (char *)LongCopy(millitime, (char *)t);
00638                                         *t++ = '.'; *t = 0;
00639                                         t = (char *)NumCopy(timepart, (UBYTE *)t);
00640                                         *t++ = ')'; *t = 0;
00641                                         if ( t >= stopper ) {
00642                                             num = t - Out;
00643                                             WriteString(ERROROUT, (UBYTE *)Out, num);
00644                                             num = 0; t = Out;
00645                                         }
00646                                     }
00647                                     else if ( *s == 'w' ) { /* number of the thread */
00648                                         t = (char *)NumCopy(identity, (UBYTE *)t);
00649                                     }
00650 #elif defined(WITHMPI)
00651                                     else if ( *s == 'W' ) { /* number of the thread with time */
00652                                         LONG millitime;
00653                                         WORD timepart;
00654                                         t = (char *)NumCopy(PF.me, (UBYTE *)t);
00655                                         millitime = TimeCPU(1);
00656                                         timepart = (WORD)(millitime%1000);
00657                                         millitime /= 1000;
00658                                         timepart /= 10;
00659                                         *t++ = '('; *t = 0;
00660                                         t = (char *)LongCopy(millitime, (char *)t);
00661                                         *t++ = '.'; *t = 0;
00662                                         t = (char *)NumCopy(timepart, (UBYTE *)t);
00663                                         *t++ = ')'; *t = 0;
00664                                         if ( t >= stopper ) {
00665                                             num = t - Out;
00666                                             WriteString(ERROROUT, (UBYTE *)Out, num);
00667                                             num = 0; t = Out;
00668                                         }
00669                                     }
00670                                     else if ( *s == 'w' ) { /* number of the thread */
00671                                         t = (char *)NumCopy(PF.me, (UBYTE *)t);
00672                                     }
00673 #else
00674                                     else if ( *s == 'w' ) { }
00675                                     else if ( *s == 'W' ) { }
00676 #endif
00677                                     else if ( FG.cTable[(int)*s] == 1 ) {
00678                                         x = *s++ - '0';
00679                                         while ( FG.cTable[(int)*s] == 1 )
00680                                             x = 10 * x + *s++ - '0';
00681
00682                                         if ( *s == 'l' || *s == 'd' ) {
00683                                             if ( *s == 'l' ) { y = va_arg(ap, LONG); }
00684                                             else { y = va_arg(ap, int); }
00685                                             if ( y < 0 ) { y = -y; w = 1; }
00686                                             else w = 0;
00687                                             u = t + x;
00688                                             do { *--u = y%10+'0'; y /= 10; } while ( y && u > t );
00689                                             if ( w && u > t ) *--u = '-';
00690                                             while ( --u >= t ) *u = ' ';
00691                                             t += x;
00692                                         }
00693                                     else if ( *s == 's' ) {
00694                                         u = va_arg(ap, char *);

```

```

00695         i = 0;
00696         while ( *u ) { i++; u++; }
00697         if ( i > x ) i = x;
00698         while ( x > i ) { *t++ = ' '; x--; }
00699         t += x;
00700         while ( --i >= 0 ) { *--t = *--u; }
00701         t += x;
00702     }
00703     else if ( *s == 'p' ) {
00704         POSITION *pp;
00705         /*#ifdef __GLIBC_HAVE_LONG_LONG */
00706             off_t ly;
00707         /*
00708         #else
00709             LONG ly;
00710         #endif
00711         */
00712         pp = va_arg(ap, POSITION *);
00713         ly = BASEPOSITION(*pp);
00714         u = t + x;
00715         do { *--u = ly%10+'0'; ly /= 10; } while ( ly && u > t );
00716         while ( --u >= t ) *u = ' ';
00717         t += x;
00718     }
00719     else if ( *s == 'i' ) {
00720         w = va_arg(ap, int);
00721         u = t + x;
00722         do { *--u = (char)(w%10+'0'); w /= 10; } while ( u > t );
00723         t += x;
00724     }
00725     else {
00726         w = va_arg(ap, int);
00727         u = t + x;
00728         do { *--u = (char)(w%10+'0'); w /= 10; } while ( w && u > t );
00729         while ( --u >= t ) *u = ' ';
00730         t += x;
00731     }
00732 }
00733 else if ( *s == 'x' ) {
00734     char ccc;
00735     y = va_arg(ap, LONG);
00736     i = 2*sizeof(LONG);
00737     while ( --i > 0 ) {
00738         ccc = ( y >> (i*4) ) & 0xF;
00739         if ( ccc ) break;
00740     }
00741     do {
00742         ccc = ( y >> (i*4) ) & 0xF;
00743         *t++ = hex[(int)ccc];
00744     } while ( --i >= 0 );
00745 }
00746 else if ( *s == '#' ) *t++ = *s;
00747 else if ( *s == '%' ) *t++ = *s;
00748 else if ( *s == 0 ) { *t++ = 0; break; }
00749 else if ( *s == '&' ) {
00750     *t++ = *s;
00751 }
00752 else {
00753     *t++ = '%';
00754     s--;
00755 }
00756 s++;
00757 }
00758 }
00759 num = t - Out;
00760 WriteString(ERROROUT, (UBYTE *)Out, num);
00761 va_end(ap);
00762 if ( specialerror == 1 ) {
00763     MesPrint("!!!Wrong object in Print statement!!!");
00764     MesPrint("!!!Object encountered is of a different type as in the format specifier");
00765 }
00766 AO.OutputLine = oldoutfill;
00767 /*[19apr2004 mt]:*/
00768 WriteFile=OldWrite;
00769 /*:[19apr2004 mt]*/
00770 return(-1);
00771 }
00772 /*
00773 #] MesPrint :
00774 #[ Warning :
00775 */
00776 void Warning(char *s)
00777 {
00778     iswarning = 1;
00779     if ( AC.WarnFlag ) MesPrint("&Warning: %s",s);

```

```

00782     iswarning = 0;
00783 }
00784
00785 /*
00786     #] Warning :
00787     #[ HighWarning :
00788 */
00789
00790 void HighWarning(char *s)
00791 {
00792     iswarning = 1;
00793     if ( AC.WarnFlag >= 2 ) MesPrint("&Warning: %s",s);
00794     iswarning = 0;
00795 }
00796
00797 /*
00798     #] HighWarning :
00799     #[ MesCall :
00800 */
00801
00802 int MesCall(char *s)
00803 {
00804     return(MesPrint((char *)"Called from %s",s));
00805 }
00806
00807 /*
00808     #] MesCall :
00809     #[ MesCerr :
00810 */
00811
00812 int MesCerr(char *s, UBYTE *t)
00813 {
00814     UBYTE *u, c;
00815     WORD i = 11;
00816     u = t;
00817     while ( *u && --i >= 0 ) u--;
00818     u++;
00819     c = *++t;
00820     *t = 0;
00821     MesPrint("&Illegal %s: %s",s,u);
00822     *t = c;
00823     return(-1);
00824 }
00825
00826 /*
00827     #] MesCerr :
00828     #[ MesComp :
00829 */
00830
00831 int MesComp(char *s, UBYTE *p, UBYTE *q)
00832 {
00833     UBYTE c;
00834     c = *++q; *q = 0;
00835     MesPrint("&%s: %s",s,p);
00836     *q = c;
00837     return(-1);
00838 }
00839
00840 /*
00841     #] MesComp :
00842     #[ PrintTerm :
00843 */
00844
00845 void PrintTerm(WORD *term, char *where)
00846 {
00847     UBYTE OutBuf[140];
00848     WORD *t, x;
00849     int i;
00850     AO.OutFill = AO.OutputLine = OutBuf;
00851     t = term;
00852     AO.OutSkip = 3;
00853     FiniLine();
00854     TokenToLine((UBYTE *)where);
00855     TokenToLine((UBYTE *)" ": );
00856     i = *t;
00857     while ( --i >= 0 ) {
00858         x = *t++;
00859         if ( x < 0 ) {
00860             x = -x;
00861             TokenToLine((UBYTE *)"--");
00862         }
00863         TalToLine((UWORD)(x));
00864         TokenToLine((UBYTE *)" " );
00865     }
00866     AO.OutSkip = 0;
00867     FiniLine();
00868 }

```

```

00869
00870 /*
00871     #] PrintTerm :
00872     #[ PrintTermC :
00873 */
00874
00875 void PrintTermC(WORD *term, char *where)
00876 {
00877     UBYTE OutBuf[140];
00878     WORD *t, x;
00879     int i;
00880     if ( *term >= 0 ) {
00881         PrintTerm(term,where);
00882         return;
00883     }
00884     AO.OutFill = AO.OutputLine = OutBuf;
00885     t = term;
00886     AO.OutSkip = 3;
00887     FiniLine();
00888     TokenToLine((UBYTE *)where);
00889     TokenToLine((UBYTE *)": ");
00890     i = t[1]+2;
00891     while ( --i >= 0 ) {
00892         x = *t++;
00893         if ( x < 0 ) {
00894             x = -x;
00895             TokenToLine((UBYTE *)"-");
00896         }
00897         TalToLine((UWORD)(x));
00898         TokenToLine((UBYTE *)" ");
00899     }
00900     AO.OutSkip = 0;
00901     FiniLine();
00902 }
00903
00904 /*
00905     #] PrintTermC :
00906     #[ PrintSubTerm :
00907 */
00908
00909 void PrintSubTerm(WORD *term, char *where)
00910 {
00911     UBYTE OutBuf[140];
00912     WORD *t;
00913     int i;
00914     AO.OutFill = AO.OutputLine = OutBuf;
00915     t = term;
00916     AO.OutSkip = 3;
00917     FiniLine();
00918     TokenToLine((UBYTE *)where);
00919     TokenToLine((UBYTE *)": ");
00920     i = t[1];
00921     while ( --i >= 0 ) { TalToLine((UWORD)(*t++)); TokenToLine((UBYTE *)" "); }
00922     AO.OutSkip = 0;
00923     FiniLine();
00924 }
00925
00926 /*
00927     #] PrintSubTerm :
00928     #[ PrintWords :
00929 */
00930
00931 void PrintWords(WORD *buffer, LONG number)
00932 {
00933     UBYTE OutBuf[140];
00934     WORD *t;
00935     AO.OutFill = AO.OutputLine = OutBuf;
00936     t = buffer;
00937     AO.OutSkip = 3;
00938     FiniLine();
00939     while ( --number >= 0 ) { TalToLine((UWORD)(*t++)); TokenToLine((UBYTE *)" "); }
00940     AO.OutSkip = 0;
00941     FiniLine();
00942 }
00943
00944 /*
00945     #] PrintWords :
00946     #[ PrintSeq :
00947 */
00948
00949 void PrintSeq(WORD *a,char *text)
00950 {
00951     MesPrint(" %s:",text);
00952     while ( *a ) {
00953         MesPrint("      %a",a[0],a);
00954         a += *a;
00955     }

```

```

00956 }
00957
00958 /*
00959     #] PrintSeq :
00960     #] exit :
00961 */

```

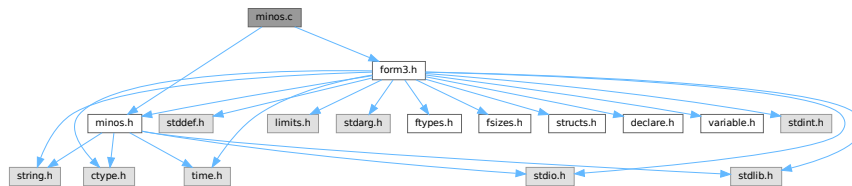
4.71 minos.c File Reference

```

#include "form3.h"
#include "minos.h"

```

Include dependency graph for minos.c:



Macros

- `#define` [CFD](#)(y, s, type, x, j)
- `#define` [CTD](#)(y, s, type, x, j)

Functions

- `int` [minosread](#) (FILE *f, char *buffer, MLONG size)
- `int` [minoswrite](#) (FILE *f, char *buffer, MLONG size)
- `char *` [str_dup](#) (char *str)
- `void` [convertblock](#) (INDEXBLOCK *in, INDEXBLOCK *out, int mode)
- `void` [convertnamesblock](#) (NAMESBLOCK *in, NAMESBLOCK *out, int mode)
- `void` [convertiniinfo](#) (INIINFO *in, INIINFO *out, int mode)
- `FILE *` [LocateBase](#) (char **name, char **newname, char *iomode)
- `int` [ReadIndex](#) (DBASE *d)
- `int` [WriteIndexBlock](#) (DBASE *d, MLONG num)
- `int` [WriteNamesBlock](#) (DBASE *d, MLONG num)
- `int` [WriteIndex](#) (DBASE *d)
- `int` [WriteIniInfo](#) (DBASE *d)
- `int` [ReadIniInfo](#) (DBASE *d)
- `DBASE *` [GetDbase](#) (char *filename, MLONG rwmode)
- `DBASE *` [NewDbase](#) (char *name, MLONG number)
- `void` [FreeTableBase](#) (DBASE *d)
- `int` [ComposeTableNames](#) (DBASE *d)
- `DBASE *` [OpenDbase](#) (char *filename)
- `MLONG` [AddTableName](#) (DBASE *d, char *name, TABLES T)
- `MLONG` [GetTableName](#) (DBASE *d, char *name)
- `int` [PutTableNames](#) (DBASE *d)
- `int` [AddToIndex](#) (DBASE *d, MLONG number)
- `MLONG` [AddObject](#) (DBASE *d, MLONG tablename, char *arguments, char *rhs)
- `MLONG` [FindTableNumber](#) (DBASE *d, char *name)
- `int` [WriteObject](#) (DBASE *d, MLONG tablename, char *arguments, char *rhs, MLONG number)
- `char *` [ReadObject](#) (DBASE *d, MLONG tablename, char *arguments)
- `char *` [ReadijObject](#) (DBASE *d, MLONG i, MLONG j, char *arguments)
- `int` [ExistsObject](#) (DBASE *d, MLONG tablename, char *arguments)
- `int` [DeleteObject](#) (DBASE *d, MLONG tablename, char *arguments)

Variables

- int `withoutflush` = 0

4.71.1 Detailed Description

These are the low level functions for the database part of the tablebases. These routines have been copied (and then adapted) from the minos database program. This file goes together with [minos.h](#)

Definition in file [minos.c](#).

4.71.2 Macro Definition Documentation

4.71.2.1 CFD

```
#define CFD(  
    y,  
    s,  
    type,  
    x,  
    j )
```

Value:

```
for(x=0, j=0; j<((int)sizeof(type)); j++) \
{ x=(x<<8)+((*(s++)&0x00FF)); y=x;
```

Definition at line 55 of file [minos.c](#).

4.71.2.2 CTD

```
#define CTD(  
    y,  
    s,  
    type,  
    x,  
    j )
```

Value:

```
x=y; for(j=sizeof(type)-1; j>=0; j--) { s[j]=x&0xFF; \
x>>=8; } s += sizeof(type);
```

Definition at line 57 of file [minos.c](#).

4.71.3 Function Documentation

4.71.3.1 minosread()

```
int minosread(  
    FILE * f,  
    char * buffer,  
    MLONG size )
```

Definition at line 66 of file [minos.c](#).

4.71.3.2 minoswrite()

```
int minoswrite (
    FILE * f,
    char * buffer,
    MLONG size )
```

Definition at line 83 of file [minos.c](#).

4.71.3.3 str_dup()

```
char * str_dup (
    char * str )
```

Definition at line 101 of file [minos.c](#).

4.71.3.4 convertblock()

```
void convertblock (
    INDEXBLOCK * in,
    INDEXBLOCK * out,
    int mode )
```

Definition at line 120 of file [minos.c](#).

4.71.3.5 convertnamesblock()

```
void convertnamesblock (
    NAMESBLOCK * in,
    NAMESBLOCK * out,
    int mode )
```

Definition at line 171 of file [minos.c](#).

4.71.3.6 convertiniinfo()

```
void convertiniinfo (
    INIINFO * in,
    INIINFO * out,
    int mode )
```

Definition at line 197 of file [minos.c](#).

4.71.3.7 LocateBase()

```
FILE * LocateBase (
    char ** name,
    char ** newname,
    char * iomode )
```

Definition at line 225 of file [minos.c](#).

4.71.3.8 ReadIndex()

```
int ReadIndex (
    DBASE * d )
```

Definition at line 288 of file [minos.c](#).

4.71.3.9 WriteIndexBlock()

```
int WriteIndexBlock (
    DBASE * d,
    MLONG num )
```

Definition at line 399 of file [minos.c](#).

4.71.3.10 WriteNamesBlock()

```
int WriteNamesBlock (
    DBASE * d,
    MLONG num )
```

Definition at line 420 of file [minos.c](#).

4.71.3.11 WriteIndex()

```
int WriteIndex (
    DBASE * d )
```

Definition at line 443 of file [minos.c](#).

4.71.3.12 WriteIniInfo()

```
int WriteIniInfo (
    DBASE * d )
```

Definition at line 506 of file [minos.c](#).

4.71.3.13 ReadIniInfo()

```
int ReadIniInfo (
    DBASE * d )
```

Definition at line 523 of file [minos.c](#).

4.71.3.14 GetDbase()

```
DBASE * GetDbase (
    char * filename,
    MLONG rwmode )
```

Definition at line 546 of file [minos.c](#).

4.71.3.15 NewDbase()

```
DBASE * NewDbase (
    char * name,
    MLONG number )
```

Definition at line 602 of file [minos.c](#).

4.71.3.16 FreeTableBase()

```
void FreeTableBase (
    DBASE * d )
```

Definition at line 739 of file [minos.c](#).

4.71.3.17 ComposeTableNames()

```
int ComposeTableNames (
    DBASE * d )
```

Definition at line 771 of file [minos.c](#).

4.71.3.18 OpenDbase()

```
DBASE * OpenDbase (
    char * filename )
```

Definition at line 820 of file [minos.c](#).

4.71.3.19 AddTableName()

```
MLONG AddTableName (
    DBASE * d,
    char * name,
    TABLES T )
```

Definition at line 854 of file [minos.c](#).

4.71.3.20 GetTableName()

```
MLONG GetTableName (
    DBASE * d,
    char * name )
```

Definition at line 937 of file [minos.c](#).

4.71.3.21 PutTableNames()

```
int PutTableNames (
    DBASE * d )
```

Definition at line 969 of file [minos.c](#).

4.71.3.22 AddToIndex()

```
int AddToIndex (
    DBASE * d,
    MLONG number )
```

Definition at line 1065 of file [minos.c](#).

4.71.3.23 AddObject()

```
MLONG AddObject (
    DBASE * d,
    MLONG tablename,
    char * arguments,
    char * rhs )
```

Definition at line 1152 of file [minos.c](#).

4.71.3.24 FindTableNumber()

```
MLONG FindTableNumber (
    DBASE * d,
    char * name )
```

Definition at line 1166 of file [minos.c](#).

4.71.3.25 WriteObject()

```
int WriteObject (
    DBASE * d,
    MLONG tablename,
    char * arguments,
    char * rhs,
    MLONG number )
```

Definition at line 1194 of file [minos.c](#).

4.71.3.26 ReadObject()

```
char * ReadObject (
    DBASE * d,
    MLONG tablenumber,
    char * arguments )
```

Definition at line 1294 of file [minos.c](#).

4.71.3.27 ReadijObject()

```
char * ReadijObject (
    DBASE * d,
    MLONG i,
    MLONG j,
    char * arguments )
```

Definition at line 1375 of file [minos.c](#).

4.71.3.28 ExistsObject()

```
int ExistsObject (
    DBASE * d,
    MLONG tablenumber,
    char * arguments )
```

Definition at line 1427 of file [minos.c](#).

4.71.3.29 DeleteObject()

```
int DeleteObject (
    DBASE * d,
    MLONG tablenumber,
    char * arguments )
```

Definition at line 1459 of file [minos.c](#).

4.71.4 Variable Documentation

4.71.4.1 withoutflush

```
int withoutflush = 0
```

Definition at line 45 of file [minos.c](#).

4.72 minos.c

[Go to the documentation of this file.](#)

```

00001
00007 /* #[ License : */
00008 /*
00009  * Copyright (C) 1984-2026 J.A.M. Vermaseren
00010  * When using this file you are requested to refer to the publication
00011  * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00012  * This is considered a matter of courtesy as the development was paid
00013  * for by FOM the Dutch physics granting agency and we would like to
00014  * be able to track its scientific use to convince FOM of its value
00015  * for the community.
00016  *
00017  * This file is part of FORM.
00018  *
00019  * FORM is free software: you can redistribute it and/or modify it under the
00020  * terms of the GNU General Public License as published by the Free Software
00021  * Foundation, either version 3 of the License, or (at your option) any later
00022  * version.
00023  *
00024  * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00025  * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00026  * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00027  * details.
00028  *
00029  * You should have received a copy of the GNU General Public License along
00030  * with FORM. If not, see <http://www.gnu.org/licenses/>.
00031 */
00032 /* #[ License : */
00033 /*
00034  *#[ Includes :
00035
00036  *File contains the lowlevel routines for the management of primitive
00037  *database-like files. The structures are explained in the file manage.h
00038  *Original file for minos made by J.Vermaseren, april-1994.
00039
00040  *Note: the minos primitives for writing are never invoked in parallel.
00041 */
00042
00043 #include "form3.h"
00044 #include "minos.h"
00045 int withoutflush = 0;
00046
00047 /*
00048  *#[ Includes :
00049  *#[ Variables :
00050 */
00051
00052 static INDEXBLOCK scratchblock;
00053 static NAMESBLOCK scratchnamesblock;
00054
00055 #define CFD(y,s,type,x,j) for(x=0,j=0;j<((int)sizeof(type));j++) \
00056 {x=(x<<8)+((*(s++)&0x00FF);} y=x;
00057 #define CTD(y,s,type,x,j) x=y;for(j=sizeof(type)-1;j>=0;j--){s[j]=x&0xFF; \
00058 x>>=8;} s += sizeof(type);
00059
00060 /*
00061  *#[ Variables :
00062  *#[ Utilities :
00063  *#[ minosread :
00064 */
00065
00066 int minosread(FILE *f,char *buffer,MLONG size)
00067 {
00068     MLONG x;
00069     while ( size > 0 ) {
00070         x = fread(buffer,sizeof(char),size,f);
00071         if ( x <= 0 ) return(-1);
00072         buffer += x;
00073         size -= x;
00074     }
00075     return(0);
00076 }
00077
00078 /*
00079  *#[ minosread :
00080  *#[ minoswrite :
00081 */
00082
00083 int minoswrite(FILE *f,char *buffer,MLONG size)
00084 {
00085     MLONG x;
00086     while ( size > 0 ) {
00087         x = fwrite(buffer,sizeof(char),size,f);

```

```

00088         if ( x <= 0 ) return(-1);
00089         buffer += x;
00090         size -= x;
00091     }
00092     if ( withoutflush == 0 ) fflush(f);
00093     return(0);
00094 }
00095
00096 /*
00097     #] minoswrite :
00098     #[ str_dup :
00099 */
00100
00101 char *str_dup(char *str)
00102 {
00103     char *s, *t;
00104     int i;
00105     s = str;
00106     while ( *s ) s++;
00107     i = s - str + 1;
00108     if ( ( s = (char *)Malloc1((size_t)i,"a string copy") ) == 0 ) return(0);
00109     t = s;
00110     while ( *str ) *t++ = *str++;
00111     *t = 0;
00112     return(s);
00113 }
00114
00115 /*
00116     #] str_dup :
00117     #[ convertblock :
00118 */
00119
00120 void convertblock(INDEXBLOCK *in,INDEXBLOCK *out,int mode)
00121 {
00122     char *s,*t;
00123     MLONG i, x;
00124     int j;
00125     OBJECTS *obj;
00126     switch ( mode ) {
00127     case TODISK:
00128         s = (char *)out;
00129         CTD(in->flags,s,MLONG,x,j)
00130         CTD(in->previousblock,s,MLONG,x,j)
00131         CTD(in->position,s,MLONG,x,j)
00132         for ( i = 0, obj = in->objects; i < NUMOBJECTS; i++, obj++ ) {
00133             CTD(obj->position,s,MLONG,x,j)
00134             CTD(obj->size,s,MLONG,x,j)
00135             CTD(obj->date,s,MLONG,x,j)
00136             CTD(obj->tablenumber,s,MLONG,x,j)
00137             CTD(obj->uncompressed,s,MLONG,x,j)
00138             CTD(obj->spare1,s,MLONG,x,j)
00139             CTD(obj->spare2,s,MLONG,x,j)
00140             CTD(obj->spare3,s,MLONG,x,j)
00141             t = obj->element;
00142             for ( j = 0; j < ELEMENTSIZE; j++ ) *s++ = *t++;
00143         }
00144         break;
00145     case FROMDISK:
00146         s = (char *)in;
00147         CFD(out->flags,s,MLONG,x,j)
00148         CFD(out->previousblock,s,MLONG,x,j)
00149         CFD(out->position,s,MLONG,x,j)
00150         for ( i = 0, obj = out->objects; i < NUMOBJECTS; i++, obj++ ) {
00151             CFD(obj->position,s,MLONG,x,j)
00152             CFD(obj->size,s,MLONG,x,j)
00153             CFD(obj->date,s,MLONG,x,j)
00154             CFD(obj->tablenumber,s,MLONG,x,j)
00155             CFD(obj->uncompressed,s,MLONG,x,j)
00156             CFD(obj->spare1,s,MLONG,x,j)
00157             CFD(obj->spare2,s,MLONG,x,j)
00158             CFD(obj->spare3,s,MLONG,x,j)
00159             t = obj->element;
00160             for ( j = 0; j < ELEMENTSIZE; j++ ) *t++ = *s++;
00161         }
00162         break;
00163     }
00164 }
00165
00166 /*
00167     #] convertblock :
00168     #[ convertnamesblock :
00169 */
00170
00171 void convertnamesblock(NAMESBLOCK *in,NAMESBLOCK *out,int mode)
00172 {
00173     char *s;
00174     MLONG x;

```

```

00175     int j;
00176     switch ( mode ) {
00177     case TODISK:
00178         s = (char *)out;
00179         CTD(in->previousblock,s,MLONG,x,j)
00180         CTD(in->position,s,MLONG,x,j)
00181         for ( j = 0; j < NAMETABLESIZE; j++ ) out->names[j] = in->names[j];
00182         break;
00183     case FROMDISK:
00184         s = (char *)in;
00185         CFD(out->previousblock,s,MLONG,x,j)
00186         CFD(out->position,s,MLONG,x,j)
00187         for ( j = 0; j < NAMETABLESIZE; j++ ) out->names[j] = in->names[j];
00188         break;
00189     }
00190 }
00191
00192 /*
00193     #] convertnamesblock :
00194     #[ convertiniinfo :
00195 */
00196
00197 void convertiniinfo(INIINFO *in,INIINFO *out,int mode)
00198 {
00199     char *s;
00200     MLONG i, x, *y;
00201     int j;
00202     switch ( mode ) {
00203     case TODISK:
00204         s = (char *)out; y = (MLONG *)in;
00205         for ( i = sizeof(INIINFO)/sizeof(MLONG); i > 0; i-- ) {
00206             CTD(*y,s,MLONG,x,j)
00207             y++;
00208         }
00209         break;
00210     case FROMDISK:
00211         s = (char *)in; y = (MLONG *)out;
00212         for ( i = sizeof(INIINFO)/sizeof(MLONG); i > 0; i-- ) {
00213             CFD(*y,s,MLONG,x,j)
00214             y++;
00215         }
00216         break;
00217     }
00218 }
00219
00220 /*
00221     #] convertiniinfo :
00222     #[ LocateBase :
00223 */
00224
00225 FILE *LocateBase(char **name, char **newname, char *iomode)
00226 {
00227     FILE *handle;
00228     int namesize, i;
00229     UBYTE *s, *to, *u1, *u2, *indir;
00230
00231     if ( ( handle = fopen(*name,iomode) ) != 0 ) {
00232         *newname = (char *)strDup1((UBYTE *)(*name),"LocateBase");
00233         return(handle);
00234     }
00235     namesize = 2; s = (UBYTE *)(*name);
00236     while ( *s ) { s++; namesize++; }
00237     indir = AM.IncDir;
00238     if ( indir ) {
00239         s = indir; i = 0;
00240         while ( *s ) { s++; i++; }
00241         *newname = (char *)Malloca1(namesize+i,"LocateBase");
00242         s = indir; to = (UBYTE *)(*newname);
00243         while ( *s ) *to++ = *s++;
00244         if ( to > (UBYTE *)(*newname) && to[-1] != SEPARATOR ) *to++ = SEPARATOR;
00245         s = (UBYTE *)(*name);
00246         while ( *s ) *to++ = *s++;
00247         *to = 0;
00248         if ( ( handle = fopen(*newname,iomode) ) != 0 ) {
00249             return(handle);
00250         }
00251         M_free(*newname,"LocateBase, incdir/file");
00252     }
00253     if ( AM.Path ) {
00254         u1 = AM.Path;
00255         while ( *u1 ) {
00256             u2 = u1; i = 0;
00257             while ( *u1 && *u1 != ':' ) {
00258                 if ( *u1 == '\\\\' ) u1++;
00259                 u1++; i++;
00260             }
00261             *newname = (char *)Malloca1(namesize+i,"LocateBase");

```



```

00262         s = u2; to = (UBYTE *) (*newname);
00263         while ( s < u1 ) {
00264             if ( *s == '\\\\' ) s++;
00265             *to++ = *s++;
00266         }
00267         if ( to > (UBYTE *) (*newname) && to[-1] != SEPARATOR ) *to++ = SEPARATOR;
00268         s = (UBYTE *) (*name);
00269         while ( *s ) *to++ = *s++;
00270         *to = 0;
00271         if ( ( handle = fopen(*newname,iomode) ) != 0 ) {
00272             return(handle);
00273         }
00274         M_free(*newname,"LocateBase Path/file");
00275         if ( *u1 ) u1++;
00276     }
00277 }
00278 /* Error1("LocateBase: Cannot find file",*name); */
00279 return(0);
00280 }
00281
00282 /*
00283     #] LocateBase :
00284     #] Utilities :
00285     #[ ReadIndex :
00286 */
00287
00288 int ReadIndex(DBASE *d)
00289 {
00290     MLONG i;
00291     INDEXBLOCK **ib;
00292     NAMESBLOCK **ina;
00293     MLONG position, size;
00294 /*
00295     Allocate the pieces one by one (makes it easier to give it back)
00296 */
00297     if ( d->info.numberofindexblocks <= 0 ) return(0);
00298     if ( sizeof(INDEXBLOCK)*d->info.numberofindexblocks > MAXINDEXSIZE ) {
00299         MesPrint("We need more than %ld bytes for the index.\n",MAXINDEXSIZE);
00300         MesPrint("The file %s may not be a proper database\n",d->name);
00301         return(-1);
00302     }
00303     size = sizeof(INDEXBLOCK *)*d->info.numberofindexblocks;
00304     if ( ( ib = (INDEXBLOCK **)Malloca1(size,"tb,index") ) == 0 ) return(-1);
00305     for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
00306         if ( ( ib[i] = (INDEXBLOCK *)Malloca1(sizeof(INDEXBLOCK),"index block") ) == 0 ) {
00307             for ( --i; i >= 0; i-- ) M_free(ib[i],"tb,indexblock");
00308             M_free(ib,"tb,index");
00309             return(-1);
00310         }
00311     }
00312     size = sizeof(NAMESBLOCK *)*d->info.numberofnamesblocks;
00313     if ( ( ina = (NAMESBLOCK **)Malloca1(size,"tb,indexnames") ) == 0 ) return(-1);
00314     for ( i = 0; i < d->info.numberofnamesblocks; i++ ) {
00315         if ( ( ina[i] = (NAMESBLOCK *)Malloca1(sizeof(NAMESBLOCK),"index names block") ) == 0 ) {
00316             for ( --i; i >= 0; i-- ) M_free(ina[i],"index names block");
00317             M_free(ina,"tb,indexnames");
00318             for ( i = 0; i < d->info.numberofindexblocks; i++ ) M_free(ib[i],"tb,indexblock");
00319             M_free(ib,"tb,index");
00320             return(-1);
00321         }
00322     }
00323 /*
00324     Read the index blocks, from the back to the front. The links are only
00325     reliable that way.
00326 */
00327     position = d->info.lastindexblock;
00328     for ( i = d->info.numberofindexblocks - 1; i >= 0; i-- ) {
00329         fseek(d->handle,position,SEEK_SET);
00330         if ( minosread(d->handle,(char *)(&scratchblock),sizeof(INDEXBLOCK)) ) {
00331             MesPrint("Error while reading file %s\n",d->name);
00332             thisiswrong:
00333             for ( i = 0; i < d->info.numberofnamesblocks; i++ ) M_free(ina[i],"index names block");
00334             M_free(ina,"tb,indexnames");
00335             for ( i = 0; i < d->info.numberofindexblocks; i++ ) M_free(ib[i],"tb,indexblock");
00336             M_free(ib,"tb,index");
00337             return(-1);
00338         }
00339         convertblock(&scratchblock,ib[i],FROMDISK);
00340         if ( ib[i]->position != position ||
00341             ( ib[i]->previousblock <= 0 && i > 0 ) ) {
00342             MesPrint("File %s has inconsistent contents\n",d->name);
00343             goto thisiswrong;
00344         }
00345         position = ib[i]->previousblock;
00346     }
00347     d->info.firstindexblock = ib[0]->position;
00348     for ( i = 0; i < d->info.numberofindexblocks; i++ ) {

```

```

00349         ib[i]->flags &= MCLEANFLAG;
00350     }
00351     /*
00352     Read the names blocks, from the back to the front. The links are only
00353     reliable that way.
00354     */
00355     position = d->info.lastnameblock;
00356     for ( i = d->info.numberofnamesblocks - 1; i >= 0; i-- ) {
00357         fseek(d->handle,position,SEEK_SET);
00358         if ( minosread(d->handle,(char *)(&scratchnamesblock),sizeof(NAMESBLOCK)) ) {
00359             MesPrint("Error while reading file %s\n",d->name);
00360             goto thisiswrong;
00361         }
00362         convertnamesblock(&scratchnamesblock,ina[i],FROMDISK);
00363         if ( ina[i]->position != position ||
00364             ( ina[i]->previousblock <= 0 && i > 0 ) ) {
00365             MesPrint("File %s has inconsistent contents\n",d->name);
00366             goto thisiswrong;
00367         }
00368         position = ina[i]->previousblock;
00369     }
00370     d->info.firstnameblock = ina[0]->position;
00371     /*
00372     Give the old info back to the system.
00373     */
00374     if ( d->iblocks ) {
00375         for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
00376             if ( d->iblocks[i] ) M_free(d->iblocks[i],"d->iblocks[i]");
00377         }
00378         M_free(d->iblocks,"d->iblocks");
00379     }
00380     if ( d->nblocks ) {
00381         for ( i = 0; i < d->info.numberofnamesblocks; i++ ) {
00382             if ( d->nblocks[i] ) M_free(d->nblocks[i],"d->nblocks[i]");
00383         }
00384         M_free(d->nblocks,"d->nblocks");
00385     }
00386     /*
00387     And substitute the new blocks
00388     */
00389     d->iblocks = ib;
00390     d->nblocks = ina;
00391     return(0);
00392 }
00393
00394 /*
00395 #] ReadIndex :
00396 #[ WriteIndexBlock :
00397 */
00398
00399 int WriteIndexBlock(DBASE *d,MLONG num)
00400 {
00401     if ( num >= d->info.numberofindexblocks ) {
00402         MesPrint("Illegal number specified for number of index blocks\n");
00403         return(-1);
00404     }
00405     fseek(d->handle,d->iblocks[num]->position,SEEK_SET);
00406     convertblock(d->iblocks[num],&scratchblock,TODISK);
00407     if ( minoswrite(d->handle,(char *)(&scratchblock),sizeof(INDEXBLOCK)) ) {
00408         MesPrint("Error while writing an index block in file %s\n",d->name);
00409         MesPrint("File may be unreliable now\n");
00410         return(-1);
00411     }
00412     return(0);
00413 }
00414
00415 /*
00416 #] WriteIndexBlock :
00417 #[ WriteNamesBlock :
00418 */
00419
00420 int WriteNamesBlock(DBASE *d,MLONG num)
00421 {
00422     if ( num >= d->info.numberofnamesblocks ) {
00423         MesPrint("Illegal number specified for number of names blocks\n");
00424         return(-1);
00425     }
00426     fseek(d->handle,d->nblocks[num]->position,SEEK_SET);
00427     convertnamesblock(d->nblocks[num],&scratchnamesblock,TODISK);
00428     if ( minoswrite(d->handle,(char *)(&scratchnamesblock),sizeof(NAMESBLOCK)) ) {
00429         MesPrint("Error while writing a names block in file %s\n",d->name);
00430         MesPrint("File may be unreliable now\n");
00431         return(-1);
00432     }
00433     return(0);
00434 }
00435

```

```

00436 /*
00437 #] WriteNamesBlock :
00438 #[ WriteIndex :
00439
00440 Problem here is to get the links right.
00441 */
00442
00443 int WriteIndex(DBASE *d)
00444 {
00445     MLONG i, position;
00446     if ( d->iblocks == 0 ) return(0);
00447     if ( d->nblocks == 0 ) return(0);
00448     for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
00449         if ( d->iblocks[i] == 0 ) {
00450             MesPrint("Error: unassigned index blocks. Cannot write\n");
00451             return(-1);
00452         }
00453     }
00454     for ( i = 0; i < d->info.numberofnamesblocks; i++ ) {
00455         if ( d->nblocks[i] == 0 ) {
00456             MesPrint("Error: unassigned names blocks. Cannot write\n");
00457             return(-1);
00458         }
00459     }
00460     d->info.lastindexblock = -1;
00461     for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
00462         position = d->iblocks[i]->position;
00463         if ( position <= 0 ) {
00464             fseek(d->handle,0,SEEK_END);
00465             position = ftell(d->handle);
00466             d->iblocks[i]->position = position;
00467             if ( i <= 0 ) d->iblocks[i]->previousblock = -1;
00468             else d->iblocks[i]->previousblock = d->iblocks[i-1]->position;
00469         }
00470         else fseek(d->handle,position,SEEK_SET);
00471         convertblock(d->iblocks[i],&scratchblock,TODISK);
00472         if ( minoswrite(d->handle,(char *)(&scratchblock),sizeof(INDEXBLOCK)) ) {
00473             MesPrint("Error while writing index of file %s",d->name);
00474             d->iblocks[i]->position = -1;
00475             return(-1);
00476         }
00477         d->info.lastindexblock = position;
00478     }
00479     d->info.lastnameblock = -1;
00480     for ( i = 0; i < d->info.numberofnamesblocks; i++ ) {
00481         position = d->nblocks[i]->position;
00482         if ( position <= 0 ) {
00483             fseek(d->handle,0,SEEK_END);
00484             position = ftell(d->handle);
00485             d->nblocks[i]->position = position;
00486             if ( i <= 0 ) d->nblocks[i]->previousblock = -1;
00487             else d->nblocks[i]->previousblock = d->nblocks[i-1]->position;
00488         }
00489         else fseek(d->handle,position,SEEK_SET);
00490         convertnamesblock(d->nblocks[i],&scratchnamesblock,TODISK);
00491         if ( minoswrite(d->handle,(char *)(&scratchnamesblock),sizeof(NAMESBLOCK)) ) {
00492             MesPrint("Error while writing index of file %s",d->name);
00493             d->nblocks[i]->position = -1;
00494             return(-1);
00495         }
00496         d->info.lastnameblock = position;
00497     }
00498     return(0);
00499 }
00500
00501 /*
00502 #] WriteIndex :
00503 #[ WriteIniInfo :
00504 */
00505
00506 int WriteIniInfo(DBASE *d)
00507 {
00508     INIINFO inf;
00509     fseek(d->handle,0,SEEK_SET);
00510     convertiniinfo(&(d->info),&inf,TODISK);
00511     if ( minoswrite(d->handle,(char *)(&inf),sizeof(INIINFO)) ) {
00512         MesPrint("Error while writing masterindex of file %s",d->name);
00513         return(-1);
00514     }
00515     return(0);
00516 }
00517
00518 /*
00519 #] WriteIniInfo :
00520 #[ ReadIniInfo :
00521 */
00522

```

```

00523 int ReadIniInfo(DBASE *d)
00524 {
00525     INIINFO inf;
00526     fseek(d->handle,0,SEEK_SET);
00527     if ( minosread(d->handle,(char *)(&inf),sizeof(INIINFO)) ) {
00528         MesPrint("Error while reading masterindex of file %s",d->name);
00529         return(-1);
00530     }
00531     convertiniinfo(&inf,&(d->info),FROMDISK);
00532     if ( d->info.entriesinindex < 0
00533         || d->info.numberofindexblocks < 0
00534         || d->info.lastindexblock < 0 ) {
00535         MesPrint("The file %s is not a proper database\n",d->name);
00536         return(-1);
00537     }
00538     return(0);
00539 }
00540
00541 /*
00542 #] ReadIniInfo :
00543 #[ GetDbase :
00544 */
00545
00546 DBASE *GetDbase(char *filename, MLONG rwmode)
00547 {
00548     FILE *f;
00549     DBASE *d;
00550     char *newname;
00551     if ( rwmode == 0 ) {
00552         if ( ( f = LocateBase(&filename,&newname,"rb") ) == 0 ) {
00553
00554             MesPrint("&Trying to open non-existent TableBase in readonly mode: %s", filename);
00555             Terminate(-1);
00556         }
00557     } else {
00558         if ( ( f = LocateBase(&filename,&newname,"r+b") ) == 0 ) {
00559
00560             return(NewDbase(filename,0));
00561         }
00562     }
00563
00564     /* setbuf(f,0); */
00565     d = (DBASE *)From0List(&(AC.TableBaseList));
00566     d->mode = 0;
00567     d->tablenamessize = 0;
00568     d->topnumber = 0;
00569     d->tablenamefill = 0;
00570     d->iblocks = 0;
00571     d->nblocks = 0;
00572     d->tablenames = 0;
00573     d->rwmode = rwmode;
00574
00575     d->info.entriesinindex = 0;
00576     d->info.numberofindexblocks = 0;
00577     d->info.firstindexblock = 0;
00578     d->info.lastindexblock = 0;
00579     d->info.numberoftables = 0;
00580     d->info.numberofnamesblocks = 0;
00581     d->info.firstnameblock = 0;
00582     d->info.lastnameblock = 0;
00583
00584     d->name = str_dup(filename); /* For the moment just for the error messages */
00585     d->handle = f;
00586     if ( ReadIniInfo(d) || ReadIndex(d) ) { M_free(d,"index-d"); fclose(f); return(0); }
00587     if ( ComposeTableNames(d) < 0 ) { FreeTableBase(d); fclose(f); return(0); }
00588     // free allocation from previous str_dup
00589     M_free(d->name, "from str_dup");
00590     d->name = str_dup(filename);
00591     d->fullname = newname;
00592     return(d);
00593 }
00594
00595 /*
00596 #] GetDbase :
00597 #[ NewDbase :
00598
00599     Creates a new database with 'number' entries in the index.
00600 */
00601
00602 DBASE *NewDbase(char *name,MLONG number)
00603 {
00604     FILE *f;
00605     DBASE *d;
00606     MLONG numblocks, numnameblocks, i;
00607     char *s;
00608     /*-----change 10-feb-2003 */
00609     int j, jj;

```

```

00610     MLONG t = (MLONG) (time(0));
00611     /*-----*/
00612     if ( number < 0 ) number = 0;
00613     if ( ( f = fopen(name,"w+b") ) == 0 ) {
00614         MesPrint("Could not create a new file with name %s\n",name);
00615         return(0);
00616     }
00617     numblocks = (number+NUMOBJECTS-1)/NUMOBJECTS;
00618     numnameblocks = 1;
00619     if ( numblocks <= 0 ) numblocks = 1;
00620     if ( numnameblocks <= 0 ) numnameblocks = 1;
00621     d = (DBASE *)From0List(&(AC.TableBaseList));
00622     if ( ( d->iblocks = (INDEXBLOCK **)Malloca1(numblocks*sizeof(INDEXBLOCK *),
00623 "new database") ) == 0 ) {
00624         NumTableBases--;
00625         return(0);
00626     }
00627     d->tablenames = 0;
00628     d->tablenamessize = 0;
00629     d->topnumber = 0;
00630     d->tablenameefill = 0;
00631     d->rwmode = 1;
00632
00633     d->mode = 0;
00634     if ( ( d->nblocks = (NAMESBLOCK **)Malloca1(sizeof(NAMESBLOCK *)*numnameblocks,
00635 "new database") ) == 0 ) {
00636         M_free(d->iblocks,"new database");
00637         NumTableBases--;
00638         return(0);
00639     }
00640     if ( ( f = fopen(name,"w+b") ) == 0 ) {
00641         MesPrint("Could not create new file %s\n",name);
00642         NumTableBases--;
00643         return(0);
00644     }
00645     /* setbuf(f,0); */
00646     d->name = str_dup(name);
00647     d->fullname = str_dup(name);
00648     d->handle = f;
00649
00650     d->info.entriesinindex = number;
00651     d->info.numberofindexblocks = numblocks;
00652     d->info.numberofnamesblocks = numnameblocks;
00653     d->info.firstindexblock = 0;
00654     d->info.lastindexblock = 0;
00655     d->info.numberoftables = 0;
00656     d->info.firstnameblock = 0;
00657     d->info.lastnameblock = 0;
00658
00659     if ( WriteIniInfo(d) ) {
00660 getout:
00661         fclose(f);
00662         remove(d->fullname);
00663         if ( d->name ) { M_free(d->name,"name tablebase"); d->name = 0; }
00664         if ( d->fullname ) { M_free(d->fullname,"fullname tablebase"); d->fullname = 0; }
00665         M_free(d->nblocks,"new database");
00666         M_free(d->iblocks,"new database");
00667         NumTableBases--;
00668         return(0);
00669     }
00670     for ( i = 0; i < numblocks; i++ ) {
00671         if ( ( d->iblocks[i] = (INDEXBLOCK *)Malloca1(sizeof(INDEXBLOCK),
00672 "index blocks of new database") ) == 0 ) {
00673             while ( --i >= 0 ) M_free(d->iblocks[i],"index blocks of new database");
00674             goto getout;
00675         }
00676         if ( i > 0 ) d->iblocks[i]->previousblock = d->iblocks[i-1]->position;
00677         else d->iblocks[i]->previousblock = -1;
00678         d->iblocks[i]->position = ftell(f);
00679         // Initialise, to keep valgrind happy
00680         d->iblocks[i]->flags = -1;
00681     /*-----change 10-feb-2003 */
00682     /*
00683         Zero things properly. We don't want garbage in the file.
00684     */
00685     for ( j = 0; j < NUMOBJECTS; j++ ) {
00686         d->iblocks[i]->objects[j].date = t;
00687         d->iblocks[i]->objects[j].size = 0;
00688         d->iblocks[i]->objects[j].position = -1;
00689         d->iblocks[i]->objects[j].tablename = 0;
00690         d->iblocks[i]->objects[j].uncompressed = 0;
00691         d->iblocks[i]->objects[j].spare1 = 0;
00692         d->iblocks[i]->objects[j].spare2 = 0;
00693         d->iblocks[i]->objects[j].spare3 = 0;
00694         for ( jj = 0; jj < ELEMENTSIZE; jj++ ) d->iblocks[i]->objects[j].element[jj] = 0;
00695     }
00696     convertblock(d->iblocks[i],&scratchblock,TODISK);

```

```

00697         if ( minoswrite(d->handle, (char *)(&scratchblock), sizeof(INDEXBLOCK)) ) {
00698             MesPrint("Error while writing new index blocks\n");
00699             goto getout;
00700         }
00701     }
00702     for ( i = 0; i < numnameblocks; i++ ) {
00703         if ( ( d->nblocks[i] = (NAMESBLOCK *)Malloc1(sizeof(NAMESBLOCK),
00704             "names blocks of new database") ) == 0 ) {
00705             while ( --i >= 0 ) { M_free(d->nblocks[i], "names blocks of new database"); }
00706             for ( i = 0; i < numblocks; i++ ) M_free(d->iblocks[i], "index blocks of new database");
00707             goto getout;
00708         }
00709         if ( i > 0 ) d->nblocks[i]->previousblock = d->nblocks[i-1]->position;
00710         else d->nblocks[i]->previousblock = -1;
00711         d->nblocks[i]->position = ftell(f);
00712         s = d->nblocks[i]->names;
00713         for ( j = 0; j < NAMETABLESIZE; j++ ) *s++ = 0;
00714         convertnamesblock(d->nblocks[i], &scratchnamesblock, TODISK);
00715         if ( minoswrite(d->handle, (char *)(&scratchnamesblock), sizeof(NAMESBLOCK)) ) {
00716             MesPrint("Error while writing new names blocks\n");
00717             for ( i = 0; i < numnameblocks; i++ ) M_free(d->nblocks[i], "names blocks of new
database");
00718             for ( i = 0; i < numblocks; i++ ) M_free(d->iblocks[i], "index blocks of new database");
00719             goto getout;
00720         }
00721     }
00722     d->info.firstindexblock = d->iblocks[0]->position;
00723     d->info.lastindexblock = d->iblocks[numblocks-1]->position;
00724     d->info.firstnameblock = d->nblocks[0]->position;
00725     d->info.lastnameblock = d->nblocks[numnameblocks-1]->position;
00726     if ( WriteIniInfo(d) ) {
00727         for ( i = 0; i < numnameblocks; i++ ) M_free(d->nblocks[i], "names blocks of new database");
00728         for ( i = 0; i < numblocks; i++ ) M_free(d->iblocks[i], "index blocks of new database");
00729         goto getout;
00730     }
00731     return(d);
00732 }
00733
00734 /*
00735 #] NewDbase :
00736 #[ FreeTableBase :
00737 */
00738
00739 void FreeTableBase(DBASE *d)
00740 {
00741     int i, j, *old, *newL;
00742     LIST *L;
00743     for ( i = 0; i < d->info.numberofnamesblocks; i++ ) M_free(d->nblocks[i], "nblocks[i]");
00744     for ( i = 0; i < d->info.numberofindexblocks; i++ ) M_free(d->iblocks[i], "iblocks[i]");
00745     M_free(d->nblocks, "nblocks");
00746     M_free(d->iblocks, "iblocks");
00747     if ( d->tablenames ) M_free(d->tablenames, "d->tablenames");
00748     if ( d->name ) M_free(d->name, "d->name");
00749     if ( d->fullname ) M_free(d->fullname, "d->fullname");
00750     i = d - tablebases;
00751     if ( i < ( NumTableBases - 1 ) ) {
00752         L = &(AC.TableBaseList);
00753         j = ( ( NumTableBases - i - 1 ) * L->size ) / sizeof(int);
00754         old = (int *)d; newL = (int *) (d+1);
00755         while ( --j >= 0 ) *newL++ = *old++;
00756         j = L->size / sizeof(int);
00757         while ( --j >= 0 ) *newL++ = 0;
00758     }
00759     NumTableBases--;
00760     M_free(d, "tb,d");
00761 }
00762
00763 /*
00764 #] FreeTableBase :
00765 #[ ComposeTableNames :
00766
00767     The nameblocks are supposed to be in memory.
00768     Hence we have to go through them
00769 */
00770
00771 int ComposeTableNames(DBASE *d)
00772 {
00773     MLONG nsize = 0;
00774     int i, j, k;
00775     char *s, *t, *ss;
00776     d->topnumber = 0;
00777     i = 0; s = d->nblocks[i]->names; j = NAMETABLESIZE;
00778     while ( *s ) {
00779         if ( *s ) d->topnumber++;
00780         for ( k = 0; k < 2; k++ ) { /* name and argtail */
00781             while ( *s ) {
00782                 j--;

```

```

00783         if ( j <= 0 ) {
00784             i++; if ( i >= d->info.numberofnamesblocks ) goto gotall;
00785             s = d->nblocks[i]->names; j = NAMETABLESIZE;
00786         }
00787         else s++;
00788     }
00789     j--;
00790     if ( j <= 0 ) {
00791         i++; if ( i >= d->info.numberofnamesblocks ) goto gotall;
00792         s = d->nblocks[i]->names; j = NAMETABLESIZE;
00793     }
00794     else s++;
00795 }
00796 }
00797 gotall:;
00798 nsize = (d->info.numberofnamesblocks-1)*NAMETABLESIZE +
00799         (s-d->nblocks[i]->names)+1;
00800 if ( ( d->tablenames = (char *)Malloc1((2*nsize+30)*sizeof(char),"tablenames") )
00801     == 0 ) { return(-1); }
00802 t = d->tablenames;
00803 d->tablenamessize = 2*nsize+30;
00804 d->tablenamefill = nsize-1;
00805 for ( k = 0; k < i; k++ ) {
00806     ss = d->nblocks[k]->names;
00807     for ( j = 0; j < NAMETABLESIZE; j++ ) *t++ = *ss++;
00808 }
00809 ss = d->nblocks[i]->names;
00810 while ( ss < s ) *t++ = *ss++;
00811 *t = 0;
00812 return(0);
00813 }
00814
00815 /*
00816 #] ComposeTableNames :
00817 #[ OpenDbase :
00818 */
00819
00820 DBASE *OpenDbase(char *filename)
00821 {
00822     FILE *f;
00823     DBASE *d;
00824     char *newname;
00825     if ( ( f = LocateBase(&filename,&newname,"r+b") ) == 0 ) {
00826         MesPrint("Cannot open file %s\n",filename);
00827         return(0);
00828     }
00829     /* setbuf(f,0); */
00830     d = (DBASE *)From0List(&(AC.TableBaseList));
00831     d->name = filename; /* For the moment just for the error messages */
00832     d->handle = f;
00833     if ( ReadIniInfo(d) || ReadIndex(d) ) { M_free(d,"OpenDbase"); fclose(f); return(0); }
00834     if ( ComposeTableNames(d) ) {
00835         FreeTableBase(d);
00836         fclose(f);
00837         return(0);
00838     }
00839     d->name = str_dup(filename);
00840     d->fullname = newname;
00841     return(d);
00842 }
00843
00844 /*
00845 #] OpenDbase :
00846 #[ AddTableName :
00847
00848 Adds a name of a table. Writes the namelist to disk.
00849 Returns the number of this tablename in the database.
00850 If the name was already in the table we return its value in negative.
00851 Zero is an error!
00852 */
00853
00854 MLONG AddTableName(DBASE *d,char *name, TABLES T)
00855 {
00856     char *s, *t, *tt;
00857     int namesize, tailsize;
00858     MLONG newsize, i, num;
00859     /*
00860     First search for the name in what we have already
00861     */
00862     if ( d->tablenames ) {
00863         num = 0;
00864         s = d->tablenames;
00865         while ( *s ) {
00866             num++;
00867             t = name;
00868             while ( ( *s == *t ) && *t ) { s++; t++; }
00869             if ( *s == *t ) { return(-num); }

```

```

00870         while ( *s ) s++;
00871         s++;
00872         while ( *s ) s++;
00873         s++;
00874     }
00875 }
00876 /*
00877 This name has to be added
00878 */
00879 MesPrint("We add the name %s\n",name);
00880 t = name;
00881 while ( *t ) { t++; }
00882 namesize = t-name;
00883 if ( ( t = (char *) (T->argtail) ) != 0 ) {
00884     while ( *t ) { t++; }
00885     tailsizsize = t - (char *) (T->argtail);
00886 }
00887 else { tailsizsize = 0; }
00888 if ( d->tablenames == 0 ) {
00889     if ( ComposeTableNames(d) ) {
00890         FreeTableBase(d);
00891         M_free(d,"AddTableName");
00892         return(0);
00893     }
00894 }
00895 d->info.numberoftables++;
00896 while ( ( d->tablenameefill+namesize+tailsizsize+3 > d->tablenamessize )
00897     || ( d->tablenames == 0 ) ) {
00898     newsizsize = 2*d->tablenamessize + 2*namesize + 2*tailsizsize + 6;
00899     if ( ( t = (char *) Malloc1(newsizsize*sizeof(char),"AddTableName") ) == 0 )
00900         return(0);
00901     tt = t;
00902     if ( d->tablenames ) {
00903         s = d->tablenames;
00904         for ( i = 0; i < d->tablenameefill; i++ ) *t++ = *s++;
00905         *t = 0;
00906         M_free(d->tablenames,"d->tablenames");
00907     }
00908     d->tablenames = tt;
00909     d->tablenamessize = newsizsize;
00910 }
00911 s = d->tablenames + d->tablenameefill;
00912 t = name;
00913 while ( *t ) *s++ = *t++;
00914 *s++ = 0;
00915 t = (char *) (T->argtail);
00916 while ( *t ) *s++ = *t++;
00917 *s++ = 0;
00918 *s = 0;
00919 d->tablenameefill = s - d->tablenames;
00920 d->topnumber++;
00921 /*
00922 Now we have to synchronize
00923 */
00924 if ( PutTableNames(d) ) return(0);
00925 return(d->topnumber);
00926 }
00927
00928 /*
00929 #] AddTableName :
00930 #[ GetTableName :
00931
00932 Gets a name of a table.
00933 Returns the number of this tablename in the database.
00934 Zero -> error
00935 */
00936
00937 MLONG GetTableName(DBASE *d,char *name)
00938 {
00939     char *s, *t;
00940     MLONG num;
00941     /*
00942     search for the name in what we have
00943     */
00944     if ( d->tablenames ) {
00945         num = 0;
00946         s = d->tablenames;
00947         while ( *s ) {
00948             num++;
00949             t = name;
00950             while ( ( *s == *t ) && *t ) { s++; t++; }
00951             if ( *s == *t ) { return(num); }
00952             while ( *s ) s++;
00953             s++;
00954             while ( *s ) s++;
00955             s++;
00956         }

```



```

00957     }
00958     return(0);
00959 }
00960
00961 /*
00962  #] GetTableName :
00963  #[ PutTableNames :
00964
00965     Takes the names string in d->tablenames and puts it in the nblocks
00966     pieces. Writes what has been changed to disk.
00967 */
00968
00969 int PutTableNames(DBASE *d)
00970 {
00971     NAMESBLOCK **nnew;
00972     int i, j, firstdif;
00973     MLONG m;
00974     char *s, *t;
00975     /*
00976     Determine how many blocks are needed.
00977     */
00978     MLONG numblocks = d->tablenamefill/NAMETABLESIZE + 1;
00979     if ( d->info.numberofnamesblocks < numblocks ) {
00980     /*
00981         We need more blocks. First make sure of the space for nblocks.
00982     */
00983         if ( ( nnew = (NAMESBLOCK **)Malloc1(sizeof(NAMESBLOCK *)*numblocks,
00984             "new names block") ) == 0 ) {
00985             return(-1);
00986         }
00987         for ( i = 0; i < d->info.numberofnamesblocks; i++ ) {
00988             nnew[i] = d->nblocks[i];
00989         }
00990         free(d->nblocks);
00991         d->nblocks = nnew;
00992         for ( ; i < numblocks; i++ ) {
00993             if ( ( d->nblocks[i] = (NAMESBLOCK *)Malloc1(sizeof(NAMESBLOCK),
00994                 "additional names blocks ") ) == 0 ) {
00995                 FreeTableBase(d);
00996                 return(-1);
00997             }
00998             d->nblocks[i]->previousblock = -1;
00999             d->nblocks[i]->position = -1;
01000             s = d->nblocks[i]->names;
01001             for ( j = 0; j < NAMETABLESIZE; j++ ) *s++ = 0;
01002         }
01003         d->info.numberofnamesblocks = numblocks;
01004     }
01005     /*
01006     Now look till where the new contents agree with the old.
01007     */
01008     firstdif = 0;
01009     i = 0; t = d->nblocks[i]->names; j = 0; s = d->tablenames;
01010     for ( m = 0; m < d->tablenamefill; m++ ) {
01011         if ( *s == *t ) {
01012             s++; t++; j++;
01013             if ( j >= NAMETABLESIZE ) {
01014                 i++;
01015                 t = d->nblocks[i]->names;
01016                 j = 0;
01017             }
01018         }
01019         else {
01020             firstdif = i;
01021             for ( ; m < d->tablenamefill; m++ ) {
01022                 *t++ = *s++; j++;
01023                 if ( j >= NAMETABLESIZE ) {
01024                     i++;
01025                     t = d->nblocks[i]->names;
01026                     j = 0;
01027                 }
01028             }
01029             *t = 0;
01030             break;
01031         }
01032     }
01033     for ( i = 0; i < d->info.numberofnamesblocks; i++ ) {
01034         if ( i == firstdif ) break;
01035         if ( d->nblocks[i]->position < 0 ) { firstdif = i; break; }
01036     }
01037     /*
01038     Now we have to (re)write the blocks, starting at firstdif.
01039     */
01040     for ( i = firstdif; i < d->info.numberofnamesblocks; i++ ) {
01041         if ( i > 0 ) d->nblocks[i]->previousblock = d->nblocks[i-1]->position;
01042         else d->nblocks[i]->previousblock = -1;
01043         if ( d->nblocks[i]->position < 0 ) {

```

```

01044         fseek(d->handle,0,SEEK_END);
01045         d->nblocks[i]->position = ftell(d->handle);
01046     }
01047     else fseek(d->handle,d->nblocks[i]->position,SEEK_SET);
01048     convertnamesblock(d->nblocks[i],&scratchnamesblock,TODISK);
01049     if ( minoswrite(d->handle,(char *)(&scratchnamesblock),sizeof(NAMESBLOCK)) ) {
01050         MesPrint("Error while writing names blocks\n");
01051         FreeTableBase(d);
01052         return(-1);
01053     }
01054 }
01055 d->info.lastnameblock = d->nblocks[d->info.numberofnamesblocks-1]->position;
01056 d->info.firstnameblock = d->nblocks[0]->position;
01057 return(WriteIniInfo(d));
01058 }
01059
01060 /*
01061 #] PutTableNames :
01062 #[ AddToIndex :
01063 */
01064
01065 int AddToIndex(DBASE *d,MLONG number)
01066 {
01067     MLONG i, oldnumofindexblocks = d->info.numberofindexblocks;
01068     MLONG j, newnumofindexblocks, jj;
01069     INDEXBLOCK **ib;
01070     MLONG t = (MLONG)(time(0));
01071     if ( number == 0 ) return(0);
01072     else if ( number < 0 ) {
01073         if ( d->info.entriesinindex < -number ) {
01074             MesPrint("There are only %ld entries in the index of file %s\n",
01075                 d->info.entriesinindex,d->name);
01076             return(-1);
01077         }
01078         d->info.entriesinindex += number;
01079     dowrite:
01080         if ( WriteIniInfo(d) ) {
01081             d->info.entriesinindex -= number;
01082             MesPrint("File may be corrupted\n");
01083             return(-1);
01084         }
01085     }
01086     else if ( d->info.entriesinindex+number <=
01087         NUMOBJECTS*d->info.numberofindexblocks ) {
01088         d->info.entriesinindex += number;
01089         goto dowrite;
01090     }
01091     else {
01092         d->info.entriesinindex += number;
01093         newnumofindexblocks = d->info.numberofindexblocks + ((number -
01094             (NUMOBJECTS*d->info.numberofindexblocks - d->info.entriesinindex))
01095             +NUMOBJECTS-1)/NUMOBJECTS;
01096         if ( ( ib = (INDEXBLOCK **)Malloc1(sizeof(INDEXBLOCK *)*newnumofindexblocks,
01097             "index") ) == 0 ) return(-1);
01098         for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01099             ib[i] = d->iblocks[i];
01100         }
01101         for ( i = d->info.numberofindexblocks; i < newnumofindexblocks; i++ ) {
01102             if ( ( ib[i] = (INDEXBLOCK *)Malloc1(sizeof(INDEXBLOCK),"index block") ) == 0 ) {
01103                 FreeTableBase(d);
01104                 return(-1);
01105             }
01106             if ( i > 0 ) ib[i]->previousblock = ib[i-1]->position;
01107             else ib[i]->previousblock = -1;
01108         }
01109     /*
01110     Zero things properly. We don't want garbage in the file.
01111     */
01112     for ( j = 0; j < NUMOBJECTS; j++ ) {
01113         ib[i]->objects[j].date = t;
01114         ib[i]->objects[j].size = 0;
01115         ib[i]->objects[j].position = -1;
01116         ib[i]->objects[j].tablennumber = 0;
01117         ib[i]->objects[j].uncompressed = 0;
01118         ib[i]->objects[j].spare1 = 0;
01119         ib[i]->objects[j].spare2 = 0;
01120         ib[i]->objects[j].spare3 = 0;
01121         for ( jj = 0; jj < ELEMENTSIZE; jj++ ) ib[i]->objects[j].element[jj] = 0;
01122     }
01123     fseek(d->handle,0,SEEK_END);
01124     ib[i]->position = ftell(d->handle);
01125     convertblock(ib[i],&scratchblock,TODISK);
01126     if ( minoswrite(d->handle,(char *)(&scratchblock),sizeof(INDEXBLOCK)) ) {
01127         MesPrint("Error while writing new index of file %s",d->name);
01128         FreeTableBase(d);
01129         return(-1);
01130     }
01131 }

```

```

01131         d->info.lastindexblock = ib[newnumofindexblocks-1]->position;
01132         d->info.firstindexblock = ib[0]->position;
01133         d->info.numberofindexblocks = newnumofindexblocks;
01134         if ( WriteIniInfo(d) ) {
01135             d->info.numberofindexblocks = oldnumofindexblocks;
01136             d->info.entriesinindex -= number;
01137             MesPrint("File may be corrupted\n");
01138             FreeTableBase(d);
01139             return(-1);
01140         }
01141         M_free(d->iblocks,"AddToIndex");
01142         d->iblocks = ib;
01143     }
01144     return(0);
01145 }
01146
01147 /*
01148 #] AddToIndex :
01149 #[ AddObject :
01150 */
01151
01152 MLONG AddObject(DBASE *d,MLONG tablenumber,char *arguments,char *rhs)
01153 {
01154     MLONG number;
01155     number = d->info.entriesinindex;
01156     if ( AddToIndex(d,1) ) return(-1);
01157     if ( WriteObject(d,tablenumber,arguments,rhs,number) ) return(-1);
01158     return(number);
01159 }
01160
01161 /*
01162 #] AddObject :
01163 #[ FindTableNumber :
01164 */
01165
01166 MLONG FindTableNumber(DBASE *d,char *name)
01167 {
01168     char *s = d->tablenames, *t, *ss;
01169     MLONG num = 0;
01170     ss = d->tablenames + d->tablenamefill;
01171     while ( s < ss ) {
01172         num++;
01173         t = name;
01174         while ( *s == *t && *t ) {
01175             s++; t++;
01176         }
01177         if ( *s == 0 && *t == 0 ) return(num);
01178         while ( *s ) s++;
01179         s++;
01180     }
01181     /* Skip also the argument tail
01182     */
01183     while ( *s ) s++;
01184     s++;
01185 }
01186 return(-1); /* Name not found */
01187 }
01188
01189 /*
01190 #] FindTableNumber :
01191 #[ WriteObject :
01192 */
01193
01194 int WriteObject(DBASE *d,MLONG tablenumber,char *arguments,char *rhs,MLONG number)
01195 {
01196     char *s, *a;
01197 #ifdef WITH2LIB
01198     char *buffer = 0;
01199     uLongf newsize = 0, oldsize = 0;
01200     uLong ssize;
01201     int error = 0;
01202 #endif
01203     MLONG i, j, position, size, n;
01204     OBJECTS *obj;
01205     if ( ( d->mode & INPUTONLY ) == INPUTONLY ) {
01206         MesPrint("Not allowed to write to input\n");
01207         return(-1);
01208     }
01209     if ( number >= d->info.entriesinindex ) {
01210         MesPrint("Reference to non-existing object number %ld\n",number+1);
01211         return(0);
01212     }
01213     j = number/NUMOBJECTS;
01214     i = number%NUMOBJECTS;
01215     obj = &(d->iblocks[j]->objects[i]);
01216     a = arguments;
01217     while ( *a ) a++;

```

```

01218     a++; n = a - arguments;
01219     if ( n > ELEMENTSIZE ) {
01220         MesPrint("Table element %s has more than %ld characters.\n",arguments,
01221             (MLONG)ELEMENTSIZE);
01222         return(-1);
01223     }
01224     s = obj->element;
01225     a = arguments;
01226     while ( *a ) *s++ = *a++;
01227     *s++ = 0;
01228     while ( n < ELEMENTSIZE ) { *s++ = 0; n++; }
01229     obj->spare1 = obj->spare2 = obj->spare3 = 0;
01230
01231     fseek(d->handle,0,SEEK_END);
01232     position = ftell(d->handle);
01233     s = rhs;
01234     while ( *s ) s++;
01235     s++;
01236     size = s - rhs;
01237 #ifdef WITHZLIB
01238     if ( ( d->mode & NOCOMPRESS ) == 0 ) {
01239         newsize = size + size/1000 + 20;
01240         if ( ( buffer = (char *)Malloca1(newsize*sizeof(char),"compress buffer") )
01241             == 0 ) {
01242             MesPrint("No compress used for element %s in file %s\n",arguments,d->name);
01243         }
01244     }
01245     else buffer = 0;
01246     if ( buffer ) {
01247         ssize = size;
01248 #ifdef WITHZSTD
01249         // Force the use of zlib for compressed Tablebase entries, so that tablebases created
01250         // with zstd-supported FORM builds can be used by zstd-unsupported FORM builds.
01251         const int old_isUsingZSTDcompression = ZWRAP_isUsingZSTDcompression();
01252         ZWRAP_useZSTDcompression(0);
01253 #endif
01254         if ( ( error = compress((Bytef *)buffer,&newsize,(Bytef *)rhs,ssize) ) != Z_OK ) {
01255             MesPrint("Error = %d\n",error);
01256             MesPrint("Due to error no compress used for element %s in file %s\n",arguments,d->name);
01257             M_free(buffer,"tb,WriteObject");
01258             buffer = 0;
01259         }
01260 #ifdef WITHZSTD
01261         ZWRAP_useZSTDcompression(old_isUsingZSTDcompression);
01262 #endif
01263     }
01264     if ( buffer ) {
01265         rhs = buffer;
01266         oldsize = size;
01267         size = newsize;
01268     }
01269 #endif
01270     if ( minoswrite(d->handle,rhs,size) ) {
01271         MesPrint("Error while writing rhs\n");
01272         return(-1);
01273     }
01274     obj->position = position;
01275     obj->size = size;
01276     obj->date = (MLONG) (time(0));
01277     obj->tablenumber = tablenumber;
01278 #ifdef WITHZLIB
01279     obj->uncompressed = oldsize;
01280     if ( buffer ) M_free(buffer,"tb,WriteObject");
01281 #else
01282     obj->uncompressed = 0;
01283 #endif
01284     return(WriteIndexBlock(d,j));
01285 }
01286
01287 /*
01288  #] WriteObject :
01289  #[ ReadObject :
01290
01291  Returns a pointer to the proper rhs
01292 */
01293
01294 char *ReadObject(DBASE *d,MLONG tablenumber,char *arguments)
01295 {
01296     OBJECTS *obj;
01297     MLONG i, j;
01298     char *buffer1, *s, *t;
01299 #ifdef WITHZLIB
01300     char *buffer2 = 0;
01301     ulongf finallength = 0;
01302 #endif
01303     if ( tablenumber > d->topnumber ) {
01304         MesPrint("Reference to non-existing table number in tablebase %s: %ld\n",

```

```

01305         d->name,tablename);
01306         return(0);
01307     }
01308     /*
01309     Start looking for the object
01310     */
01311     for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01312         for ( j = 0; j < NUMOBJECTS; j++ ) {
01313             if ( d->iblocks[i]->objects[j].tablename != tablename ) continue;
01314             s = arguments; t = d->iblocks[i]->objects[j].element;
01315             while ( *s == *t && *s ) { s++; t++; }
01316             if ( *t == 0 && *s == 0 ) goto foundelement;
01317         }
01318     }
01319     s = d->tablenames; i = 1;
01320     while ( *s ) {
01321         if ( i == tablename ) break;
01322         while ( *s ) s++;
01323         s++;
01324         while ( *s ) s++;
01325         s++;
01326         i++;
01327     }
01328     MesPrint("%s(%s) not found in tablebase %s\n",s,arguments,d->name);
01329     return(0);
01330
01331 foundelement:;
01332     obj = &(d->iblocks[i]->objects[j]);
01333     fseek(d->handle,obj->position,SEEK_SET);
01334     if ( ( buffer1 = (char *)Malloc1(obj->size,"reading rhs buffer1") ) == 0 ) {
01335         return(0);
01336     }
01337     #ifdef WITHZLIB
01338     if ( obj->uncompressed > 0 ) {
01339         if ( ( buffer2 = (char *)Malloc1(obj->uncompressed,"reading rhs buffer2") ) == 0 ) {
01340             return(0);
01341         }
01342     }
01343     else buffer2 = 0;
01344     #endif
01345     if ( minosread(d->handle,buffer1,obj->size) ) {
01346         MesPrint("Could not read rhs %s in file %s\n",arguments,d->name);
01347         M_free(buffer1,"tb,ReadObject");
01348     }
01349     #ifdef WITHZLIB
01350     if ( buffer2 ) M_free(buffer2,"tb,ReadObject");
01351     #endif
01352     return(0);
01353 }
01354 #ifdef WITHZLIB
01355 if ( buffer2 == 0 ) return(buffer1);
01356 finallength = obj->uncompressed;
01357 if ( uncompress((Bytef *)buffer2,&finallength,(Bytef *)buffer1,obj->size) != Z_OK ) {
01358     MesPrint("Cannot uncompress element %s in file %s\n",arguments,d->name);
01359     M_free(buffer1,"tb,ReadObject"); M_free(buffer2,"tb,ReadObject");
01360     return(0);
01361 }
01362 M_free(buffer1,"tb,ReadObject");
01363 return(buffer2);
01364 #else
01365 return(buffer1);
01366 #endif
01367 }
01368 /*
01369 #] ReadObject :
01370 #[ ReadijObject :
01371
01372 Returns a pointer to the proper rhs
01373 */
01374
01375 char *ReadijObject(DBASE *d,MLONG i,MLONG j,char *arguments)
01376 {
01377     OBJECTS *obj;
01378     char *buffer1;
01379     #ifdef WITHZLIB
01380     char *buffer2 = 0;
01381     uLongf finallength = 0;
01382     #endif
01383     obj = &(d->iblocks[i]->objects[j]);
01384     fseek(d->handle,obj->position,SEEK_SET);
01385     if ( ( buffer1 = (char *)Malloc1(obj->size,"reading rhs buffer1") ) == 0 ) {
01386         return(0);
01387     }
01388     #ifdef WITHZLIB
01389     if ( obj->uncompressed > 0 ) {
01390         if ( ( buffer2 = (char *)Malloc1(obj->uncompressed,"reading rhs buffer2") ) == 0 ) {
01391             return(0);

```

```

01392     }
01393 }
01394 else buffer2 = 0;
01395 #endif
01396 if ( minosread(d->handle,buffer1,obj->size) ) {
01397     MesPrint("Could not read rhs %s in file %s\n",arguments,d->name);
01398     if ( buffer1 ) M_free(buffer1,"rhs buffer1");
01399 #ifdef WITHZLIB
01400     if ( buffer2 ) M_free(buffer2,"rhs buffer2");
01401 #endif
01402     return(0);
01403 }
01404 #ifdef WITHZLIB
01405 if ( buffer2 == 0 ) return(buffer1);
01406 finallength = obj->uncompressed;
01407 if ( uncompress((Bytef *)buffer2,&finallength,(Bytef *)buffer1,obj->size) != Z_OK ) {
01408     MesPrint("Cannot uncompress element %s in file %s\n",arguments,d->name);
01409     if ( buffer1 ) M_free(buffer1,"rhs buffer1");
01410     if ( buffer2 ) M_free(buffer2,"rhs buffer2");
01411     return(0);
01412 }
01413 M_free(buffer1,"rhs buffer1");
01414 return(buffer2);
01415 #else
01416 return(buffer1);
01417 #endif
01418 }
01419
01420 /*
01421  #] ReadijObject :
01422  #[ ExistsObject :
01423
01424  Returns 1 if Object exists
01425 */
01426
01427 int ExistsObject(DBASE *d,MLONG tablenumber,char *arguments)
01428 {
01429     MLONG i, j;
01430     char *s, *t;
01431     if ( tablenumber > d->topnumber ) {
01432         MesPrint("Reference to non-existing table number in tablebase %s: %ld\n",
01433             d->name,tablenumber);
01434         return(0);
01435     }
01436     /*
01437     Start looking for the object
01438     */
01439     for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01440         for ( j = 0; j < NUMOBJECTS; j++ ) {
01441             if ( d->iblocks[i]->objects[j].tablenumber != tablenumber ) continue;
01442             s = arguments; t = d->iblocks[i]->objects[j].element;
01443             while ( *s == *t && *s ) { s++; t++; }
01444             if ( *t == 0 && *s == 0 ) return(1);
01445         }
01446     }
01447     return(0);
01448 }
01449
01450 /*
01451  #] ExistsObject :
01452  #[ DeleteObject :
01453
01454  Returns 1 if Object has been deleted.
01455  We leave a hole. Actually the object is still there but has been
01456  inactivated. It can be reactivated by calling this routine again.
01457 */
01458
01459 int DeleteObject(DBASE *d,MLONG tablenumber,char *arguments)
01460 {
01461     MLONG i, j;
01462     char *s, *t;
01463     if ( tablenumber > d->topnumber ) {
01464         MesPrint("Reference to non-existing table number in tablebase %s: %ld\n",
01465             d->name,tablenumber);
01466         return(0);
01467     }
01468     /*
01469     Start looking for the object
01470     */
01471     for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01472         for ( j = 0; j < NUMOBJECTS; j++ ) {
01473             if ( d->iblocks[i]->objects[j].tablenumber != tablenumber ) continue;
01474             s = arguments; t = d->iblocks[i]->objects[j].element;
01475             while ( *s == *t && *s ) { s++; t++; }
01476             if ( *t == 0 && *s == 0 ) {
01477                 d->iblocks[i]->objects[j].tablenumber =
01478                     -d->iblocks[i]->objects[j].tablenumber - 1;

```

```

01479             return(1);
01480         }
01481     }
01482 }
01483     return(0);
01484 }
01485
01486 /*
01487  #] DeleteObject :
01488 */

```

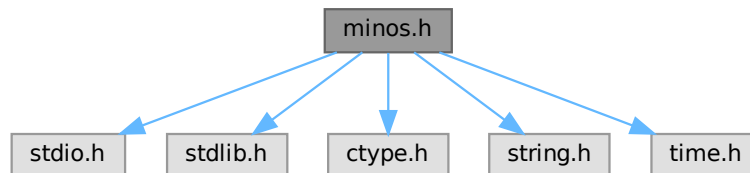
4.73 minos.h File Reference

```

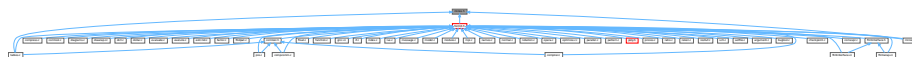
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <time.h>

```

Include dependency graph for minos.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct [iniinfo](#)
- struct [objects](#)
- struct [indexblock](#)
- struct [nameblock](#)
- struct [dbase](#)

Macros

- #define [TODISK](#) 0
- #define [FROMDISK](#) 1
- #define [MDIRTYFLAG](#) 1
- #define [MCLEANFLAG](#) (~MDIRTYFLAG)
- #define [INANDOUT](#) 0
- #define [INPUTONLY](#) 1
- #define [OUTPUTONLY](#) 2
- #define [NOCOMPRESS](#) 4

Typedefs

- typedef struct [iniinfo](#) **INIINFO**
- typedef struct [objects](#) **OBJECTS**
- typedef struct [indexblock](#) **INDEXBLOCK**
- typedef struct [nameblock](#) **NAMESBLOCK**
- typedef struct [dbase](#) **DBASE**

Functions

- int [minosread](#) (FILE *f, char *buffer, MLONG size)
- int [minoswrite](#) (FILE *f, char *buffer, MLONG size)

Variables

- int [withoutflush](#)

4.73.1 Detailed Description

Contains all needed declarations and definitions for the tablebase low level file routines. These have been taken from the minos database system and modified somewhat.

!!!CAUTION!!! Changes in this file will most likely have consequences for the recovery mechanism (see [checkpoint.c](#)). You need to care for the code in [checkpoint.c](#) as well and modify the code there accordingly!

Definition in file [minos.h](#).

4.73.2 Macro Definition Documentation

4.73.2.1 TODISK

```
#define TODISK 0
```

Definition at line [145](#) of file [minos.h](#).

4.73.2.2 FROMDISK

```
#define FROMDISK 1
```

Definition at line [146](#) of file [minos.h](#).

4.73.2.3 MDIRTYFLAG

```
#define MDIRTYFLAG 1
```

Definition at line [147](#) of file [minos.h](#).

4.73.2.4 MCLEANFLAG

```
#define MCLEANFLAG (~MDIRTYFLAG)
```

Definition at line 148 of file [minos.h](#).

4.73.2.5 INANDOUT

```
#define INANDOUT 0
```

Definition at line 149 of file [minos.h](#).

4.73.2.6 INPUTONLY

```
#define INPUTONLY 1
```

Definition at line 150 of file [minos.h](#).

4.73.2.7 OUTPUTONLY

```
#define OUTPUTONLY 2
```

Definition at line 151 of file [minos.h](#).

4.73.2.8 NOCOMPRESS

```
#define NOCOMPRESS 4
```

Definition at line 152 of file [minos.h](#).

4.73.3 Function Documentation

4.73.3.1 minosread()

```
int minosread (
    FILE * f,
    char * buffer,
    MLONG size )
```

Definition at line 66 of file [minos.c](#).

4.73.3.2 minoswrite()

```
int minoswrite (
    FILE * f,
    char * buffer,
    MLONG size )
```

Definition at line 83 of file [minos.c](#).

4.73.4 Variable Documentation

4.73.4.1 withoutflush

int withoutflush [extern]

Definition at line 45 of file [minos.c](#).

4.74 minos.h

[Go to the documentation of this file.](#)

```

00001 #ifndef __MANAGE_H__
00002
00003 #define __MANAGE_H__
00004
00017 /* #[ License : */
00018 /*
00019  * Copyright (C) 1984-2026 J.A.M. Vermaseren
00020  * When using this file you are requested to refer to the publication
00021  * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00022  * This is considered a matter of courtesy as the development was paid
00023  * for by FOM the Dutch physics granting agency and we would like to
00024  * be able to track its scientific use to convince FOM of its value
00025  * for the community.
00026  *
00027  * This file is part of FORM.
00028  *
00029  * FORM is free software: you can redistribute it and/or modify it under the
00030  * terms of the GNU General Public License as published by the Free Software
00031  * Foundation, either version 3 of the License, or (at your option) any later
00032  * version.
00033  *
00034  * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00035  * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00036  * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00037  * details.
00038  *
00039  * You should have received a copy of the GNU General Public License along
00040  * with FORM. If not, see <http://www.gnu.org/licenses/>.
00041 */
00042 /* #[ License : */
00043
00044 #include <stdio.h>
00045 #include <stdlib.h>
00046 #include <ctype.h>
00047 #include <string.h>
00048 #include <time.h>
00049
00050 /*
00051  * The following typedef has been moved to form3.h where all the sizes
00052  * are defined for the various memory models.
00053  * We want MLONG to have a more or less fixed size.
00054  * In form3.h we try to fix it at 8 bytes.
00055  * This should make files exchangeable
00056  * between various 32-bits and 64-bits systems. At 4 bytes it might have
00057  * problems with files of more than 2 Gbytes.
00058
00059 typedef long MLONG;
00060 */
00061
00062 #if BITSINWORD == 32
00063     #define NUMOBJECTS 1024
00064     #define MAXINDEXSIZE 1000000000L
00065     #define NAMETABLESIZE 1008
00066     #define ELEMENTSIZE 256
00067 #elif BITSINWORD == 16
00068     #define NUMOBJECTS 100
00069     #define MAXINDEXSIZE 33000000L
00070     #define NAMETABLESIZE 1008
00071     #define ELEMENTSIZE 128
00072 #else
00073     #error Only 64-bit and 32-bit platforms are supported.
00074 #endif
00075
00076
00077 int minosread(FILE *f,char *buffer,MLONG size);
00078 int minoswrite(FILE *f,char *buffer,MLONG size);

```

```

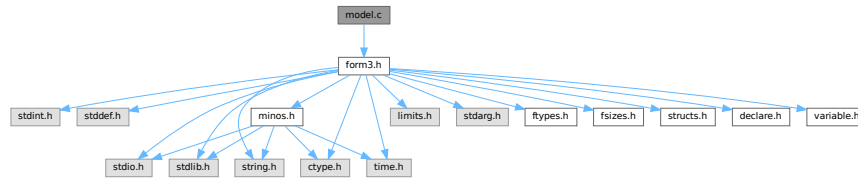
00079
00080 /*
00081     ELEMENTSIZE should make a nice number of sizeof(OBJECTS)
00082     Usually this will be much too much, but there are cases.....
00083 */
00084
00085 typedef struct iniinfo {
00086     /* should contains only MLONG variables or convertiniinfo should be modified */
00087     MLONG     entriesinindex;
00088     MLONG     numberofindexblocks;
00089     MLONG     firstindexblock;
00090     MLONG     lastindexblock;
00091     MLONG     numberoftables;
00092     MLONG     numberofnamesblocks;
00093     MLONG     firstnameblock;
00094     MLONG     lastnameblock;
00095 } INIINFO;
00096
00097 typedef struct objects {
00098     /* if any changes, convertblock should be adapted too!!!! */
00099     MLONG position;          /* position of RHS= */
00100     MLONG size;              /* size on disk (could be compressed) */
00101     MLONG date;              /* Time stamp */
00102     MLONG tablenumber;        /* Number of table. Refers to name in special index */
00103     MLONG uncompressed;      /* uncompressed size if compressed. If not: 0 */
00104     MLONG spare1;
00105     MLONG spare2;
00106     MLONG spare3;
00107     char element[ELEMENTSIZE]; /* table element in character form */
00108 } OBJECTS;
00109
00110 typedef struct indexblock {
00111     MLONG     flags;
00112     MLONG     previousblock;
00113     MLONG     position;
00114     OBJECTS   objects[NUMOBJECTS];
00115 } INDEXBLOCK;
00116
00117 typedef struct nameblock {
00118     MLONG     previousblock;
00119     MLONG     position;
00120     char     names[NAMETABLESIZE];
00121 } NAMESBLOCK;
00122
00123 typedef struct dbase {
00124     INIINFO    info;
00125     MLONG      mode;
00126     MLONG      tablenamessize;
00127     MLONG      topnumber;
00128     MLONG      tablenamefill;
00129     MLONG      rwmode;
00130     INDEXBLOCK **iblocks;
00131     NAMESBLOCK **nblocks;
00132     FILE        *handle;
00133     char        *name;
00134     char        *fullname;
00135     char        *tablenames;
00136 } DBASE;
00137
00138 /*
00139 typedef int (*SFUN)(char *);
00140 typedef struct compile {
00141     char *keyword;
00142     SFUN func;
00143 } MCFUNCTION;
00144 */
00145 #define TODISK 0
00146 #define FROMDISK 1
00147 #define MDIRTYFLAG 1
00148 #define MCLEANFLAG (~MDIRTYFLAG)
00149 #define INANDOUT 0
00150 #define INPUTONLY 1
00151 #define OUTPUTONLY 2
00152 #define NOCOMPRESS 4
00153
00154 extern int withoutflush;
00155
00156 #endif

```

4.75 model.c File Reference

```
#include "form3.h"
```

Include dependency graph for model.c:



Functions

- [VERTEX](#) * [CreateVertex](#) ([MODEL](#) *m)
- [UBYTE](#) * [ReadParticle](#) ([UBYTE](#) *s, [VERTEX](#) *v, [MODEL](#) *m, int par)
- int [CoModel](#) ([UBYTE](#) *s)
- int [CoParticle](#) ([UBYTE](#) *s)
- int [CoVertex](#) ([UBYTE](#) *s)
- int [CoEndModel](#) ([UBYTE](#) *s)

4.75.1 Detailed Description

Implementation of "Model", required by the `diagrams_` function.

Definition in file [model.c](#).

4.75.2 Function Documentation

4.75.2.1 CreateVertex()

```
VERTEX * CreateVertex (
    MODEL * m )
```

Definition at line 76 of file [model.c](#).

4.75.2.2 ReadParticle()

```
UBYTE * ReadParticle (
    UBYTE * s,
    VERTEX * v,
    MODEL * m,
    int par )
```

Definition at line 106 of file [model.c](#).

4.75.2.3 CoModel()

```
int CoModel (
    UBYTE * s )
```

Definition at line 149 of file [model.c](#).

4.75.2.4 CoParticle()

```
int CoParticle (
    UBYTE * s )
```

Definition at line 224 of file [model.c](#).

4.75.2.5 CoVertex()

```
int CoVertex (
    UBYTE * s )
```

Definition at line 306 of file [model.c](#).

4.75.2.6 CoEndModel()

```
int CoEndModel (
    UBYTE * s )
```

Definition at line 418 of file [model.c](#).

4.76 model.c

[Go to the documentation of this file.](#)

```
00001
00005 /* # [ License : */
00006 /*
00007 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00008 * When using this file you are requested to refer to the publication
00009 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010 * This is considered a matter of courtesy as the development was paid
00011 * for by FOM the Dutch physics granting agency and we would like to
00012 * be able to track its scientific use to convince FOM of its value
00013 * for the community.
00014 *
00015 * This file is part of FORM.
00016 *
00017 * FORM is free software: you can redistribute it and/or modify it under the
00018 * terms of the GNU General Public License as published by the Free Software
00019 * Foundation, either version 3 of the License, or (at your option) any later
00020 * version.
00021 *
00022 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00025 * details.
00026 *
00027 * You should have received a copy of the GNU General Public License along
00028 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00029 */
00030 /* # ] License : */
00031 /*
00032 # [ Explanations :
```

```

00033
00034 This is a second generation much simplified version of model.c
00035 Syntax:
00036     Model QED;
00037         Particle electron,positron,-2;
00038         Particle photon,+3;
00039         Vertex electron,positron,photon: g;
00040     EndModel;
00041 or:
00042     Model QCD;
00043         Particle quark,antiquark,-2;
00044         Particle ghost,antighost,-1;
00045         Particle gluon,+3;
00046         Vertex quark,antiquark,gluon: g;
00047         Vertex qghost,antighost,gluon: g;
00048         Vertex gluon,gluon,gluon: g;
00049         Vertex gluon,gluon,gluon,gluon: g^2;
00050     EndModel;
00051 Remarks:
00052 1: if no second particle is given, a particle and its antiparticle are the same
00053 2: Spin statistics is by dimension of SU(2) representation with a sign.
00054 3: In a vertex we need to mention the coupling constants.
00055 4: Internally a particle gives a vertex with only two lines.
00056 5: All particles must have been declared before any vertex.
00057
00058 The diagrams should be provided as a collection of nodes and edges
00059 with momenta with a direction substituted. The other parameters (like
00060 Lorentz indices, color indices etc.) can then be provided in a procedure
00061 that should go with this mode definition.
00062 In principle the edges are not needed if there are no 'insertions', but
00063 because we would like to have the insertions as well we need the edges.
00064
00065 #] Explanations :
00066 #[ Includes : model.c
00067 */
00068
00069 #include "form3.h"
00070
00071 /*
00072 #] Includes :
00073 #[ CreateVertex :
00074 */
00075
00076 VERTEX *CreateVertex(MODEL *m)
00077 {
00078     VERTEX *v, **new;
00079     WORD newsize;
00080     int i;
00081     if ( m->invertices >= m->sizevertices ) {
00082         if ( m->sizevertices == 0 ) newsize = 20;
00083         else newsize = 2*m->sizevertices;
00084         new = (VERTEX **)Malloc1(newsize*sizeof(VERTEX *),"m->vertices");
00085         for ( i = 0; i < m->sizevertices; i++ ) new[i] = m->vertices[i];
00086         if ( m->sizevertices > 0 ) M_free(m->vertices,"m->vertices");
00087         m->vertices = new;
00088         m->sizevertices = newsize;
00089     }
00090     v = (VERTEX *)Malloc1(sizeof(VERTEX),"VERTEX");
00091     m->vertices[m->invertices++] = v;
00092     v->nparticles = 0;
00093     v->ncouplings = 0;
00094     v->type = 0;
00095     v->error = 0;
00096     v->externonly = 0;
00097     v->spare = 0;
00098     return(v);
00099 }
00100
00101 /*
00102 #] CreateVertex :
00103 #[ ReadParticle :
00104 */
00105
00106 UBYTE *ReadParticle(UBYTE *s, VERTEX *v, MODEL *m, int par)
00107 {
00108     UBYTE *name, c;
00109     PARTICLE *p = v->particles + v->nparticles++;
00110     WORD type, funnum;
00111     int i, j;
00112     SKIPBLANKS(s)
00113     name = s; s = SkipName(s); c = *s; *s = 0;
00114     if ( GetVar(name,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND ) {
00115         p->number = AddFunction(name,0,VERTEXFUNCTION,0,0,0,-1,-1) + FUNCTION;
00116     }
00117     else if ( par == 1 && type == CFUNCTION && functions[funnum].spec == VERTEXFUNCTION ) {
00118         p->number = funnum+FUNCTION;
00119     }

```

```

00120     else if ( par == 0 && type == CFUNCTION && functions[funnum].spec == VERTEXFUNCTION ) {
00121     /*
00122         We should check whether this particle exists already in this model.
00123         If so, this will be an error.
00124     */
00125         for ( i = 0; i < m->nparticles-1; i++ ) {
00126             for ( j = 0; j < m->vertices[i]->nparticles; j++ ) {
00127                 if ( m->vertices[i]->particles[j].number == funnum ) {
00128                     MesPrint("&Illegal attempt to redefine a particle in the same model: %s",name);
00129                     v->error = 1;
00130                 }
00131             }
00132         }
00133         p->number = funnum+FUNCTION;
00134     }
00135     else {
00136         MesPrint("&Name of particle previously declared as another variable: %s",name);
00137         v->error = 1;
00138     }
00139     *s = c;
00140     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
00141     return(s);
00142 }
00143
00144 /*
00145     #] ReadParticle :
00146     #[ CoModel :
00147 */
00148
00149 int CoModel(UBYTE *s)
00150 {
00151     UBYTE *name, c;
00152     MODEL *m = (MODEL *)Malloc1(sizeof(MODEL),"Model");
00153     WORD numberofset, *e;
00154     SETS set;
00155     SKIPBLANKS(s)
00156     AC.ModelLevel++;
00157     int i;
00158 /*
00159     We now have to add the model to the list of models.
00160 */
00161     if ( AC.modelspace == 0 || AC.nummodels >= AC.modelspace ) {
00162         int newspace = 2*AC.modelspace+2, i;
00163         MODEL **models = (MODEL **)Malloc1((sizeof(MODEL *))*newspace,"AC.models");
00164         for ( i = 0; i < AC.modelspace; i++ ) models[i] = AC.models[i];
00165         if ( AC.models ) M_free(AC.models,"AC.models");
00166         AC.modelspace = newspace;
00167         AC.models = models;
00168     }
00169
00170     AC.models[AC.nummodels++] = m;
00171
00172     m->vertices = 0;
00173     m->couplings = 0;
00174     m->nparticles = m->nvertices = m->sizevertices = m->invertices = 0;
00175     m->sizecouplings = 0;
00176     m->error = 0;
00177     m->grccmodel = NULL;
00178
00179     name = s;
00180     if ( FG.cTable[*s] == 0 ) {
00181         while ( FG.cTable[*s] <= 1 ) s++;
00182         c = *s; *s = 0;
00183         m->name = strdup(name,"Model name");
00184         *s = c;
00185         SKIPBLANKS(s)
00186         if ( *s != 0 ) {
00187             MesPrint("&Illegal option in model statement: %s",s);
00188             return(1);
00189         }
00190     }
00191     else {
00192         m->name = strdup((UBYTE *) "---", "Model name");
00193         m->error = 1;
00194         MesPrint("&Illegal name for model: %s",name);
00195         return(1);
00196     }
00197 /*
00198     Now make it a set with one element. The type of the set is rather special
00199 */
00200     if ( GetName(AC.varnames,m->name,&numberofset,NOAUTO) != NAMENOTFOUND ) {
00201         MesPrint("&Name conflict with name %s of model",m->name);
00202         return(1);
00203     }
00204     numberofset = AddSet(name,0);
00205     set = Sets + numberofset;
00206     set->type = CMODEL;

```

```

00207     e = (WORD *)FromVarList (&AC.SetElementList);
00208     *e = AC.nummodels-1;
00209     set->last++;
00210     AC.SetList.numtemp = AC.SetList.num;
00211     AC.SetElementList.numtemp = AC.SetElementList.num;
00212
00213     for ( i = 0; i <= MAXLEGS; i++ ) m->legcouple[i] = 0;
00214     m->legcouple[2] = 1;
00215
00216     return(0);
00217 }
00218
00219 /*
00220 #] CoModel :
00221 #[ CoParticle :
00222 */
00223
00224 int CoParticle(UBYTE *s)
00225 {
00226     MODEL *m;
00227     VERTEX *v;
00228     int i;
00229     if ( AC.nummodels <= 0 ) {
00230         MesPrint("&No open model for particle statement");
00231         return(1);
00232     }
00233     m = AC.models[AC.nummodels-1];
00234     if ( m->nvertices > 0 ) {
00235         MesPrint("&In a model description Particle statements should be before Vertex statements.");
00236         m->error = 1;
00237         return(1);
00238     }
00239     v = CreateVertex(m);
00240     m->nparticles++;
00241     s = ReadParticle(s,v,m,0);
00242     v->particles[0].type = 1;
00243     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
00244     if ( v->error == 0 ) {
00245         if ( FG.cTable[*s] == 0 ) {
00246             s = ReadParticle(s,v,m,0);
00247             v->particles[1].type = -1;
00248             if ( v->particles[0].number == v->particles[1].number ) {
00249                 v->particles[0].type = 0;
00250                 v->particles[1].type = 0;
00251             }
00252         }
00253         else { /* Particle = antiparticle */
00254             v->particles[1] = v->particles[0];
00255             v->nparticles++;
00256             v->particles[0].type = 0;
00257             v->particles[1].type = 0;
00258         }
00259     }
00260     if ( v->error == 0 && ( *s == '+' || *s == '-' ) ) {
00261         int x = 0, sign = ( *s == '-' ) ? -1: 1;
00262         s++;
00263         while ( FG.cTable[*s] == 1 ) { x = 10*x + *s++ - '0'; }
00264         if ( x == 0 ) {
00265             v->error = 1;
00266             MesPrint("&Spin goes by dimension of SU(2) representation. Zero is not allowed.");
00267         }
00268         else { v->particles[0].spin = v->particles[1].spin = sign*x; }
00269         SKIPBLANKS(s)
00270     }
00271     else if ( v->error == 0 ) {
00272         v->particles[0].spin = v->particles[1].spin = 1;
00273         v->particles[0].type = 0;
00274         v->particles[1].type = 0;
00275     }
00276 /*
00277 Here will come future options
00278 */
00279 while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
00280 while ( v->error == 0 && *s != 0 && FG.cTable[*s] == 0 ) {
00281     UBYTE *opt = s, c;
00282     while ( FG.cTable[*s] == 0 ) s++;
00283     c = *s; *s = 0;
00284     if ( StrICont(opt,(UBYTE *)"external") == 0 ) { v->externonly = 1; }
00285     else {
00286         MesPrint("&Unrecognized option %s in particle statement.",opt);
00287         v->error = 1;
00288     }
00289     *s = c;
00290     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
00291 }
00292 /*
00293 The couplings array will be used for registering in what kind of vertices the

```



```

00294     particle is involved. (example: in QCD the quarks only in 2 and 3-point vertices)
00295 */
00296     for ( i = 0; i < MAXPARTICLES; i++ ) v->couplings[i] = 0;
00297     v->couplings[2] = 1;
00298     return(v->error);
00299 }
00300
00301 /*
00302 #] CoParticle :
00303 #[ CoVertex :
00304 */
00305
00306 int CoVertex(UBYTE *s)
00307 {
00308     UBYTE *ss, *name, c;
00309     WORD type;
00310     MODEL *m;
00311     VERTEX *v;
00312     if ( AC.ModelLevel <= 0 ) {
00313         MesPrint("&No open model for vertex statement");
00314         return(1);
00315     }
00316     m = AC.models[AC.nummodels-1];
00317 /*
00318     Get an object of type VERTEX
00319 */
00320     v = CreateVertex(m);
00321     m->nvertices++;
00322     SKIPBLANKS(s);
00323     while ( *s && *s != ':' ) {
00324         if ( v->error == 0 ) {
00325             s = ReadParticle(s,v,m,1);
00326             if ( v->error ) m->error = 1;
00327         }
00328         else {
00329             while ( *s && *s != ':' ) {
00330                 if ( *s == '[' ) { SKIPBRAL(s) }
00331                 else if ( *s == '(' ) { SKIPBRA3(s) }
00332                 else if ( *s == 0 ) {
00333                     v->error = 1;
00334                     MesPrint("&No coupling constant in vertex statement.");
00335                     break;
00336                 }
00337             }
00338             break;
00339         }
00340     }
00341     if ( v->error == 0 ) {
00342         if ( *s == ':' ) { /* read symbols and powers */
00343             s++; SKIPBLANKS(s);
00344             while ( *s ) {
00345                 if ( v->ncouplings >= 2*MAXCOUPLINGS ) {
00346                     MesPrint("&More than %d coupling constants in vertex.",(WORD)MAXCOUPLINGS);
00347                     MesPrint("      Recompile with a larger value for MAXCOUPLINGS.");
00348                     return(1);
00349                 }
00350                 ss = s; s = SkipName(s); c = *s; *s = 0; name = ConstructName(ss,0);
00351                 /* Forbid couplings which have no symbol or an overall numerical factor.
00352                 if ( *name == 0 ) {
00353                     v->error = 1;
00354                     MesPrint("&Invalid coupling constant in vertex statement.");
00355                 }
00356                 if ( GetVar(name,&type,&v->couplings[v->ncouplings],CSYMBOL,WITHAUTO)
00357                     == NAMENOTFOUND ) {
00358                     WORD minpow = -MAXPOWER;
00359                     WORD maxpow = MAXPOWER;
00360                     WORD cplx = 0, dim = 0;
00361                     v->couplings[v->ncouplings] = AddSymbol(name,minpow,maxpow,cplx,dim);
00362                 }
00363                 *s = c;
00364                 v->ncouplings++;
00365             /*
00366             Now possibly a power
00367             */
00368             if ( *s == '^' ) { /* we need an integer */
00369                 WORD x = 0; WORD sign = 1;
00370                 s++;
00371                 while ( *s == '-' || *s == '+' ) {
00372                     if ( *s == '-' ) sign = -sign;
00373                     s++;
00374                 }
00375                 if ( sign == -1 ) {
00376                     v->error = 1;
00377                     MesPrint("&Invalid negative power of coupling constant.");
00378                 }
00379                 while ( FG.cTable[*s] == 1 ) x = 10*x + *s++ - '0';
00380                 v->couplings[v->ncouplings++] = x;

```

```

00381         }
00382         else {
00383             v->couplings[v->ncouplings++] = 1;
00384         }
00385         SKIPBLANKS(s);
00386         if ( *s == '*' ) { s++; continue; }
00387         if ( *s != ',' ) break;
00388         s++;
00389         SKIPBLANKS(s);
00390     }
00391 }
00392 else {
00393     v->error = 1;
00394     MesPrint("%A vertex statement needs at least one coupling constant.");
00395 }
00396 }
00397 if ( v->error == 0 ) {
00398 /*
00399     Register which types of vertices are possible for each particle.
00400 */
00401     int i, j;
00402     for ( i = 0; i < v->nparticles; i++ ) {
00403         for ( j = 0; j < m->nparticles; j++ ) {
00404             if ( m->vertices[j]->particles[0].number == v->particles[i].number ) break;
00405             if ( m->vertices[j]->particles[1].number == v->particles[i].number ) break;
00406         }
00407         m->vertices[j]->couplings[v->nparticles] = 1;
00408     }
00409 }
00410 return(v->error);
00411 }
00412
00413 /*
00414 #] CoVertex :
00415 #[ CoEndModel :
00416 */
00417
00418 int CoEndModel(UBYTE *s)
00419 {
00420     int i, j, k, kk;
00421     WORD csize = 0, *newcouplings, newsize;
00422     MODEL *m;
00423     VERTEX *v;
00424     if ( AC.ModelLevel <= 0 ) {
00425         MesPrint("&EndModel statement without matching Model statement");
00426         return(1);
00427     }
00428     m = AC.models[AC.nummodels-1];
00429 /*
00430     Now create a list of all coupling constants.
00431     Note that we do not expect an astronomical number of them and hence
00432     we use a simple insertion algorithm.
00433 */
00434     m->ncouplings = 0;
00435     for ( i = 0; i < m->nvertices; i++ ) {
00436         v = m->vertices[i+m->nparticles];
00437         m->legcouple[v->nparticles] = 1;
00438         if ( m->ncouplings + v->ncouplings > csize ) {
00439             if ( csize == 0 ) newsize = v->ncouplings + 10;
00440             else newsize = 2*csize;
00441             newcouplings = (WORD *)Malloc1(newsize*sizeof(WORD), "m->couplings");
00442             for ( k = 0; k < m->ncouplings; k++ ) newcouplings[k] = m->couplings[k];
00443             if ( csize > 0 ) M_free(m->couplings, "m->couplings");
00444             m->couplings = newcouplings;
00445             csize = newsize;
00446         }
00447         for ( j = 0; j < v->ncouplings; j += 2 ) {
00448             WORD sym = v->couplings[j];
00449             for ( k = 0; k < m->ncouplings; k++ ) {
00450                 if ( sym == m->couplings[k] ) break;
00451                 if ( sym < m->couplings[k] ) {
00452                     for ( kk = m->ncouplings; kk > k; k-- )
00453                         m->couplings[kk] = m->couplings[kk-1];
00454                     m->couplings[k] = sym;
00455                     m->ncouplings++;
00456                     break;
00457                 }
00458             }
00459             if ( k >= m->ncouplings ) m->couplings[m->ncouplings++] = sym;
00460         }
00461     }
00462     AC.ModelLevel--;
00463     DUMMYUSE(s)
00464     return(LoadModel(m));
00465 }
00466
00467 /*

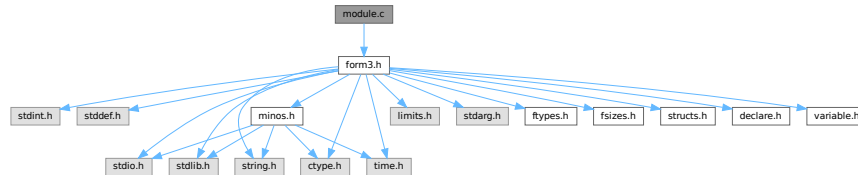
```

```
00468      #] CoEndModel :
00469      */
```

4.77 module.c File Reference

```
#include "form3.h"
```

Include dependency graph for module.c:



Functions

- int [ModuleInstruction](#) (int *moduletype, int *specialtype)
- int [CoModuleOption](#) (UBYTE *s)
- int [CoModOption](#) (UBYTE *s)
- void [SetSpecialMode](#) (int moduletype, int specialtype)
- int [ExecModule](#) (int moduletype)
- int [ExecStore](#) (void)
- void [FullCleanUp](#) (void)
- int [DoPolyfun](#) (UBYTE *s)
- int [DoPolyratfun](#) (UBYTE *s)
- int [DoNoParallel](#) (UBYTE *s)
- int [DoParallel](#) (UBYTE *s)
- int [DoModSum](#) (UBYTE *s)
- int [DoModMax](#) (UBYTE *s)
- int [DoModMin](#) (UBYTE *s)
- int [DoModLocal](#) (UBYTE *s)
- int [DoProcessBucket](#) (UBYTE *s)
- UBYTE * [DoModDollar](#) (UBYTE *s, int type)
- int [DoinParallel](#) (UBYTE *s)
- int [DonotinParallel](#) (UBYTE *s)
- int [DoExecStatement](#) (void)
- int [DoPipeStatement](#) (void)

4.77.1 Detailed Description

A number of routines that deal with the moduleoption statement and the execution of modules. Additionally there are the execution of the exec and pipe instructions.

Definition in file [module.c](#).

4.77.2 Function Documentation

4.77.2.1 ModuleInstruction()

```
int ModuleInstruction (
    int * moduletype,
    int * specialtype )
```

Definition at line 76 of file [module.c](#).

4.77.2.2 CoModuleOption()

```
int CoModuleOption (
    UBYTE * s )
```

Definition at line 143 of file [module.c](#).

4.77.2.3 CoModOption()

```
int CoModOption (
    UBYTE * s )
```

Definition at line 212 of file [module.c](#).

4.77.2.4 SetSpecialMode()

```
void SetSpecialMode (
    int moduletype,
    int specialtype )
```

Definition at line 257 of file [module.c](#).

4.77.2.5 ExecModule()

```
int ExecModule (
    int moduletype )
```

Definition at line 274 of file [module.c](#).

4.77.2.6 ExecStore()

```
int ExecStore (
    void )
```

Definition at line 299 of file [module.c](#).

4.77.2.7 FullCleanUp()

```
void FullCleanUp (
    void )
```

Definition at line 314 of file [module.c](#).

4.77.2.8 DoPolyfun()

```
int DoPolyfun (
    UBYTE * s )
```

Definition at line 395 of file [module.c](#).

4.77.2.9 DoPolyratfun()

```
int DoPolyratfun (
    UBYTE * s )
```

Definition at line 458 of file [module.c](#).

4.77.2.10 DoNoParallel()

```
int DoNoParallel (
    UBYTE * s )
```

Definition at line 529 of file [module.c](#).

4.77.2.11 DoParallel()

```
int DoParallel (
    UBYTE * s )
```

Definition at line 544 of file [module.c](#).

4.77.2.12 DoModSum()

```
int DoModSum (
    UBYTE * s )
```

Definition at line 559 of file [module.c](#).

4.77.2.13 DoModMax()

```
int DoModMax (
    UBYTE * s )
```

Definition at line 579 of file [module.c](#).

4.77.2.14 DoModMin()

```
int DoModMin (
    UBYTE * s )
```

Definition at line 599 of file [module.c](#).

4.77.2.15 DoModLocal()

```
int DoModLocal (
    UBYTE * s )
```

Definition at line 619 of file [module.c](#).

4.77.2.16 DoProcessBucket()

```
int DoProcessBucket (
    UBYTE * s )
```

Definition at line 639 of file [module.c](#).

4.77.2.17 DoModDollar()

```
UBYTE * DoModDollar (
    UBYTE * s,
    int type )
```

Definition at line 657 of file [module.c](#).

4.77.2.18 DoinParallel()

```
int DoinParallel (
    UBYTE * s )
```

Definition at line 758 of file [module.c](#).

4.77.2.19 DonotinParallel()

```
int DonotinParallel (
    UBYTE * s )
```

Definition at line 768 of file [module.c](#).

4.77.2.20 DoExecStatement()

```
int DoExecStatement (
    void )
```

Definition at line 780 of file [module.c](#).

4.77.2.21 DoPipeStatement()

```
int DoPipeStatement (
    void )
```

Definition at line 797 of file [module.c](#).

4.78 module.c

[Go to the documentation of this file.](#)

```
00001
00007 /* #[] License : */
00008 /*
00009 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00010 * When using this file you are requested to refer to the publication
00011 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00012 * This is considered a matter of courtesy as the development was paid
00013 * for by FOM the Dutch physics granting agency and we would like to
00014 * be able to track its scientific use to convince FOM of its value
00015 * for the community.
00016 *
00017 * This file is part of FORM.
00018 *
00019 * FORM is free software: you can redistribute it and/or modify it under the
00020 * terms of the GNU General Public License as published by the Free Software
00021 * Foundation, either version 3 of the License, or (at your option) any later
00022 * version.
00023 *
00024 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00025 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00026 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00027 * details.
00028 *
00029 * You should have received a copy of the GNU General Public License along
00030 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00031 */
00032 /* #[] License : */
00033 /*
00034 * #[] Includes :
00035 */
00036
00037 #include "form3.h"
00038
00039 /*
00040 * #[] Includes :
00041 * #[] Modules :
00042 * #[] ModuleInstruction :
00043 *
00044 * Enters after the . of .sort etc
00045 * We have word[[(options)][commentary];]
00046 * Word is one of 'clear end global sort store'
00047 * Options are 'polyfun, endloopifunchanged, endloopifzero'
00048 * Additions for solving equations can be added to 'word'
00049 *
00050 * The flag in the moduleoptions is for telling whether the current
00051 * option can be followed by others. If it is > 0 it cannot.
00052 */
00053
00054 static KEYWORD ModuleWords[] = {
00055     {"clear", (TFUN)0, CLEARMODULE, 0}
00056     , {"end", (TFUN)0, ENDMODULE, 0}
00057     , {"global", (TFUN)0, GLOBALMODULE, 0}
00058     , {"sort", (TFUN)0, SORTMODULE, 0}
00059     , {"store", (TFUN)0, STOREMODULE, 0}
00060 };
00061
00062 static KEYWORD ModuleOptions[] = {
00063     {"inparallel", DoinParallel, 1, 1}
00064     , {"local", DoModLocal, MODLOCAL, 0}
00065     , {"maximum", DoModMax, MODMAX, 0}
00066     , {"minimum", DoModMin, MODMIN, 0}
00067     , {"noparallel", DoNoParallel, NOPARALLEL_USER, 0}
00068     , {"notinparallel", DonotinParallel, 0, 1}
00069     , {"parallel", DoParallel, PARALLELFLAG, 0}
00070     , {"polyfun", DoPolyfun, POLYFUN, 0}
00071     , {"polyratfun", DoPolyratfun, POLYFUN, 0}
00072     , {"processbucketsize", DoProcessBucket, 0, 0}
00073     , {"sum", DoModSum, MODSUM, 0}
```

```

00074 };
00075
00076 int ModuleInstruction(int *moduletype, int *specialtype)
00077 {
00078     UBYTE *t, *s, *u, c;
00079     KEYWORD *key;
00080     int addit = 0, error = 0, i, j;
00081     DUMMYUSE(specialtype);
00082     LoadInstruction(0);
00083     AC.firstctypemessage = 0;
00084     s = AP.preStart; SKIPBLANKS(s)
00085     t = EndOfToken(s); c = *t; *t = 0;
00086     AC.origin = FROMPOINTINSTRUCTION;
00087     key = FindKeyWord(AP.preStart, ModuleWords, sizeof(ModuleWords)/sizeof(KEYWORD));
00088     if ( key == 0 ) {
00089         MesPrint("@Unrecognized module terminator: %s",s);
00090         error = 1;
00091         key = ModuleWords;
00092         while ( StrCmp((UBYTE *)key->name, (UBYTE *)"end") ) key++;
00093     }
00094     *t = c;
00095     *moduletype = key->type;
00096     SKIPBLANKS(t);
00097     while ( *t == '(' ) { /* There are options */
00098         s = t+1; SKIPBRA3(t)
00099         if ( *t == 0 ) {
00100             MesPrint("@Improper options field in . instruction");
00101             error = 1;
00102         }
00103         else {
00104             *t = 0;
00105             if ( CoModOption(s) ) error = 1;
00106             *t++ = ')';
00107         }
00108     }
00109     if ( *t == ':' ) { /* There is an 'advertisement' */
00110         t++;
00111         SKIPBLANKS(t)
00112         s = t; i = 0;
00113         while ( *t && *t != ';' ) {
00114             if ( *t == '\\\\' ) t++;
00115             t++; i++;
00116         }
00117         u = t;
00118         while ( u > s && u[-1] == ' ' ) { u--; i--; }
00119         if ( *u == '\\\\' ) { u++; i++; }
00120         for ( j = COMMERCIALSIZE-1; j >= 0; j-- ) {
00121             if ( i <= 0 ) break;
00122             AC.Commercial[j] = *--u; i--;
00123             if ( u > s && u[-1] == '\\\\' ) u--;
00124         }
00125         for ( ; j >= 0; j-- ) AC.Commercial[j] = ' ';
00126         AC.Commercial[COMMERCIALSIZE] = 0;
00127         addit += 2;
00128     }
00129     if ( addit && *t != ';' ) {
00130         MesPrint("@Improper ending of . instruction");
00131         error = -1;
00132     }
00133     return(error);
00134 }
00135
00136 /*
00137     #] ModuleInstruction :
00138     #[ CoModuleOption :
00139
00140     ModuleOption, options;
00141 */
00142
00143 int CoModuleOption(UBYTE *s)
00144 {
00145     UBYTE *t, *tt, c;
00146     KEYWORD *option;
00147     int error = 0, polyflag = 0;
00148     AC.origin = FROMMODULEOPTION;
00149     if ( *s ) do {
00150         s = ToToken(s);
00151         t = EndOfToken(s);
00152         c = *t; *t = 0;
00153         option = FindKeyWord(s, ModuleOptions,
00154             sizeof(ModuleOptions)/sizeof(KEYWORD));
00155         if ( option == 0 ) {
00156             if ( polyflag ) {
00157                 *t = c; t++; s = SkipAName(t);
00158                 polyflag = 0;
00159                 continue;
00160             }

```



```

00161         else {
00162             MesPrint("@Unrecognized module option: %s",s);
00163             error = 1;
00164             polyflag = 0;
00165             *t = c;
00166         }
00167     }
00168     else {
00169         *t = c;
00170         SKIPBLANKS(t)
00171         if ( (option->func)(t) ) error = 1;
00172     }
00173     if ( StrCmp((UBYTE *) (option->name), (UBYTE *) ("polyfun")) == 0
00174         || StrCmp((UBYTE *) (option->name), (UBYTE *) ("polyratfun")) == 0 ) {
00175         polyflag = 1;
00176         t++;
00177         t = SkipAName(t);
00178     }
00179     else polyflag = 0;
00180     if ( option->flags > 0 ) return(error);
00181     while ( *t ) {
00182         if ( *t == ',' ) {
00183             tt = t+1;
00184             while ( *tt == ',' ) tt++;
00185             if ( *tt != '$' ) break;
00186             t = tt+1;
00187         }
00188         if ( *t == ')' ) break;
00189         if ( *t == '(' ) SKIPBRA3(t)
00190         else if ( *t == '{' ) SKIPBRA2(t)
00191         else if ( *t == '[' ) SKIPBRA1(t)
00192         t++;
00193     }
00194     s = t;
00195 } while ( *s == ',' );
00196 if ( *s ) {
00197     MesPrint("@Unrecognized module option: %s",s);
00198     error = 1;
00199 }
00200 return(error);
00201 }
00202
00203 /*
00204     #] CoModuleOption :
00205     #[ CoModOption :
00206
00207     To be called from a .instruction.
00208     Only recognizes polyfun. The newer ones should be via the
00209     ModuleOption statement.
00210 */
00211
00212 int CoModOption(UBYTE *s)
00213 {
00214     UBYTE *t,c;
00215     int error = 0;
00216     AC.origin = FROMPOINTINSTRUCTION;
00217     if ( *s ) do {
00218         s = ToToken(s);
00219         t = EndOfToken(s);
00220         c = *t; *t = 0;
00221         if ( StrICmp(s, (UBYTE *) "polyfun") == 0 ) {
00222             *t = c;
00223             SKIPBLANKS(t)
00224             if ( DoPolyfun(t) ) error = 1;
00225         }
00226         else if ( StrICmp(s, (UBYTE *) "polyratfun") == 0 ) {
00227             *t = c;
00228             SKIPBLANKS(t)
00229             if ( DoPolyratfun(t) ) error = 1;
00230         }
00231         else {
00232             MesPrint("@Unrecognized module option in .instruction: %s",s);
00233             error = 1;
00234             *t = c;
00235         }
00236         while ( *t ) {
00237             if ( *t == ',' || *t == ')' ) break;
00238             if ( *t == '(' ) SKIPBRA3(t)
00239             else if ( *t == '{' ) SKIPBRA2(t)
00240             else if ( *t == '[' ) SKIPBRA1(t)
00241             t++;
00242         }
00243         s = t;
00244     } while ( *s == ',' );
00245     if ( *s ) {
00246         MesPrint("@Unrecognized module option in .instruction: %s",s);
00247         error = 1;

```

```

00248     }
00249     return(error);
00250 }
00251
00252 /*
00253     #] CoModOption :
00254     #[ SetSpecialMode :
00255 */
00256
00257 void SetSpecialMode(int moduletype, int specialtype)
00258 {
00259     DUMMYUSE(moduletype); DUMMYUSE(specialtype);
00260 }
00261
00262 /*
00263     #] SetSpecialMode :
00264     #[ MakeGlobal :
00265 */
00266 void MakeGlobal()
00267 {
00268 }
00269
00270     #] MakeGlobal :
00271     #[ ExecModule :
00272 */
00273
00274 int ExecModule(int moduletype)
00275 {
00276     /*
00277     Here we check module options and other things that should
00278     be done at the start of a module.
00279     */
00280     #ifndef WITHFLOAT
00281         if ( ( AC.tMaxWeight != 0 && AC.tMaxWeight != AC.MaxWeight )
00282             || ( AC.tDefaultPrecision != 0 && AC.tDefaultPrecision != AC.DefaultPrecision ) ) {
00283             AC.DefaultPrecision = AC.tDefaultPrecision;
00284             AC.tDefaultPrecision = 0;
00285             AC.MaxWeight = AC.tMaxWeight;
00286             AC.tMaxWeight = 0;
00287             ClearMZVTables();
00288             SetupMZVTables();
00289         }
00290     #endif
00291     return (DoExecute(moduletype,0));
00292 }
00293
00294 /*
00295     #] ExecModule :
00296     #[ ExecStore :
00297 */
00298
00299 int ExecStore(void)
00300 {
00301     return(0);
00302 }
00303
00304 /*
00305     #] ExecStore :
00306     #[ FullCleanUp :
00307 */
00308     Remark 27-oct-2005 by JV
00309     This routine (and CleanUp in startup.c) may still need some work:
00310         What to do with preprocessor variables
00311         What to do with files we write to
00312 */
00313
00314 void FullCleanUp(void)
00315 {
00316     int j;
00317
00318     while ( AC.CurrentStream->previous >= 0 )
00319         AC.CurrentStream = CloseStream(AC.CurrentStream);
00320     AP.PreSwitchLevel = AP.PreIfLevel = 0;
00321
00322     for ( j = NumProcedures-1; j >= 0; j-- ) {
00323         if ( Procedures[j].name ) M_free(Procedures[j].name,"name of procedure");
00324         if ( Procedures[j].p.buffer ) M_free(Procedures[j].p.buffer,"buffer of procedure");
00325     }
00326     NumProcedures = 0;
00327
00328     while ( NumPre > AP.gNumPre ) {
00329         NumPre--;
00330         M_free(PreVar[NumPre].name,"PreVar[NumPre].name");
00331         PreVar[NumPre].name = PreVar[NumPre].value = 0;
00332     }
00333
00334     AC.DidClean = 0;

```

```

00335     for ( j = 0; j < NumExpressions; j++ ) {
00336         AC.exprnames->namenode[Expressions[j].node].type = CDELETE;
00337         AC.DidClean = 1;
00338     }
00339
00340     CompactifyTree(AC.exprnames,EXPRNAMES);
00341
00342     for ( j = AO.NumDictionaries-1; j >= 0; j-- ) {
00343         RemoveDictionary(AO.Dictionaries[j]);
00344         M_free(AO.Dictionaries[j],"Dictionary");
00345     }
00346     AO.NumDictionaries = AO.gNumDictionaries = 0;
00347     M_free(AO.Dictionaries,"Dictionaries");
00348     AO.Dictionaries = 0;
00349     AO.SizeDictionaries = 0;
00350     AP.OpenDictionary = 0;
00351     AO.CurrentDictionary = 0;
00352
00353     AP.ComChar = AP.cComChar;
00354     if ( AP.procedureExtension ) M_free(AP.procedureExtension,"procedureextension");
00355     AP.procedureExtension = strdup(AP.cprocedureExtension,"procedureextension");
00356
00357     AC.StatsFlag = AM.gStatsFlag = AM.ggStatsFlag;
00358     AC.extrasymbols = AM.gextrasymbols = AM.ggextrasymbols;
00359     AC.extrasym[0] = AM.gextrasym[0] = AM.ggextrasym[0] = 'Z';
00360     AC.extrasym[1] = AM.gextrasym[1] = AM.ggextrasym[1] = 0;
00361     AC.NoSpacesInNumbers = AM.gNoSpacesInNumbers = AM.ggNoSpacesInNumbers;
00362     AO.IndentSpace = AM.gIndentSpace = AM.ggIndentSpace;
00363     AC.ThreadStats = AM.gThreadStats = AM.ggThreadStats;
00364     AC.OldFactArgFlag = AM.gOldFactArgFlag = AM.ggOldFactArgFlag;
00365     AC.FinalStats = AM.gFinalStats = AM.ggFinalStats;
00366     AC.OldGCDflag = AM.gOldGCDflag = AM.ggOldGCDflag;
00367     AC.ThreadsFlag = AM.gThreadsFlag = AM.ggThreadsFlag;
00368     if ( AC.ThreadsFlag && AM.totalnumberofthreads > 1 ) AS.MultiThreaded = 1;
00369     AC.ThreadBucketSize = AM.gThreadBucketSize = AM.ggThreadBucketSize;
00370     AC.ThreadBalancing = AM.gThreadBalancing = AM.ggThreadBalancing;
00371     AC.ThreadSortFileSynch = AM.gThreadSortFileSynch = AM.ggThreadSortFileSynch;
00372     AC.ShortStatsMax = AM.gShortStatsMax = AM.ggShortStatsMax;
00373     AC.SizeCommuteInSet = AM.gSizeCommuteInSet = 0;
00374 #ifdef WITHFLOAT
00375     AC.MaxWeight = AM.gMaxWeight = AM.ggMaxWeight;
00376     AC.DefaultPrecision = AM.gDefaultPrecision = AM.ggDefaultPrecision;
00377 #endif
00378
00379     NumExpressions = 0;
00380     if ( DeleteStore(0) < 0 ) {
00381         MesPrint("@Cannot restart the storage file");
00382         Terminate(-1);
00383     }
00384     RemoveDollars();
00385     CleanUp(1);
00386     ResetVariables(2);
00387     IniVars();
00388 }
00389
00390 /*
00391     #] FullCleanUp :
00392     #[ DoPolyfun :
00393 */
00394
00395 int DoPolyfun(UBYTE *s)
00396 {
00397     GETIDENTITY
00398     UBYTE *t, c;
00399     WORD funnum, eqsign = 0;
00400     if ( AC.origin == FROMPOINTINSTRUCTION ) {
00401         if ( *s == 0 || *s == ',' || *s == ')' ) {
00402             AR.PolyFun = 0; AR.PolyFunType = 0;
00403             return(0);
00404         }
00405         if ( *s != '=' ) {
00406             MesPrint("@Proper use in point instructions is: PolyFun[=functionname]");
00407             return(-1);
00408         }
00409         eqsign = 1;
00410     }
00411     else {
00412         if ( *s == 0 ) {
00413             AR.PolyFun = 0; AR.PolyFunType = 0;
00414             return(0);
00415         }
00416         if ( *s != '=' && *s != ',' ) {
00417             MesPrint("@Proper use is: PolyFun[{ ,= }functionname]");
00418             return(-1);
00419         }
00420         if ( *s == '=' ) eqsign = 1;
00421     }
}

```

```

00422     s++;
00423     SKIPBLANKS(s)
00424     t = EndOfToken(s);
00425     c = *t; *t = 0;
00426
00427     if ( GetName(AC.varnames,s,&funnum,WITHAUTO) != CFUNCTION ) {
00428         if ( AC.origin != FROMPOINTINSTRUCTION && eqsign == 0 ) {
00429             AR.PolyFun = 0; AR.PolyFunType = 0;
00430             return(0);
00431         }
00432         MesPrint("@ %s is not a properly declared function",s);
00433         *t = c;
00434         return(-1);
00435     }
00436     if ( functions[funnum].spec > 0 || functions[funnum].commute != 0 ) {
00437         MesPrint("@The PolyFun must be a regular commuting function!");
00438         *t = c;
00439         return(-1);
00440     }
00441     AR.PolyFun = funnum+FUNCTION; AR.PolyFunType = 1;
00442     *t = c;
00443     SKIPBLANKS(t)
00444     if ( *t && *t != ',' && *t != ')' ) {
00445         t++; c = *t; *t = 0;
00446         MesPrint("@Improper ending of end-of-module instruction: %s",s);
00447         *t = c;
00448         return(-1);
00449     }
00450     return(0);
00451 }
00452
00453 /*
00454     #] DoPolyfun :
00455     #[ DoPolyratfun :
00456 */
00457
00458 int DoPolyratfun(UBYTE *s)
00459 {
00460     GETIDENTITY
00461     UBYTE *t, c;
00462     WORD funnum;
00463     if ( AC.origin == FROMPOINTINSTRUCTION ) {
00464         if ( *s == 0 || *s == ',' || *s == ')' ) {
00465             AR.PolyFun = 0; AR.PolyFunType = 0; AR.PolyFunInv = 0; AR.PolyFunExp = 0;
00466             return(0);
00467         }
00468         if ( *s != '=' ) {
00469             MesPrint("@Proper use in point instructions is:
PolyRatFun[=functionname[+functionname]]");
00470             return(-1);
00471         }
00472     }
00473     else {
00474         if ( *s == 0 ) {
00475             AR.PolyFun = 0; AR.PolyFunType = 0; AR.PolyFunInv = 0; AR.PolyFunExp = 0;
00476             return(0);
00477         }
00478         if ( *s != '=' && *s != ',' ) {
00479             MesPrint("@Proper use is: PolyRatFun[{ ,= }functionname[+functionname]]");
00480             return(-1);
00481         }
00482     }
00483     s++;
00484     SKIPBLANKS(s)
00485     t = EndOfToken(s);
00486     c = *t; *t = 0;
00487
00488     if ( GetName(AC.varnames,s,&funnum,WITHAUTO) != CFUNCTION ) {
00489 Error1:;
00490         MesPrint("@ %s is not a properly declared function",s);
00491         *t = c;
00492         return(-1);
00493     }
00494     if ( functions[funnum].spec > 0 || functions[funnum].commute != 0 ) {
00495 Error2:;
00496         MesPrint("@The PolyRatFun must be a regular commuting function!");
00497         *t = c;
00498         return(-1);
00499     }
00500     AR.PolyFun = funnum+FUNCTION; AR.PolyFunType = 2;
00501     AR.PolyFunInv = 0;
00502     AR.PolyFunExp = 0;
00503     AC.PolyRatFunChanged = 1;
00504     *t = c;
00505     if ( *t == '+' ) {
00506         t++; s = t;
00507         t = EndOfToken(s);

```

```

00508         c = *t; *t = 0;
00509         if ( GetName(AC.varnames,s,&funnum,WITHAUTO) != CFUNCTION ) goto Error1;
00510         if ( functions[funnum].spec > 0 || functions[funnum].commute != 0 ) goto Error2;
00511         AR.PolyFunInv = funnum+FUNCTION;
00512         *t = c;
00513     }
00514     SKIPBLANKS(t)
00515     if ( *t && *t != ',' && *t != ')' ) {
00516         t++; c = *t; *t = 0;
00517         MesPrint("@Improper ending of end-of-module instruction: %s",s);
00518         *t = c;
00519         return(-1);
00520     }
00521     return(0);
00522 }
00523
00524 /*
00525     #] DoPolyratfun :
00526     #[ DoNoParallel :
00527 */
00528
00529 int DoNoParallel(UBYTE *s)
00530 {
00531     if ( *s == 0 || *s == ',' || *s == ')' ) {
00532         AC.mparallelfldflag |= NOPARALLEL_USER;
00533         return(0);
00534     }
00535     MesPrint("@NoParallel should not have extra parameters");
00536     return(-1);
00537 }
00538
00539 /*
00540     #] DoNoParallel :
00541     #[ DoParallel :
00542 */
00543
00544 int DoParallel(UBYTE *s)
00545 {
00546     if ( *s == 0 || *s == ',' || *s == ')' ) {
00547         AC.mparallelfldflag &= ~NOPARALLEL_USER;
00548         return(0);
00549     }
00550     MesPrint("@Parallel should not have extra parameters");
00551     return(-1);
00552 }
00553
00554 /*
00555     #] DoParallel :
00556     #[ DoModSum :
00557 */
00558
00559 int DoModSum(UBYTE *s)
00560 {
00561     while ( *s == ',' ) s++;
00562     if ( *s != '$' ) {
00563         MesPrint("@Module Sum should mention which $-variables");
00564         return(-1);
00565     }
00566     s = DoModDollar(s,MODSUM);
00567     if ( s && *s != 0 && *s != ')' ) {
00568         MesPrint("@Irregular end of Sum option of Module statement");
00569         return(-1);
00570     }
00571     return(0);
00572 }
00573
00574 /*
00575     #] DoModSum :
00576     #[ DoModMax :
00577 */
00578
00579 int DoModMax(UBYTE *s)
00580 {
00581     while ( *s == ',' ) s++;
00582     if ( *s != '$' ) {
00583         MesPrint("@Module Maximum should mention which $-variables");
00584         return(-1);
00585     }
00586     s = DoModDollar(s,MODMAX);
00587     if ( s && *s != 0 ) {
00588         MesPrint("@Irregular end of Maximum option of Module statement");
00589         return(-1);
00590     }
00591     return(0);
00592 }
00593
00594 /*

```

```

00595         #] DoModMax :
00596         #[ DoModMin :
00597     */
00598
00599 int DoModMin(UBYTE *s)
00600 {
00601     while ( *s == ',' ) s++;
00602     if ( *s != '$' ) {
00603         MesPrint("@Module Minimum should mention which $-variables");
00604         return(-1);
00605     }
00606     s = DoModDollar(s,MODMIN);
00607     if ( s && *s != 0 ) {
00608         MesPrint("@Irregular end of Minimum option of Module statement");
00609         return(-1);
00610     }
00611     return(0);
00612 }
00613
00614 /*
00615         #] DoModMin :
00616         #[ DoModLocal :
00617     */
00618
00619 int DoModLocal(UBYTE *s)
00620 {
00621     while ( *s == ',' ) s++;
00622     if ( *s != '$' ) {
00623         MesPrint("@ModuleOption Local should mention which $-variables");
00624         return(-1);
00625     }
00626     s = DoModDollar(s,MODLOCAL);
00627     if ( s && *s != 0 ) {
00628         MesPrint("@Irregular end of Local option of ModuleOption statement");
00629         return(-1);
00630     }
00631     return(0);
00632 }
00633
00634 /*
00635         #] DoModLocal :
00636         #[ DoProcessBucket :
00637     */
00638
00639 int DoProcessBucket(UBYTE *s)
00640 {
00641     LONG x;
00642     while ( *s == ',' || *s == '=' ) s++;
00643     ParseNumber(x,s)
00644     if ( *s && *s != ' ' && *s != '\t' ) {
00645         MesPrint("&Numerical value expected for ProcessBucketSize");
00646         return(1);
00647     }
00648     AC.mProcessBucketSize = x;
00649     return(0);
00650 }
00651
00652 /*
00653         #] DoProcessBucket :
00654         #[ DoModDollar :
00655     */
00656
00657 UBYTE * DoModDollar(UBYTE *s, int type)
00658 {
00659     UBYTE *name, c;
00660     WORD number;
00661     MODOPTDOLLAR *md;
00662     while ( *s == '$' ) {
00663     /*
00664         Read the name of the dollar
00665         Mark the type
00666     */
00667         s++;
00668         name = s;
00669         if ( FG.cTable[*s] == 0 ) {
00670             while ( FG.cTable[*s] == 0 || FG.cTable[*s] == 1 ) s++;
00671             c = *s; *s = 0;
00672             number = GetDollar(name);
00673             if ( number < 0 ) {
00674                 number = AddDollar(s,0,0,0);
00675                 Warning("Undefined $-variable in module statement");
00676             }
00677             md = (MODOPTDOLLAR *)FromList(&AC.ModOptDolList);
00678             md->number = number;
00679             md->type = type;
00680 #ifdef WITHPTHREADS
00681             if ( type == MODLOCAL ) {

```

```

00682         int j, i;
00683         DOLLARS dglobal, dlocal;
00684         md->dstruct = (DOLLARS)Malloc1(
00685             sizeof(struct DoLLaRs)*AM.totalnumberofthreads,"Local DOLLARS");
00686     /*
00687         Now copy the global dollar into the local copies.
00688         This can be nontrivial if the value needs an allocation.
00689         We don't really need the locks.
00690     */
00691     dglobal = Dollars + number;
00692     for ( j = 0; j < AM.totalnumberofthreads; j++ ) {
00693         dlocal = md->dstruct + j;
00694         dlocal->index = dglobal->index;
00695         dlocal->node = dglobal->node;
00696         dlocal->type = dglobal->type;
00697         dlocal->name = dglobal->name;
00698         dlocal->size = dglobal->size;
00699         dlocal->where = dglobal->where;
00700         if ( dlocal->size > 0 ) {
00701             dlocal->where = (WORD *)Malloc1((dlocal->size+1)*sizeof(WORD),"Local dollar
value");
00702             for ( i = 0; i < dlocal->size; i++ )
00703                 dlocal->where[i] = dglobal->where[i];
00704             dlocal->where[dlocal->size] = 0;
00705         }
00706         dlocal->pthreadlockread = dummylock;
00707         dlocal->pthreadlockwrite = dummylock;
00708         dlocal->nfactors = dglobal->nfactors;
00709         if ( dglobal->nfactors > 1 ) {
00710             int nsize;
00711             WORD *t, *m;
00712             dlocal->factors = (FACDOLLAR
*)Malloc1(dglobal->nfactors*sizeof(FACDOLLAR),"Dollar factors");
00713             for ( i = 0; i < dglobal->nfactors; i++ ) {
00714                 nsize = dglobal->factors[i].size;
00715                 dlocal->factors[i].type = DOLUNDEFINED;
00716                 dlocal->factors[i].value = dglobal->factors[i].value;
00717                 if ( ( dlocal->factors[i].size = nsize ) > 0 ) {
00718                     dlocal->factors[i].where = t = (WORD
*)Malloc1(sizeof(WORD)*(nsize+1),"DollarCopyFactor");
00719                     m = dglobal->factors[i].where;
00720                     NCOPY(t,m,nsize);
00721                     *t = 0;
00722                 }
00723                 else {
00724                     dlocal->factors[i].where = 0;
00725                 }
00726             }
00727         }
00728         else { dlocal->factors = 0; }
00729     }
00730 }
00731 else {
00732     md->dstruct = 0;
00733 }
00734 #endif
00735 *s = c;
00736 }
00737 else {
00738     MesPrint("&Illegal name for $-variable in module option");
00739     while ( *s != ',' && *s != 0 && *s != ')' ) s++;
00740 }
00741 while ( *s == ',' ) s++;
00742 }
00743 return(s);
00744 }
00745
00746 /*
00747     #] DoModDollar :
00748     #[ DoinParallel :
00749
00750     The idea is that we should have the commands
00751     ModuleOption,InParallel;
00752     ModuleOption,InParallel,name1,name2,...,namen;
00753     ModuleOption,NotInParallel,name1,name2,...,namen;
00754     The advantage over the InParallel statement is that this statement
00755     comes after the definition of the expressions.
00756 */
00757
00758 int DoinParallel(UBYTE *s)
00759 {
00760     return(DoInParallel(s,1));
00761 }
00762
00763 /*
00764     #] DoinParallel :
00765     #[ DonotinParallel :

```

```

00766 */
00767
00768 int DonotinParallel(UBYTE *s)
00769 {
00770     return(DoInParallel(s,0));
00771 }
00772
00773 /*
00774     #] DonotinParallel :
00775     #] Modules :
00776     #[ External :
00777     #[ DoExecStatement :
00778 */
00779
00780 int DoExecStatement(void)
00781 {
00782     #ifdef WITHSYSTEM
00783         FLUSHCONSOLE;
00784         if ( system((char *) (AP.preStart)) ) return(-1);
00785         return(0);
00786     #else
00787         Error0("External programs not implemented on this computer/system");
00788         return(-1);
00789     #endif
00790 }
00791
00792 /*
00793     #] DoExecStatement :
00794     #[ DoPipeStatement :
00795 */
00796
00797 int DoPipeStatement(void)
00798 {
00799     #ifdef WITHPIPE
00800         FLUSHCONSOLE;
00801         if ( OpenStream(AP.preStart,PIPESTREAM,0,PREENOACTION) == 0 ) return(-1);
00802         return(0);
00803     #else
00804         Error0("Pipes not implemented on this computer/system");
00805         return(-1);
00806     #endif
00807 }
00808
00809 /*
00810     #] DoPipeStatement :
00811     #] External :
00812 */

```

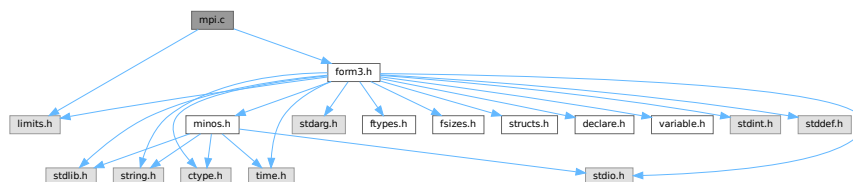
4.79 mpi.c File Reference

```

#include <limits.h>
#include "form3.h"

```

Include dependency graph for mpi.c:



Data Structures

- struct [longMultiStruct](#)

Macros

- `#define PF_PACKSIZE 1600`
- `#define MPI_ERRCODE_CHECK(err)`

Typedefs

- `typedef struct longMultiStruct PF_LONGMULTI`

Functions

- `LONG PF_RealTime (int i)`
- `int PF_LibInit (int *argcp, char ***argvp)`
- `int PF_LibTerminate (int error)`
- `int PF_Probe (int *src)`
- `int PF_ISendSbuf (int to, int tag)`
- `int PF_RecvWbuf (WORD *b, LONG *s, int *src)`
- `int PF_IRecvRbuf (PF_BUFFER *r, int bn, int from)`
- `int PF_WaitRbuf (PF_BUFFER *r, int bn, LONG *size)`
- `int PF_Bcast (void *buffer, int count)`
- `int PF_RawSend (int dest, void *buf, LONG l, int tag)`
- `LONG PF_RawRecv (int *src, void *buf, LONG thesize, int *tag)`
- `int PF_RawProbe (int *src, int *tag, int *bytesize)`
- `int PF_PrintPackBuf (char *s, int size)`
- `int PF_PreparePack (void)`
- `int PF_Pack (const void *buffer, size_t count, MPI_Datatype type)`
- `int PF_Unpack (void *buffer, size_t count, MPI_Datatype type)`
- `int PF_PackString (const UBYTE *str)`
- `int PF_UnpackString (UBYTE *str)`
- `int PF_Send (int to, int tag)`
- `int PF_Receive (int src, int tag, int *psrc, int *ptag)`
- `int PF_Broadcast (void)`
- `int PF_PrepareLongSinglePack (void)`
- `int PF_LongSinglePack (const void *buffer, size_t count, MPI_Datatype type)`
- `int PF_LongSingleUnpack (void *buffer, size_t count, MPI_Datatype type)`
- `int PF_LongSingleSend (int to, int tag)`
- `int PF_LongSingleReceive (int src, int tag, int *psrc, int *ptag)`
- `int PF_PrepareLongMultiPack (void)`
- `int PF_LongMultiPackImpl (const void *buffer, size_t count, size_t eSize, MPI_Datatype type)`
- `int PF_LongMultiUnpackImpl (void *buffer, size_t count, size_t eSize, MPI_Datatype type)`
- `int PF_LongMultiBroadcast (void)`

Variables

- `LONG PF_maxDollarChunkSize = 0`

4.79.1 Detailed Description

MPI dependent functions of parform

This file contains all the functions for the parallel version of form3 that explicitly need to call mpi routines. This is the only file that really needs to be linked to the mpi-library!

Definition in file [mpi.c](#).

4.79.2 Macro Definition Documentation

4.79.2.1 PF_PACKSIZE

```
#define PF_PACKSIZE 1600
```

Definition at line 58 of file [mpi.c](#).

4.79.2.2 MPI_ERRCODE_CHECK

```
#define MPI_ERRCODE_CHECK(  
    err )
```

Value:

```
do { \
    int _tmp_err = (err); \
    if ( _tmp_err != MPI_SUCCESS ) return _tmp_err != 0 ? _tmp_err : -1; \
} while (0)
```

A macro which exits the caller with a non-zero return value if *err* is not MPI_SUCCESS.

Parameters

<i>err</i>	The return value of a MPI function to be checked.

Remarks

The MPI standard defines MPI_SUCCESS == 0. Then (_tmp_err == 0) appears twice and we can expect the second evaluation will be eliminated by the compiler optimization.

Definition at line 84 of file [mpi.c](#).

4.79.3 Function Documentation

4.79.3.1 PF_RealTime()

```
LONG PF_RealTime (  
    int i )
```

Returns the realtime in 1/100 sec. as a LONG.

Parameters

<i>i</i>	the timer will be reset if i == 0.

Returns

the real elapsed time in 1/100 second.

Definition at line 101 of file [mpi.c](#).

Referenced by [PF_Init\(\)](#).

4.79.3.2 PF_LibInit()

```
int PF_LibInit (
    int * argcp,
    char *** argvp )
```

Performs all library dependent initializations.

Parameters

<i>argcp</i>	pointer to the number of arguments.
<i>argvp</i>	pointer to the arguments.

Returns

0 if OK, nonzero on error.

Definition at line 123 of file [mpi.c](#).

Referenced by [PF_Init\(\)](#).

4.79.3.3 PF_LibTerminate()

```
int PF_LibTerminate (
    int error )
```

Exits mpi, when there is an error either indicated or happening, returnvalue is 1, else returnvalue is 0.

Parameters

<i>error</i>	an error code.

Returns

0 if OK, nonzero on error.

Definition at line 209 of file [mpi.c](#).

Referenced by [PF_Terminate\(\)](#).

4.79.3.4 PF_Probe()

```
int PF_Probe (
    int * src )
```

Probes the next incoming message. If *src* == PF_ANY_SOURCE this operation is blocking, otherwise nonblocking.

Parameters

<i>in, out</i>	<i>src</i>	the source process number. In output, the process number of actual found source.

Returns

the tag value of the next incoming message if found, 0 if a nonblocking probe (input *src* != PF_ANY_SOURCE) did not find any messages. A negative returned value indicates an error.

Definition at line 230 of file [mpi.c](#).

4.79.3.5 PF_ISendSbuf()

```
int PF_ISendSbuf (
    int to,
    int tag )
```

Nonblocking send operation. It sends everything from *buff* to *fill* of the active buffer. Depending on *tag* it also can do waiting for other sends to finish or set the active buffer to the next one.

Parameters

<i>to</i>	the destination process number.
<i>tag</i>	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 261 of file [mpi.c](#).

Referenced by [FlushOut\(\)](#), [PF_Processor\(\)](#), and [PutOut\(\)](#).

4.79.3.6 PF_RecvWbuf()

```
int PF_RecvWbuf (
    WORD * b,
    LONG * s,
    int * src )
```

Blocking receive of a WORD buffer.

Parameters

<i>out</i>	<i>b</i>	the buffer to store the received data.
<i>in, out</i>	<i>s</i>	the size of the buffer. The output value is the actual size of the received data.
<i>in, out</i>	<i>src</i>	the source process number. The output value is the process number of actual source.

Returns

the received message tag. A negative value indicates an error.

Definition at line 337 of file [mpi.c](#).

4.79.3.7 PF_IRecvRbuf()

```
int PF_IRecvRbuf (
    PF_BUFFER * r,
    int bn,
    int from )
```

Posts nonblocking receive for the active receive buffer. The buffer is filled from `full` to `stop`.

Parameters

<i>r</i>	the PF_BUFFER struct for the nonblocking receive.
<i>bn</i>	the index of the cyclic buffer.
<i>from</i>	the source process number.

Returns

0 if OK, nonzero on error.

Definition at line 366 of file [mpi.c](#).

4.79.3.8 PF_WaitRbuf()

```
int PF_WaitRbuf (
    PF_BUFFER * r,
    int bn,
    LONG * size )
```

Waits for the buffer *bn* to finish a pending nonblocking receive. It returns the received tag and in **size* the number of field received. If the receive is already finished, just return the flag and size, else wait for it to finish, but also check for other pending receives.

Parameters

	<i>r</i>	the PF_BUFFER struct for the pending nonblocking receive.
	<i>bn</i>	the index of the cyclic buffer.
out	<i>size</i>	the actual size of received data.

Returns

the received message tag. A negative value indicates an error.

Definition at line 400 of file [mpi.c](#).

4.79.3.9 PF_Bcast()

```
int PF_Bcast (
    void * buffer,
    int count )
```

Broadcasts a message from the master to slaves.

Parameters

<i>in, out</i>	<i>buffer</i>	the starting address of buffer. The contents in this buffer on the master will be transferred to those on the slaves.
	<i>count</i>	the length of the buffer in bytes.

Returns

0 if OK, nonzero on error.

Definition at line 440 of file [mpi.c](#).

Referenced by [PF_BroadcastBuffer\(\)](#), and [PF_BroadcastNumber\(\)](#).

4.79.3.10 PF_RawSend()

```
int PF_RawSend (
    int dest,
    void * buf,
    LONG l,
    int tag )
```

Sends *l* bytes from *buf* to *dest*. Returns 0 on success, or -1.

Parameters

<i>dest</i>	the destination process number.
<i>buf</i>	the send buffer.
<i>l</i>	the size of data in the send buffer in bytes.
<i>tag</i>	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 463 of file [mpi.c](#).

Referenced by [PF_MUnlock\(\)](#), and [PF_SendFile\(\)](#).

4.79.3.11 PF_RawRecv()

```
LONG PF_RawRecv (
    int * src,
```

```
void * buf,
LONG thesize,
int * tag )
```

Receives not more than *thesize* bytes from *src*, returns the actual number of received bytes, or -1 on failure.

Parameters

in, out	<i>src</i>	the source process number. In output, that of the actual received message.
out	<i>buf</i>	the receive buffer.
	<i>thesize</i>	the size of the receive buffer in bytes.
out	<i>tag</i>	the message tag of the actual received message.

Returns

the actual sizeof received data in bytes, or -1 on failure.

Definition at line 484 of file [mpi.c](#).

Referenced by [PF_RecvFile\(\)](#).

4.79.3.12 PF_RawProbe()

```
int PF_RawProbe (
    int * src,
    int * tag,
    int * bytesize )
```

Probes an incoming message.

Parameters

in, out	<i>src</i>	the source process number. In output, that of the actual received message.
in, out	<i>tag</i>	the message tag. In output, that of the actual received message.
out	<i>bytesize</i>	the size of incoming data in bytes.

Returns

0 if OK, nonzero on error.

Definition at line 508 of file [mpi.c](#).

4.79.3.13 PF_PrintPackBuf()

```
int PF_PrintPackBuf (
    char * s,
    int size )
```

Prints the contents in the pack buffer.

Parameters

<i>s</i>	a message to be shown.
<i>size</i>	the length of the buffer to be shown.

Returns

0 if OK, nonzero on error.

Definition at line 594 of file [mpi.c](#).

4.79.3.14 PF_PrepPack()

```
int PF_PrepPack (
    void )
```

Prepares for the next pack operations on the sender.

Returns

0 if OK, nonzero on error.

Definition at line 624 of file [mpi.c](#).

Referenced by [Optimize\(\)](#), [PF_BroadcastPreDollar\(\)](#), [PF_BroadcastString\(\)](#), and [PF_Init\(\)](#).

4.79.3.15 PF_Pack()

```
int PF_Pack (
    const void * buffer,
    size_t count,
    MPI_Datatype type )
```

Adds data into the pack buffer.

Parameters

<i>buffer</i>	the pointer to the buffer storing the data to be packed.
<i>count</i>	the number of elements in the buffer.
<i>type</i>	the data type of elements in the buffer.

Returns

0 if OK, nonzero on error.

Definition at line 642 of file [mpi.c](#).

References [MPI_ERRCODE_CHECK](#).

Referenced by [Optimize\(\)](#), [PF_BroadcastPreDollar\(\)](#), and [PF_Init\(\)](#).

4.79.3.16 PF_Unpack()

```
int PF_Unpack (
    void * buffer,
    size_t count,
    MPI_Datatype type )
```

Retrieves the next data in the pack buffer.

Parameters

out	<i>buffer</i>	the pointer to the buffer to store the unpacked data.
	<i>count</i>	the number of elements of data to be received.
	<i>type</i>	the data type of elements of data to be received.

Returns

0 if OK, nonzero on error.

Definition at line 671 of file [mpi.c](#).

References [MPI_ERRCODE_CHECK](#).

Referenced by [Optimize\(\)](#), [PF_BroadcastPreDollar\(\)](#), and [PF_Init\(\)](#).

4.79.3.17 PF_PackString()

```
int PF_PackString (
    const UBYTE * str )
```

Packs a string *str* into the packed buffer `PF_packbuf`, including the trailing zero.

The first element (`PF_INT`) is the length of the packed portion of the string. If the string does not fit to the buffer `PF_packbuf`, the function packs only the initial portion. It returns the number of packed bytes, so if `(str[length-1]!='\0')` then the whole string fits to the buffer, if not, then the rest (`str+length`) must be packed and send again. On error, the function returns the negative error code.

One exception: the string `"\0\0"` is used as an image of the NULL, so all 3 characters will be packed.

Parameters

<i>str</i>	a string to be packed.

Returns

the number of packed bytes, or a negative value on failure.

Definition at line 706 of file [mpi.c](#).

Referenced by [PF_BroadcastString\(\)](#).

4.79.3.18 PF_UnpackString()

```
int PF_UnpackString (
    UBYTE * str )
```

Unpacks a string to *str* from the packed buffer PF_packbuf, including the trailing zero.

It returns the number of unpacked bytes, so if (str[length-1]=='\0') then the whole string was unpacked, if not, then the rest must be appended to (str+length). On error, the function returns the negative error code.

Parameters

out	<i>str</i>	the buffer to store the unpacked string

Returns

the number of unpacked bytes, or a negative value on failure.

Definition at line 774 of file [mpi.c](#).

Referenced by [PF_BroadcastString\(\)](#).

4.79.3.19 PF_Send()

```
int PF_Send (
    int to,
    int tag )
```

Sends the contents in the pack buffer to the process specified by *to*.

Example:

```
if ( PF.me == SRC ) {
    PF_PreparePack();
    // Packing operations here...
    PF_Send(DEST, TAG);
}
else if ( PF.me == DEST ) {
    PF_Receive(SRC, TAG, &actual_src, &actual_tag);
    // Unpacking operations here...
}
```

Parameters

<i>to</i>	the destination process number.
<i>tag</i>	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 822 of file [mpi.c](#).

References [MPI_ERRCODE_CHECK](#).

4.79.3.20 PF_Receive()

```
int PF_Receive (
    int src,
    int tag,
    int * psrc,
    int * ptag )
```

Receives data into the pack buffer from the process specified by *src*. This function allows &*src* == *psrc* or &*tag* == *ptag*. Either *psrc* or *ptag* can be NULL.

See the example of [PF_Send\(\)](#).

Parameters

	<i>src</i>	the source process number (can be PF_ANY_SOURCE).
	<i>tag</i>	the source message tag (can be PF_ANY_TAG).
out	<i>psrc</i>	the actual source process number of received message.
out	<i>ptag</i>	the received message tag.

Returns

0 if OK, nonzero on error.

Definition at line 848 of file [mpi.c](#).

References [MPI_ERRCODE_CHECK](#).

4.79.3.21 PF_Broadcast()

```
int PF_Broadcast (
    void )
```

Broadcasts the contents in the pack buffer on the master to those on the slaves.

Example:

```
if ( PF.me == MASTER ) {
    PF_PrepPack();
    // Packing operations here...
}
PF_Broadcast();
if ( PF.me != MASTER ) {
    // Unpacking operations here...
}
```

Returns

0 if OK, nonzero on error.

Definition at line 883 of file [mpi.c](#).

References [MPI_ERRCODE_CHECK](#).

Referenced by [Optimize\(\)](#), [PF_BroadcastPreDollar\(\)](#), [PF_BroadcastString\(\)](#), and [PF_Init\(\)](#).

4.79.3.22 PF_PrepareLongSinglePack()

```
int PF_PrepareLongSinglePack (
    void )
```

Prepares for the next long-single-pack operations on the sender.

Returns

0 if OK, nonzero on error.

Definition at line 1451 of file [mpi.c](#).

Referenced by [PF_CollectModifiedDollars\(\)](#), and [PF_Processor\(\)](#).

4.79.3.23 PF_LongSinglePack()

```
int PF_LongSinglePack (
    const void * buffer,
    size_t count,
    MPI_Datatype type )
```

Adds data into the "long single" pack buffer.

Parameters

<i>buffer</i>	the pointer to the buffer storing the data to be packed.
<i>count</i>	the number of elements in the buffer.
<i>type</i>	the data type of elements in the buffer.

Returns

0 if OK, nonzero on error.

Definition at line 1469 of file [mpi.c](#).

Referenced by [PF_CollectModifiedDollars\(\)](#), and [PF_Processor\(\)](#).

4.79.3.24 PF_LongSingleUnpack()

```
int PF_LongSingleUnpack (
    void * buffer,
    size_t count,
    MPI_Datatype type )
```

Retrieves the next data in the "long single" pack buffer.

Parameters

out	<i>buffer</i>	the pointer to the buffer to store the unpacked data.
	<i>count</i>	the number of elements of data to be received.
	<i>type</i>	the data type of elements of data to be received.

Returns

0 if OK, nonzero on error.

Definition at line 1503 of file [mpi.c](#).

Referenced by [PF_CollectModifiedDollars\(\)](#), and [PF_Processor\(\)](#).

4.79.3.25 PF_LongSingleSend()

```
int PF_LongSingleSend (  
    int to,  
    int tag )
```

Sends the contents in the "long single" pack buffer to the process specified by *to*.

Example:

```
if ( PF.me == SRC ) {  
    PF_PrepareLongSinglePack();  
    // Packing operations here...  
    PF_LongSingleSend(DEST, TAG);  
}  
else if ( PF.me == DEST ) {  
    PF_LongSingleReceive(SRC, TAG, &actual_src, &actual_tag);  
    // Unpacking operations here...  
}
```

Parameters

<i>to</i>	the destination process number.
<i>tag</i>	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 1540 of file [mpi.c](#).

Referenced by [PF_CollectModifiedDollars\(\)](#), and [PF_Processor\(\)](#).

4.79.3.26 PF_LongSingleReceive()

```
int PF_LongSingleReceive (  
    int src,  
    int tag,  
    int * psrc,  
    int * ptag )
```

Receives data into the "long single" pack buffer from the process specified by *src*. This function allows *&src == psrc* or *&tag == ptag*. Either *psrc* or *ptag* can be NULL.

See the example of [PF_LongSingleSend\(\)](#).

Parameters

	<i>src</i>	the source process number (can be PF_ANY_SOURCE).
	<i>tag</i>	the source message tag (can be PF_ANY_TAG).
out	<i>psrc</i>	the actual source process number of received message.
out	<i>ptag</i>	the received message tag.

Returns

0 if OK, nonzero on error.

Definition at line 1583 of file [mpi.c](#).

Referenced by [PF_CollectModifiedDollars\(\)](#), and [PF_Processor\(\)](#).

4.79.3.27 PF_PrepareLongMultiPack()

```
int PF_PrepareLongMultiPack (
    void )
```

Prepares for the next long-multi-pack operations on the sender.

Returns

0 if OK, nonzero on error.

Definition at line 1643 of file [mpi.c](#).

Referenced by [Optimize\(\)](#), [PF_BroadcastCBuf\(\)](#), [PF_BroadcastExpFlags\(\)](#), [PF_BroadcastModifiedDollars\(\)](#), and [PF_BroadcastRedefinedPreVars\(\)](#).

4.79.3.28 PF_LongMultiPackImpl()

```
int PF_LongMultiPackImpl (
    const void * buffer,
    size_t count,
    size_t eSize,
    MPI_Datatype type )
```

Adds data into the "long multi" pack buffer.

Parameters

<i>buffer</i>	the pointer to the buffer storing the data to be packed.
<i>count</i>	the number of elements in the buffer.
<i>eSize</i>	the byte size of each element of data.
<i>type</i>	the data type of elements in the buffer.

Returns

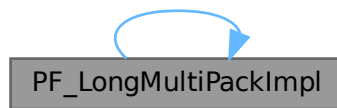
0 if OK, nonzero on error.

Definition at line 1662 of file [mpi.c](#).

References [PF_LongMultiPackImpl\(\)](#).

Referenced by [PF_LongMultiPackImpl\(\)](#).

Here is the call graph for this function:

**4.79.3.29 PF_LongMultiUnpackImpl()**

```
int PF_LongMultiUnpackImpl (
    void * buffer,
    size_t count,
    size_t eSize,
    MPI_Datatype type )
```

Retrieves the next data in the "long multi" pack buffer.

Parameters

out	<i>buffer</i>	the pointer to the buffer to store the unpacked data.
	<i>count</i>	the number of elements of data to be received.
	<i>eSize</i>	the byte size of each element of data.
	<i>type</i>	the data type of elements of data to be received.

Returns

0 if OK, nonzero on error.

Definition at line 1721 of file [mpi.c](#).

References [PF_LongMultiUnpackImpl\(\)](#).

Referenced by [PF_LongMultiUnpackImpl\(\)](#).

Here is the call graph for this function:



4.79.3.30 PF_LongMultiBroadcast()

```
int PF_LongMultiBroadcast (
    void )
```

Broadcasts the contents in the "long multi" pack buffer on the master to those on the slaves.

Example:

```
if ( PF.me == MASTER ) {
    PF_PrepareLongMultiPack();
    // Packing operations here...
}
PF_LongMultiBroadcast();
if ( PF.me != MASTER ) {
    // Unpacking operations here...
}
```

Returns

0 if OK, nonzero on error.

Definition at line 1807 of file [mpi.c](#).

Referenced by [Optimize\(\)](#), [PF_BroadcastCBuf\(\)](#), [PF_BroadcastExpFlags\(\)](#), [PF_BroadcastModifiedDollars\(\)](#), and [PF_BroadcastRedefinedPreVars\(\)](#).

4.79.4 Variable Documentation

4.79.4.1 PF_maxDollarChunkSize

```
LONG PF_maxDollarChunkSize = 0
```

Definition at line 69 of file [mpi.c](#).

4.80 mpi.c

[Go to the documentation of this file.](#)

```

00001
00009 /* #[] License : */
00010 /*
00011 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00012 *   When using this file you are requested to refer to the publication
00013 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014 *   This is considered a matter of courtesy as the development was paid
00015 *   for by FOM the Dutch physics granting agency and we would like to
00016 *   be able to track its scientific use to convince FOM of its value
00017 *   for the community.
00018 *
00019 *   This file is part of FORM.
00020 *
00021 *   FORM is free software: you can redistribute it and/or modify it under the
00022 *   terms of the GNU General Public License as published by the Free Software
00023 *   Foundation, either version 3 of the License, or (at your option) any later
00024 *   version.
00025 *
00026 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00029 *   details.
00030 *
00031 *   You should have received a copy of the GNU General Public License along
00032 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00033 */
00034 /* #] License : */
00035 /*
00036     #[] Includes and variables :
00037 */
00038
00039 #include <limits.h>
00040 #include "form3.h"
00041
00042 #ifdef MPICH_PROFILING
00043 # include "mpe.h"
00044 #endif
00045
00046 #ifdef MPIDEBUGGING
00047 #include "mpidbg.h"
00048 #endif
00049
00050 /*[12oct2005 mt]:*/
00051 /*
00052     Today there was some cleanup, some stuff is moved into another place
00053     in this file, and PF_packsize is removed and PF_packsize is used
00054     instead. It is rather difficult to proper comment it, so not all these
00055     changing are marked by "[12oct2005 mt]"
00056 */
00057
00058 #define PF_PACKSIZE 1600
00059
00060 /*
00061     Size in bytes, will be initialized soon as
00062     PF_packsize=PF_PACKSIZE/sizeof(int)*sizeof(int); for possible
00063     future developing we prefer to do this initialization not here,
00064     but in PF_LibInit:
00065 */
00066
00067 static int PF_packsize = 0;
00068 static MPI_Status PF_status;
00069 LONG PF_maxDollarChunkSize = 0; /*:[04oct2005 mt]*/
00070
00071 static int PF_ShortPackInit(void);
00072 static int PF_longPackInit(void); /*:[12oct2005 mt]*/
00073
00084 #define MPI_ERRCODE_CHECK(err) \
00085     do { \
00086         int _tmp_err = (err); \
00087         if ( _tmp_err != MPI_SUCCESS ) return _tmp_err != 0 ? _tmp_err : -1; \
00088     } while (0)
00089
00090 /*
00091     #] Includes and variables :
00092     #[] PF_RealTime :
00093 */
00094
00101 LONG PF_RealTime(int i)
00102 {
00103     static double starttime;
00104     if ( i == PF_RESET ) {
00105         starttime = MPI_Wtime();

```

```

00106         return((LONG)0);
00107     }
00108     return((LONG)( 100. * (MPI_Wtime() - starttime) ));
00109 }
00110
00111 /*
00112 #] PF_RealTime :
00113 #[ PF_LibInit :
00114 */
00115
00123 int PF_LibInit(int *argcp, char ***argvp)
00124 {
00125     int ret;
00126     ret = MPI_Init(argcp,argvp);
00127     if ( ret != MPI_SUCCESS ) return(ret);
00128     ret = MPI_Comm_rank(PF_COMM,&PF.me);
00129     if ( ret != MPI_SUCCESS ) return(ret);
00130     ret = MPI_Comm_size(PF_COMM,&PF.numtasks);
00131     if ( ret != MPI_SUCCESS ) return(ret);
00132
00133     /* Initialization of packed communications. */
00134     PF_packsize = PF_PACKSIZE/sizeof(int)*sizeof(int);
00135     if ( PF_ShortPackInit() ) return -1;
00136     if ( PF_longPackInit() ) return -1;
00137
00138     /*Block*/
00139     int bytes, totalbytes=0;
00140 /*
00141     There is one problem with maximal possible packing: there is no API to
00142     convert bytes to the record number. So, here we calculate the buffer
00143     size needed for storing dollarvars:
00144
00145     LONG PF_maxDollarChunkSize is the size for the portion of the dollar
00146     variable buffer suitable for broadcasting. This variable should be
00147     visible from parallel.c
00148
00149     Evaluate PF_Pack(numterms,1,PF_INT):
00150 */
00151     if ( ( ret = MPI_Pack_size(1,PF_INT,PF_COMM,&bytes) )!=MPI_SUCCESS )
00152         return(ret);
00153
00154     totalbytes+=bytes;
00155 /*
00156     Evaluate PF_Pack( newsize,1,PF_LONG):
00157 */
00158     if ( ( ret = MPI_Pack_size(1,PF_LONG,PF_COMM,&bytes) )!=MPI_SUCCESS )
00159         return(ret);
00160
00161     totalbytes += bytes;
00162 /*
00163     Now available room is PF_packsize-totalbytes
00164 */
00165     totalbytes = PF_packsize-totalbytes;
00166 /*
00167     Now totalbytes is the size of chunk in bytes.
00168     Evaluate this size in number of records:
00169
00170     Rough estimate:
00171 */
00172     PF_maxDollarChunkSize=totalbytes/sizeof(WORD);
00173 /*
00174     Go to the up limit:
00175 */
00176     do {
00177         if ( ( ret = MPI_Pack_size(
00178             ++PF_maxDollarChunkSize,PF_WORD,PF_COMM,&bytes) )!=MPI_SUCCESS )
00179             return(ret);
00180     } while ( bytes<totalbytes );
00181 /*
00182     Now the chunk size is too large
00183     And now evaluate the exact value:
00184 */
00185     do {
00186         if ( ( ret = MPI_Pack_size(
00187             --PF_maxDollarChunkSize,PF_WORD,PF_COMM,&bytes) )!=MPI_SUCCESS )
00188             return(ret);
00189     } while ( bytes>totalbytes );
00190 /*
00191     Now PF_maxDollarChunkSize is the size of chunk of PF_WORD fitting the
00192     buffer <= (PF_packsize-PF_INT-PF_LONG)
00193 */
00194     /*Block*/
00195     return(0);
00196 }
00197 /*
00198 #] PF_LibInit :
00199 #[ PF_LibTerminate :

```

```

00200 */
00201
00209 int PF_LibTerminate(int error)
00210 {
00211     DUMMYUSE(error);
00212     return(MPI_Finalize());
00213 }
00214
00215 /*
00216 #] PF_LibTerminate :
00217 #[ PF_Probe :
00218 */
00219
00230 int PF_Probe(int *src)
00231 {
00232     int ret, flag;
00233     if ( *src == PF_ANY_SOURCE ) { /*Blocking call*/
00234         ret = MPI_Probe(*src,MPI_ANY_TAG,PF_COMM,&PF_status);
00235         flag = 1;
00236     }
00237     else { /*Non-blocking call*/
00238         ret = MPI_Iprobe(*src,MPI_ANY_TAG,PF_COMM,&flag,&PF_status);
00239     }
00240     *src = PF_status.MPI_SOURCE;
00241     if ( ret != MPI_SUCCESS ) { if ( ret > 0 ) ret *= -1; return(ret); }
00242     if ( !flag ) return(0);
00243     return(PF_status.MPI_TAG);
00244 }
00245
00246 /*
00247 #] PF_Probe :
00248 #[ PF_IsendSbuf :
00249 */
00250
00261 int PF_IsendSbuf(int to, int tag)
00262 {
00263     PF_BUFFER *s = PF.sbuf;
00264     int a = s->active;
00265     int size = s->fill[a] - s->buff[a];
00266     int r = 0;
00267
00268     static int finished;
00269
00270     s->fill[a] = s->buff[a];
00271     if ( s->numbufs == 1 ) {
00272         r = MPI_Ssend(s->buff[a],size,PF_WORD,MASTER,tag,PF_COMM);
00273         if ( r != MPI_SUCCESS ) {
00274             fprintf(stderr,"%d|&d] PF_IsendSbuf: MPI_Ssend returns: %d \n",
00275                 PF.me,(int)AC.CModule,r);
00276             fflush(stderr);
00277             return(r);
00278         }
00279         return(0);
00280     }
00281
00282     switch ( tag ) { /* things to do before sending */
00283     case PF_TERM_MSGTAG:
00284         if ( PF.sbuf->request[to] != MPI_REQUEST_NULL)
00285             r = MPI_Wait(&PF.sbuf->request[to],&PF.sbuf->retstat[to]);
00286         if ( r != MPI_SUCCESS ) return(r);
00287         break;
00288     default:
00289         break;
00290     }
00291
00292     r = MPI_Isend(s->buff[a],size,PF_WORD,to,tag,PF_COMM,&s->request[a]);
00293
00294     if ( r != MPI_SUCCESS ) return(r);
00295
00296     switch ( tag ) { /* things to do after initialising sending */
00297     case PF_TERM_MSGTAG:
00298         finished = 0;
00299         break;
00300     case PF_ENDSORT_MSGTAG:
00301         if ( ++finished == PF.numtasks - 1 )
00302             r = MPI_Waitall(s->numbufs,s->request,s->status);
00303         if ( r != MPI_SUCCESS ) return(r);
00304         break;
00305     case PF_BUFFER_MSGTAG:
00306         if ( ++s->active >= s->numbufs ) s->active = 0;
00307         while ( s->request[s->active] != MPI_REQUEST_NULL ) {
00308             r = MPI_Waitsome(s->numbufs,s->request,&size,s->index,s->retstat);
00309             if ( r != MPI_SUCCESS ) return(r);
00310         }
00311         break;
00312     case PF_ENDBUFFER_MSGTAG:
00313         if ( ++s->active >= s->numbufs ) s->active = 0;

```

```

00314         r = MPI_Waitall(s->numbufs,s->request,s->status);
00315         if ( r != MPI_SUCCESS ) return(r);
00316         break;
00317     default:
00318         return(-99);
00319         break;
00320 }
00321 return(0);
00322 }
00323
00324 /*
00325     #] PF_IsendSbuf :
00326     #[ PF_RecvWbuf :
00327 */
00328
00337 int PF_RecvWbuf(WORD *b, LONG *s, int *src)
00338 {
00339     int i, r = 0;
00340
00341     r = MPI_Recv(b, (int)*s, PF_WORD, *src, PF_ANY_MSGTAG, PF_COMM, &PF_status);
00342     if ( r != MPI_SUCCESS ) { if ( r > 0 ) r *= -1; return(r); }
00343
00344     r = MPI_Get_count(&PF_status, PF_WORD, &i);
00345     if ( r != MPI_SUCCESS ) { if ( r > 0 ) r *= -1; return(r); }
00346
00347     *s = (LONG)i;
00348     *src = PF_status.MPI_SOURCE;
00349     return(PF_status.MPI_TAG);
00350 }
00351
00352 /*
00353     #] PF_RecvWbuf :
00354     #[ PF_IRecvRbuf :
00355 */
00356
00366 int PF_IRecvRbuf(PF_BUFFER *r, int bn, int from)
00367 {
00368     int ret;
00369     r->type[bn] = PF_WORD;
00370
00371     if ( r->numbufs == 1 ) {
00372         r->tag[bn] = MPI_ANY_TAG;
00373         r->from[bn] = from;
00374     }
00375     else {
00376         ret = MPI_Irecv(r->full[bn], (int)(r->stop[bn] - r->full[bn]), PF_WORD, from,
00377             MPI_ANY_TAG, PF_COMM, &r->request[bn]);
00378         if (ret != MPI_SUCCESS) { if (ret > 0) ret *= -1; return(ret); }
00379     }
00380     return(0);
00381 }
00382
00383 /*
00384     #] PF_IRecvRbuf :
00385     #[ PF_WaitRbuf :
00386 */
00387
00400 int PF_WaitRbuf(PF_BUFFER *r, int bn, LONG *size)
00401 {
00402     int ret, rsize;
00403
00404     if ( r->numbufs == 1 ) {
00405         *size = r->stop[bn] - r->full[bn];
00406         ret = MPI_Recv(r->full[bn], (int)*size, r->type[bn], r->from[bn], r->tag[bn],
00407             PF_COMM, &(r->status[bn]));
00408         if ( ret != MPI_SUCCESS ) { if ( ret > 0 ) ret *= -1; return(ret); }
00409         ret = MPI_Get_count(&(r->status[bn]), r->type[bn], &rsize);
00410         if ( ret != MPI_SUCCESS ) { if ( ret > 0 ) ret *= -1; return(ret); }
00411         if ( rsize > *size ) return(-99);
00412         *size = (LONG)rsize;
00413     }
00414     else {
00415         while ( r->request[bn] != MPI_REQUEST_NULL ) {
00416             ret = MPI_Waitsome(r->numbufs, r->request, &rsize, r->index, r->retstat);
00417             if ( ret != MPI_SUCCESS ) { if ( ret > 0 ) ret *= -1; return(ret); }
00418             while ( --rsize >= 0 ) r->status[r->index[rsize]] = r->retstat[rsize];
00419         }
00420         ret = MPI_Get_count(&(r->status[bn]), r->type[bn], &rsize);
00421         if ( ret != MPI_SUCCESS ) { if ( ret > 0 ) ret *= -1; return(ret); }
00422         *size = (LONG)rsize;
00423     }
00424     return(r->status[bn].MPI_TAG);
00425 }
00426
00427 /*
00428     #] PF_WaitRbuf :
00429     #[ PF_Bcast :

```

```

00430 */
00431
00440 int PF_Bcast(void *buffer, int count)
00441 {
00442     if ( MPI_Bcast(buffer, count, MPI_BYTE, MASTER, PF_COMM) != MPI_SUCCESS )
00443         return(-1);
00444     return(0);
00445 }
00446
00447 /*
00448     #] PF_Bcast :
00449     #[ PF_RawSend :
00450 */
00451
00463 int PF_RawSend(int dest, void *buf, LONG l, int tag)
00464 {
00465     int ret=MPI_Ssend(buf, (int)l, MPI_BYTE, dest, tag, PF_COMM);
00466     if ( ret != MPI_SUCCESS ) return(-1);
00467     return(0);
00468 }
00469 /*
00470     #] PF_RawSend :
00471     #[ PF_RawRecv :
00472 */
00473
00484 LONG PF_RawRecv(int *src, void *buf, LONG thesize, int *tag)
00485 {
00486     MPI_Status stat;
00487     int ret=MPI_Recv(buf, (int)thesize, MPI_BYTE, *src, MPI_ANY_TAG, PF_COMM, &stat);
00488     if ( ret != MPI_SUCCESS ) return(-1);
00489     if ( MPI_Get_count(&stat, MPI_BYTE, &ret) != MPI_SUCCESS ) return(-1);
00490     *tag = stat.MPI_TAG;
00491     *src = stat.MPI_SOURCE;
00492     return(ret);
00493 }
00494
00495 /*
00496     #] PF_RawRecv :
00497     #[ PF_RawProbe :
00498 */
00499
00508 int PF_RawProbe(int *src, int *tag, int *bytesize)
00509 {
00510     MPI_Status stat;
00511     int srcval = src != NULL ? *src : PF_ANY_SOURCE;
00512     int tagval = tag != NULL ? *tag : PF_ANY_MSGTAG;
00513     int ret = MPI_Probe(srcval, tagval, PF_COMM, &stat);
00514     if ( ret != MPI_SUCCESS ) return -1;
00515     if ( src != NULL ) *src = stat.MPI_SOURCE;
00516     if ( tag != NULL ) *tag = stat.MPI_TAG;
00517     if ( bytesize != NULL ) {
00518         ret = MPI_Get_count(&stat, MPI_BYTE, bytesize);
00519         if ( ret != MPI_SUCCESS ) return -1;
00520     }
00521     return 0;
00522 }
00523
00524 /*
00525     #] PF_RawProbe :
00526     #[ The pack buffer :
00527         #[ Variables :
00528 */
00529
00530 /*
00531     * The pack buffer with the fixed size (= PF_packsize).
00532 */
00533 static UBYTE *PF_packbuf = NULL;
00534 static UBYTE *PF_packstop = NULL;
00535 static int PF_packpos = 0;
00536
00537 /*
00538     #] Variables :
00539     #[ PF_ShortPackInit :
00540 */
00541
00548 static int PF_ShortPackInit(void)
00549 {
00550     PF_packbuf = (UBYTE *)Malloc1(sizeof(UBYTE) * PF_packsize, "PF_ShortPackInit");
00551     if ( PF_packbuf == NULL ) return -1;
00552     PF_packstop = PF_packbuf + PF_packsize;
00553     return 0;
00554 }
00555
00556 /*
00557     #] PF_ShortPackInit :
00558     #[ PF_InitPackBuf :
00559 */

```

```

00560
00566 static inline int PF_InitPackBuf(void)
00567 {
00568 /*
00569      This is definitely not the best place for allocating the
00570      buffer! Moved to PF_LibInit():
00571
00572      if ( PF_packbuf == 0 ) {
00573          PF_packbuf = (UBYTE *)Malloc1(sizeof(UBYTE)*PF.packsize,"PF_InitPackBuf");
00574          if ( PF_packbuf == 0 ) return(-1);
00575          PF_packstop = PF_packbuf + PF.packsize;
00576      }
00577 */
00578     PF_packpos = 0;
00579     return(0);
00580 }
00581
00582 /*
00583      #] PF_InitPackBuf :
00584      #[ PF_PrintPackBuf :
00585 */
00586
00594 int PF_PrintPackBuf(char *s, int size)
00595 {
00596 #ifdef NOMESPRINTYET
00597 /*
00598      The use of printf should be discouraged. The results are flushed to
00599      the output at unpredictable moments. We should use printf only
00600      during startup when MesPrint doesn't have its buffers and output
00601      channels initialized.
00602 */
00603     int i;
00604     printf("[%d] %s: ",PF.me,s);
00605     for(i=0;i<size;i++) printf("%d ",PF_packbuf[i]);
00606     printf("\n");
00607 #else
00608     MesPrint("[%d] %s: %a",PF.me,s,size,(WORD *) (PF_packbuf));
00609 #endif
00610     return(0);
00611 }
00612
00613 /*
00614      #] PF_PrintPackBuf :
00615      #[ PF_PreparePack :
00616 */
00617
00624 int PF_PreparePack(void)
00625 {
00626     return PF_InitPackBuf();
00627 }
00628
00629 /*
00630      #] PF_PreparePack :
00631      #[ PF_Pack :
00632 */
00633
00642 int PF_Pack(const void *buffer, size_t count, MPI_Datatype type)
00643 {
00644     int err, bytes;
00645
00646     if ( count > INT_MAX ) return -99;
00647
00648     err = MPI_Pack_size((int)count, type, PF_COMM, &bytes);
00649     MPI_ERRCODE_CHECK(err);
00650     if ( PF_packpos + bytes > PF_packstop - PF_packbuf ) return -99;
00651
00652     err = MPI_Pack((void *)buffer, (int)count, type, PF_packbuf, PF_packsize, &PF_packpos, PF_COMM);
00653     MPI_ERRCODE_CHECK(err);
00654
00655     return 0;
00656 }
00657
00658 /*
00659      #] PF_Pack :
00660      #[ PF_Unpack :
00661 */
00662
00671 int PF_Unpack(void *buffer, size_t count, MPI_Datatype type)
00672 {
00673     int err;
00674
00675     if ( count > INT_MAX ) return -99;
00676
00677     err = MPI_Unpack(PF_packbuf, PF_packsize, &PF_packpos, buffer, (int)count, type, PF_COMM);
00678     MPI_ERRCODE_CHECK(err);
00679
00680     return 0;

```

```

00681 }
00682
00683 /*
00684     #] PF_Unpack :
00685     #[ PF_PackString :
00686 */
00687
00706 int PF_PackString(const UBYTE *str)
00707 {
00708     int ret,buflength,bytes,length;
00709     /*
00710         length will be packed in the beginning.
00711         Decrement buffer size by the length of the field "length":
00712     */
00713     if ( ( ret = MPI_Pack_size(1,PF_INT,PF_COMM,&bytes) ) != MPI_SUCCESS )
00714         return(ret);
00715     buflength = PF_packsize - bytes;
00716     /*
00717         Calculate the string length (INCLUDING the trailing zero!):
00718     */
00719     for ( length = 0; length < buflength; length++ ) {
00720         if ( str[length] == '\0' ) {
00721             length++; /* since the trailing zero must be accounted */
00722             break;
00723         }
00724     }
00725     /*
00726         The string "\0!\0" is used as an image of the NULL.
00727     */
00728     if ( ( str[0] == '\0' ) /* empty string */
00729         && ( str[1] == '!' ) /* Special case? */
00730         && ( str[2] == '\0' ) /* Yes, pass 3 initial symbols */
00731         ) length += 2; /* all 3 characters will be packed */
00732     length++; /* Will be decremented in the following loop */
00733     /*
00734         The problem: packed size of byte may be not equal 1! So first, suppose
00735         it is 1, and if this is not the case decrease the length of the string
00736         until it fits the buffer:
00737     */
00738     do {
00739         if ( ( ret = MPI_Pack_size(--length,PF_BYTE,PF_COMM,&bytes) )
00740             != MPI_SUCCESS ) return(ret);
00741     } while ( bytes > buflength );
00742     /*
00743         Note, now if str[length-1] == '\0' then the string fits to the buffer
00744         (INCLUDING the trailing zero!);if not, the rest must be packed further!
00745
00746         Pack the length to PF_packbuf:
00747     */
00748     if ( ( ret = MPI_Pack(&length,1,PF_INT,PF_packbuf,PF_packsize,
00749         &PF_packpos,PF_COMM) ) != MPI_SUCCESS ) return(ret);
00750     /*
00751         Pack the string to PF_packbuf:
00752     */
00753     if ( ( ret = MPI_Pack((UBYTE *)str,length,PF_BYTE,PF_packbuf,PF_packsize,
00754         &PF_packpos,PF_COMM) ) != MPI_SUCCESS ) return(ret);
00755     return(length);
00756 }
00757
00758 /*
00759     #] PF_PackString :
00760     #[ PF_UnpackString :
00761 */
00762
00774 int PF_UnpackString(UBYTE *str)
00775 {
00776     int ret,length;
00777     /*
00778         Unpack the length:
00779     */
00780     if( (ret = MPI_Unpack(PF_packbuf,PF_packsize,&PF_packpos,
00781         &length,1,PF_INT,PF_COMM)) != MPI_SUCCESS )
00782         return(ret);
00783     /*
00784         Unpack the string:
00785     */
00786     if ( ( ret = MPI_Unpack(PF_packbuf,PF_packsize,&PF_packpos,
00787         str,length,PF_BYTE,PF_COMM) ) != MPI_SUCCESS ) return(ret);
00788     /*
00789         Now if str[length-1]=='\0' then the whole string
00790         (INCLUDING the trailing zero!) was unpacked ;if not, the rest
00791         must be unpacked to str+length.
00792     */
00793     return(length);
00794 }
00795
00796 /*

```

```

00797         #] PF_UnpackString :
00798         #[ PF_Send :
00799     */
00800
00822 int PF_Send(int to, int tag)
00823 {
00824     int err;
00825     err = MPI_Ssend(PF_packbuf, PF_packpos, MPI_PACKED, to, tag, PF_COMM);
00826     MPI_ERRCODE_CHECK(err);
00827     return 0;
00828 }
00829
00830 /*
00831     #] PF_Send :
00832     #[ PF_Receive :
00833 */
00834
00848 int PF_Receive(int src, int tag, int *psrc, int *ptag)
00849 {
00850     int err;
00851     MPI_Status status;
00852     PF_InitPackBuf();
00853     err = MPI_Recv(PF_packbuf, PF_packsize, MPI_PACKED, src, tag, PF_COMM, &status);
00854     MPI_ERRCODE_CHECK(err);
00855     if ( psrc ) *psrc = status.MPI_SOURCE;
00856     if ( ptag ) *ptag = status.MPI_TAG;
00857     return 0;
00858 }
00859
00860 /*
00861     #] PF_Receive :
00862     #[ PF_Broadcast :
00863 */
00864
00883 int PF_Broadcast(void)
00884 {
00885     int err;
00886     /*
00887     * If PF_SHORTBROADCAST is defined, then the broadcasting will be performed in
00888     * 2 steps. First, the size of the buffer will be broadcast, then the buffer of
00889     * exactly used size. This should be faster with slow connections, but slower on
00890     * SMP shmem MPI because of the latency.
00891     */
00892     #ifdef PF_SHORTBROADCAST
00893         int pos = PF_packpos;
00894     #endif
00895     if ( PF.me != MASTER ) {
00896         err = PF_InitPackBuf();
00897         if ( err ) return err;
00898     }
00899     #ifdef PF_SHORTBROADCAST
00900         err = MPI_Bcast(&pos, 1, MPI_INT, MASTER, PF_COMM);
00901         MPI_ERRCODE_CHECK(err);
00902         err = MPI_Bcast(PF_packbuf, pos, MPI_PACKED, MASTER, PF_COMM);
00903     #else
00904         err = MPI_Bcast(PF_packbuf, PF_packsize, MPI_PACKED, MASTER, PF_COMM);
00905     #endif
00906     MPI_ERRCODE_CHECK(err);
00907     return 0;
00908 }
00909
00910 /*
00911     #] PF_Broadcast :
00912     #] The pack buffer :
00913     #[ Long pack stuff :
00914     #[ Explanations :
00915
00916     The problems here are:
00917     1. We need to send/receive long dollar variables. For
00918     preprocessor-defined dollarvars we used multiply
00919     packing/broadcasting (see parallel.c:PF_BroadcastPreDollar())
00920     since each variable must be broadcast immediately. For run-time
00921     the changed dollar variables, collecting and broadcasting are
00922     performed at the end of the module and all modified dollarvars
00923     are transferred "at once", that is why the size of packed and
00924     transferred buffers may be really very large.
00925     2. There is some strange feature of MPI_Bcast() on Altix MPI
00926     implementation, namely, sometimes it silently fails with big
00927     buffers. For better performance, it would be useful to send one
00928     big buffer instead of several small ones (since the latency is more
00929     important than the bandwidth). That is why we need two different
00930     sets of routines: for long point-to-point communication we collect
00931     big re-allocatable buffer, the corresponding routines have the
00932     prefix PF_longSingle, and for broadcasting we pack data into
00933     several smaller buffers, the corresponding routines have the
00934     prefix PF_longMulti.
00935     Note, from portability reasons we cannot split large packed

```



```

00936     buffer into small chunks, send them and collect back on the other
00937     side, see "Advice to users" on page 180 MPI--The Complete Reference
00938     Volumel, second edition.
00939     OPTIMIZING:
00940     We assume, for most communications, the single buffer of size
00941     PF_packsize is enough.
00942
00943     How does it work:
00944     For point-to-point, there is one big re-allocatable
00945     buffer PF_longPackBuf with two integer positions: PF_longPackPos
00946     and PF_longPackTop (due to re-allocatable character of the buffer,
00947     it is better to use integers rather than pointers).
00948     Each time of re-allocation, the size of the buffer
00949     PF_longPackBuf is incremented by the same size of a "standard" chunk
00950     PF_packsize.
00951     For broadcasting there is one linked list (PF_longMultiRoot),
00952     which contains either positions of a chunk of PF_longPackBuf, or
00953     it's own buffer. This is done for better memory utilisation:
00954     longSingle and longMulti are never used simultaneously.
00955     When a new cell is needed for longMulti packing, we increment
00956     the counter PF_longPackN and just follow the list. If it is not
00957     possible, we allocate the cell's own buffer and link it to the end
00958     of the list PF_longMultiRoot.
00959     When PF_longPackPos is reallocated, we link new chunks into
00960     existing PF_longMultiRoot list before the first longMulti allocated
00961     cell's own buffer. The pointer PF_longMultiLastChunk points to the last
00962     cell of PF_longMultiRoot containing the pointer to the chunk of
00963     PF_longPackBuf.
00964     Initialization PF_longPackBuf is made by the function
00965     PF_longSingleReset(). In the begin of the PF_longPackBuf it packs
00966     the size of the last sent buffer. Upon sending, the program checks,
00967     whether there was at list one re-allocation (PF_longPackN>1) .
00968     If so, the sender first packs and sends small buffer
00969     (PF_longPackSmallBuf) containing one integer number -- the
00970     _negative_ new size of the send buffer. Getting the buffer, a
00971     receiver unpacks one integer and checks whether it is <0 . If so,
00972     the receiver will repeat receiving, but first it checks whether
00973     it has enough buffer and increase it, if necessary.
00974     Initialization PF_longMultiRoot is made by the function
00975     PF_longMultiReset(). In the begin of the first chunk it packs
00976     one integer -- the number 1. Upon sending, the program checks,
00977     how many cells were packed (PF_longPackN). If more than 1, the
00978     sender packs to the next cell the integer PF_longPackN, than
00979     packs PF_longPackN pairs of integers -- the information about how many
00980     times chunk on each cell was accessed by the packing procedure,
00981     this information is contained by the nPacks field of the cell
00982     structure, and how many non-complete items was at the end of this
00983     chunk the structure field lastLen. Then the sender sends first
00984     this auxiliary chunk.
00985     The receiver unpacks the integer from obtained chunk and, if this
00986     integer is more than 1, it gets more chunks, unpacking information
00987     from the first auxiliary chunk into the corresponding nPacks
00988     fields. Unpacking information from multiple chunks, the receiver
00989     knows, when the chunk is expired and it must switch to the next cell,
00990     successively decrementing corresponding nPacks field.
00991
00992     XXX: There are still some flaws:
00993     PF_LongSingleSend/PF_LongSingleReceive may fail, for example, for data
00994     transfers from the master to many slaves. Suppose that the master sends big
00995     data to slaves, which needs an increase of the buffer of the receivers. For
00996     the first data transfer, the master sends the new buffer size as the first
00997     message, and then sends the data as the second message, because
00998     PF_LongSinglePack records the increase of the buffer size on the master. For
00999     the next time, however, the master sends the data without sending the new
01000     buffer size, and then MPI_Recv fails due to the data overflow.
01001     In parallel.c, they are used for the communication from slaves to the
01002     master. In this case, this problem does not occur because the master always
01003     has enough buffer.
01004     The maximum size that PF_LongMultiBroadcast can broadcast is limited to
01005     around 320kB because the current implementation tries to pack all
01006     information of chained buffers into one buffer, whose size is PF_packsize
01007     = 1600B.
01008
01009     #] Explanations :
01010     #[ Variables :
01011 */
01012
01013 typedef struct longMultiStruct {
01014     UBYTE *buffer; /* NULL if */
01015     int bufpos; /* if >=0, PF_longPackBuf+bufpos is the chunk start */
01016     int packpos; /* the current position */
01017     int nPacks; /* How many times PF_longPack operates on this cell */
01018     int lastLen; /* if > 0, the last packing didn't fit completely to this
01019                  chunk, only lastLen items was packed, the rest is in
01020                  the next cell. */
01021     struct longMultiStruct *next; /* next linked cell, or NULL */
01022 } PF_LONGMULTI;

```

```

01023
01024 static UBYTE *PF_longPackBuf = NULL;
01025 static void *PF_longPackSmallBuf = NULL;
01026 static int PF_longPackPos = 0;
01027 static int PF_longPackTop = 0;
01028 static PF_LONGMULTI *PF_longMultiRoot = NULL;
01029 static PF_LONGMULTI *PF_longMultiTop = NULL;
01030 static PF_LONGMULTI *PF_longMultiLastChunk = NULL;
01031 static int PF_longPackN = 0;
01032
01033 /*
01034     #] Variables :
01035     #[ Long pack private functions :
01036     #[ PF_longMultiNewCell :
01037 */
01038
01039 static inline int PF_longMultiNewCell(void)
01040 {
01041     /*
01042         Allocate a new cell:
01043     */
01044     PF_longMultiTop->next = (PF_LONGMULTI *)
01045         Malloc1(sizeof(PF_LONGMULTI), "PF_longMultiCell");
01046     if ( PF_longMultiTop->next == NULL ) return(-1);
01047     /*
01048         Allocate a private buffer:
01049     */
01050     PF_longMultiTop->next->buffer=(UBYTE*)
01051         Malloc1(sizeof(UBYTE)*PF_packsize, "PF_longMultiChunk");
01052     if ( PF_longMultiTop->next->buffer == NULL ) return(-1);
01053     /*
01054         For the private buffer position is -1:
01055     */
01056     PF_longMultiTop->next->bufpos = -1;
01057     /*
01058         This is the last cell in the chain:
01059     */
01060     PF_longMultiTop->next->next = NULL;
01061     /*
01062         packpos and nPacks are not initialized!
01063     */
01064     return(0);
01065 }
01066
01067 /*
01068     #] PF_longMultiNewCell :
01069     #[ PF_longMultiPack2NextCell :
01070 */
01071 static inline int PF_longMultiPack2NextCell(void)
01072 {
01073     /*
01074         Is there a free cell in the chain?
01075     */
01076     if ( PF_longMultiTop->next == NULL ) {
01077         /*
01078             No, allocate the new cell with a private buffer:
01079         */
01080         if ( PF_longMultiNewCell() ) return(-1);
01081     }
01082     /*
01083         Move to the next cell in the chain:
01084     */
01085     PF_longMultiTop = PF_longMultiTop->next;
01086     /*
01087         if >=0, the cell buffer is the chunk of PF_longPackBuf, initialize it:
01088     */
01089     if ( PF_longMultiTop->bufpos > -1 )
01090         PF_longMultiTop->buffer = PF_longPackBuf+PF_longMultiTop->bufpos;
01091     /*
01092         else -- the cell has it's own private buffer.
01093         Initialize the cell fields:
01094     */
01095     PF_longMultiTop->nPacks = 0;
01096     PF_longMultiTop->lastLen = 0;
01097     PF_longMultiTop->packpos = 0;
01098     return(0);
01099 }
01100
01101 /*
01102     #] PF_longMultiPack2NextCell :
01103     #[ PF_longMultiNewChunkAdded :
01104 */
01105
01106 static inline int PF_longMultiNewChunkAdded(int n)
01107 {
01108     /*
01109         Store the list tail:

```

```

01110 */
01111 PF_LONGMULTI *MemCell = PF_longMultiLastChunk->next;
01112 int pos = PF_longPackTop;
01113
01114 while ( n-- > 0 ) {
01115 /*
01116     Allocate a new cell:
01117 */
01118     PF_longMultiLastChunk->next = (PF_LONGMULTI *)
01119         Malloc1(sizeof(PF_LONGMULTI), "PF_longMultiCell");
01120     if ( PF_longMultiLastChunk->next == NULL ) return(-1);
01121 /*
01122     Update the Last Chunk Pointer:
01123 */
01124     PF_longMultiLastChunk = PF_longMultiLastChunk->next;
01125 /*
01126     Initialize the new cell:
01127 */
01128     PF_longMultiLastChunk->bufpos = pos;
01129     pos += PF_packsize;
01130     PF_longMultiLastChunk->buffer = NULL;
01131     PF_longMultiLastChunk->packpos = 0;
01132     PF_longMultiLastChunk->nPacks = 0;
01133     PF_longMultiLastChunk->lastLen = 0;
01134 }
01135 /*
01136     Hitch the tail:
01137 */
01138     PF_longMultiLastChunk->next = MemCell;
01139     return(0);
01140 }
01141
01142 /*
01143     #] PF_longMultiNewChunkAdded :
01144     #[ PF_longCopyChunk :
01145 */
01146
01147 static inline void PF_longCopyChunk(int *to, int *from, int n)
01148 {
01149     NCOPYI(to, from, n)
01150 /* for ( ; n > 0; n-- ) *to++ = *from++; */
01151 }
01152
01153 /*
01154     #] PF_longCopyChunk :
01155     #[ PF_longAddChunk :
01156
01157     The chunk must be increased by n*PF_packsize.
01158 */
01159
01160 static int PF_longAddChunk(int n, int mustRealloc)
01161 {
01162     UBYTE *newbuf;
01163     if ( ( newbuf = (UBYTE*)Malloc1(sizeof(UBYTE)*(PF_longPackTop+n*PF_packsize),
01164         "PF_longPackBuf") ) == NULL ) return(-1);
01165 /*
01166     Allocate and chain a new cell for longMulti:
01167 */
01168     if ( PF_longMultiNewChunkAdded(n) ) return(-1);
01169 /*
01170     Copy the content to the new buffer:
01171 */
01172     if ( mustRealloc ) {
01173         PF_longCopyChunk((int*)newbuf, (int*)PF_longPackBuf, PF_longPackTop/sizeof(int));
01174     }
01175 /*
01176     Note, PF_packsize is multiple by sizeof(int) by construction!
01177 */
01178     PF_longPackTop += (n*PF_packsize);
01179 /*
01180     Free the old buffer and store the new one:
01181 */
01182     M_free(PF_longPackBuf, "PF_longPackBuf");
01183     PF_longPackBuf = newbuf;
01184 /*
01185     Count number of re-allocs:
01186 */
01187     PF_longPackN += n;
01188     return(0);
01189 }
01190
01191 /*
01192     #] PF_longAddChunk :
01193     #[ PF_longMultiHowSplit :
01194
01195     "count" of "type" elements in an input buffer occupy "bytes" bytes.
01196     We know from the algorithm, that it is too many. How to split

```

```

01197     the buffer so that the head fits to rest of a storage buffer?*/
01198 static inline int PF_longMultiHowSplit(int count, MPI_Datatype type, int bytes)
01199 {
01200     int ret, items, totalbytes;
01201
01202     if ( count < 2 ) return(0); /* Nothing to split */
01203 /*
01204     A rest of a storage buffer:
01205 */
01206     totalbytes = PF_packsize - PF_longMultiTop->packpos;
01207 /*
01208     Rough estimate:
01209 */
01210     items = (int)((double)totalbytes*count/bytes);
01211 /*
01212     Go to the up limit:
01213 */
01214     do {
01215         if ( ( ret = MPI_Pack_size(++items,type,PF_COMM,&bytes) )
01216             !=MPI_SUCCESS ) return(ret);
01217     } while ( bytes < totalbytes );
01218 /*
01219     Now the value of "items" is too large
01220     And now evaluate the exact value:
01221 */
01222     do {
01223         if ( ( ret = MPI_Pack_size(--items,type,PF_COMM,&bytes) )
01224             !=MPI_SUCCESS ) return(ret);
01225         if ( items == 0 ) /* Nothing about MPI_Pack_size(0) == 0 in standards! */
01226             return(0);
01227     } while ( bytes > totalbytes );
01228     return(items);
01229 }
01230 /*
01231     #] PF_longMultiHowSplit :
01232     #[ PF_longPackInit :
01233 */
01234
01235 static int PF_longPackInit(void)
01236 {
01237     int ret;
01238     PF_longPackBuf = (UBYTE*)Malloc1(sizeof(UBYTE)*PF_packsize,"PF_longPackBuf");
01239     if ( PF_longPackBuf == NULL ) return(-1);
01240 /*
01241     PF_longPackTop is not initialized yet, use in as a return value:
01242 */
01243     ret = MPI_Pack_size(1,MPI_INT,PF_COMM,&PF_longPackTop);
01244     if ( ret != MPI_SUCCESS ) return(ret);
01245
01246     PF_longPackSmallBuf =
01247         (void*)Malloc1(sizeof(UBYTE)*PF_longPackTop,"PF_longPackSmallBuf");
01248
01249     PF_longPackTop = PF_packsize;
01250     PF_longMultiRoot =
01251         (PF_LONGMULTI *)Malloc1(sizeof(PF_LONGMULTI),"PF_longMultiRoot");
01252     if ( PF_longMultiRoot == NULL ) return(-1);
01253     PF_longMultiRoot->bufpos = 0;
01254     PF_longMultiRoot->buffer = NULL;
01255     PF_longMultiRoot->next = NULL;
01256     PF_longMultiLastChunk = PF_longMultiRoot;
01257
01258     PF_longPackPos = 0;
01259     PF_longMultiRoot->packpos = 0;
01260     PF_longMultiTop = PF_longMultiRoot;
01261     PF_longPackN = 1;
01262     return(0);
01263 }
01264
01265 /*
01266     #] PF_longPackInit :
01267     #[ PF_longMultiPreparePrefix :
01268 */
01269
01270 static inline int PF_longMultiPreparePrefix(void)
01271 {
01272     int ret;
01273     PF_LONGMULTI *thePrefix;
01274     int i = PF_longPackN;
01275 /*
01276     Here we have PF_longPackN>1!
01277     New cell (at the list end) to create the auxiliary chunk:
01278 */
01279     if ( PF_longMultiPack2NextCell() ) return(-1);
01280 /*
01281     Store the pointer to the chunk we will proceed:
01282 */
01283     thePrefix = PF_longMultiTop;

```

```

01284 /*
01285     Pack PF_longPackN:
01286 */
01287     ret = MPI_Pack(&(PF_longPackN),
01288                   1,
01289                   MPI_INT,
01290                   thePrefix->buffer,
01291                   PF_packsize,
01292                   &(thePrefix->packpos),
01293                   PF_COMM);
01294     if ( ret != MPI_SUCCESS ) return(ret);
01295 /*
01296     And start from the beginning:
01297 */
01298     for ( PF_longMultiTop = PF_longMultiRoot; i > 0; i-- ) {
01299 /*
01300     Pack number of Pack hits:
01301 */
01302         ret = MPI_Pack(&(PF_longMultiTop->nPacks),
01303                       1,
01304                       MPI_INT,
01305                       thePrefix->buffer,
01306                       PF_packsize,
01307                       &(thePrefix->packpos),
01308                       PF_COMM);
01309 /*
01310     Pack the length of the last fit portion:
01311 */
01312         ret |= MPI_Pack(&(PF_longMultiTop->lastLen),
01313                       1,
01314                       MPI_INT,
01315                       thePrefix->buffer,
01316                       PF_packsize,
01317                       &(thePrefix->packpos),
01318                       PF_COMM);
01319 /*
01320     Check the size -- not necessary, MPI_Pack did it.
01321 */
01322         if ( ret != MPI_SUCCESS ) return(ret);
01323 /*
01324     Go to the next cell:
01325 */
01326         PF_longMultiTop = PF_longMultiTop->next;
01327     }
01328     PF_longMultiTop = thePrefix;
01329 /*
01330     PF_longMultiTop is ready!
01331 */
01332     return(0);
01333 }
01334 }
01335
01336 /*
01337     #] PF_longMultiPreparePrefix :
01338     #[ PF_longMultiProcessPrefix :
01339 */
01340
01341 static inline int PF_longMultiProcessPrefix(void)
01342 {
01343     int ret,i;
01344 /*
01345     We have PF_longPackN records packed in PF_longMultiRoot->buffer,
01346     pairs nPacks and lastLen. Loop through PF_longPackN cells,
01347     unpacking these integers into proper fields:
01348 */
01349     for ( PF_longMultiTop = PF_longMultiRoot, i = 0; i < PF_longPackN; i++ ) {
01350 /*
01351     Go to th next cell, allocating, when necessary:
01352 */
01353         if ( PF_longMultiPack2NextCell() ) return(-1);
01354 /*
01355     Unpack the number of Pack hits:
01356 */
01357         ret = MPI_Unpack(PF_longMultiRoot->buffer,
01358                         PF_packsize,
01359                         &( PF_longMultiRoot->packpos),
01360                         &(PF_longMultiTop->nPacks),
01361                         1,
01362                         MPI_INT,
01363                         PF_COMM);
01364         if ( ret != MPI_SUCCESS ) return(ret);
01365 /*
01366     Unpack the length of the last fit portion:
01367 */
01368         ret = MPI_Unpack(PF_longMultiRoot->buffer,
01369                         PF_packsize,
01370                         &( PF_longMultiRoot->packpos),

```

```

01371             &(PF_longMultiTop->lastLen),
01372             1,
01373             MPI_INT,
01374             PF_COMM);
01375         if ( ret != MPI_SUCCESS ) return(ret);
01376     }
01377     return(0);
01378 }
01379
01380 /*
01381     #] PF_longMultiProcessPrefix :
01382     #[ PF_longSingleReset :
01383 */
01384
01392 static inline int PF_longSingleReset(int is_sender)
01393 {
01394     int ret;
01395     PF_longPackPos=0;
01396     if ( is_sender ) {
01397         ret = MPI_Pack(&PF_longPackTop,1,MPI_INT,
01398             PF_longPackBuf,PF_longPackTop,&PF_longPackPos,PF_COMM);
01399         if ( ret != MPI_SUCCESS ) return(ret);
01400         PF_longPackN = 1;
01401     }
01402     else {
01403         PF_longPackN=0;
01404     }
01405     return(0);
01406 }
01407
01408 /*
01409     #] PF_longSingleReset :
01410     #[ PF_longMultiReset :
01411 */
01412
01420 static inline int PF_longMultiReset(int is_sender)
01421 {
01422     int ret = 0, theone = 1;
01423     PF_longMultiRoot->packpos = 0;
01424     if ( is_sender ) {
01425         ret = MPI_Pack(&theone,1,MPI_INT,
01426             PF_longPackBuf,PF_longPackTop,&(PF_longMultiRoot->packpos),PF_COMM);
01427         PF_longPackN = 1;
01428     }
01429     else {
01430         PF_longPackN = 0;
01431     }
01432     PF_longMultiRoot->nPacks = 0; /* The auxiliary field is not counted */
01433     PF_longMultiRoot->lastLen = 0;
01434     PF_longMultiTop = PF_longMultiRoot;
01435     PF_longMultiRoot->buffer = PF_longPackBuf;
01436     return ret;
01437 }
01438
01439 /*
01440     #] PF_longMultiReset :
01441     #] Long pack private functions :
01442     #[ PF_PrepareLongSinglePack :
01443 */
01444
01451 int PF_PrepareLongSinglePack(void)
01452 {
01453     return PF_longSingleReset(1);
01454 }
01455
01456 /*
01457     #] PF_PrepareLongSinglePack :
01458     #[ PF_LongSinglePack :
01459 */
01460
01469 int PF_LongSinglePack(const void *buffer, size_t count, MPI_Datatype type)
01470 {
01471     int ret, bytes;
01472     /* XXX: Limited by int size. */
01473     if ( count > INT_MAX ) return -99;
01474     ret = MPI_Pack_size((int)count,type,PF_COMM,&bytes);
01475     if ( ret != MPI_SUCCESS ) return(ret);
01476
01477     while ( PF_longPackPos+bytes > PF_longPackTop ) {
01478         if ( PF_longAddChunk(1, 1) ) return(-1);
01479     }
01480     /*
01481         PF_longAddChunk(1, 1) means, the chunk must
01482         be increased by 1 and re-allocated
01483     */
01484     ret = MPI_Pack((void *)buffer,(int)count,type,
01485         PF_longPackBuf,PF_longPackTop,&PF_longPackPos,PF_COMM);

```

```

01486     if ( ret != MPI_SUCCESS ) return(ret);
01487     return(0);
01488 }
01489
01490 /*
01491     #] PF_LongSinglePack :
01492     #[ PF_LongSingleUnpack :
01493 */
01494
01503 int PF_LongSingleUnpack(void *buffer, size_t count, MPI_Datatype type)
01504 {
01505     int ret;
01506     /* XXX: Limited by int size. */
01507     if ( count > INT_MAX ) return -99;
01508     ret = MPI_Unpack(PF_longPackBuf, PF_longPackTop, &PF_longPackPos,
01509                     buffer, (int)count, type, PF_COMM);
01510     if ( ret != MPI_SUCCESS ) return(ret);
01511     return(0);
01512 }
01513
01514 /*
01515     #] PF_LongSingleUnpack :
01516     #[ PF_LongSingleSend :
01517 */
01518
01540 int PF_LongSingleSend(int to, int tag)
01541 {
01542     int ret, pos = 0;
01543     /*
01544         Note, here we assume that this function couldn't be used
01545         with to == PF_ANY_SOURCE!
01546     */
01547     if ( PF_longPackN > 1 ) {
01548         /* The buffer was incremented, pack send the new size first: */
01549         int tmp = -PF_longPackTop;
01550         /*
01551             Negative value means there will be the second buffer
01552         */
01553         ret = MPI_Pack(&tmp, 1, PF_INT,
01554                       PF_longPackSmallBuf, PF_longPackTop, &pos, PF_COMM);
01555         if ( ret != MPI_SUCCESS ) return(ret);
01556         ret = MPI_Ssend(PF_longPackSmallBuf, pos, MPI_PACKED, to, tag, PF_COMM);
01557         if ( ret != MPI_SUCCESS ) return(ret);
01558     }
01559     ret = MPI_Ssend(PF_longPackBuf, PF_longPackPos, MPI_PACKED, to, tag, PF_COMM);
01560     if ( ret != MPI_SUCCESS ) return(ret);
01561     return(0);
01562 }
01563
01564 /*
01565     #] PF_LongSingleSend :
01566     #[ PF_LongSingleReceive :
01567 */
01568
01583 int PF_LongSingleReceive(int src, int tag, int *psrc, int *ptag)
01584 {
01585     int ret, missed, oncemore;
01586     MPI_Status status;
01587     PF_longSingleReset(0);
01588     do {
01589         ret = MPI_Recv(PF_longPackBuf, PF_longPackTop, MPI_PACKED, src, tag,
01590                       PF_COMM, &status);
01591         if ( ret != MPI_SUCCESS ) return(ret);
01592     /*
01593         The source and tag must be specified here for the case if
01594         MPI_Recv is performed more than once:
01595     */
01596         src = status.MPI_SOURCE;
01597         tag = status.MPI_TAG;
01598         if ( psrc ) *psrc = status.MPI_SOURCE;
01599         if ( ptag ) *ptag = status.MPI_TAG;
01600     /*
01601         Now we got either small buffer with the new PF_longPackTop,
01602         or just a regular chunk.
01603     */
01604         ret = MPI_Unpack(PF_longPackBuf, PF_longPackTop, &PF_longPackPos,
01605                           &missed, 1, MPI_INT, PF_COMM);
01606         if ( ret != MPI_SUCCESS ) return(ret);
01607     /*
01608         if ( missed < 0 ) { /* The small buffer was received. */
01609             oncemore = 1; /* repeat receiving afterwards */
01610             /* Reallocate the buffer and get the data */
01611             missed = -missed;
01612         */
01613         restore after unpacking small from buffer:
01614     */
01615         PF_longPackPos = 0;

```

```

01616     }
01617     else {
01618         oncemore = 0; /* That's all, no repetition */
01619     }
01620     if ( missed > PF_longPackTop ) {
01621         /*
01622          * The room must be increased. We need a re-allocation for the
01623          * case that there is no repetition.
01624          */
01625         if ( PF_longAddChunk( (missed-PF_longPackTop)/PF_packsize, !oncemore ) )
01626             return(-1);
01627     }
01628     } while ( oncemore );
01629     return(0);
01630 }
01631
01632 /*
01633      #] PF_LongSingleReceive :
01634      #[ PF_PrepareLongMultiPack :
01635 */
01636
01643 int PF_PrepareLongMultiPack(void)
01644 {
01645     return PF_longMultiReset(1);
01646 }
01647
01648 /*
01649      #] PF_PrepareLongMultiPack :
01650      #[ PF_LongMultiPackImpl :
01651 */
01652
01662 int PF_LongMultiPackImpl(const void*buffer, size_t count, size_t eSize, MPI_Datatype type)
01663 {
01664     int ret, items;
01665
01666     /* XXX: Limited by int size. */
01667     if ( count > INT_MAX ) return -99;
01668
01669     ret = MPI_Pack_size((int)count,type,PF_COMM,&items);
01670     if ( ret != MPI_SUCCESS ) return(ret);
01671
01672     if ( PF_longMultiTop->packpos + items <= PF_packsize ) {
01673         ret = MPI_Pack((void *)buffer,(int)count,type,PF_longMultiTop->buffer,
01674                       PF_packsize,&(PF_longMultiTop->packpos),PF_COMM);
01675         if ( ret != MPI_SUCCESS ) return(ret);
01676         PF_longMultiTop->nPacks++;
01677         return(0);
01678     }
01679     /*
01680         The data do not fit to the rest of the buffer.
01681         There are two possibilities here: go to the next cell
01682         immediately, or first try to pack some portion. The function
01683         PF_longMultiHowSplit() returns the number of items could be
01684         packed in the end of the current cell:
01685     */
01686     if ( ( items = PF_longMultiHowSplit((int)count,type,items) ) < 0 ) return(items);
01687
01688     if ( items > 0 ) { /* store the head */
01689         ret = MPI_Pack((void *)buffer,items,type,PF_longMultiTop->buffer,
01690                       PF_packsize,&(PF_longMultiTop->packpos),PF_COMM);
01691         if ( ret != MPI_SUCCESS ) return(ret);
01692         PF_longMultiTop->nPacks++;
01693         PF_longMultiTop->lastLen = items;
01694     }
01695     /*
01696         Now the rest should be packed to the new cell.
01697         Slide to the new cell:
01698     */
01699     if ( PF_longMultiPack2NextCell() ) return(-1);
01700     PF_longPackN++;
01701     /*
01702         Pack the rest to the next cell:
01703     */
01704     return(PF_LongMultiPackImpl((char *)buffer+items*eSize,count-items,eSize,type));
01705 }
01706
01707 /*
01708      #] PF_LongMultiPackImpl :
01709      #[ PF_LongMultiUnpackImpl :
01710 */
01711
01721 int PF_LongMultiUnpackImpl(void *buffer, size_t count, size_t eSize, MPI_Datatype type)
01722 {
01723     int ret;
01724
01725     /* XXX: Limited by int size. */
01726     if ( count > INT_MAX ) return -99;

```



```

01727
01728     if ( PF_longPackN < 2 ) { /* Just unpack the buffer from the single cell */
01729         ret = MPI_Unpack(
01730             PF_longMultiTop->buffer,
01731             PF_packsize,
01732             &(PF_longMultiTop->packpos),
01733             buffer,
01734             count,type,PF_COMM);
01735         if ( ret != MPI_SUCCESS ) return(ret);
01736         return(0);
01737     }
01738     /*
01739     More than one cell is in use.
01740     */
01741     if ( ( PF_longMultiTop->nPacks > 1 ) /* the cell is not expired */
01742         || /* The last cell contains exactly required portion: */
01743         ( ( PF_longMultiTop->nPacks == 1 ) && ( PF_longMultiTop->lastLen == 0 ) )
01744     ) { /* Just unpack the buffer from the current cell */
01745         ret = MPI_Unpack(
01746             PF_longMultiTop->buffer,
01747             PF_packsize,
01748             &(PF_longMultiTop->packpos),
01749             buffer,
01750             count,type,PF_COMM);
01751         if ( ret != MPI_SUCCESS ) return(ret);
01752         (PF_longMultiTop->nPacks)--;
01753         return(0);
01754     }
01755     if ( ( PF_longMultiTop->nPacks == 1 ) && ( PF_longMultiTop->lastLen != 0 ) ) {
01756         /*
01757         Unpack the head:
01758         */
01759         ret = MPI_Unpack(
01760             PF_longMultiTop->buffer,
01761             PF_packsize,
01762             &(PF_longMultiTop->packpos),
01763             buffer,
01764             PF_longMultiTop->lastLen,type,PF_COMM);
01765         if ( ret != MPI_SUCCESS ) return(ret);
01766         /*
01767         Decrement the counter by read items:
01768         */
01769         count -= PF_longMultiTop->lastLen;
01770         if ( count <= 0 ) return(-1); /*Something is wrong! */
01771         /*
01772         Shift the output buffer position:
01773         */
01774         buffer = (char *)buffer + PF_longMultiTop->lastLen * eSize;
01775         (PF_longMultiTop->nPacks)--;
01776     }
01777     /*
01778     Here PF_longMultiTop->nPacks == 0
01779     */
01780     if ( ( PF_longMultiTop = PF_longMultiTop->next ) == NULL ) return(-1);
01781     return(PF_LongMultiUnpackImpl(buffer,count,eSize,type));
01782 }
01783
01784 /*
01785     #] PF_LongMultiUnpackImpl :
01786     #[ PF_LongMultiBroadcast :
01787     */
01788
01807 int PF_LongMultiBroadcast(void)
01808 {
01809     int ret, i;
01810
01811     if ( PF.me == MASTER ) {
01812         /*
01813         PF_longPackN is the number of packed chunks. If it is more
01814         than 1, we have to pack a new one and send it first
01815         */
01816         if ( PF_longPackN > 1 ) {
01817             if ( PF_longMultiPreparePrefix() ) return(-1);
01818             ret = MPI_Bcast((void*)PF_longMultiTop->buffer,
01819                 PF_packsize,MPI_PACKED,MASTER,PF_COMM);
01820             if ( ret != MPI_SUCCESS ) return(ret);
01821             /*
01822             PF_longPackN was not incremented by PF_longMultiPreparePrefix()!
01823             */
01824         }
01825         /*
01826         Now we start from the beginning:
01827         */
01828         PF_longMultiTop = PF_longMultiRoot;
01829         /*
01830         Just broadcast all the chunks:
01831         */

```

```

01832         for ( i = 0; i < PF_longPackN; i++ ) {
01833             ret = MPI_Bcast((void*)PF_longMultiTop->buffer,
01834                             PF_packsize,MPI_PACKED,MASTER,PF_COMM);
01835             if ( ret != MPI_SUCCESS ) return(ret);
01836             PF_longMultiTop = PF_longMultiTop->next;
01837         }
01838         return(0);
01839     }
01840     /*
01841         else - the slave
01842     */
01843     PF_longMultiReset(0);
01844     /*
01845         Get the first chunk; it can be either the only data chunk, or
01846         an auxiliary chunk, if the data do not fit the single chunk:
01847     */
01848     ret = MPI_Bcast((void*)PF_longMultiRoot->buffer,
01849                     PF_packsize,MPI_PACKED,MASTER,PF_COMM);
01850     if ( ret != MPI_SUCCESS ) return(ret);
01851
01852     ret = MPI_Unpack((void*)PF_longMultiRoot->buffer,
01853                     PF_packsize,
01854                     &(PF_longMultiRoot->packpos),
01855                     &PF_longPackN,1,MPI_INT,PF_COMM);
01856     if ( ret != MPI_SUCCESS ) return(ret);
01857     /*
01858         Now in PF_longPackN we have the number of cells used
01859         for broadcasting. If it is >1, then we have to allocate
01860         enough cells, initialize them and receive all the chunks.
01861     */
01862     if ( PF_longPackN < 2 ) /* That's all, the single chunk is received. */
01863         return(0);
01864     /*
01865         Here we have to get PF_longPackN chunks. But, first,
01866         initialize cells by info from the received auxiliary chunk.
01867     */
01868     if ( PF_longMultiProcessPrefix() ) return(-1);
01869     /*
01870         Now we have free PF_longPackN cells, starting
01871         from PF_longMultiRoot->next, with properly initialized
01872         nPacks and lastLen fields. Get chunks:
01873     */
01874     for ( PF_longMultiTop = PF_longMultiRoot->next, i = 0; i < PF_longPackN; i++ ) {
01875         ret = MPI_Bcast((void*)PF_longMultiTop->buffer,
01876                         PF_packsize,MPI_PACKED,MASTER,PF_COMM);
01877         if ( ret != MPI_SUCCESS ) return(ret);
01878         if ( i == 0 ) { /* The first chunk, it contains extra "1". */
01879             int tmp;
01880             /*
01881                 Extract this 1 into tmp and forget about it.
01882             */
01883             ret = MPI_Unpack((void*)PF_longMultiTop->buffer,
01884                             PF_packsize,
01885                             &(PF_longMultiTop->packpos),
01886                             &tmp,1,MPI_INT,PF_COMM);
01887             if ( ret != MPI_SUCCESS ) return(ret);
01888         }
01889         PF_longMultiTop = PF_longMultiTop->next;
01890     }
01891     /*
01892         multiUnPack starts with PF_longMultiTop, skip auxiliary chunk in
01893         PF_longMultiRoot:
01894     */
01895     PF_longMultiTop = PF_longMultiRoot->next;
01896     return(0);
01897 }
01898
01899 /*
01900     #] PF_LongMultiBroadcast :
01901     #] Long pack stuff :
01902 */

```

4.81 mpidbg.h File Reference

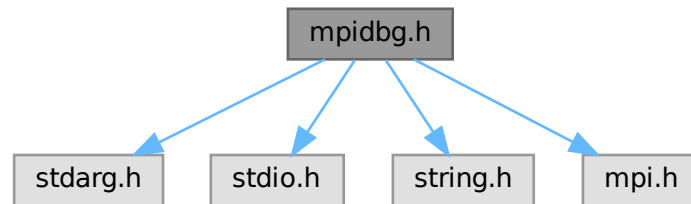
```

#include <stdarg.h>
#include <stdio.h>
#include <string.h>

```

```
#include <mpi.h>
```

Include dependency graph for mpidbg.h:



Macros

- #define [MPIDBG_RANK](#) MPIDBG_Get_rank()
- #define [MPIDBG_Out](#)(...) MPIDBG_Out(file, line, func, __VA_ARGS__)
- #define [MPIDBG_EXTARG](#) const char *file, int line, const char *func
- #define [MPI_Send](#)(...) MPIDBG_Send(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Recv](#)(...) MPIDBG_Recv(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Bsend](#)(...) MPIDBG_Bsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Ssend](#)(...) MPIDBG_Ssend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Rsend](#)(...) MPIDBG_Rsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Isend](#)(...) MPIDBG_Isend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Ibsend](#)(...) MPIDBG_Ibsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Issend](#)(...) MPIDBG_Issend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Irsend](#)(...) MPIDBG_Irsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Irecv](#)(...) MPIDBG_Irecv(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Wait](#)(...) MPIDBG_Wait(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Test](#)(...) MPIDBG_Test(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Waitany](#)(...) MPIDBG_Waitany(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Testany](#)(...) MPIDBG_Testany(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Waitall](#)(...) MPIDBG_Waitall(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Testall](#)(...) MPIDBG_Testall(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Waitsome](#)(...) MPIDBG_Waitsome(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Testsome](#)(...) MPIDBG_Testsome(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Iprobe](#)(...) MPIDBG_Iprobe(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Probe](#)(...) MPIDBG_Probe(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Cancel](#)(...) MPIDBG_Cancel(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Test_cancelled](#)(...) MPIDBG_Test_cancelled(__VA_ARGS__, __FILE__, __LINE__, __func__,
↵
__)
- #define [MPI_Barrier](#)(...) MPIDBG_Barrier(__VA_ARGS__, __FILE__, __LINE__, __func__)
- #define [MPI_Bcast](#)(...) MPIDBG_Bcast(__VA_ARGS__, __FILE__, __LINE__, __func__)

4.81.1 Detailed Description

MPI APIs with the logging feature. NOTE: This file needs C99.

Definition in file [mpidbg.h](#).

4.81.2 Macro Definition Documentation

4.81.2.1 MPIDBG_RANK

```
#define MPIDBG_RANK MPIDBG_Get_rank()
```

Definition at line 55 of file [mpidbg.h](#).

4.81.2.2 MPIDBG_Out

```
#define MPIDBG_Out(  
    ... ) MPIDBG_Out(file, line, func, __VA_ARGS__)
```

Definition at line 72 of file [mpidbg.h](#).

4.81.2.3 MPIDBG_EXTARG

```
#define MPIDBG_EXTARG const char *file, int line, const char *func
```

Definition at line 139 of file [mpidbg.h](#).

4.81.2.4 MPI_Send

```
#define MPI_Send(  
    ... ) MPIDBG_Send(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 157 of file [mpidbg.h](#).

4.81.2.5 MPI_Recv

```
#define MPI_Recv(  
    ... ) MPIDBG_Recv(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 178 of file [mpidbg.h](#).

4.81.2.6 MPI_Bsend

```
#define MPI_Bsend(  
    ... ) MPIDBG_Bsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 197 of file [mpidbg.h](#).

4.81.2.7 MPI_Ssend

```
#define MPI_Ssend(  
    ... ) MPIDBG_Ssend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 216 of file [mpidbg.h](#).

4.81.2.8 MPI_Rsend

```
#define MPI_Rsend(  
    ... ) MPIDBG_Rsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 235 of file [mpidbg.h](#).

4.81.2.9 MPI_Isend

```
#define MPI_Isend(  
    ... ) MPIDBG_Isend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 254 of file [mpidbg.h](#).

4.81.2.10 MPI_Ibsend

```
#define MPI_Ibsend(  
    ... ) MPIDBG_Ibsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 273 of file [mpidbg.h](#).

4.81.2.11 MPI_Issend

```
#define MPI_Issend(  
    ... ) MPIDBG_Issend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 292 of file [mpidbg.h](#).

4.81.2.12 MPI_Irsend

```
#define MPI_Irsend(  
    ... ) MPIDBG_Irsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 311 of file [mpidbg.h](#).

4.81.2.13 MPI_Irecv

```
#define MPI_Irecv(  
    ... ) MPIDBG_Irecv(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 330 of file [mpidbg.h](#).

4.81.2.14 MPI_Wait

```
#define MPI_Wait(  
    ... ) MPIDBG_Wait(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 353 of file [mpidbg.h](#).

4.81.2.15 MPI_Test

```
#define MPI_Test(  
    ... ) MPIDBG_Test(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 381 of file [mpidbg.h](#).

4.81.2.16 MPI_Waitany

```
#define MPI_Waitany(  
    ... ) MPIDBG_Waitany(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 407 of file [mpidbg.h](#).

4.81.2.17 MPI_Testany

```
#define MPI_Testany(  
    ... ) MPIDBG_Testany(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 438 of file [mpidbg.h](#).

4.81.2.18 MPI_Waitall

```
#define MPI_Waitall(  
    ... ) MPIDBG_Waitall(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 462 of file [mpidbg.h](#).

4.81.2.19 MPI_Testall

```
#define MPI_Testall(  
    ... ) MPIDBG_Testall(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 491 of file [mpidbg.h](#).

4.81.2.20 MPI_Waitsome

```
#define MPI_Waitsome(  
    ... ) MPIDBG_Waitsome(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 515 of file [mpidbg.h](#).

4.81.2.21 MPI_Testsome

```
#define MPI_Testsome(  
    ... ) MPIDBG_Testsome(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 539 of file [mpidbg.h](#).

4.81.2.22 MPI_Iprobe

```
#define MPI_Iprobe(  
    ... ) MPIDBG_Iprobe(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 565 of file [mpidbg.h](#).

4.81.2.23 MPI_Probe

```
#define MPI_Probe(  
    ... ) MPIDBG_Probe(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 586 of file [mpidbg.h](#).

4.81.2.24 MPI_Cancel

```
#define MPI_Cancel(  
    ... ) MPIDBG_Cancel(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 605 of file [mpidbg.h](#).

4.81.2.25 MPI_Test_cancelled

```
#define MPI_Test_cancelled(  
    ... ) MPIDBG_Test_cancelled(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 629 of file [mpidbg.h](#).

4.81.2.26 MPI_Barrier

```
#define MPI_Barrier(  
    ... ) MPIDBG_Barrier(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 648 of file [mpidbg.h](#).

4.81.2.27 MPI_Bcast

```
#define MPI_Bcast(  
    ... ) MPIDBG_Bcast(__VA_ARGS__, __FILE__, __LINE__, __func__)
```

Definition at line 667 of file [mpidbg.h](#).

4.82 mpidbg.h

[Go to the documentation of this file.](#)

```

00001 #ifndef MPIDBG_H
00002 #define MPIDBG_H
00003
00010 /* #[] License : */
00011 /*
00012  * Copyright (C) 1984-2026 J.A.M. Vermaseren
00013  * When using this file you are requested to refer to the publication
00014  * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00015  * This is considered a matter of courtesy as the development was paid
00016  * for by FOM the Dutch physics granting agency and we would like to
00017  * be able to track its scientific use to convince FOM of its value
00018  * for the community.
00019  *
00020  * This file is part of FORM.
00021  *
00022  * FORM is free software: you can redistribute it and/or modify it under the
00023  * terms of the GNU General Public License as published by the Free Software
00024  * Foundation, either version 3 of the License, or (at your option) any later
00025  * version.
00026  *
00027  * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00028  * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00029  * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00030  * details.
00031  *
00032  * You should have received a copy of the GNU General Public License along
00033  * with FORM. If not, see <http://www.gnu.org/licenses/>.
00034 */
00035 /* #[] License : */
00036
00037 /*
00038  #[] Includes :
00039 */
00040
00041 #include <stdarg.h>
00042 #include <stdio.h>
00043 #include <string.h>
00044 #include <mpi.h>
00045
00046 /*
00047  #[] Includes :
00048  #[] Utilities :
00049  #[] MPIDBG_RANK :
00050 */
00051
00052 static inline int MPIDBG_Get_rank(void) {
00053     return PF.me; /* Assume we are working with ParFORM. */
00054 }
00055 #define MPIDBG_RANK MPIDBG_Get_rank()
00056
00057 /*
00058  #[] MPIDBG_RANK :
00059  #[] MPIDBG_Out :
00060 */
00061
00062 static inline void MPIDBG_Out(const char *file, int line, const char *func, const char *fmt, ...) {
00063     char buf[1024]; /* Enough. */
00064     va_list ap;
00065     va_start(ap, fmt);
00066     snprintf(buf, 1024, "*** [%d] %10s %4d @ %-16s: ", MPIDBG_RANK, file, line, func);
00067     vsnprintf(buf + strlen(buf), 1024 - strlen(buf), fmt, ap);
00068     va_end(ap);
00069     /* Assume fprintf with a line will work well even in multi-processes. */
00070     fprintf(stderr, "%s\n", buf);
00071 }
00072 #define MPIDBG_Out(...) MPIDBG_Out(file, line, func, __VA_ARGS__)
00073
00074 /*
00075  #[] MPIDBG_Out :
00076  #[] MPIDBG_sprint_requests :
00077 */
00078
00079 static inline void MPIDBG_sprint_requests(char *buf, int count, const MPI_Request *requests)
00080 {
00081     /* Assume sprintf never fail and returns >= 0... */
00082     buf += sprintf(buf, "(");
00083     int i, first = 1;
00084     for (i = 0; i < count; i++) {
00085         if (first) {
00086             first = 0;
00087         }
00088         else {

```



```

00089         buf += sprintf(buf, ",");
00090     }
00091     if ( requests[i] != MPI_REQUEST_NULL ) {
00092         buf += sprintf(buf, "%d", i);
00093     }
00094     else {
00095         buf += sprintf(buf, "*");
00096     }
00097 }
00098 buf += sprintf(buf, " ");
00099 }
00100
00101 /*
00102     #] MPIDBG_sprint_requests :
00103     #[ MPIDBG_sprint_statuses :
00104 */
00105
00106 static inline void MPIDBG_sprint_statuses(char *buf, int count, const MPI_Request *old_requests, const
MPI_Request *new_requests, const MPI_Status *statuses)
00107 {
00108     /* Assume sprintf never fail and returns >= 0... */
00109     buf += sprintf(buf, "(");
00110     int i, first = 1;
00111     for ( i = 0; i < count; i++ ) {
00112         if ( first ) {
00113             first = 0;
00114         }
00115         else {
00116             buf += sprintf(buf, ",");
00117         }
00118         if ( old_requests[i] != MPI_REQUEST_NULL && new_requests[i] == MPI_REQUEST_NULL ) {
00119             int ret_size = 0;
00120             MPI_Get_count((MPI_Status *)&statuses[i], MPI_BYTE, &ret_size);
00121             buf += sprintf(buf, "(source=%d,tag=%d,size=%d)", statuses[i].MPI_SOURCE,
statuses[i].MPI_TAG, ret_size);
00122         } else {
00123             buf += sprintf(buf, "*");
00124         }
00125     }
00126     buf += sprintf(buf, " ");
00127 }
00128
00129 /*
00130     #] MPIDBG_sprint_statuses :
00131     #] Utilities :
00132     #[ MPI APIs :
00133 */
00134
00135 /*
00136 * The followings are the MPI APIs with the logging.
00137 */
00138
00139 #define MPIDBG_EXTARG const char *file, int line, const char *func
00140
00141 /*
00142     #[ MPI_Send :
00143 */
00144
00145 static inline int MPIDBG_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm
comm, MPIDBG_EXTARG)
00146 {
00147     MPIDBG_Out("MPI_Send: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00148     int ret = MPI_Send(buf, count, datatype, dest, tag, comm);
00149     if ( ret == MPI_SUCCESS ) {
00150         MPIDBG_Out("MPI_Send: OK");
00151     }
00152     else {
00153         MPIDBG_Out("MPI_Send: Failed");
00154     }
00155     return ret;
00156 }
00157 #define MPI_Send(...) MPIDBG_Send(__VA_ARGS__, __FILE__, __LINE__, __func__)
00158
00159 /*
00160     #] MPI_Send :
00161     #[ MPI_Recv :
00162 */
00163
00164 static inline int MPIDBG_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag,
MPI_Comm comm, MPI_Status *status, MPIDBG_EXTARG)
00165 {
00166     MPIDBG_Out("MPI_Recv: src=%d dest=%d tag=%d", source, MPIDBG_RANK, tag);
00167     int ret = MPI_Recv(buf, count, datatype, source, tag, comm, status);
00168     if ( ret == MPI_SUCCESS ) {
00169         int ret_count = 0;
00170         MPI_Get_count(status, datatype, &ret_count);
00171         MPIDBG_Out("MPI_Recv: OK src=%d dest=%d tag=%d count=%d", status->MPI_SOURCE, MPIDBG_RANK,

```

```

        status->MPI_TAG, ret_count);
00172     }
00173     else {
00174         MPIDBG_Out("MPI_Recv: Failed");
00175     }
00176     return ret;
00177 }
00178 #define MPI_Recv(...) MPIDBG_Recv(__VA_ARGS__, __FILE__, __LINE__, __func__)
00179
00180 /*
00181     #] MPI_Recv :
00182     #[ MPI_Bsend :
00183 */
00184
00185 static inline int MPIDBG_Bsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPIDBG_EXTARG)
00186 {
00187     MPIDBG_Out("MPI_Bsend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00188     int ret = MPI_Bsend(buf, count, datatype, dest, tag, comm);
00189     if ( ret == MPI_SUCCESS ) {
00190         MPIDBG_Out("MPI_Bsend: OK");
00191     }
00192     else {
00193         MPIDBG_Out("MPI_Bsend: Failed");
00194     }
00195     return ret;
00196 }
00197 #define MPI_Bsend(...) MPIDBG_Bsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
00198
00199 /*
00200     #] MPI_Bsend :
00201     #[ MPI_Ssend :
00202 */
00203
00204 static inline int MPIDBG_Ssend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPIDBG_EXTARG)
00205 {
00206     MPIDBG_Out("MPI_Ssend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00207     int ret = MPI_Ssend(buf, count, datatype, dest, tag, comm);
00208     if ( ret == MPI_SUCCESS ) {
00209         MPIDBG_Out("MPI_Ssend: OK");
00210     }
00211     else {
00212         MPIDBG_Out("MPI_Ssend: Failed");
00213     }
00214     return ret;
00215 }
00216 #define MPI_Ssend(...) MPIDBG_Ssend(__VA_ARGS__, __FILE__, __LINE__, __func__)
00217
00218 /*
00219     #] MPI_Ssend :
00220     #[ MPI_Rsend :
00221 */
00222
00223 static inline int MPIDBG_Rsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPIDBG_EXTARG)
00224 {
00225     MPIDBG_Out("MPI_Rsend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00226     int ret = MPI_Rsend(buf, count, datatype, dest, tag, comm);
00227     if ( ret == MPI_SUCCESS ) {
00228         MPIDBG_Out("MPI_Rsend: OK");
00229     }
00230     else {
00231         MPIDBG_Out("MPI_Rsend: Failed");
00232     }
00233     return ret;
00234 }
00235 #define MPI_Rsend(...) MPIDBG_Rsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
00236
00237 /*
00238     #] MPI_Rsend :
00239     #[ MPI_Isend :
00240 */
00241
00242 static inline int MPIDBG_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPI_Request *request, MPIDBG_EXTARG)
00243 {
00244     MPIDBG_Out("MPI_Isend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00245     int ret = MPI_Isend(buf, count, datatype, dest, tag, comm, request);
00246     if ( ret == MPI_SUCCESS ) {
00247         MPIDBG_Out("MPI_Isend: OK");
00248     }
00249     else {
00250         MPIDBG_Out("MPI_Isend: Failed");
00251     }
00252     return ret;
00253 }

```

```

00254 #define MPI_Isend(...) MPIDBG_Isend(__VA_ARGS__, __FILE__, __LINE__, __func__)
00255
00256 /*
00257     #] MPI_Isend :
00258     #[ MPI_Ibsend :
00259 */
00260
00261 static inline int MPIDBG_Ibsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm, MPI_Request *request, MPIDBG_EXTARG)
00262 {
00263     MPIDBG_Out("MPI_Ibsend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00264     int ret = MPI_Ibsend(buf, count, datatype, dest, tag, comm, request);
00265     if ( ret == MPI_SUCCESS ) {
00266         MPIDBG_Out("MPI_Ibsend: OK");
00267     }
00268     else {
00269         MPIDBG_Out("MPI_Ibsend: Failed");
00270     }
00271     return ret;
00272 }
00273 #define MPI_Ibsend(...) MPIDBG_Ibsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
00274
00275 /*
00276     #] MPI_Ibsend :
00277     #[ MPI_Issend :
00278 */
00279
00280 static inline int MPIDBG_Issend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm, MPI_Request *request, MPIDBG_EXTARG)
00281 {
00282     MPIDBG_Out("MPI_Issend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00283     int ret = MPI_Issend(buf, count, datatype, dest, tag, comm, request);
00284     if ( ret == MPI_SUCCESS ) {
00285         MPIDBG_Out("MPI_Issend: OK");
00286     }
00287     else {
00288         MPIDBG_Out("MPI_Issend: Failed");
00289     }
00290     return ret;
00291 }
00292 #define MPI_Issend(...) MPIDBG_Issend(__VA_ARGS__, __FILE__, __LINE__, __func__)
00293
00294 /*
00295     #] MPI_Issend :
00296     #[ MPI_Irsend :
00297 */
00298
00299 static inline int MPIDBG_Irsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm, MPI_Request *request, MPIDBG_EXTARG)
00300 {
00301     MPIDBG_Out("MPI_Irsend: src=%d dest=%d tag=%d count=%d", MPIDBG_RANK, dest, tag, count);
00302     int ret = MPI_Irsend(buf, count, datatype, dest, tag, comm, request);
00303     if ( ret == MPI_SUCCESS ) {
00304         MPIDBG_Out("MPI_Irsend: OK");
00305     }
00306     else {
00307         MPIDBG_Out("MPI_Irsend: Failed");
00308     }
00309     return ret;
00310 }
00311 #define MPI_Irsend(...) MPIDBG_Irsend(__VA_ARGS__, __FILE__, __LINE__, __func__)
00312
00313 /*
00314     #] MPI_Irsend :
00315     #[ MPI_Irecv :
00316 */
00317
00318 static inline int MPIDBG_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag,
    MPI_Comm comm, MPI_Request *request, MPIDBG_EXTARG)
00319 {
00320     MPIDBG_Out("MPI_Irecv: src=%d dest=%d tag=%d", source, MPIDBG_RANK, tag);
00321     int ret = MPI_Irecv(buf, count, datatype, source, tag, comm, request);
00322     if ( ret == MPI_SUCCESS ) {
00323         MPIDBG_Out("MPI_Irecv: OK dest=%d", MPIDBG_RANK);
00324     }
00325     else {
00326         MPIDBG_Out("MPI_Irecv: Failed");
00327     }
00328     return ret;
00329 }
00330 #define MPI_Irecv(...) MPIDBG_Irecv(__VA_ARGS__, __FILE__, __LINE__, __func__)
00331
00332 /*
00333     #] MPI_Irecv :
00334     #[ MPI_Wait :
00335 */
00336

```

```

00337 static inline int MPIDBG_Wait(MPI_Request *request, MPI_Status *status, MPIDBG_EXTARG)
00338 {
00339     char buf[256 * 1]; /* Enough. */
00340     MPI_Request old_request = *request;
00341     MPIDBG_sprint_requests(buf, 1, request);
00342     MPIDBG_Out("MPI_Wait: rank=%d request=%s", MPIDBG_RANK, buf);
00343     int ret = MPI_Wait(request, status);
00344     if ( ret == MPI_SUCCESS ) {
00345         MPIDBG_sprint_statuses(buf, 1, request, &old_request, status);
00346         MPIDBG_Out("MPI_Wait: OK rank=%d result=%s", MPIDBG_RANK, buf);
00347     }
00348     else {
00349         MPIDBG_Out("MPI_Wait: Failed");
00350     }
00351     return ret;
00352 }
00353 #define MPI_Wait(...) MPIDBG_Wait(__VA_ARGS__, __FILE__, __LINE__, __func__)
00354
00355 /*
00356     #] MPI_Wait :
00357     #[ MPI_Test :
00358 */
00359
00360 static inline int MPIDBG_Test(MPI_Request *request, int *flag, MPI_Status *status, MPIDBG_EXTARG)
00361 {
00362     char buf[256 * 1]; /* Enough. */
00363     MPI_Request old_request = *request;
00364     MPIDBG_sprint_requests(buf, 1, request);
00365     MPIDBG_Out("MPI_Test: rank=%d request=%s", MPIDBG_RANK, buf);
00366     int ret = MPI_Test(request, flag, status);
00367     if ( ret == MPI_SUCCESS ) {
00368         if ( *flag ) {
00369             MPIDBG_sprint_statuses(buf, 1, request, &old_request, status);
00370             MPIDBG_Out("MPI_Test: OK rank=%d result=%s", MPIDBG_RANK, buf);
00371         }
00372         else {
00373             MPIDBG_Out("MPI_Test: OK flag=false");
00374         }
00375     }
00376     else {
00377         MPIDBG_Out("MPI_Test: Failed");
00378     }
00379     return ret;
00380 }
00381 #define MPI_Test(...) MPIDBG_Test(__VA_ARGS__, __FILE__, __LINE__, __func__)
00382
00383 /*
00384     #] MPI_Test :
00385     #[ MPI_Waitany :
00386 */
00387
00388 static inline int MPIDBG_Waitany(int count, MPI_Request *array_of_requests, int *index, MPI_Status
*status, MPIDBG_EXTARG)
00389 {
00390     char buf[256]; /* Enough. */
00391     MPI_Request old_requests[count];
00392     memcpy(old_requests, array_of_requests, sizeof(MPI_Request) * count);
00393     MPIDBG_sprint_requests(buf, count, array_of_requests);
00394     MPIDBG_Out("MPI_Waitany: rank=%d request=%s", MPIDBG_RANK, buf);
00395     int ret = MPI_Waitany(count, array_of_requests, index, status);
00396     if ( ret == MPI_SUCCESS ) {
00397         MPI_Status statuses[count];
00398         statuses[*index] = *status;
00399         MPIDBG_sprint_statuses(buf, count, old_requests, array_of_requests, statuses);
00400         MPIDBG_Out("MPI_Waitany: OK rank=%d result=%s", MPIDBG_RANK, buf);
00401     }
00402     else {
00403         MPIDBG_Out("MPI_Waitany: Failed");
00404     }
00405     return ret;
00406 }
00407 #define MPI_Waitany(...) MPIDBG_Waitany(__VA_ARGS__, __FILE__, __LINE__, __func__)
00408
00409 /*
00410     #] MPI_Waitany :
00411     #[ MPI_Testany :
00412 */
00413
00414 static inline int MPIDBG_Testany(int count, MPI_Request *array_of_requests, int *index, int *flag,
MPI_Status *status, MPIDBG_EXTARG)
00415 {
00416     char buf[256]; /* Enough. */
00417     MPI_Request old_requests[count];
00418     memcpy(old_requests, array_of_requests, sizeof(MPI_Request) * count);
00419     MPIDBG_sprint_requests(buf, count, array_of_requests);
00420     MPIDBG_Out("MPI_Testany: rank=%d request=%s", MPIDBG_RANK, buf);
00421     int ret = MPI_Testany(count, array_of_requests, index, flag, status);

```

```

00422     if ( ret == MPI_SUCCESS ) {
00423         if ( *flag ) {
00424             MPI_Status statuses[count];
00425             statuses[*index] = *status;
00426             MPIDBG_sprint_statuses(buf, count, old_requests, array_of_requests, statuses);
00427             MPIDBG_Out("MPI_Testany: OK rank=%d result=%s", MPIDBG_RANK, buf);
00428         }
00429         else {
00430             MPIDBG_Out("MPI_Testany: OK flag=false");
00431         }
00432     }
00433     else {
00434         MPIDBG_Out("MPI_Testany: Failed");
00435     }
00436     return ret;
00437 }
00438 #define MPI_Testany(...) MPIDBG_Testany(__VA_ARGS__, __FILE__, __LINE__, __func__)
00439
00440 /*
00441     #] MPI_Testany :
00442     #[ MPI_Waitall :
00443 */
00444
00445 static inline int MPIDBG_Waitall(int count, MPI_Request *array_of_requests, MPI_Status
*array_of_statuses, MPIDBG_EXTARG)
00446 {
00447     char buf[256 * count]; /* Enough. */
00448     MPI_Request old_requests[count];
00449     memcpy(old_requests, array_of_requests, sizeof(MPI_Request) * count);
00450     MPIDBG_sprint_requests(buf, count, array_of_requests);
00451     MPIDBG_Out("MPI_Waitall: rank=%d request=%s", MPIDBG_RANK, buf);
00452     int ret = MPI_Waitall(count, array_of_requests, array_of_statuses);
00453     if ( ret == MPI_SUCCESS ) {
00454         MPIDBG_sprint_statuses(buf, count, old_requests, array_of_requests, array_of_statuses);
00455         MPIDBG_Out("MPI_Waitall: OK rank=%d result=%s", MPIDBG_RANK, buf);
00456     }
00457     else {
00458         MPIDBG_Out("MPI_Waitall: Failed");
00459     }
00460     return ret;
00461 }
00462 #define MPI_Waitall(...) MPIDBG_Waitall(__VA_ARGS__, __FILE__, __LINE__, __func__)
00463
00464 /*
00465     #] MPI_Waitall :
00466     #[ MPI_Testall :
00467 */
00468
00469 static inline int MPIDBG_Testall(int count, MPI_Request *array_of_requests, int *flag, MPI_Status
*array_of_statuses, MPIDBG_EXTARG)
00470 {
00471     char buf[256 * count]; /* Enough. */
00472     MPI_Request old_requests[count];
00473     memcpy(old_requests, array_of_requests, sizeof(MPI_Request) * count);
00474     MPIDBG_sprint_requests(buf, count, array_of_requests);
00475     MPIDBG_Out("MPI_Testall: rank=%d request=%s", MPIDBG_RANK, buf);
00476     int ret = MPI_Testall(count, array_of_requests, flag, array_of_statuses);
00477     if ( ret == MPI_SUCCESS ) {
00478         if ( *flag ) {
00479             MPIDBG_sprint_statuses(buf, count, old_requests, array_of_requests, array_of_statuses);
00480             MPIDBG_Out("MPI_Testall: OK rank=%d result=%s", MPIDBG_RANK, buf);
00481         }
00482         else {
00483             MPIDBG_Out("MPI_Testall: OK flag=false");
00484         }
00485     }
00486     else {
00487         MPIDBG_Out("MPI_Testall: Failed");
00488     }
00489     return ret;
00490 }
00491 #define MPI_Testall(...) MPIDBG_Testall(__VA_ARGS__, __FILE__, __LINE__, __func__)
00492
00493 /*
00494     #] MPI_Testall :
00495     #[ MPI_Waitsome :
00496 */
00497
00498 static inline int MPIDBG_Waitsome(int incount, MPI_Request *array_of_requests, int *outcount, int
*array_of_indices, MPI_Status *array_of_statuses, MPIDBG_EXTARG)
00499 {
00500     char buf[256 * incount]; /* Enough. */
00501     MPI_Request old_requests[incount];
00502     memcpy(old_requests, array_of_requests, sizeof(MPI_Request) * incount);
00503     MPIDBG_sprint_requests(buf, incount, array_of_requests);
00504     MPIDBG_Out("MPI_Waitsome: rank=%d request=%s", MPIDBG_RANK, buf);
00505     int ret = MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses);

```

```

00506     if ( ret == MPI_SUCCESS ) {
00507         MPIDBG_sprint_statuses(buf, incount, old_requests, array_of_requests, array_of_statuses);
00508         MPIDBG_Out("MPI_Waitosome: OK rank=%d result=%s", MPIDBG_RANK, buf);
00509     }
00510     else {
00511         MPIDBG_Out("MPI_Waitosome: Failed");
00512     }
00513     return ret;
00514 }
00515 #define MPI_Waitosome(...) MPIDBG_Waitosome(__VA_ARGS__, __FILE__, __LINE__, __func__)
00516
00517 /*
00518     #] MPI_Waitosome :
00519     #[ MPI_Testosome :
00520 */
00521
00522 static inline int MPIDBG_Testosome(int incount, MPI_Request *array_of_requests, int *outcount, int
    *array_of_indices, MPI_Status *array_of_statuses, MPIDBG_EXTARG)
00523 {
00524     char buf[256 * incount]; /* Enough. */
00525     MPI_Request old_requests[incount];
00526     memcpy(old_requests, array_of_requests, sizeof(MPI_Request) * incount);
00527     MPIDBG_sprint_requests(buf, incount, array_of_requests);
00528     MPIDBG_Out("MPI_Testosome: rank=%d request=%s", MPIDBG_RANK, buf);
00529     int ret = MPI_Testosome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses);
00530     if ( ret == MPI_SUCCESS ) {
00531         MPIDBG_sprint_statuses(buf, incount, old_requests, array_of_requests, array_of_statuses);
00532         MPIDBG_Out("MPI_Testosome: OK rank=%d result=%s", MPIDBG_RANK, buf);
00533     }
00534     else {
00535         MPIDBG_Out("MPI_Testosome: Failed");
00536     }
00537     return ret;
00538 }
00539 #define MPI_Testosome(...) MPIDBG_Testosome(__VA_ARGS__, __FILE__, __LINE__, __func__)
00540
00541 /*
00542     #] MPI_Testosome :
00543     #[ MPI_Iprobe :
00544 */
00545
00546 static inline int MPIDBG_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status,
    MPIDBG_EXTARG)
00547 {
00548     MPIDBG_Out("MPI_Iprobe: src=%d dest=%d tag=%d", source, MPIDBG_RANK, tag);
00549     int ret = MPI_Iprobe(source, tag, comm, flag, status);
00550     if ( ret == MPI_SUCCESS ) {
00551         if ( *flag ) {
00552             int ret_size = 0;
00553             MPI_Get_count(status, MPI_BYTE, &ret_size);
00554             MPIDBG_Out("MPI_Iprobe: OK src=%d dest=%d tag=%d size=%d", status->MPI_SOURCE,
    MPIDBG_RANK, status->MPI_TAG, ret_size);
00555         }
00556         else {
00557             MPIDBG_Out("MPI_Iprobe: OK flag=false");
00558         }
00559     }
00560     else {
00561         MPIDBG_Out("MPI_Iprobe: Failed");
00562     }
00563     return ret;
00564 }
00565 #define MPI_Iprobe(...) MPIDBG_Iprobe(__VA_ARGS__, __FILE__, __LINE__, __func__)
00566
00567 /*
00568     #] MPI_Iprobe :
00569     #[ MPI_Probe :
00570 */
00571
00572 static inline int MPIDBG_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status, MPIDBG_EXTARG)
00573 {
00574     MPIDBG_Out("MPI_Probe: src=%d dest=%d tag=%d", source, MPIDBG_RANK, tag);
00575     int ret = MPI_Probe(source, tag, comm, status);
00576     if ( ret == MPI_SUCCESS ) {
00577         int ret_size = 0;
00578         MPI_Get_count(status, MPI_BYTE, &ret_size);
00579         MPIDBG_Out("MPI_Probe: OK src=%d dest=%d tag=%d size=%d", status->MPI_SOURCE, MPIDBG_RANK,
    status->MPI_TAG, ret_size);
00580     }
00581     else {
00582         MPIDBG_Out("MPI_Probe: Failed");
00583     }
00584     return ret;
00585 }
00586 #define MPI_Probe(...) MPIDBG_Probe(__VA_ARGS__, __FILE__, __LINE__, __func__)
00587
00588 /*

```

```

00589         #] MPI_Probe :
00590         #[ MPI_Cancel :
00591     */
00592
00593     static inline int MPIDBG_Cancel(MPI_Request *request, MPIDBG_EXTARG)
00594     {
00595         MPIDBG_Out("MPI_Cancel: rank=%d", MPIDBG_RANK);
00596         int ret = MPI_Cancel(request);
00597         if ( ret == MPI_SUCCESS ) {
00598             MPIDBG_Out("MPI_Cancel: OK");
00599         }
00600         else {
00601             MPIDBG_Out("MPI_Cancel: Failed");
00602         }
00603         return ret;
00604     }
00605     #define MPI_Cancel(...) MPIDBG_Cancel(__VA_ARGS__, __FILE__, __LINE__, __func__)
00606
00607     /*
00608         #] MPI_Cancel :
00609         #[ MPI_Test_cancelled :
00610     */
00611
00612     static inline int MPIDBG_Test_cancelled(MPI_Status *status, int *flag, MPIDBG_EXTARG)
00613     {
00614         MPIDBG_Out("MPI_Test_cancelled: rank=%d", MPIDBG_RANK);
00615         int ret = MPI_Test_cancelled(status, flag);
00616         if ( ret == MPI_SUCCESS ) {
00617             if ( *flag ) {
00618                 MPIDBG_Out("MPI_Test_cancelled: OK flag=true");
00619             }
00620             else {
00621                 MPIDBG_Out("MPI_Test_cancelled: OK flag=false");
00622             }
00623         }
00624         else {
00625             MPIDBG_Out("MPI_Test_cancelled: Failed");
00626         }
00627         return ret;
00628     }
00629     #define MPI_Test_cancelled(...) MPIDBG_Test_cancelled(__VA_ARGS__, __FILE__, __LINE__, __func__)
00630
00631     /*
00632         #] MPI_Test_cancelled :
00633         #[ MPI_Barrier :
00634     */
00635
00636     static inline int MPIDBG_Barrier(MPI_Comm comm, MPIDBG_EXTARG)
00637     {
00638         MPIDBG_Out("MPI_Barrier: rank=%d", MPIDBG_RANK);
00639         int ret = MPI_Barrier(comm);
00640         if ( ret == MPI_SUCCESS ) {
00641             MPIDBG_Out("MPI_Barrier: OK");
00642         }
00643         else {
00644             MPIDBG_Out("MPI_Barrier: Failed");
00645         }
00646         return ret;
00647     }
00648     #define MPI_Barrier(...) MPIDBG_Barrier(__VA_ARGS__, __FILE__, __LINE__, __func__)
00649
00650     /*
00651         #] MPI_Barrier :
00652         #[ MPI_Bcast :
00653     */
00654
00655     static inline int MPIDBG_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm
comm, MPIDBG_EXTARG)
00656     {
00657         MPIDBG_Out("MPI_Bcast: root=%d count=%d", root, count);
00658         int ret = MPI_Bcast(buffer, count, datatype, root, comm);
00659         if ( ret == MPI_SUCCESS ) {
00660             MPIDBG_Out("MPI_Bcast: OK");
00661         }
00662         else {
00663             MPIDBG_Out("MPI_Bcast: Failed");
00664         }
00665         return ret;
00666     }
00667     #define MPI_Bcast(...) MPIDBG_Bcast(__VA_ARGS__, __FILE__, __LINE__, __func__)
00668
00669     /*
00670         #] MPI_Bcast :
00671         #] MPI APIs :
00672     */
00673
00674     #endif /* MPIDBG_H */

```

4.83 mytime.cc

```

00001 /* #[ License : */
00002 /*
00003  * Copyright (C) 1984-2026 J.A.M. Vermaseren
00004  * When using this file you are requested to refer to the publication
00005  * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00006  * This is considered a matter of courtesy as the development was paid
00007  * for by FOM the Dutch physics granting agency and we would like to
00008  * be able to track its scientific use to convince FOM of its value
00009  * for the community.
00010  *
00011  * This file is part of FORM.
00012  *
00013  * FORM is free software: you can redistribute it and/or modify it under the
00014  * terms of the GNU General Public License as published by the Free Software
00015  * Foundation, either version 3 of the License, or (at your option) any later
00016  * version.
00017  *
00018  * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00019  * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00020  * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00021  * details.
00022  *
00023  * You should have received a copy of the GNU General Public License along
00024  * with FORM. If not, see <http://www.gnu.org/licenses/>.
00025  */
00026 /* #[ License : */
00027
00028 #ifdef HAVE_CONFIG_H
00029 #include <config.h>
00030 #endif
00031
00032 // A timing routine for debugging. Only on Unix (where sys/time.h is available).
00033 #ifdef UNIX
00034
00035 #include <sys/time.h>
00036 #include <cstdlib>
00037 #include <stdio>
00038 #include <string>
00039
00040 #ifndef timersub
00041 /* timersub is not in POSIX, but presents on most BSD derivatives.
00042    This implementation is borrowed from glibc. (TU 23 Oct 2011) */
00043 #define timersub(a, b, result) \
00044     do { \
00045         (result)->tv_sec = (a)->tv_sec - (b)->tv_sec; \
00046         (result)->tv_usec = (a)->tv_usec - (b)->tv_usec; \
00047         if ((result)->tv_usec < 0) { \
00048             --(result)->tv_sec; \
00049             (result)->tv_usec += 1000000; \
00050         } \
00051     } while (0)
00052 #endif
00053
00054 bool starttime_set = false;
00055 timeval starttime;
00056
00057 double thetime () {
00058     if (!starttime_set) {
00059         gettimeofday(&starttime, NULL);
00060         starttime_set=true;
00061     }
00062     timeval now,diff;
00063     gettimeofday(&now, NULL);
00064     timersub(&now,&starttime,&diff);
00065     return diff.tv_sec+diff.tv_usec/1000000.0;
00066 }
00067
00068 std::string thetime_str() {
00069     char res[10];
00070     snprintf(res,10,"%0.4lf", thetime());
00071     return res;
00072 }
00073
00074 #endif // UNIX

```

4.84 mytime.h

```

00001 #pragma once
00002 /* #[ License : */
00003 /*

```



```

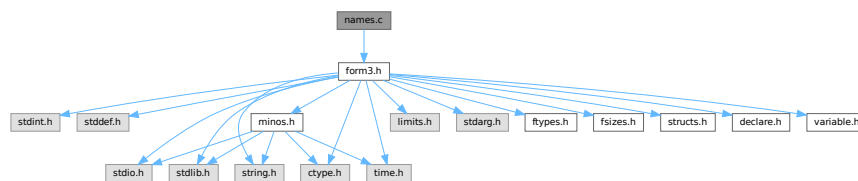
00004 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00005 * When using this file you are requested to refer to the publication
00006 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00007 * This is considered a matter of courtesy as the development was paid
00008 * for by FOM the Dutch physics granting agency and we would like to
00009 * be able to track its scientific use to convince FOM of its value
00010 * for the community.
00011 *
00012 * This file is part of FORM.
00013 *
00014 * FORM is free software: you can redistribute it and/or modify it under the
00015 * terms of the GNU General Public License as published by the Free Software
00016 * Foundation, either version 3 of the License, or (at your option) any later
00017 * version.
00018 *
00019 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00020 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00021 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00022 * details.
00023 *
00024 * You should have received a copy of the GNU General Public License along
00025 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00026 */
00027 /* #] License : */
00028
00029 double thetime();
00030 std::string thetime_str();

```

4.85 names.c File Reference

```
#include "form3.h"
```

Include dependency graph for names.c:



Functions

- **NAMENODE** * **GetNode** (**NAMETREE** *nametree, UBYTE *name)
- int **AddName** (**NAMETREE** *nametree, UBYTE *name, WORD type, WORD number, int *nodenum)
- int **GetName** (**NAMETREE** *nametree, UBYTE *namein, WORD *number, int par)
- UBYTE * **GetFunction** (UBYTE *s, WORD *funnum)
- UBYTE * **GetNumber** (UBYTE *s, WORD *num)
- int **GetLastExprName** (UBYTE *name, WORD *number)
- int **GetOName** (**NAMETREE** *nametree, UBYTE *name, WORD *number, int par)
- int **GetAutoName** (UBYTE *name, WORD *number)
- int **GetVar** (UBYTE *name, WORD *type, WORD *number, int wantedtype, int par)
- int **EntVar** (WORD type, UBYTE *name, WORD x, WORD y, WORD z, WORD d)
- int **GetDollar** (UBYTE *name)
- void **DumpTree** (**NAMETREE** *nametree)
- void **DumpNode** (**NAMETREE** *nametree, WORD node, WORD depth)
- int **CompactifyTree** (**NAMETREE** *nametree, WORD par)
- void **CopyTree** (**NAMETREE** *newtree, **NAMETREE** *oldtree, WORD node, WORD par)
- void **LinkTree** (**NAMETREE** *tree, WORD offset, WORD numnodes)
- **NAMETREE** * **MakeNameTree** (void)

- void [FreeNameTree](#) ([NAMETREE](#) *n)
- void [ClearWildcardNames](#) (void)
- int [AddWildcardName](#) (UBYTE *name)
- int [GetWildcardName](#) (UBYTE *name)
- int [AddSymbol](#) (UBYTE *name, int minpow, int maxpow, int cplx, int dim)
- int [CoSymbol](#) (UBYTE *s)
- int [AddIndex](#) (UBYTE *name, int dim, int dim4)
- int [CoIndex](#) (UBYTE *s)
- UBYTE * [DoDimension](#) (UBYTE *s, int *dim, int *dim4)
- int [CoDimension](#) (UBYTE *s)
- int [AddVector](#) (UBYTE *name, int cplx, int dim)
- int [CoVector](#) (UBYTE *s)
- int [AddFunction](#) (UBYTE *name, int comm, int istensor, int cplx, int symprop, int dim, int argmax, int argmin)
- int [CoCommutelnSet](#) (UBYTE *s)
- int [CoFunction](#) (UBYTE *s, int comm, int istensor)
- int [CoNFunction](#) (UBYTE *s)
- int [CoCFunction](#) (UBYTE *s)
- int [CoNTensor](#) (UBYTE *s)
- int [CoCTensor](#) (UBYTE *s)
- int [DoTable](#) (UBYTE *s, int par)
- int [CoTable](#) (UBYTE *s)
- int [CoNTable](#) (UBYTE *s)
- int [CoCTable](#) (UBYTE *s)
- void [EmptyTable](#) ([TABLES](#) T)
- int [AddSet](#) (UBYTE *name, WORD dim)
- int [DoElements](#) (UBYTE *s, [SETS](#) set, UBYTE *name)
- int [CoSet](#) (UBYTE *s)
- int [DoTempSet](#) (UBYTE *from, UBYTE *to)
- int [CoAuto](#) (UBYTE *inp)
- int [AddDollar](#) (UBYTE *name, WORD type, WORD *start, LONG size)
- int [ReplaceDollar](#) (WORD number, WORD newtype, WORD *newstart, LONG newsize)
- int [AddDubious](#) (UBYTE *name)
- int [MakeDubious](#) ([NAMETREE](#) *nametree, UBYTE *name, WORD *number)
- int [NameConflict](#) (int type, UBYTE *name)
- int [AddExpression](#) (UBYTE *name, int x, int y)
- int [GetLabel](#) (UBYTE *name)
- void [ResetVariables](#) (int par)
- void [RemoveDollars](#) (void)
- void [Globalize](#) (int par)
- int [TestName](#) (UBYTE *name)

4.85.1 Detailed Description

The complete names administration. All variables with a name have to pass here to be properly registered, have structs of the proper type assigned to them etc. The file also contains the utility routines for maintaining the balanced trees that make searching for names rather fast.

Definition in file [names.c](#).

4.85.2 Function Documentation

4.85.2.1 GetNode()

```
NAMENODE * GetNode (
    NAMETREE * nametree,
    UBYTE * name )
```

Definition at line 49 of file [names.c](#).

4.85.2.2 AddName()

```
int AddName (
    NAMETREE * nametree,
    UBYTE * name,
    WORD type,
    WORD number,
    int * nodenum )
```

Definition at line 71 of file [names.c](#).

4.85.2.3 GetName()

```
int GetName (
    NAMETREE * nametree,
    UBYTE * namein,
    WORD * number,
    int par )
```

Definition at line 252 of file [names.c](#).

4.85.2.4 GetFunction()

```
UBYTE * GetFunction (
    UBYTE * s,
    WORD * funnum )
```

Definition at line 342 of file [names.c](#).

4.85.2.5 GetNumber()

```
UBYTE * GetNumber (
    UBYTE * s,
    WORD * num )
```

Definition at line 388 of file [names.c](#).

4.85.2.6 GetLastExprName()

```
int GetLastExprName (
    UBYTE * name,
    WORD * number )
```

Definition at line [438](#) of file [names.c](#).

4.85.2.7 GetOName()

```
int GetOName (
    NAMETREE * nametree,
    UBYTE * name,
    WORD * number,
    int par )
```

Definition at line [460](#) of file [names.c](#).

4.85.2.8 GetAutoName()

```
int GetAutoName (
    UBYTE * name,
    WORD * number )
```

Definition at line [479](#) of file [names.c](#).

4.85.2.9 GetVar()

```
int GetVar (
    UBYTE * name,
    WORD * type,
    WORD * number,
    int wantedtype,
    int par )
```

Definition at line [524](#) of file [names.c](#).

4.85.2.10 EntVar()

```
int EntVar (
    WORD type,
    UBYTE * name,
    WORD x,
    WORD y,
    WORD z,
    WORD d )
```

Definition at line [557](#) of file [names.c](#).

4.85.2.11 GetDollar()

```
int GetDollar (
    UBYTE * name )
```

Definition at line 590 of file [names.c](#).

4.85.2.12 DumpTree()

```
void DumpTree (
    NAMETREE * nametree )
```

Definition at line 602 of file [names.c](#).

4.85.2.13 DumpNode()

```
void DumpNode (
    NAMETREE * nametree,
    WORD node,
    WORD depth )
```

Definition at line 615 of file [names.c](#).

4.85.2.14 CompactifyTree()

```
int CompactifyTree (
    NAMETREE * nametree,
    WORD par )
```

Definition at line 634 of file [names.c](#).

4.85.2.15 CopyTree()

```
void CopyTree (
    NAMETREE * newtree,
    NAMETREE * oldtree,
    WORD node,
    WORD par )
```

Definition at line 696 of file [names.c](#).

4.85.2.16 LinkTree()

```
void LinkTree (
    NAMETREE * tree,
    WORD offset,
    WORD numnodes )
```

Definition at line 784 of file [names.c](#).

4.85.2.17 MakeNameTree()

```
NAMETREE * MakeNameTree (
    void )
```

Definition at line [815](#) of file [names.c](#).

4.85.2.18 FreeNameTree()

```
void FreeNameTree (
    NAMETREE * n )
```

Definition at line [834](#) of file [names.c](#).

4.85.2.19 ClearWildcardNames()

```
void ClearWildcardNames (
    void )
```

Definition at line [849](#) of file [names.c](#).

4.85.2.20 AddWildcardName()

```
int AddWildcardName (
    UBYTE * name )
```

Definition at line [854](#) of file [names.c](#).

4.85.2.21 GetWildcardName()

```
int GetWildcardName (
    UBYTE * name )
```

Definition at line [898](#) of file [names.c](#).

4.85.2.22 AddSymbol()

```
int AddSymbol (
    UBYTE * name,
    int minpow,
    int maxpow,
    int cplx,
    int dim )
```

Definition at line [920](#) of file [names.c](#).

4.85.2.23 CoSymbol()

```
int CoSymbol (
    UBYTE * s )
```

Definition at line 946 of file [names.c](#).

4.85.2.24 AddIndex()

```
int AddIndex (
    UBYTE * name,
    int dim,
    int dim4 )
```

Definition at line 1088 of file [names.c](#).

4.85.2.25 CoIndex()

```
int CoIndex (
    UBYTE * s )
```

Definition at line 1112 of file [names.c](#).

4.85.2.26 DoDimension()

```
UBYTE * DoDimension (
    UBYTE * s,
    int * dim,
    int * dim4 )
```

Definition at line 1160 of file [names.c](#).

4.85.2.27 CoDimension()

```
int CoDimension (
    UBYTE * s )
```

Definition at line 1226 of file [names.c](#).

4.85.2.28 AddVector()

```
int AddVector (
    UBYTE * name,
    int cplx,
    int dim )
```

Definition at line 1244 of file [names.c](#).

4.85.2.29 CoVector()

```
int CoVector (
    UBYTE * s )
```

Definition at line 1267 of file [names.c](#).

4.85.2.30 AddFunction()

```
int AddFunction (
    UBYTE * name,
    int comm,
    int istensor,
    int cplx,
    int symprop,
    int dim,
    int argmax,
    int argmin )
```

Definition at line 1326 of file [names.c](#).

4.85.2.31 CoCommuteInSet()

```
int CoCommuteInSet (
    UBYTE * s )
```

Definition at line 1356 of file [names.c](#).

4.85.2.32 CoFunction()

```
int CoFunction (
    UBYTE * s,
    int comm,
    int istensor )
```

Definition at line 1446 of file [names.c](#).

4.85.2.33 CoNFunction()

```
int CoNFunction (
    UBYTE * s )
```

Definition at line 1626 of file [names.c](#).

4.85.2.34 CoCFunction()

```
int CoCFunction (
    UBYTE * s )
```

Definition at line 1627 of file [names.c](#).

4.85.2.35 CoNTensor()

```
int CoNTensor (
    UBYTE * s )
```

Definition at line 1628 of file [names.c](#).

4.85.2.36 CoCTensor()

```
int CoCTensor (
    UBYTE * s )
```

Definition at line 1629 of file [names.c](#).

4.85.2.37 DoTable()

```
int DoTable (
    UBYTE * s,
    int par )
```

Definition at line 1668 of file [names.c](#).

4.85.2.38 CoTable()

```
int CoTable (
    UBYTE * s )
```

Definition at line 2050 of file [names.c](#).

4.85.2.39 CoNTable()

```
int CoNTable (
    UBYTE * s )
```

Definition at line 2060 of file [names.c](#).

4.85.2.40 CoCTable()

```
int CoCTable (
    UBYTE * s )
```

Definition at line 2070 of file [names.c](#).

4.85.2.41 EmptyTable()

```
void EmptyTable (
    TABLES T )
```

Definition at line 2080 of file [names.c](#).

4.85.2.42 AddSet()

```
int AddSet (
    UBYTE * name,
    WORD dim )
```

Definition at line [2124](#) of file [names.c](#).

4.85.2.43 DoElements()

```
int DoElements (
    UBYTE * s,
    SETS set,
    UBYTE * name )
```

Definition at line [2157](#) of file [names.c](#).

4.85.2.44 CoSet()

```
int CoSet (
    UBYTE * s )
```

Definition at line [2357](#) of file [names.c](#).

4.85.2.45 DoTempSet()

```
int DoTempSet (
    UBYTE * from,
    UBYTE * to )
```

Definition at line [2482](#) of file [names.c](#).

4.85.2.46 CoAuto()

```
int CoAuto (
    UBYTE * inp )
```

Definition at line [2574](#) of file [names.c](#).

4.85.2.47 AddDollar()

```
int AddDollar (
    UBYTE * name,
    WORD type,
    WORD * start,
    LONG size )
```

Definition at line [2604](#) of file [names.c](#).

4.85.2.48 ReplaceDollar()

```
int ReplaceDollar (
    WORD number,
    WORD newtype,
    WORD * newstart,
    LONG newsize )
```

Definition at line [2648](#) of file [names.c](#).

4.85.2.49 AddDubious()

```
int AddDubious (
    UBYTE * name )
```

Definition at line [2681](#) of file [names.c](#).

4.85.2.50 MakeDubious()

```
int MakeDubious (
    NAMETREE * nametree,
    UBYTE * name,
    WORD * number )
```

Definition at line [2695](#) of file [names.c](#).

4.85.2.51 NameConflict()

```
int NameConflict (
    int type,
    UBYTE * name )
```

Definition at line [2730](#) of file [names.c](#).

4.85.2.52 AddExpression()

```
int AddExpression (
    UBYTE * name,
    int x,
    int y )
```

Definition at line [2746](#) of file [names.c](#).

4.85.2.53 GetLabel()

```
int GetLabel (
    UBYTE * name )
```

Definition at line [2789](#) of file [names.c](#).

4.85.2.54 ResetVariables()

```
void ResetVariables (
    int par )
```

Definition at line 2834 of file [names.c](#).

4.85.2.55 RemoveDollars()

```
void RemoveDollars (
    void )
```

Definition at line 3130 of file [names.c](#).

4.85.2.56 Globalize()

```
void Globalize (
    int par )
```

Definition at line 3157 of file [names.c](#).

4.85.2.57 TestName()

```
int TestName (
    UBYTE * name )
```

Definition at line 3256 of file [names.c](#).

4.86 names.c

[Go to the documentation of this file.](#)

```
00001
00009 /* # [ License : */
00010 /*
00011 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00012 * When using this file you are requested to refer to the publication
00013 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014 * This is considered a matter of courtesy as the development was paid
00015 * for by FOM the Dutch physics granting agency and we would like to
00016 * be able to track its scientific use to convince FOM of its value
00017 * for the community.
00018 *
00019 * This file is part of FORM.
00020 *
00021 * FORM is free software: you can redistribute it and/or modify it under the
00022 * terms of the GNU General Public License as published by the Free Software
00023 * Foundation, either version 3 of the License, or (at your option) any later
00024 * version.
00025 *
00026 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00029 * details.
00030 *
00031 * You should have received a copy of the GNU General Public License along
00032 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00033 */
00034 /* # ] License : */
00035 /*
00036 # [ Includes :
```

```

00037 */
00038
00039 #include "form3.h"
00040
00041 /* EXTERNLOCK(dummylock) */
00042
00043 /*
00044     #] Includes :
00045     #[ GetNode :
00046 */
00047
00048
00049 NAMENODE *GetNode(NAMETREE *nametree, UBYTE *name)
00050 {
00051     NAMENODE *n;
00052     int node, newnode, i;
00053     if ( nametree->namenode == 0 ) return(0);
00054     newnode = nametree->headnode;
00055     do {
00056         node = newnode;
00057         n = nametree->namenode+node;
00058         if ( ( i = StrCmp(name,nametree->namebuffer+n->name) ) < 0 )
00059             newnode = n->left;
00060         else if ( i > 0 ) newnode = n->right;
00061         else { return(n); }
00062     } while ( newnode >= 0 );
00063     return(0);
00064 }
00065
00066 /*
00067     #] GetNode :
00068     #[ AddName :
00069 */
00070
00071 int AddName(NAMETREE *nametree, UBYTE *name, WORD type, WORD number, int *nodenum)
00072 {
00073     NAMENODE *n, *nn, *nnn;
00074     UBYTE *s, *ss, *sss;
00075     LONG *cl,*c2, j, newsize;
00076     int node, newnode, node3, r, rr = 0, i, retval = 0;
00077     if ( nametree->namenode == 0 ) {
00078         s = name; i = 1; while ( *s ) { i++; s++; }
00079         j = INITNAMESIZE;
00080         if ( i > j ) j = i;
00081         nametree->namenode = (NAMENODE *)Malloc1(INITNODESIZE*sizeof(NAMENODE),
00082             "new nametree in AddName");
00083         nametree->namebuffer = (UBYTE *)Malloc1(j,
00084             "new namebuffer in AddName");
00085         nametree->nodesize = INITNODESIZE;
00086         nametree->namesize = j;
00087         nametree->namefill = i;
00088         nametree->nodefill = 1;
00089         nametree->headnode = 0;
00090         n = nametree->namenode;
00091         n->parent = n->left = n->right = -1;
00092         n->balance = 0;
00093         n->type = type;
00094         n->number = number;
00095         n->name = 0;
00096         s = name;
00097         ss = nametree->namebuffer;
00098         while ( *s ) *ss++ = *s++;
00099         *ss = 0;
00100         *nodenum = 0;
00101         return(retval);
00102     }
00103     newnode = nametree->headnode;
00104     do {
00105         node = newnode;
00106         n = nametree->namenode+node;
00107         if ( StrCmp(name,nametree->namebuffer+n->name) < 0 ) {
00108             newnode = n->left; r = -1;
00109         }
00110         else {
00111             newnode = n->right; r = 1;
00112         }
00113     } while ( newnode >= 0 );
00114     /*
00115     We are at the insertion point. Add the node.
00116     */
00117     if ( nametree->nodefill >= nametree->nodesize ) { /* Double allocation */
00118         newsize = nametree->nodesize * 2;
00119         if ( newsize > MAXINNAMETREE ) newsize = MAXINNAMETREE;
00120         if ( nametree->nodefill >= MAXINNAMETREE ) {
00121             MesPrint("!!!More than %1 names in one object", (LONG)MAXINNAMETREE);
00122             Terminate(-1);
00123         }
00124     }

```

```

00124     nnn = (NAMENODE *)Malloc1(2*((LONG)newsiz* sizeof(NAMENODE)),
00125     "extra names in AddName");
00126     c1 = (LONG *)nnn; c2 = (LONG *)nametree->namenode;
00127     i = (nametree->nodefill * sizeof(NAMENODE))/sizeof(LONG);
00128     while ( --i >= 0 ) *c1++ = *c2++;
00129     M_free(nametree->namenode, "nametree->namenode");
00130     nametree->namenode = nnn;
00131     nametree->nodesize = newsiz;
00132     n = nametree->namenode+node;
00133 }
00134 *nodenum = newnode = nametree->nodefill++;
00135 nn = nametree->namenode+newnode;
00136 nn->parent = node;
00137 if ( r < 0 ) n->left = newnode; else n->right = newnode;
00138 nn->left = nn->right = -1;
00139 nn->type = type;
00140 nn->number = number;
00141 nn->balance = 0;
00142 i = 1; s = name; while ( *s ) { i++; s++; }
00143 while ( nametree->namefill + i >= nametree->namesize ) { /* Double alloc */
00144     sss = (UBYTE *)Malloc1(2*nametree->namesize,
00145     "extra names in AddName");
00146     s = sss; ss = nametree->namebuffer; j = nametree->namefill;
00147     while ( --j >= 0 ) *s++ = *ss++;
00148     M_free(nametree->namebuffer, "nametree->namebuffer");
00149     nametree->namebuffer = sss;
00150     nametree->namesize *= 2;
00151 }
00152 s = nametree->namebuffer+nametree->namefill;
00153 nn->name = nametree->namefill;
00154 retval = nametree->namefill;
00155 nametree->namefill += i;
00156 while ( *name ) *s++ = *name++;
00157 *s = 0;
00158 /*
00159 Adjust the balance factors
00160 */
00161 while ( node >= 0 ) {
00162     n = nametree->namenode + node;
00163     if ( newnode == n->left ) rr = -1;
00164     else rr = 1;
00165     if ( n->balance == -rr ) { n->balance = 0; return(retval); }
00166     else if ( n->balance == rr ) break;
00167     n->balance = rr;
00168     newnode = node;
00169     node = n->parent;
00170 }
00171 if ( node < 0 ) return(retval);
00172 /*
00173 We have to rebalance the tree. There are two basic operations.
00174 n/node is the unbalanced node. newnode is its child.
00175 rr is the old balance of n/node.
00176 */
00177 nn = nametree->namenode + newnode;
00178 if ( nn->balance == -rr ) { /* The difficult case */
00179     if ( rr > 0 ) {
00180         node3 = nn->left;
00181         nnn = nametree->namenode + node3;
00182         nnn->parent = n->parent;
00183         n->parent = nn->parent = node3;
00184         if ( nnn->right >= 0 ) nametree->namenode[nnn->right].parent = newnode;
00185         if ( nnn->left >= 0 ) nametree->namenode[nnn->left].parent = node;
00186         n->right = nnn->left; nnn->left = node;
00187         nn->left = nnn->right; nnn->right = newnode;
00188         if ( nnn->balance > 0 ) { n->balance = -1; nn->balance = 0; }
00189         else if ( nnn->balance == 0 ) { n->balance = nn->balance = 0; }
00190         else { nn->balance = 1; n->balance = 0; }
00191     }
00192     else {
00193         node3 = nn->right;
00194         nnn = nametree->namenode + node3;
00195         nnn->parent = n->parent;
00196         n->parent = nn->parent = node3;
00197         if ( nnn->right >= 0 ) nametree->namenode[nnn->right].parent = node;
00198         if ( nnn->left >= 0 ) nametree->namenode[nnn->left].parent = newnode;
00199         n->left = nnn->right; nnn->right = node;
00200         nn->right = nnn->left; nnn->left = newnode;
00201         if ( nnn->balance < 0 ) { n->balance = 1; nn->balance = 0; }
00202         else if ( nnn->balance == 0 ) { n->balance = nn->balance = 0; }
00203         else { nn->balance = -1; n->balance = 0; }
00204     }
00205     nnn->balance = 0;
00206     if ( nnn->parent >= 0 ) {
00207         nn = nametree->namenode + nnn->parent;
00208         if ( node == nn->left ) nn->left = node3;
00209         else nn->right = node3;
00210     }

```

```

00211     if ( node == nametree->headnode ) nametree->headnode = node3;
00212 }
00213 else if ( nn->balance == rr ) { /* The easy case */
00214     nn->parent = n->parent; n->parent = newnode;
00215     if ( rr > 0 ) {
00216         if ( nn->left >= 0 ) nametree->namenode[nn->left].parent = node;
00217         n->right = nn->left; nn->left = node;
00218     }
00219     else {
00220         if ( nn->right >= 0 ) nametree->namenode[nn->right].parent = node;
00221         n->left = nn->right; nn->right = node;
00222     }
00223     if ( nn->parent >= 0 ) {
00224         nnn = nametree->namenode + nn->parent;
00225         if ( node == nnn->left ) nnn->left = newnode;
00226         else nnn->right = newnode;
00227     }
00228     nn->balance = n->balance = 0;
00229     if ( node == nametree->headnode ) nametree->headnode = newnode;
00230 }
00231 #ifdef DEBUGON
00232 else { /* Cannot be. Code here for debugging only */
00233     MesPrint("We ran into an impossible case in AddName\n");
00234     DumpTree(nametree);
00235     Terminate(-1);
00236 }
00237 #endif
00238 return(retval);
00239 }
00240
00241 /*
00242 #] AddName :
00243 #[ GetName :
00244
00245 When AutoDeclare is an active statement.
00246 If par == WITHAUTO and the variable is not found we have to check:
00247 1: that nametree != AC.exprnames && nametree != AC.dollarnames
00248 2: check that the variable is not in AC.exprnames after all.
00249 3: call GetAutoName and return its values.
00250 */
00251
00252 int GetName(NAMETREE *nametree, UBYTE *namein, WORD *number, int par)
00253 {
00254     NAMENODE *n;
00255     int node, newnode, i;
00256     UBYTE *s, *t, *u, *name;
00257     /* name = ConstructName(namein,0); */
00258     name = namein;
00259     if ( nametree->namenode == 0 || nametree->namefill == 0 ) goto NotFound;
00260     newnode = nametree->headnode;
00261     do {
00262         node = newnode;
00263         n = nametree->namenode+node;
00264         if ( ( i = StrCmp(name,nametree->namebuffer+n->name) ) < 0 )
00265             newnode = n->left;
00266         else if ( i > 0 ) newnode = n->right;
00267         else {
00268             *number = n->number;
00269             return(n->type);
00270         }
00271     } while ( newnode >= 0 );
00272     s = name;
00273     while ( *s ) s++;
00274     if ( s > name && s[-1] == '_' && nametree == AC.varnames ) {
00275     /*
00276         The Kronecker delta d_ is very special. It is not really a function.
00277     */
00278     if ( s == name+2 && ( *name == 'd' || *name == 'D' ) ) {
00279         *number = DELTA-FUNCTION;
00280         return(CDELTA);
00281     }
00282     /*
00283         Test for N#_? type variables (summed indices)
00284     */
00285     if ( s > name+2 && *name == 'N' ) {
00286         t = name+1; i = 0;
00287         while ( FG.cTable[*t] == 1 ) i = 10*i + *t++ - '0';
00288         if ( s == t+1 ) {
00289             *number = i + AM.IndDum - AM.OffsetIndex;
00290             return(CINDEX);
00291         }
00292     }
00293     /*
00294         Now test for any built in object
00295     */
00296     newnode = nametree->headnode;
00297     do {

```

```

00298         node = newnode;
00299         n = nametree->namenode+node;
00300         if ( ( i = StrHICmp(name,nametree->namebuffer+n->name) ) < 0 )
00301             newnode = n->left;
00302         else if ( i > 0 ) newnode = n->right;
00303         else {
00304             *number = n->number; return(n->type);
00305         }
00306     } while ( newnode >= 0 );
00307 /*
00308     Now we test for the extra symbols of the type STR###_
00309     The string sits in AC.extrasym and is followed by digits.
00310     The name is only legal if the number is in the
00311     range 1,...,cbuf[AM.sbufnum].numrhs
00312 */
00313     t = name; u = AC.extrasym;
00314     while ( *t == *u ) { t++; u++; }
00315     if ( *u == 0 && *t != 0 ) { /* potential hit */
00316         WORD x = 0;
00317         while ( FG.cTable[*t] == 1 ) {
00318             x = 10*x + (*t++ - '0');
00319         }
00320         if ( *t == '_' && x > 0 && x <= cbuf[AM.sbufnum].numrhs ) { /* Hit */
00321             *number = MAXVARIABLES-x;
00322             return(CSYMBOL);
00323         }
00324     }
00325 }
00326 NotFound:;
00327 if ( par != WITHAUTO || nametree == AC.autonames ) return(NAMENOTFOUND);
00328 return(GetAutoName(name,number));
00329 }
00330
00331 /*
00332 #] GetName :
00333 #[ GetFunction :
00334
00335 Gets either a function or a $ that should expand into a function
00336 during runtime. In the case of the $ the value in funnum is -dolnum-1.
00337 The return value is the position after the name of the function or the $.
00338 */
00339
00340 static WORD one = 1;
00341
00342 UBYTE *GetFunction(UBYTE *s,WORD *funnum)
00343 {
00344     int type;
00345     WORD numfun;
00346     UBYTE *t1, c;
00347     if ( *s == '$' ) {
00348         t1 = s+1; while ( FG.cTable[*t1] < 2 ) t1++;
00349         c = *t1; *t1 = 0;
00350         if ( ( type = GetName(AC.dollarnames,s+1,&numfun,NOAUTO) ) == CDOLLAR ) {
00351             *funnum = -numfun-2;
00352         }
00353         else {
00354             MesPrint("%s is undefined",s);
00355             numfun = AddDollar(s+1,DOLINDEX,&one,1);
00356             *funnum = 0;
00357         }
00358     }
00359     else {
00360         t1 = SkipAName(s);
00361         c = *t1; *t1 = 0;
00362         if ( ( ( type = GetName(AC.varnames,s,&numfun,WITHAUTO) ) != CFUNCTION )
00363             || ( functions[numfun].spec > 0 ) ) {
00364             MesPrint("%s should be a regular function",s);
00365             *funnum = 0;
00366             if ( type < 0 ) {
00367                 if ( GetName(AC.exprnames,s,&numfun,NOAUTO) == NAMENOTFOUND )
00368                     AddFunction(s,0,0,0,0,0,-1,-1);
00369             }
00370             *t1 = c;
00371             return(t1);
00372         }
00373         *funnum = numfun+FUNCTION;
00374     }
00375     *t1 = c;
00376     return(t1);
00377 }
00378
00379 /*
00380 #] GetFunction :
00381 #[ GetNumber :
00382
00383 Gets either a number or a $ that should expand into a number
00384 during runtime. In the case of the $ the value in num is -dolnum-2.

```



```

00385     The return value is the position after the number or the $.
00386 */
00387
00388 UBYTE *GetNumber(UBYTE *s,WORD *num)
00389 {
00390     int type;
00391     WORD numfun;
00392     UBYTE *t1, c;
00393     while ( *s == '+' ) s++;
00394     if ( *s == '$' ) {
00395         t1 = s+1; while ( FG.cTable[*t1] < 2 ) t1++;
00396         c = *t1; *t1 = 0;
00397         if ( ( type = GetName(AC.dollarnames,s+1,&numfun,NOAUTO) ) == CDOLLAR ) {
00398             *num = -numfun-2;
00399         }
00400         else {
00401             MesPrint("%s is undefined",s);
00402             numfun = AddDollar(s+1,DOLINDEX,&one,1);
00403             *num = -1;
00404         }
00405     }
00406     else if ( *s >= '0' && *s <= '9' ) {
00407         ULONG x = *s++ - '0';
00408         while ( *s >= '0' && *s <= '9' ) { x = 10*x + (*s++-'0'); }
00409         t1 = s;
00410         if ( x >= MAXPOSITIVE ) goto illegal;
00411         *num = (WORD)x;
00412         return(t1);
00413     }
00414     else {
00415         if ( *s == '-' ) { s++; }
00416         if ( *s >= '0' && *s <= '9' ) { while ( *s >= '0' && *s <= '9' ) s++; t1 = s; }
00417         else { t1 = SkipAName(s); }
00418     illegal:
00419         *num = -1;
00420         MesPrint("&Illegal option in Canonicalize statement. Should be a nonnegative number or $
variable.");
00421         return(t1);
00422     }
00423     *t1 = c;
00424     return(t1);
00425 }
00426
00427 /*
00428 #] GetNumber :
00429 #[ GetLastExprName :
00430
00431 When AutoDeclare is an active statement.
00432 If par == WITHAUTO and the variable is not found we have to check:
00433 1: that nametree != AC.exprnames && nametree != AC.dollarnames
00434 2: check that the variable is not in AC.exprnames after all.
00435 3: call GetAutoName and return its values.
00436 */
00437
00438 int GetLastExprName(UBYTE *name, WORD *number)
00439 {
00440     int i;
00441     EXPRESSIONS e;
00442     for ( i = NumExpressions; i > 0; i-- ) {
00443         e = Expressions+i-1;
00444         if ( StrCmp(AC.exprnames->namebuffer+e->name,name) == 0 ) {
00445             *number = i-1;
00446             return(1);
00447         }
00448     }
00449     return(0);
00450 }
00451
00452 /*
00453 #] GetLastExprName :
00454 #[ GetOName :
00455
00456 Adds the proper offsets, so we do not have to do that in the calling
00457 routine.
00458 */
00459
00460 int GetOName(NAMETREE *nametree, UBYTE *name, WORD *number, int par)
00461 {
00462     int retval = GetName(nametree,name,number,par);
00463     switch ( retval ) {
00464         case CVECTOR: *number += AM.OffsetVector; break;
00465         case CINDEX: *number += AM.OffsetIndex; break;
00466         case CFUNCTION: *number += FUNCTION; break;
00467         default: break;
00468     }
00469     return(retval);
00470 }

```

```

00471
00472 /*
00473 #] GetOName :
00474 #[ GetAutoName :
00475
00476 This routine gets the automatic declarations
00477 */
00478
00479 int GetAutoName(UBYTE *name, WORD *number)
00480 {
00481     UBYTE *s, c;
00482     int type;
00483     if ( GetName(AC.exprnames,name,number,NOAUTO) != NAMENOTFOUND )
00484         return(NAMENOTFOUND);
00485     s = name;
00486     while ( *s ) { s++; }
00487     if ( s[-1] == '_' ) {
00488         return(NAMENOTFOUND);
00489     }
00490     while ( s > name ) {
00491         c = *s; *s = 0;
00492         type = GetName(AC.autonames,name,number,NOAUTO);
00493         *s = c;
00494         switch(type) {
00495             case CSYMBOL: {
00496                 SYMBOLS sym = ((SYMBOLS) (AC.AutoSymbolList.lijst)) + *number;
00497                 *number = AddSymbol(name,sym->minpower,sym->maxpower,sym->complex,sym->dimension);
00498                 return(type); }
00499             case CVECTOR: {
00500                 VECTORS vec = ((VECTORS) (AC.AutoVectorList.lijst)) + *number;
00501                 *number = AddVector(name,vec->complex,vec->dimension);
00502                 return(type); }
00503             case CINDEX: {
00504                 INDICES ind = ((INDICES) (AC.AutoIndexList.lijst)) + *number;
00505                 *number = AddIndex(name,ind->dimension,ind->nmin4);
00506                 return(type); }
00507             case CFUNCTION: {
00508                 FUNCTIONS fun = ((FUNCTIONS) (AC.AutoFunctionList.lijst)) + *number;
00509                 *number =
AddFunction(name,fun->commute,fun->spec,fun->complex,fun->symmetric,fun->dimension,fun->maxnumargs,fun->minnumargs);
00510                 return(type); }
00511             default:
00512                 break;
00513         }
00514         s--;
00515     }
00516     return(NAMENOTFOUND);
00517 }
00518
00519 /*
00520 #] GetAutoName :
00521 #[ GetVar :
00522 */
00523
00524 int GetVar(UBYTE *name, WORD *type, WORD *number, int wantedtype, int par)
00525 {
00526     WORD funnum;
00527     int typ;
00528     if ( ( typ = GetName(AC.varnames,name,number,par) ) != wantedtype ) {
00529         if ( typ != NAMENOTFOUND ) {
00530             if ( wantedtype == -1 ) {
00531                 *type = typ;
00532                 return(1);
00533             }
00534             NameConflict(typ,name);
00535             MakeDubious(AC.varnames,name,&funnum);
00536             return(-1);
00537         }
00538         if ( ( typ = GetName(AC.exprnames,name,&funnum,par) ) != NAMENOTFOUND ) {
00539             if ( typ == wantedtype || wantedtype == -1 ) {
00540                 *number = funnum; *type = typ; return(1);
00541             }
00542             NameConflict(typ,name);
00543             return(-1);
00544         }
00545         return(NAMENOTFOUND);
00546     }
00547     if ( typ == -1 ) { return(0); }
00548     *type = typ;
00549     return(1);
00550 }
00551
00552 /*
00553 #] GetVar :
00554 #[ EntVar :
00555 */
00556

```

```

00557 int EntVar(WORD type, UBYTE *name, WORD x, WORD y, WORD z, WORD d)
00558 {
00559     switch ( type ) {
00560         case CSYMBOL:
00561             return(AddSymbol(name,y,z,x,d));
00562             break;
00563         case CINDEK:
00564             return(AddIndex(name,x,z));
00565             break;
00566         case CVECTOR:
00567             return(AddVector(name,x,d));
00568             break;
00569         case CFUNCTION:
00570             return(AddFunction(name,y,z,x,0,d,-1,-1));
00571             break;
00572         case CSET:
00573             AC.SetList.numtemp++;
00574             return(AddSet(name,d));
00575             break;
00576         case CEXPRESSION:
00577             return(AddExpression(name,x,y));
00578             break;
00579         default:
00580             break;
00581     }
00582     return(-1);
00583 }
00584
00585 /*
00586 #] EntVar :
00587 #[ GetDollar :
00588 */
00589
00590 int GetDollar(UBYTE *name)
00591 {
00592     WORD number;
00593     if ( GetName(AC.dollarnames,name,&number,NOAUTO) == NAMENOTFOUND ) return(-1);
00594     return((int)number);
00595 }
00596
00597 /*
00598 #] GetDollar :
00599 #[ DumpTree :
00600 */
00601
00602 void DumpTree(NAMETREE *nametree)
00603 {
00604     if ( nametree->headnode >= 0
00605         && nametree->namebuffer && nametree->namenode ) {
00606         DumpNode(nametree,nametree->headnode,0);
00607     }
00608 }
00609
00610 /*
00611 #] DumpTree :
00612 #[ DumpNode :
00613 */
00614
00615 void DumpNode(NAMETREE *nametree, WORD node, WORD depth)
00616 {
00617     NAMENODE *n;
00618     int i;
00619     char *name;
00620     n = nametree->namenode + node;
00621     if ( n->left >= 0 ) DumpNode(nametree,n->left,depth+1);
00622     for ( i = 0; i < depth; i++ ) printf(" ");
00623     name = (char *) (nametree->namebuffer+n->name);
00624     printf("%s(%d): (%d) (%d) [%d]\n",
00625         name,node,n->parent,n->left,n->right,n->balance);
00626     if ( n->right >= 0 ) DumpNode(nametree,n->right,depth+1);
00627 }
00628
00629 /*
00630 #] DumpNode :
00631 #[ CompactifyTree :
00632 */
00633
00634 int CompactifyTree(NAMETREE *nametree,WORD par)
00635 {
00636     NAMETREE newtree;
00637     NAMENODE *n;
00638     LONG i, j, ns, k;
00639     UBYTE *s;
00640
00641     for ( i = 0, j = 0, k = 0, n = nametree->namenode, ns = 0;
00642         i < nametree->nodedefill; i++, n++ ) {
00643         if ( n->type != CDELETE ) {

```

```

00644         s = nametree->namebuffer+n->name;
00645         while ( *s ) { s++; ns++; }
00646         j++;
00647     }
00648     else k++;
00649 }
00650 if ( k == 0 ) return(0);
00651 if ( j == 0 ) {
00652     if ( nametree->namebuffer ) M_free(nametree->namebuffer,"nametree->namebuffer");
00653     if ( nametree->namenode ) M_free(nametree->namenode,"nametree->namenode");
00654     nametree->namebuffer = 0;
00655     nametree->namenode = 0;
00656     nametree->namesize = nametree->namefill =
00657     nametree->nodesize = nametree->nodefill =
00658     nametree->oldnamefill = nametree->oldnodefill = 0;
00659     nametree->globalnamefill = nametree->globalnodefill =
00660     nametree->clearnamefill = nametree->clearnodefill = 0;
00661     nametree->headnode = -1;
00662     return(0);
00663 }
00664 ns += j;
00665 if ( j < 10 ) j = 10;
00666 if ( ns < 100 ) ns = 100;
00667 newtree->namenode = (NAMENODE *)Malloc1(2*j*sizeof(NAMENODE),"compactify nametree");
00668 newtree->nodefill = 0; newtree->nodesize = 2*j;
00669 newtree->namebuffer = (UBYTE *)Malloc1(2*ns,"compactify nametree");
00670 newtree->namefill = 0; newtree->namesize = 2*ns;
00671 CopyTree(&newtree,nametree,nametree->headnode,par);
00672 newtree->namenode[newtree->nodefill+1].parent = -1;
00673 LinkTree(&newtree,(WORD)0,newtree->nodefill);
00674 newtree->headnode = newtree->nodefill » 1;
00675 M_free(nametree->namebuffer,"nametree->namebuffer");
00676 M_free(nametree->namenode,"nametree->namenode");
00677 nametree->namebuffer = newtree->namebuffer;
00678 nametree->namenode = newtree->namenode;
00679 nametree->namesize = newtree->namesize;
00680 nametree->namefill = newtree->namefill;
00681 nametree->nodesize = newtree->nodesize;
00682 nametree->nodefill = newtree->nodefill;
00683 nametree->oldnamefill = newtree->namefill;
00684 nametree->oldnodefill = newtree->nodefill;
00685 nametree->headnode = newtree->headnode;
00686
00687 /* DumpTree(nametree); */
00688 return(0);
00689 }
00690
00691 /*
00692 #] CompactifyTree :
00693 #[ CopyTree :
00694 */
00695
00696 void CopyTree(NAMETREE *newtree, NAMETREE *oldtree, WORD node, WORD par)
00697 {
00698     NAMENODE *n, *m;
00699     UBYTE *s, *t;
00700     n = oldtree->namenode+node;
00701     if ( n->left >= 0 ) CopyTree(newtree,oldtree,n->left,par);
00702     if ( n->type != CDELETE ) {
00703         m = newtree->namenode+newtree->nodefill;
00704         m->type = n->type;
00705         m->number = n->number;
00706         m->name = newtree->namefill;
00707         m->left = m->right = -1;
00708         m->balance = 0;
00709         switch ( n->type ) {
00710             case CSYMBOL:
00711                 if ( par == AUTONAMES ) {
00712                     autosymbols[n->number].name = newtree->namefill;
00713                     autosymbols[n->number].node = newtree->nodefill;
00714                 }
00715                 else {
00716                     symbols[n->number].name = newtree->namefill;
00717                     symbols[n->number].node = newtree->nodefill;
00718                 }
00719                 break;
00720             case CINDEX :
00721                 if ( par == AUTONAMES ) {
00722                     autoindices[n->number].name = newtree->namefill;
00723                     autoindices[n->number].node = newtree->nodefill;
00724                 }
00725                 else {
00726                     indices[n->number].name = newtree->namefill;
00727                     indices[n->number].node = newtree->nodefill;
00728                 }
00729                 break;
00730             case CVECTOR:

```

```

00731         if ( par == AUTONAMES ) {
00732             autovectors[n->number].name = newtree->namefill;
00733             autovectors[n->number].node = newtree->nodefill;
00734         }
00735         else {
00736             vectors[n->number].name = newtree->namefill;
00737             vectors[n->number].node = newtree->nodefill;
00738         }
00739         break;
00740     case CFUNCTION:
00741         if ( par == AUTONAMES ) {
00742             autofunctions[n->number].name = newtree->namefill;
00743             autofunctions[n->number].node = newtree->nodefill;
00744         }
00745         else {
00746             functions[n->number].name = newtree->namefill;
00747             functions[n->number].node = newtree->nodefill;
00748         }
00749         break;
00750     case CSET:
00751         Sets[n->number].name = newtree->namefill;
00752         Sets[n->number].node = newtree->nodefill;
00753         break;
00754     case CEXPRESSION:
00755         Expressions[n->number].name = newtree->namefill;
00756         Expressions[n->number].node = newtree->nodefill;
00757         break;
00758     case CDUBIOUS:
00759         Dubious[n->number].name = newtree->namefill;
00760         Dubious[n->number].node = newtree->nodefill;
00761         break;
00762     case CDOLLAR:
00763         Dollars[n->number].name = newtree->namefill;
00764         Dollars[n->number].node = newtree->nodefill;
00765         break;
00766     default:
00767         MesPrint("Illegal variable type in CopyTree: %d",n->type);
00768         break;
00769     }
00770     newtree->nodefill++;
00771     s = newtree->namebuffer + newtree->namefill;
00772     t = oldtree->namebuffer + n->name;
00773     while ( *t ) { *s++ = *t++; newtree->namefill++; }
00774     *s = 0; newtree->namefill++;
00775 }
00776 if ( n->right >= 0 ) CopyTree(newtree,oldtree,n->right,par);
00777 }
00778
00779 /*
00780 #] CopyTree :
00781 #[ LinkTree :
00782 */
00783
00784 void LinkTree(NAMETREE *tree, WORD offset, WORD numnodes)
00785 {
00786     /*
00787     Makes the tree into a binary tree
00788     */
00789     int med,numleft,numright,medleft,medright;
00790     med = numnodes » 1;
00791     numleft = med;
00792     numright = numnodes - med - 1;
00793     medleft = numleft » 1;
00794     medright = ( numright » 1 ) + med + 1;
00795     if ( numleft > 0 ) {
00796         tree->namenode[offset+med].left = offset+medleft;
00797         tree->namenode[offset+medleft].parent = offset+med;
00798     }
00799     if ( numright > 0 ) {
00800         tree->namenode[offset+med].right = offset+medright;
00801         tree->namenode[offset+medright].parent = offset+med;
00802     }
00803     if ( numleft > 0 ) LinkTree(tree,offset,numleft);
00804     if ( numright > 0 ) LinkTree(tree,offset+med+1,numright);
00805     while ( numleft && numright ) { numleft »= 1; numright »= 1; }
00806     if ( numleft ) tree->namenode[offset+med].balance = -1;
00807     else if ( numright ) tree->namenode[offset+med].balance = 1;
00808 }
00809
00810 /*
00811 #] LinkTree :
00812 #[ MakeNameTree :
00813 */
00814
00815 NAMETREE *MakeNameTree(void)
00816 {
00817     NAMETREE *n;

```

```

00818     n = (NAMETREE *)Malloc1(sizeof(NAMETREE),"new nametree");
00819     n->namebuffer = 0;
00820     n->namenode = 0;
00821     n->namesize = n->namefill = n->nodesize = n->nodefill =
00822     n->oldnamefill = n->oldnodefill = 0;
00823     n->globalnamefill = n->globalnodefill =
00824     n->clearnamefill = n->clearnodefill = 0;
00825     n->headnode = -1;
00826     return(n);
00827 }
00828
00829 /*
00830 #] MakeNameTree :
00831 #[ FreeNameTree :
00832 */
00833
00834 void FreeNameTree(NAMETREE *n)
00835 {
00836     if ( n ) {
00837         if ( n->namebuffer ) M_free(n->namebuffer,"nametree->namebuffer");
00838         if ( n->namenode ) M_free(n->namenode,"nametree->namenode");
00839         M_free(n,"nametree");
00840     }
00841 }
00842
00843 /*
00844 #] FreeNameTree :
00845 #[ WildcardNames :
00846 */
00847
00848 void ClearWildcardNames(void)
00849 {
00850     AC.NumWildcardNames = 0;
00851 }
00852
00853 int AddWildcardName(UBYTE *name)
00854 {
00855     GETIDENTITY
00856     int size = 0, tocopy, i;
00857     UBYTE *s = name, *t, *newbuffer;
00858     while ( *s ) { s++; size++; }
00859     for ( i = 0, t = AC.WildcardNames; i < AC.NumWildcardNames; i++ ) {
00860         s = name;
00861         while ( ( *s == *t ) && *s ) { s++; t++; }
00862         if ( *s == 0 && *t == 0 ) return(i+1);
00863         while ( *t ) t++;
00864         t++;
00865     }
00866     tocopy = t - AC.WildcardNames;
00867     if ( tocopy + size + 1 > AC.WildcardBufferSize ) {
00868         if ( AC.WildcardBufferSize == 0 ) {
00869             AC.WildcardBufferSize = size+1;
00870             if ( AC.WildcardBufferSize < 100 ) AC.WildcardBufferSize = 100;
00871         }
00872         else if ( size+1 >= AC.WildcardBufferSize ) {
00873             AC.WildcardBufferSize += size+1;
00874         }
00875         else {
00876             AC.WildcardBufferSize *= 2;
00877         }
00878         newbuffer = (UBYTE *)Malloc1((LONG)AC.WildcardBufferSize,"argument list names");
00879         t = newbuffer;
00880         if ( AC.WildcardNames ) {
00881             s = AC.WildcardNames;
00882             while ( tocopy > 0 ) { *t++ = *s++; tocopy--; }
00883             M_free(AC.WildcardNames,"AC.WildcardNames");
00884         }
00885         AC.WildcardNames = newbuffer;
00886         M_free(AT.WildArgTaken,"AT.WildArgTaken");
00887         AT.WildArgTaken = (WORD *)Malloc1((LONG)AC.WildcardBufferSize*sizeof(WORD)/2
00888             ,"argument list names");
00889     }
00890     s = name;
00891     while ( *s ) *t++ = *s++;
00892     *t = 0;
00893     AC.NumWildcardNames++;
00894     return(AC.NumWildcardNames);
00895 }
00896
00897 int GetWildcardName(UBYTE *name)
00898 {
00899     UBYTE *s, *t;
00900     int i;
00901     for ( i = 0, t = AC.WildcardNames; i < AC.NumWildcardNames; i++ ) {
00902         s = name;
00903         while ( ( *s == *t ) && *s ) { s++; t++; }
00904     }

```

```

00905         if ( *s == 0 && *t == 0 ) return(i+1);
00906         while ( *t ) t++;
00907         t++;
00908     }
00909     return(0);
00910 }
00911 /*
00912 #] WildcardNames :
00913 #[ AddSymbol :
00914
00915     The actual addition. Special routine for additions 'on the fly'
00916 */
00917 int AddSymbol(UBYTE *name, int minpow, int maxpow, int cplx, int dim)
00918 {
00919     int nodenum, numsymbol = AC.Symbols->num;
00920     UBYTE *s = name;
00921     SYMBOLS sym = (SYMBOLS)FromVarList(AC.Symbols);
00922     bzero(sym, sizeof(struct SyMbOl));
00923     sym->name = AddName(*AC.activenames, name, CSYMBOL, numsymbol, &nodenum);
00924     sym->minpower = minpow;
00925     sym->maxpower = maxpow;
00926     sym->complex = cplx;
00927     sym->flags = 0;
00928     sym->node = nodenum;
00929     sym->dimension = dim;
00930     while ( *s ) s++;
00931     sym->namesize = (s-name)+1;
00932     return(numsymbol);
00933 }
00934 /*
00935 #] AddSymbol :
00936 #[ CoSymbol :
00937
00938     Symbol declarations.  name[#{R|I|C}][{[min]:[max]}]
00939     Note that we know already that the parentheses match properly
00940 */
00941 int CoSymbol(UBYTE *s)
00942 {
00943     int type, error = 0, minpow, maxpow, cplx, sgn, dim;
00944     WORD numsymbol;
00945     UBYTE *name, *oldc, c, cc;
00946     do {
00947         minpow = -MAXPOWER;
00948         maxpow = MAXPOWER;
00949         cplx = 0;
00950         dim = 0;
00951         name = s;
00952         if ( ( s = SkipAName(s) ) == 0 ) {
00953             IllForm: MesPrint("&Illegally formed name in symbol statement");
00954             error = 1;
00955             s = SkipField(name, 0);
00956             goto eol;
00957         }
00958         oldc = s; cc = c = *s; *s = 0;
00959         if ( TestName(name) ) { *s = c; goto IllForm; }
00960         if ( cc == '#' ) {
00961             s++;
00962             if ( tolower(*s) == 'r' ) cplx = VARTYPENONE;
00963             else if ( tolower(*s) == 'c' ) cplx = VARTYPECOMPLEX;
00964             else if ( tolower(*s) == 'i' ) cplx = VARTYPEIMAGINARY;
00965             else if ( ( (*s == '-' || *s == '+' || *s == '=')
00966                 && ( s[1] >= '0' && s[1] <= '9' ) )
00967                 || (*s >= '0' && *s <= '9' ) ) {
00968                 LONG x;
00969                 sgn = 0;
00970                 if ( *s == '-' ) { sgn = VARTYPEMINUS; s++; }
00971                 else if ( *s == '+' || *s == '=' ) { sgn = 0; s++; }
00972                 x = *s - '0';
00973                 while ( s[1] >= '0' && s[1] <= '9' ) {
00974                     x = 10*x + (s[1] - '0'); s++;
00975                 }
00976                 if ( x >= MAXPOWER || x <= 1 ) {
00977                     MesPrint("&Illegal value for root of unity %s", name);
00978                     error = 1;
00979                 }
00980                 else {
00981                     maxpow = x;
00982                 }
00983                 cplx = VARTYPEPEROOTOFUNITY | sgn;
00984             }
00985             else {
00986                 MesPrint("&Illegal specification for complexity of symbol %s", name);
00987             }
00988         }
00989     } while (1);
00990 }

```

```

00992         *oldc = c;
00993         error = 1;
00994         s = SkipField(s,0);
00995         goto eol;
00996     }
00997     s++; cc = *s;
00998 }
00999 if ( cc == '{' ) {
01000     s++;
01001     if ( ( *s == 'd' || *s == 'D' ) && s[1] == '=' ) {
01002         s += 2;
01003         if ( *s == '-' || *s == '+' || FG.cTable[*s] == 1 ) {
01004             ParseSignedNumber(dim,s)
01005             if ( dim < -HALFMAX || dim > HALFMAX ) {
01006                 MesPrint("&Warning: dimension of %s (%d) out of range"
01007                     ,name,dim);
01008             }
01009         }
01010         if ( *s != '}' ) goto IllDim;
01011         else s++;
01012     }
01013     else {
01014 IllDim:
01015         MesPrint("&Error: Illegal dimension field for variable %s",name);
01016         error = 1;
01017         s = SkipField(s,0);
01018         goto eol;
01019     }
01020     cc = *s;
01021     if ( cc == '(' ) {
01022         if ( ( cplx & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY ) {
01023             MesPrint("&Root of unity property for %s cannot be combined with power
restrictions",name);
01024             error = 1;
01025         }
01026         s++;
01027         if ( *s == '-' || *s == '+' || FG.cTable[*s] == 1 ) {
01028             ParseSignedNumber(minpow,s)
01029             if ( minpow < -MAXPOWER ) {
01030                 minpow = -MAXPOWER;
01031                 if ( AC.WarnFlag )
01032                     MesPrint("&Warning: minimum power of %s corrected to %d"
01033                         ,name,-MAXPOWER);
01034             }
01035         }
01036         if ( *s != ':' ) {
01037 skippar:
01038             error = 1;
01039             s = SkipField(s,1);
01040             goto eol;
01041         }
01042         else s++;
01043         if ( *s == '-' || *s == '+' || FG.cTable[*s] == 1 ) {
01044             ParseSignedNumber(maxpow,s)
01045             if ( maxpow > MAXPOWER ) {
01046                 maxpow = MAXPOWER;
01047                 if ( AC.WarnFlag )
01048                     MesPrint("&Warning: maximum power of %s corrected to %d"
01049                         ,name,MAXPOWER);
01050             }
01051         }
01052         if ( *s != ')' ) goto skippar;
01053         s++;
01054     }
01055     if ( ( AC.AutoDeclareFlag == 0 &&
01056         ( ( type = GetName(AC.exprnames,name,&numsymbol,NOAUTO) )
01057         != NAMENOTFOUND ) )
01058     || ( ( type = GetName(*AC.activenames),name,&numsymbol,NOAUTO) ) != NAMENOTFOUND ) ) {
01059         if ( type != CSYMBOL ) error = NameConflict(type,name);
01060         else {
01061             SYMBOLS sym = (SYMBOLS)(AC.Symbols->lijst) + numsymbol;
01062             if ( ( numsymbol == AC.lPolyFunVar ) && ( AC.lPolyFunType > 0 )
01063                 && ( AC.lPolyFun != 0 ) && ( minpow > -MAXPOWER || maxpow < MAXPOWER ) ) {
01064                 MesPrint("&The symbol %s is used by power expansions in the PolyRatFun!",name);
01065                 error = 1;
01066             }
01067             sym->complex = cplx;
01068             sym->minpower = minpow;
01069             sym->maxpower = maxpow;
01070             sym->dimension= dim;
01071         }
01072     }
01073     else {
01074         AddSymbol(name,minpow,maxpow,cplx,dim);
01075     }
01076     *oldc = c;
01077     while ( *s == ',' ) s++;
01078 } while ( *s );

```



```

01078     return(error);
01079 }
01080
01081 /*
01082 #] CoSymbol :
01083 #[ AddIndex :
01084
01085 The actual addition. Special routine for additions 'on the fly'
01086 */
01087
01088 int AddIndex(UBYTE *name, int dim, int dim4)
01089 {
01090     int nodenum, numindex = AC.Indices->num;
01091     INDICES ind = (INDICES)FromVarList(AC.Indices);
01092     UBYTE *s = name;
01093     bzero(ind, sizeof(struct InDeX));
01094     ind->name = AddName(*AC.activenames, name, CINDEXT, numindex, &nodenum);
01095     ind->type = 0;
01096     ind->dimension = dim;
01097     ind->flags = 0;
01098     ind->nmin4 = dim4;
01099     ind->node = nodenum;
01100     while ( *s ) s++;
01101     ind->namesize = (s-name)+1;
01102     return(numindex);
01103 }
01104
01105 /*
01106 #] AddIndex :
01107 #[ CoIndex :
01108
01109 Index declarations. name[={number|symbol[:othersymbol]}]
01110 */
01111
01112 int CoIndex(UBYTE *s)
01113 {
01114     int type, error = 0, dim, dim4;
01115     WORD numindex;
01116     UBYTE *name, *oldc, c;
01117     do {
01118         dim = AC.lDefDim;
01119         dim4 = AC.lDefDim4;
01120         name = s;
01121         if ( ( s = SkipAName(s) ) == 0 ) {
01122             IllForm: MesPrint("&Illegally formed name in index statement");
01123             error = 1;
01124             s = SkipField(name, 0);
01125             goto eol;
01126         }
01127         oldc = s; c = *s; *s = 0;
01128         if ( TestName(name) ) { *s = c; goto IllForm; }
01129         if ( c == '=' ) {
01130             s++;
01131             if ( ( s = DoDimension(s, &dim, &dim4) ) == 0 ) {
01132                 *oldc = c;
01133                 error = 1;
01134                 s = SkipField(name, 0);
01135                 goto eol;
01136             }
01137         }
01138         if ( ( AC.AutoDeclareFlag == 0 &&
01139             ( ( type = GetName(AC.exprnames, name, &numindex, NOAUTO) )
01140               != NAMENOTFOUND ) )
01141             || ( ( type = GetName(*AC.activenames, name, &numindex, NOAUTO) ) != NAMENOTFOUND ) ) ) {
01142             if ( type != CINDEXT ) error = NameConflict(type, name);
01143             else { /* reset the dimensions */
01144                 indices[numindex].dimension = dim;
01145                 indices[numindex].nmin4 = dim4;
01146             }
01147         }
01148         else AddIndex(name, dim, dim4);
01149         *oldc = c;
01150     eol: while ( *s == ',' ) s++;
01151     } while ( *s );
01152     return(error);
01153 }
01154
01155 /*
01156 #] CoIndex :
01157 #[ DoDimension :
01158 */
01159
01160 UBYTE *DoDimension(UBYTE *s, int *dim, int *dim4)
01161 {
01162     UBYTE c, *t = s;
01163     int type, error = 0;
01164     WORD numsymbol;

```

```

01165     NAMETREE **oldtree = AC.activenames;
01166     LIST* oldsymbols = AC.Symbols;
01167     *dim4 = -NMIN4SHIFT;
01168     if ( FG.cTable[*s] == 1 ) {
01169 retry:
01170         ParseNumber(*dim,s)
01171     #if ( BITSINWORD/8 < 4 )
01172         if ( *dim >= (1 << (BITSINWORD-1)) ) goto illeg;
01173     #endif
01174         *dim4 = *dim - 4;
01175         return(s);
01176     }
01177     else if ( ( (FG.cTable[*s] == 0) || ( *s == '[' ) )
01178         && ( s = SkipAName(s) ) != 0 ) {
01179         AC.activenames = &(AC.varnames);
01180         AC.Symbols = &(AC.SymbolList);
01181         c = *s; *s = 0;
01182         if ( ( type = GetName(AC.exprnames,t,&numsymbol,NOAUTO) ) != NAMENOTFOUND )
01183         || ( ( type = GetName(AC.varnames,t,&numsymbol,WITHAUTO) ) != NAMENOTFOUND ) ) {
01184             if ( type != CSYMBOL ) error = NameConflict(type,t);
01185         }
01186         else {
01187             numsymbol = AddSymbol(t,-MAXPOWER,MAXPOWER,0,0);
01188             if ( AC.WarnFlag )
01189                 MesPrint("&Warning: Implicit declaration of %s as a symbol",t);
01190         }
01191         *dim = -numsymbol;
01192         if ( ( *s = c ) == ':' ) {
01193             s++;
01194             t = s;
01195             if ( ( s = SkipAName(s) ) == 0 ) goto illeg;
01196             if ( ( ( type = GetName(AC.exprnames,t,&numsymbol,NOAUTO) ) != NAMENOTFOUND )
01197             || ( ( type = GetName(AC.varnames,t,&numsymbol,WITHAUTO) ) != NAMENOTFOUND ) ) ) {
01198                 if ( type != CSYMBOL ) error = NameConflict(type,t);
01199             }
01200             else {
01201                 numsymbol = AddSymbol(t,-MAXPOWER,MAXPOWER,0,0);
01202                 if ( AC.WarnFlag )
01203                     MesPrint("&Warning: Implicit declaration of %s as a symbol",t);
01204             }
01205             *dim4 = -numsymbol-NMIN4SHIFT;
01206         }
01207     }
01208     else if ( *s == '+' && FG.cTable[s[1]] == 1 ) {
01209         s++; goto retry;
01210     }
01211     else {
01212 illeg: MesPrint("&Illegal dimension specification. Should be number >= 0, symbol or symbol:symbol");
01213         return(0);
01214     }
01215     AC.Symbols = oldsymbols;
01216     AC.activenames = oldtree;
01217     if ( error ) return(0);
01218     return(s);
01219 }
01220
01221 /*
01222 #] DoDimension :
01223 #[ CoDimension :
01224 */
01225
01226 int CoDimension(UBYTE *s)
01227 {
01228     s = DoDimension(s,&AC.lDefDim,&AC.lDefDim4);
01229     if ( s == 0 ) return(1);
01230     if ( *s != 0 ) {
01231         MesPrint("&Argument of dimension statement should be number >= 0, symbol or symbol:symbol");
01232         return(1);
01233     }
01234     return(0);
01235 }
01236
01237 /*
01238 #] CoDimension :
01239 #[ AddVector :
01240
01241     The actual addition. Special routine for additions 'on the fly'
01242 */
01243
01244 int AddVector(UBYTE *name, int cplx, int dim)
01245 {
01246     int nodenum, numvector = AC.Vectors->num;
01247     VECTORS v = (VECTORS)FromVarList(AC.Vectors);
01248     UBYTE *s = name;
01249     bzero(v,sizeof(struct VeCtOr));
01250     v->name = AddName(*AC.activenames,name,CVECTOR,numvector,&nodenum);
01251     v->complex = cplx;

```

```

01252     v->node      = nodenum;
01253     v->dimension = dim;
01254     v->flags = 0;
01255     while ( *s ) s++;
01256     v->namesize = (s-name)+1;
01257     return(numvector);
01258 }
01259
01260 /*
01261  #] AddVector :
01262  #[ CoVector :
01263
01264  Vector declarations. The descriptor string is "(,%n)"
01265 */
01266
01267 int CoVector(UBYTE *s)
01268 {
01269     int type, error = 0, dim;
01270     WORD numvector;
01271     UBYTE *name, c, *endname;
01272     do {
01273         name = s;
01274         dim = 0;
01275         if ( ( s = SkipAName(s) ) == 0 ) {
01276 IllForm:      MesPrint("&Illegally formed name in vector statement");
01277                 error = 1;
01278                 s = SkipField(s,0);
01279             }
01280             else {
01281                 c = *s; *s = 0, endname = s;
01282                 if ( TestName(name) ) { *s = c; goto IllForm; }
01283                 if ( c == '{' ) {
01284                     s++;
01285                     if ( ( *s == 'd' || *s == 'D' ) && s[1] == '=' ) {
01286                         s += 2;
01287                         if ( *s == '-' || *s == '+' || FG.cTable[*s] == 1 ) {
01288                             ParseSignedNumber(dim,s)
01289                             if ( dim < -HALFMAX || dim > HALFMAX ) {
01290                                 MesPrint("&Warning: dimension of %s (%d) out of range"
01291                                     ,name,dim);
01292                             }
01293                         }
01294                         if ( *s != '}' ) goto IllDim;
01295                         else s++;
01296                     }
01297                     else {
01298 IllDim:      MesPrint("&Error: Illegal dimension field for variable %s",name);
01299                 error = 1;
01300                 s = SkipField(s,0);
01301                 while ( *s == ',' ) s++;
01302                 continue;
01303             }
01304             }
01305             if ( ( AC.AutoDeclareFlag == 0 &&
01306                 ( ( type = GetName(AC.exprnames,name,&numvector,NOAUTO) )
01307                   != NAMENOTFOUND ) )
01308                 || ( ( type = GetName(*AC.activenames,name,&numvector,NOAUTO) ) != NAMENOTFOUND ) ) {
01309                 if ( type != CVECTOR ) error = NameConflict(type,name);
01310             }
01311             else AddVector(name,0,dim);
01312             *endname = c;
01313         }
01314         while ( *s == ',' ) s++;
01315     } while ( *s );
01316     return(error);
01317 }
01318
01319 /*
01320  #] CoVector :
01321  #[ AddFunction :
01322
01323  The actual addition. Special routine for additions 'on the fly'
01324 */
01325
01326 int AddFunction(UBYTE *name, int comm, int istensor, int cplx, int symprop, int dim, int argmax, int
    argmin)
01327 {
01328     int nodenum, numfunction = AC.Functions->num;
01329     FUNCTIONS fun = (FUNCTIONS)FromVarList(AC.Functions);
01330     UBYTE *s = name;
01331     bzero(fun,sizeof(struct FuNcTiOn));
01332     fun->name = AddName(*AC.activenames,name,CFUNCTION,numfunction,&nodenum);
01333     fun->commute = comm;
01334     fun->spec = istensor;
01335     fun->complex = cplx;
01336     fun->tabl = 0;
01337     fun->flags = 0;

```

```

01338     fun->node = nodenum;
01339     fun->symminfo = 0;
01340     fun->symmetric = symprop;
01341     fun->dimension = dim;
01342     fun->maxnumargs = argmax;
01343     fun->minnumargs = argmin;
01344     while ( *s ) s++;
01345     fun->namesize = (s-name)+1;
01346     return(numfunction);
01347 }
01348
01349 /*
01350 #] AddFunction :
01351 #[ CoCommutateInSet :
01352
01353 Commuting, f1, ..., fn;
01354 */
01355
01356 int CoCommutateInSet(UBYTE *s)
01357 {
01358     UBYTE *name, *ss, c, *start = s;
01359     WORD number, type, *g, *gg;
01360     int error = 0, i, len = StrLen(s), len2 = 0;
01361     if ( AC.CommutateInSet != 0 ) {
01362         g = AC.CommutateInSet;
01363         while ( *g ) g += *g;
01364         len2 = g - AC.CommutateInSet;
01365         if ( len2+len+3 > AC.SizeCommutateInSet ) {
01366             gg = (WORD *)Malloca1((len2+len+3)*sizeof(WORD), "CommutateInSet");
01367             for ( i = 0; i < len2; i++ ) gg[i] = AC.CommutateInSet[i];
01368             gg[len2] = 0;
01369             M_free(AC.CommutateInSet, "CommutateInSet");
01370             AC.CommutateInSet = gg;
01371             AC.SizeCommutateInSet = len+len2+3;
01372             g = AC.CommutateInSet+len2;
01373         }
01374     }
01375     else {
01376         AC.SizeCommutateInSet = len+2;
01377         g = AC.CommutateInSet = (WORD *)Malloca1((len+3)*sizeof(WORD), "CommutateInSet");
01378         *g = 0;
01379     }
01380     gg = g++;
01381     ss = s-1;
01382     for(;;) {
01383         while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
01384         if ( *s == 0 ) {
01385             if ( s - start >= len ) break;
01386             *s = ' '; s++;
01387             *g = 0;
01388             *gg = g-gg;
01389             if ( *gg < 2 ) {
01390                 MesPrint("&There should be at least two noncommuting functions or tensors in a
commuting statement.");
01391                 error = 1;
01392             }
01393             else if ( *gg == 2 ) {
01394                 gg[2] = gg[1]; gg[3] = 0; gg[0] = 3;
01395             }
01396             gg = g++;
01397             continue;
01398         }
01399         if ( s > ss ) {
01400             if ( *s != '{' ) {
01401                 MesPrint("&The CommutateInSet statement should have sets enclosed in {}.");
01402                 error = 1;
01403                 break;
01404             }
01405             ss = s;
01406             SKIPBRA2(ss) /* Note that parentheses were tested before */
01407             *ss = 0;
01408             s++;
01409         }
01410         name = s;
01411         s = SkipAName(s);
01412         c = *s; *s = 0;
01413         if ( ( type = GetName(AC.varnames, name, &number, NOAUTO) ) != CFUNCTION ) {
01414             MesPrint("&%s is not a function or tensor", name);
01415             error = 1;
01416         }
01417         else if ( functions[number].commute == 0 ){
01418             MesPrint("&%s is not a noncommuting function or tensor", name);
01419             error = 1;
01420         }
01421         else {
01422             *g++ = number+FUNCTION;
01423             functions[number].flags |= COULDCOMMUTE;

```

```

01424         if ( number+FUNCTION >= GAMMA && number+FUNCTION <= GAMMASEVEN ) {
01425             functions[GAMMA-FUNCTION].flags |= COULDCOMMUTE;
01426             functions[GAMMAI-FUNCTION].flags |= COULDCOMMUTE;
01427             functions[GAMMAFIVE-FUNCTION].flags |= COULDCOMMUTE;
01428             functions[GAMMASIX-FUNCTION].flags |= COULDCOMMUTE;
01429             functions[GAMMASEVEN-FUNCTION].flags |= COULDCOMMUTE;
01430         }
01431     }
01432     *s = c;
01433 }
01434 return(error);
01435 }
01436
01437 /*
01438 #] CoCommuteInSet :
01439 #[ CoFunction + ...:
01440
01441 Function declarations.
01442 The second parameter indicates commutation properties.
01443 The third parameter tells whether we have a tensor.
01444 */
01445
01446 int CoFunction(UBYTE *s, int comm, int istensor)
01447 {
01448     int type, error = 0, cplx, symtype, dim, argmax, argmin;
01449     WORD numfunction, reverseorder = 0, addone;
01450     UBYTE *name, *oldc, *par, c, cc;
01451     do {
01452         symtype = cplx = 0, argmin = argmax = -1;
01453         dim = 0;
01454         name = s;
01455         if ( ( s = SkipAName(s) ) == 0 ) {
01456             IllForm: MesPrint("&Illegally formed function/tensor name");
01457             error = 1;
01458             s = SkipField(name,0);
01459             goto eol;
01460         }
01461         oldc = s; cc = c = *s; *s = 0;
01462         if ( TestName(name) ) { *s = c; goto IllForm; }
01463         if ( c == '#' ) {
01464             s++;
01465             if ( tolower(*s) == 'r' ) cplx = VARTYPENONE;
01466             else if ( tolower(*s) == 'c' ) cplx = VARTYPECOMPLEX;
01467             else if ( tolower(*s) == 'i' ) cplx = VARTYPEIMAGINARY;
01468             else {
01469                 MesPrint("&Illegal specification for complexity of %s",name);
01470                 *oldc = c;
01471                 error = 1;
01472                 s = SkipField(s,0);
01473                 goto eol;
01474             }
01475             s++; cc = *s;
01476         }
01477         if ( cc == '{' ) {
01478             s++;
01479             if ( (*s == 'd' || *s == 'D') && s[1] == '=' ) {
01480                 s += 2;
01481                 if ( (*s == '-' || *s == '+') || FG.cTable[*s] == 1 ) {
01482                     ParseSignedNumber(dim,s)
01483                     if ( dim < -HALFMAX || dim > HALFMAX ) {
01484                         MesPrint("&Warning: dimension of %s (%d) out of range",
01485                             ,name,dim);
01486                     }
01487                 }
01488                 if ( *s != '}' ) goto IllDim;
01489                 else s++;
01490             }
01491             else {
01492                 IllDim: MesPrint("&Error: Illegal dimension field for variable %s",name);
01493                 error = 1;
01494                 s = SkipField(s,0);
01495                 goto eol;
01496             }
01497             cc = *s;
01498         }
01499         if ( cc == '(' ) {
01500             s++;
01501             if ( *s == '-' ) {
01502                 reverseorder = REVERSEORDER;
01503                 s++;
01504             }
01505             else {
01506                 reverseorder = 0;
01507             }
01508             par = s;
01509             while ( FG.cTable[*s] == 0 ) s++;
01510             cc = *s; *s = 0;

```

```

01511         if ( s <= par ) {
01512 illegsym:      *s = cc;
01513                 MesPrint("&Illegal specification for symmetry of %s",name);
01514                 *oldc = c;
01515                 error = 1;
01516                 s = SkipField(s,1);
01517                 goto eol;
01518             }
01519             if ( StrICont(par,(UBYTE *)"symmetric") == 0 ) symtype = SYMMETRIC;
01520             else if ( StrICont(par,(UBYTE *)"antisymmetric") == 0 ) symtype = ANTISYMMETRIC;
01521             else if ( ( StrICont(par,(UBYTE *)"cyclesymmetric") == 0 )
01522                 || ( StrICont(par,(UBYTE *)"cyclic") == 0 ) ) symtype = CYCLESYMMETRIC;
01523             else if ( ( StrICont(par,(UBYTE *)"rcyclesymmetric") == 0 )
01524                 || ( StrICont(par,(UBYTE *)"rcyclic") == 0 )
01525                 || ( StrICont(par,(UBYTE *)"reversecyclic") == 0 ) ) symtype = RCYCLESYMMETRIC;
01526             else goto illegsym;
01527             *s = cc;
01528             if ( *s != ' ' || ( s[1] && s[1] != ',' && s[1] != '<' ) ) {
01529                 Warning("Excess information in symmetric properties currently ignored");
01530                 s = SkipField(s,1);
01531             }
01532             else s++;
01533             symtype |= reverseorder;
01534             cc = *s;
01535         }
01536 retry:;
01537         if ( cc == '<' ) {
01538             s++; addone = 0;
01539             if ( *s == '=' ) { addone++; s++; }
01540             argmax = 0;
01541             while ( FG.cTable[*s] == 1 ) { argmax = 10*argmax + *s++ - '0'; }
01542             argmax += addone;
01543             par = s;
01544             while ( FG.cTable[*s] == 0 ) s++;
01545             if ( s > par ) {
01546                 cc = *s; *s = 0;
01547                 if ( ( StrICont(par,(UBYTE *)"arguments") == 0 )
01548                     || ( StrICont(par,(UBYTE *)"args") == 0 ) ) {}
01549                 else {
01550                     Warning("Illegal information in number of arguments properties currently
01551 ignored");
01552                     }
01553                 *s = cc;
01554             }
01555             if ( argmax <= 0 ) {
01556                 MesPrint("&Error: Cannot have fewer than 0 arguments for variable %s",name);
01557                 error = 1;
01558             }
01559             cc = *s;
01560         }
01561         if ( cc == '>' ) {
01562             s++; addone = 1;
01563             if ( *s == '=' ) { addone = 0; s++; }
01564             argmin = 0;
01565             while ( FG.cTable[*s] == 1 ) { argmin = 10*argmin + *s++ - '0'; }
01566             argmin += addone;
01567             par = s;
01568             while ( FG.cTable[*s] == 0 ) s++;
01569             if ( s > par ) {
01570                 cc = *s; *s = 0;
01571                 if ( ( StrICont(par,(UBYTE *)"arguments") == 0 )
01572                     || ( StrICont(par,(UBYTE *)"args") == 0 ) ) {}
01573                 else {
01574                     Warning("Illegal information in number of arguments properties currently
01575 ignored");
01576                     }
01577                 *s = cc;
01578             }
01579             cc = *s;
01580         }
01581         if ( cc == '<' ) goto retry;
01582         if ( ( AC.AutoDeclareFlag == 0 &&
01583             ( ( type = GetName(AC.exprnames,name,&numfunction,NOAUTO) )
01584               != NAMENOTFOUND ) )
01585             || ( ( type = GetName(*AC.activenames,name,&numfunction,NOAUTO) ) != NAMENOTFOUND ) ) {
01586             if ( type != CFUNCTION ) error = NameConflict(type,name);
01587             else {
01588                 /*
01589                 FUNCTIONS fun = (FUNCTIONS) (AC.Functions->lijst) + numfunction-FUNCTION; */
01590                 FUNCTIONS fun = (FUNCTIONS) (AC.Functions->lijst) + numfunction;
01591             }
01592             if ( fun->tabl != 0 ) {
01593                 MesPrint("&Illegal attempt to change table into function");
01594                 error = 1;
01595             }
01596             fun->complex = cplx;
01597             fun->commute = comm;

```

```

01596         if ( istensor && fun->spec == 0 ) {
01597             MesPrint("&Function %s changed to tensor",name);
01598             error = 1;
01599         }
01600         else if ( istensor == 0 && fun->spec > 0 ) {
01601             MesPrint("&Tensor %s changed to function",name);
01602             error = 1;
01603         }
01604         else if ( fun->spec == VERTEXFUNCTION ) {
01605             MesPrint("&Function or Tensor %s already declared as a Particle",name);
01606             error = 1;
01607         }
01608         fun->spec = istensor;
01609         if ( fun->symmetric != symtype ) {
01610             fun->symmetric = symtype;
01611             AC.SymChangeFlag = 1;
01612         }
01613         fun->maxnumargs = argmax;
01614         fun->minnumargs = argmin;
01615     }
01616 }
01617 else {
01618     AddFunction(name,comm,istensor,cplx,symtype,dim,argmax,argmin);
01619 }
01620 *oldc = c;
01621 eol: while ( *s == ',' ) s++;
01622 } while ( *s );
01623 return(error);
01624 }
01625
01626 int CoNFunction(UBYTE *s) { return(CoFunction(s,1,0)); }
01627 int CoCFunction(UBYTE *s) { return(CoFunction(s,0,0)); }
01628 int CoNTensor(UBYTE *s) { return(CoFunction(s,1,2)); }
01629 int CoCTensor(UBYTE *s) { return(CoFunction(s,0,2)); }
01630
01631 /*
01632 #] CoFunction + ...:
01633 #[ DoTable :
01634
01635 Syntax:
01636 Table [check] [strict|relax] [zerofill] name(:1:2,...,regular arguments);
01637 name must be the name of a regular function.
01638 the table indices must be the first arguments.
01639 The parenthesis indicates 'name' as opposed to the options.
01640
01641 We leave behind:
01642 a struct tabl in the FUNCTION struct
01643 Regular table:
01644     an array tablepointers for the pointers to elements of rhs
01645     in the compiler struct cbuf[T->bufnum]
01646     an array MINMAX T->mm with the minima and maxima
01647     a prototype array
01648     an offset in the compiler buffer for the pattern to be matched
01649 Sparse table:
01650     Just the number of dimensions
01651     We will keep track of the number of defined elements in totind
01652     and in tablepointers we will have numind+1 positions for each
01653     element. The first numind elements for the indices and the
01654     last one for the element in cbuf[T->bufnum].rhs
01655
01656     If the number of dimensions is *<number>, there is not a fixed
01657     number of dimensions. Just a maximum. In that case the first
01658     index should be the number of other dimensions. This first index
01659     does not count in <number>.
01660
01661 Complication: to preserve speed we need a prototype and a pattern
01662 for each thread when we use WITHPTHREADS. This is because we write
01663 into those when looking for the pattern.
01664 */
01665
01666 static int nwarntab = 1;
01667
01668 int DoTable(UBYTE *s, int par)
01669 {
01670     GETIDENTITY
01671     UBYTE *name, *p, *inp, c;
01672     int i, j, k, sparseflag = 0, rflag = 0, checkflag = 0;
01673     int error = 0, ret, oldcbufnum, oldEside;
01674     WORD funnum, type, *OldWork, *w, *ww, *t, *tt, *flags1, oldnumrhs,oldnumlhs;
01675     LONG oldcpointer;
01676     MINMAX *mm, *mml;
01677     LONG x, y;
01678     TABLES T;
01679     CBUF *C;
01680
01681     while ( *s == ',' ) s++;
01682     do {

```

```

01683     name = s;
01684     if ( ( s = SkipAName(s) ) == 0 ) {
01685 IllForm: MesPrint("&Illegal name or option in table declaration");
01686         return(1);
01687     }
01688     c = *s; *s = 0;
01689     if ( TestName(name) ) { *s = c; goto IllForm; }
01690     *s = c;
01691     if ( *s == '(' ) break;
01692     if ( *s != ',' ) {
01693         MesPrint("&Illegal definition of table");
01694         return(1);
01695     }
01696     *s = 0;
01697 /*
01698     Secondary options
01699 */
01700     if ( StrICmp(name, (UBYTE *)("check" )) == 0 ) checkflag = 1;
01701     else if ( StrICmp(name, (UBYTE *)("zero" )) == 0 ) checkflag = 2;
01702     else if ( StrICmp(name, (UBYTE *)("one" )) == 0 ) checkflag = 3;
01703     else if ( StrICmp(name, (UBYTE *)("strict")) == 0 ) rflag = 1;
01704     else if ( StrICmp(name, (UBYTE *)("relax" )) == 0 ) rflag = -1;
01705     else if ( StrICmp(name, (UBYTE *)("zerofill" )) == 0 ) { rflag = -2; checkflag = 2; }
01706     else if ( StrICmp(name, (UBYTE *)("onefill" )) == 0 ) { rflag = -3; checkflag = 3; }
01707     else if ( StrICmp(name, (UBYTE *)("sparse")) == 0 ) sparseflag |= 1;
01708     else if ( StrICmp(name, (UBYTE *)("base")) == 0 ) sparseflag |= 3;
01709     else if ( StrICmp(name, (UBYTE *)("tablebase")) == 0 ) sparseflag |= 3;
01710     else {
01711         MesPrint("&Illegal option in table definition: '%s'",name);
01712         error = 1;
01713     }
01714     *s++ = ',';
01715     while ( *s == ',' ) s++;
01716 } while ( *s );
01717 if ( name == s || *s == 0 ) {
01718     MesPrint("&Illegal name or option in table declaration");
01719     return(1);
01720 }
01721 *s = 0; /* *s could only have been a parenthesis */
01722 if ( sparseflag ) {
01723     if ( checkflag == 1 ) rflag = 0;
01724     else if ( checkflag == 2 ) rflag = -2;
01725     else if ( checkflag == 3 ) rflag = -3;
01726     else rflag = -1;
01727 }
01728 if ( ( ret = GetVar(name,&type,&funnum,CFUNCTION,NOAUTO) ) ==
01729     NAMENOTFOUND ) {
01730     if ( par == 0 ) {
01731         funnum = EntVar(CFUNCTION,name,0,1,0,0);
01732     }
01733     else if ( par == 1 || par == 2 ) {
01734         funnum = EntVar(CFUNCTION,name,0,0,0,0);
01735     }
01736 }
01737 else if ( ret <= 0 ) {
01738     funnum = EntVar(CFUNCTION,name,0,0,0,0);
01739     error = 1;
01740 }
01741 else {
01742     if ( par == 2 ) {
01743         if ( nwarntab ) {
01744             Warning("Table now declares its (commuting) function.");
01745             Warning("Earlier definition in Function statement obsolete. Please remove.");
01746             nwarntab = 0;
01747         }
01748     }
01749     else {
01750         error = 1;
01751         MesPrint("&(N)(C)Tables should not be declared previously");
01752     }
01753 }
01754 if ( functions[funnum].spec > 0 ) {
01755     MesPrint("&Tensors cannot become tables");
01756     return(1);
01757 }
01758 if ( functions[funnum].symmetric > 0 ) {
01759     MesPrint("&Functions with nontrivial symmetrization properties cannot become tables");
01760     return(1);
01761 }
01762 if ( functions[funnum].tabl ) {
01763     MesPrint("&Redefinition of an existing table is not allowed.");
01764     return(1);
01765 }
01766 functions[funnum].tabl = T = (TABLES)Malloc1(sizeof(struct TableS),"table");
01767 /*
01768     Next we find the size of the table (if it is not sparse)
01769 */

```



```

01770     T->defined = T->mdefined = 0; T->sparse = sparseflag; T->mm = 0; T->flags = 0;
01771     T->numtree = 0; T->rootnum = 0; T->MaxTreeSize = 0;
01772     T->boomlijst = 0;
01773     T->strict = rflag;
01774     T->bounds = checkflag;
01775     T->bufnum = inicbufs();
01776     T->argtail = 0;
01777     T->sparse = 0;
01778     T->bufferssize = 8;
01779     T->buffers = (WORD *)Malloc1(sizeof(WORD)*T->bufferssize,"Table buffers");
01780     T->buffersfill = 0;
01781     T->buffers[T->buffersfill++] = T->bufnum;
01782     T->mode = 0;
01783     T->numdummies = 0;
01784     mm = T->mm;
01785     T->numind = 0;
01786     if ( rflag > 0 ) AC.MustTestTable++;
01787     T->totind = 0; /* Table hasn't been checked */
01788
01789     p = s; *s = '(';
01790     if ( sparseflag ) {
01791 /*
01792      First copy the tail, just in case we will construct a tablebase
01793      Note that we keep the ( to indicate a tail
01794      The actual arguments can be found after the comma. Before we have
01795      the dimension which the tablebase will need for consistency checking.
01796 */
01797         inp = p+1;
01798         SKIPBRA3(inp);
01799         c = *inp; *inp = 0;
01800         T->argtail = strdup1(p,"argtail");
01801         *inp = c;
01802 /*
01803      Now the regular compilation
01804 */
01805         inp = p++;
01806         if ( *p == '<' ) {
01807             WORD inc = 1;
01808             p++;
01809             if ( *p == '=' ) { inc++; p++; }
01810 /*
01811      We will use one extra number for telling how many there really are.
01812 */
01813             x = 0;
01814             while ( *p <= '9' && *p >= '0' ) x = 10*x + (*p++-'0');
01815             x = -x-inc;
01816             if ( x == -1 ) {
01817                 MesPrint("&Maximum number of dimensions in *-table should be at least one.");
01818                 error = 1;
01819                 goto FinishUp2;
01820             }
01821         }
01822         else {
01823             ParseNumber(x,p)
01824             if ( FG.cTable[p[-1]] != 1 || ( *p != ',' && *p != ')' ) ) {
01825                 p = inp;
01826                 MesPrint("&First argument in a sparse table must be a number of dimensions");
01827                 error = 1;
01828                 x = 1;
01829             }
01830         }
01831         T->numind = x;
01832         T->mm = (MINMAX *)Malloc1(ABS(x)*sizeof(MINMAX),"table dimensions");
01833         T->flags = (WORD *)Malloc1(ABS(x)*sizeof(WORD),"table flags");
01834         mm = T->mm;
01835         inp = p;
01836         if ( *inp != ')' ) inp++;
01837         T->totind = 0; /* At the moment there are this many */
01838         T->tablepointers = 0;
01839         T->reserved = 0;
01840     }
01841     else {
01842         T->numind = 0;
01843         T->totind = 1;
01844         for(;;) { /* Read the dimensions as far as they can be recognized */
01845             inp = ++p;
01846             if ( FG.cTable[*p] != 1 && *p != '+' && *p != '-' ) break;
01847             ParseSignedNumber(x,p)
01848             if ( FG.cTable[p[-1]] != 1 || *p != ':' ) break;
01849             p++;
01850             ParseSignedNumber(y,p)
01851             if ( FG.cTable[p[-1]] != 1 || ( *p != ',' && *p != ')' ) ) {
01852                 MesPrint("&Illegal dimension field in table declaration");
01853                 return(1);
01854             }
01855             mm1 = (MINMAX *)Malloc1((T->numind+1)*sizeof(MINMAX),"table dimensions");
01856             flags1 = (WORD *)Malloc1((T->numind+1)*sizeof(WORD),"table flags");

```

```

01857         for ( i = 0; i < T->numind; i++ ) { mml[i] = T->mm[i]; flagsl[i] = T->flags[i]; }
01858         if ( T->mm ) M_free(T->mm,"table dimensions");
01859         if ( T->flags ) M_free(T->flags,"table flags");
01860         T->mm = mml;
01861         T->flags = flagsl;
01862         mm = T->mm + T->numind;
01863         mm->mini = x; mm->maxi = y;
01864         T->totind *= mm->maxi-mm->mini+1;
01865         T->numind++;
01866         if ( *p == ')' ) { inp = p; break; }
01867     }
01868     w = T->tablepointers
01869     = (WORD *)Malloc1(TABLEEXTENSION*sizeof(WORD)*(T->totind),"table pointers");
01870     i = T->totind;
01871     for ( i = TABLEEXTENSION*T->totind; i > 0; i-- ) *w++ = -1; /* means: undefined */
01872     for ( i = T->numind-1, x = 1; i >= 0; i-- ) {
01873         T->mm[i].size = x; /* Defines increment in this dimension */
01874         x *= T->mm[i].maxi - T->mm[i].mini + 1;
01875     }
01876 }
01877 /*
01878 Now we redo the 'function part' and send it to the compiler.
01879 The prototype has to be picked up properly.
01880 */
01881 AT.WorkPointer++; /* We need one extra word later */
01882 OldWork = AT.WorkPointer;
01883 oldcbufnum = AC.cbufnum;
01884 AC.cbufnum = T->bufnum;
01885 C = cbuf+AC.cbufnum;
01886 oldcpointer = C->Pointer - C->Buffer;
01887 oldnumlhs = C->numlhs;
01888 oldnumrhs = C->numrhs;
01889 AddLHS(AC.cbufnum);
01890 while ( s >= name ) *--inp = *s--;
01891 w = AT.WorkPointer;
01892 AC.ProtoType = w;
01893 *w++ = SUBEXPRESSION;
01894 *w++ = SUBEXPSIZE;
01895 *w++ = 0;
01896 *w++ = 1;
01897 *w++ = AC.cbufnum;
01898 FILLSUB(w)
01899 AC.WildC = w;
01900 AC.NwildC = 0;
01901 AT.WorkPointer = w + 4*AM.MaxWildcards;
01902 if ( ( ret = CompileAlgebra(inp,LHSIDE,AC.ProtoType) ) < 0 ) {
01903     error = 1; goto FinishUp;
01904 }
01905 if ( AC.NwildC && SortWild(w,AC.NwildC) ) error = 1;
01906 w += AC.NwildC;
01907 i = w-OldWork;
01908 OldWork[1] = i;
01909 /*
01910 Basically we have to pull this pattern through Generator in case
01911 there are functions inside functions, or parentheses.
01912 We have to temporarily disable the .tabl to avoid problems with
01913 TestSub.
01914 Essential: we need to start NewSort twice to avoid the PutOut routines.
01915 The ground pattern is sitting in C->numrhs, but it could be that it
01916 has subexpressions in it. Hence it has to be worked out as the lhs in
01917 id statements (in comexpr.c).
01918 */
01919 OldWork[2] = C->numrhs;
01920 *w++ = 1; *w++ = 1; *w++ = 3;
01921 OldWork[-1] = w-OldWork+1;
01922 AT.WorkPointer = w;
01923 ww = C->rhs[C->numrhs];
01924 for ( j = 0; j < *ww; j++ ) w[j] = ww[j];
01925 AT.WorkPointer = w+*w;
01926 if ( *ww == 0 || ww[*ww] != 0 ) {
01927     MesPrint("&Illegal table pattern definition");
01928     AC.lhdollarflag = 0;
01929     error = 1;
01930 }
01931 if ( error ) goto FinishUp;
01932
01933 if ( NewSort(BHEAD0) || NewSort(BHEAD0) ) { error = 1; goto FinishUp; }
01934 AN.RepPoint = AT.RepCount + 1;
01935 AC.lhdollarflag = 0; oldEside = AR.Eside; AR.Eside = LHSIDE;
01936 AR.Cnumlhs = C->numlhs;
01937 functions[funnum].tabl = 0;
01938 if ( Generator(BHEAD w,C->numlhs) ) {
01939     functions[funnum].tabl = T;
01940     AR.Eside = oldEside;
01941     LowerSortLevel(); LowerSortLevel(); goto FinishUp;
01942 }
01943 functions[funnum].tabl = T;

```

```

01944     AR.Eside = oldEside;
01945     AT.WorkPointer = w;
01946     if ( EndSort(BHEAD w,0) < 0 ) { LowerSortLevel(); goto FinishUp; }
01947     if ( *w == 0 || *(w+w) != 0 ) {
01948         MesPrint("&Irregular pattern in table definition");
01949         error = 1;
01950         goto FinishUp;
01951     }
01952     LowerSortLevel();
01953     if ( AC.lhdollarflag ) {
01954         MesPrint("&Unexpanded dollar variables are not allowed in table definition");
01955         error = 1;
01956         goto FinishUp;
01957     }
01958     AT.WorkPointer = ww = w + *w;
01959     if ( ww[-1] != 3 || ww[-2] != 1 || ww[-3] != 1 ) {
01960         MesPrint("&Coefficient of pattern in table definition should be 1.");
01961         error = 1;
01962         goto FinishUp;
01963     }
01964     AC.DumNum = 0;
01965     /*
01966     Now we have to allocate space for prototype+pattern
01967     In the case of TFORM we need extra pointers, because each worker has its own
01968     */
01969     j = *w + ABS(T->numind)*2-3;
01970     #ifdef WITHPTHREADS
01971     { int n;
01972     T->prototypeSize = ((i+j)*sizeof(WORD)+2*sizeof(WORD *)) * AM.totalnumberofthreads;
01973     T->prototype = (WORD **)Malloc1(T->prototypeSize, "table prototype");
01974     T->pattern = T->prototype + AM.totalnumberofthreads;
01975     t = (WORD *) (T->pattern + AM.totalnumberofthreads);
01976     for ( n = 0; n < AM.totalnumberofthreads; n++ ) {
01977         T->prototype[n] = t;
01978         for ( k = 0; k < i; k++ ) *t++ = OldWork[k];
01979     }
01980     T->pattern[0] = t;
01981     j--; w++;
01982     w[1] += ABS(T->numind)*2;
01983     for ( k = 0; k < FUNHEAD; k++ ) *t++ = *w++;
01984     j -= FUNHEAD;
01985     for ( k = 0; k < ABS(T->numind); k++ ) { *t++ = -SNUMBER; *t++ = 0; j -= 2; }
01986     for ( k = 0; k < j; k++ ) *t++ = *w++;
01987     if ( sparseflag ) T->pattern[0][1] = t - T->pattern[0];
01988     k = t - T->pattern[0];
01989     for ( n = 1; n < AM.totalnumberofthreads; n++ ) {
01990         T->pattern[n] = t; tt = T->pattern[0];
01991         for ( i = 0; i < k; i++ ) *t++ = *tt++;
01992     }
01993     }
01994     #else
01995     T->prototypeSize = (i+j)*sizeof(WORD);
01996     T->prototype = (WORD *)Malloc1(T->prototypeSize, "table prototype");
01997     T->pattern = T->prototype + i;
01998     for ( k = 0; k < i; k++ ) T->prototype[k] = OldWork[k];
01999     t = T->pattern;
02000     j--; w++;
02001     w[1] += ABS(T->numind)*2;
02002     for ( k = 0; k < FUNHEAD; k++ ) *t++ = *w++;
02003     j -= FUNHEAD;
02004     for ( k = 0; k < ABS(T->numind); k++ ) { *t++ = -SNUMBER; *t++ = 0; j -= 2; }
02005     for ( k = 0; k < j; k++ ) *t++ = *w++;
02006     if ( sparseflag ) T->pattern[1] = t - T->pattern;
02007     #endif
02008     /*
02009     At this point we can pop the compilerbuffer.
02010     */
02011     C->Pointer = C->Buffer + oldcpointer;
02012     C->numrhs = oldnumrhs;
02013     C->numlhs = oldnumlhs;
02014     /*
02015     Now check whether wildcards get converted to dollars (for PARALLEL)
02016     We give a warning!
02017     */
02018     #ifdef WITHPTHREADS
02019     t = T->prototype[0];
02020     #else
02021     t = T->prototype;
02022     #endif
02023     tt = t + t[1]; t += SUBEXPSIZE;
02024     while ( t < tt ) {
02025         if ( *t == LOADDOLLAR ) {
02026             Warning("The use of $-variable assignments in tables disables parallel\
02027 execution for the whole program.");
02028             AM.hparallelflag |= NOPARALLEL_TBLDOLLAR;
02029             AC.mparallelflag |= NOPARALLEL_TBLDOLLAR;
02030             AddPotModdollar(t[2]);

```

```

02031     }
02032     t += t[1];
02033 }
02034 FinishUp;;
02035 AT.WorkPointer = OldWork - 1;
02036 AC.cbufnum = oldcbufnum;
02037 FinishUp2;;
02038 if ( T->sparse ) ClearTableTree(T);
02039 if ( ( sparseflag & 2 ) != 0 ) {
02040     if ( T->spare == 0 ) { SpareTable(T); }
02041 }
02042 return(error);
02043 }
02044
02045 /*
02046 #] DoTable :
02047 #[ CoTable :
02048 */
02049
02050 int CoTable(UBYTE *s)
02051 {
02052     return(DoTable(s,2));
02053 }
02054
02055 /*
02056 #] CoTable :
02057 #[ CoNTable :
02058 */
02059
02060 int CoNTable(UBYTE *s)
02061 {
02062     return(DoTable(s,0));
02063 }
02064
02065 /*
02066 #] CoNTable :
02067 #[ CoCTable :
02068 */
02069
02070 int CoCTable(UBYTE *s)
02071 {
02072     return(DoTable(s,1));
02073 }
02074
02075 /*
02076 #] CoCTable :
02077 #[ EmptyTable :
02078 */
02079
02080 void EmptyTable(TABLES T)
02081 {
02082     int j;
02083     if ( T->sparse ) ClearTableTree(T);
02084     if ( T->boomlijst ) M_free(T->boomlijst,"TableTree");
02085     T->boomlijst = 0;
02086     for ( j = 0; j < T->buffersfill; j++ ) { /* was <= */
02087         finishcbuf(T->buffers[j]);
02088     }
02089     if ( T->buffers ) M_free(T->buffers,"Table buffers");
02090     finishcbuf(T->bufnum);
02091     T->bufnum = inicbufs();
02092     T->bufferssize = 8;
02093     T->buffers = (WORD *)Malloc1(sizeof(WORD)*T->bufferssize,"Table buffers");
02094     T->buffersfill = 0;
02095     T->buffers[T->buffersfill++] = T->bufnum;
02096     T->defined = T->mdefined = 0; T->flags = 0;
02097     T->numtree = 0; T->rootnum = 0; T->MaxTreeSize = 0;
02098     T->spare = 0; T->reserved = 0;
02099     if ( T->spare ) {
02100         TABLES TT = T->spare;
02101         if ( TT->mm ) M_free(TT->mm,"tableminmax");
02102         if ( TT->flags ) M_free(TT->flags,"tableflags");
02103         if ( TT->tablepointers ) M_free(TT->tablepointers,"tablepointers");
02104         for ( j = 0; j < TT->buffersfill; j++ ) {
02105             finishcbuf(TT->buffers[j]);
02106         }
02107         if ( TT->boomlijst ) M_free(TT->boomlijst,"TableTree");
02108         if ( TT->buffers ) M_free(TT->buffers,"Table buffers");
02109         M_free(TT,"table");
02110         SpareTable(T);
02111     }
02112     else {
02113         WORD *w = T->tablepointers;
02114         j = T->totind;
02115         for ( j = TABLEEXTENSION*T->totind; j > 0; j-- ) *w++ = -1; /* means: undefined */
02116     }
02117 }

```

```

02118
02119 /*
02120 #] EmptyTable :
02121 #[ AddSet :
02122 */
02123
02124 int AddSet(UBYTE *name, WORD dim)
02125 {
02126     int nodenum, numset = AC.SetList.num;
02127     SETS set = (SETS)FromVarList(&AC.SetList);
02128     UBYTE *s;
02129     if ( name ) {
02130         set->name = AddName(AC.varnames, name, CSET, numset, &nodenum);
02131         s = name;
02132         while ( *s ) s++;
02133         set->namesize = (s-name)+1;
02134         set->node = nodenum;
02135     }
02136     else {
02137         set->name = 0;
02138         set->namesize = 0;
02139         set->node = -1;
02140     }
02141     set->first =
02142     set->last = AC.SetElementList.num; /* set has no elements yet */
02143     set->type = -1; /* undefined as of yet */
02144     set->dimension = dim;
02145     set->flags = 0;
02146     return(numset);
02147 }
02148
02149 /*
02150 #] AddSet :
02151 #[ DoElements :
02152
02153 Remark (25-mar-2011): If the dimension has been set (dim != MAXPOSITIVE)
02154 we want to test dimensions. Numbers count as dimension zero?
02155 */
02156
02157 int DoElements(UBYTE *s, SETS set, UBYTE *name)
02158 {
02159     int type, error = 0, x, sgn, i;
02160     WORD numset, *e;
02161     UBYTE c, *cname;
02162     while ( *s ) {
02163         if ( *s == ',' ) { s++; continue; }
02164         sgn = 0;
02165         while ( *s == '-' || *s == '+' ) {
02166             if ( *s == '-' ) sgn ^= 1;
02167             s++;
02168         }
02169         cname = s;
02170         if ( FG.cTable[*s] == 0 || *s == '_' || *s == '[' ) {
02171             if ( ( s = SkipAName(s) ) == 0 ) {
02172                 MesPrint("&Illegal name in set definition");
02173                 return(1);
02174             }
02175             c = *s; *s = 0;
02176             if ( ( ( type = GetName(AC.exprnames, cname, &numset, NOAUTO) ) == NAMENOTFOUND )
02177                 && ( ( type = GetOName(AC.varnames, cname, &numset, WITHAUTO) ) == NAMENOTFOUND ) ) {
02178                 DUBIOUSV dv;
02179                 int nodenum;
02180                 MesPrint("&%s has not been declared", cname);
02181             }
02182             /*
02183             We enter a 'dubious' declaration to cut down on errors
02184             */
02185             numset = AC.DubiousList.num;
02186             dv = (DUBIOUSV)FromVarList(&AC.DubiousList);
02187             dv->name = AddName(AC.varnames, cname, CDUBIOUS, numset, &nodenum);
02188             dv->node = nodenum;
02189             set->type = type = CDUBIOUS;
02190             set->dimension = 0;
02191             error = 1;
02192         }
02193         if ( set->type == -1 ) {
02194             if ( type == CSYMBOL ) {
02195                 for ( i = set->first; i < set->last; i++ ) {
02196                     SetElements[i] += 2*MAXPOWER;
02197                 }
02198                 set->type = type;
02199             }
02200             if ( set->type != type && set->type != CDUBIOUS
02201                 && type != CDUBIOUS ) {
02202                 if ( set->type != CNUMBER || ( type != CSYMBOL
02203                     && type != CINDEX ) ) {
02204                     MesPrint(

```

```

02205         "%s has not the same type as the other members of the set"
02206         ,cname);
02207         error = 1;
02208         set->type = CDUBIOUS;
02209     }
02210     else {
02211         if ( type == CSYMBOL ) {
02212             for ( i = set->first; i < set->last; i++ ) {
02213                 SetElements[i] += 2*MAXPOWER;
02214             }
02215         }
02216         set->type = type;
02217     }
02218 }
02219 if ( set->dimension != MAXPOSITIVE ) { /* Dimension check */
02220     switch ( set->type ) {
02221         case CSYMBOL:
02222             if ( symbols[numset].dimension != set->dimension ) {
02223                 MesPrint("&Dimension check failed in set %s, symbol %s",
02224                     VARNAME(Sets, (set-Sets)),
02225                     VARNAME(symbols, numset));
02226                 error = 1;
02227                 set->dimension = MAXPOSITIVE;
02228             }
02229             break;
02230         case CVECTOR:
02231             if ( vectors[numset-AM.OffsetVector].dimension != set->dimension ) {
02232                 MesPrint("&Dimension check failed in set %s, vector %s",
02233                     VARNAME(Sets, (set-Sets)),
02234                     VARNAME(vectors, (numset-AM.OffsetVector)));
02235                 error = 1;
02236                 set->dimension = MAXPOSITIVE;
02237             }
02238             break;
02239         case CFUNCTION:
02240             if ( functions[numset-FUNCTION].dimension != set->dimension ) {
02241                 MesPrint("&Dimension check failed in set %s, function %s",
02242                     VARNAME(Sets, (set-Sets)),
02243                     VARNAME(functions, (numset-FUNCTION)));
02244                 error = 1;
02245             }
02246             break;
02247             set->dimension = MAXPOSITIVE;
02248     }
02249 }
02250 if ( sgn ) {
02251     if ( type != CVECTOR ) {
02252         MesPrint("&Illegal use of - sign in set. Can use only with vector or number");
02253         error = 1;
02254     }
02255 }
02256 /*
02257     numset = AM.OffsetVector - numset;
02258     numset |= SPECMASK;
02259     numset = AM.OffsetVector - numset;
02260 */
02261     numset -= WILDMASK;
02262 }
02263 *s = c;
02264 if ( name == 0 && *s == '?' ) {
02265     s++;
02266     switch ( set->type ) {
02267         case CSYMBOL:
02268             numset = -numset; break;
02269         case CVECTOR:
02270             numset += WILDOFFSET; break;
02271         case CINDEX:
02272             numset |= WILDMASK; break;
02273         case CFUNCTION:
02274             numset |= WILDMASK; break;
02275     }
02276     AC.wildflag = 1;
02277 }
02278 /*
02279     Now add the element to the set.
02280 */
02281     e = (WORD *)FromVarList(&AC.SetElementList);
02282     *e = numset;
02283     (set->last)++;
02284 }
02285 else if ( FG.cTable[*s] == 1 ) {
02286     ParseNumber(x,s)
02287     if ( sgn ) x = -x;
02288     if ( x >= MAXPOWER || x <= -MAXPOWER ||
02289         ( set->type == CINDEX && ( x < 0 || x >= AM.OffsetIndex ) ) ) {
02290         MesPrint("&Illegal value for set element: %d",x);
02291         if ( AC.firstconstindex ) {
02292             MesPrint("&0 <= Fixed indices < ConstIndex(which is %d)",

```

```

02292         AM.OffsetIndex-1);
02293     MesPrint("&For setting ConstIndex, read the chapter on the setup file");
02294     AC.firstconstindex = 0;
02295     }
02296     error = 1;
02297     x = 0;
02298     }
02299 /*
02300     Check what is allowed with the type.
02301 */
02302     if ( set->type == -1 ) {
02303         if ( x < 0 || x >= AM.OffsetIndex ) {
02304             for ( i = set->first; i < set->last; i++ ) {
02305                 SetElements[i] += 2*MAXPOWER;
02306             }
02307             set->type = CSYMBOL;
02308         }
02309         else set->type = CNUMBER;
02310     }
02311     else if ( set->type == CDUBIOUS ) {}
02312     else if ( set->type == CNUMBER && x < 0 ) {
02313         for ( i = set->first; i < set->last; i++ ) {
02314             SetElements[i] += 2*MAXPOWER;
02315         }
02316         set->type = CSYMBOL;
02317     }
02318     else if ( set->type != CSYMBOL && ( x < 0 ||
02319 ( set->type != CINDEX && set->type != CNUMBER ) ) ) {
02320         MesPrint("&Illegal mixture of element types in set");
02321         error = 1;
02322         set->type = CDUBIOUS;
02323     }
02324 /*
02325     Allocate an element
02326 */
02327     e = (WORD *)FromVarList(&AC.SetElementList);
02328     (set->last)++;
02329     if ( set->type == CSYMBOL ) *e = x + 2*MAXPOWER;
02330 /*
02331     else if ( set->type == CINDEX ) *e = x; */
02332     else *e = x;
02333     }
02334     else {
02335         MesPrint("&Illegal object in list of set elements");
02336         return(1);
02337     }
02338     if ( error == 0 && ( ( set->flags & ORDEREDSET ) == ORDEREDSET ) ) {
02339 /*
02340         The set->last-set->first list of numbers must be sorted.
02341         Because we plan here potentially thousands of elements we use
02342         a simple version of splitmerge. In ordered sets we can search
02343         later with a binary search.
02344 */
02345         SimpleSplitMerge(SetElements+set->first,set->last-set->first);
02346     }
02347     return(error);
02348 }
02349
02350 /*
02351 #] DoElements :
02352 #[ CoSet :
02353
02354     Set declarations.
02355 */
02356
02357 int CoSet(UBYTE *s)
02358 {
02359     int type, error = 0, ordered = 0;
02360     UBYTE *name = s, c, *ss;
02361     SETS set;
02362     WORD numberofset, dim = MAXPOSITIVE;
02363 #ifdef WITHFLOAT
02364 /*-----*/
02365     {
02366         WORD numeq = 0;
02367         LONG x;
02368         ss = s;
02369         while ( *ss && *ss != ':' ) { if ( *ss == '=' ) numeq++; ss++; }
02370         if ( *ss == 0 && numeq == 1 ) { /* We have the Set var = value; variety */
02371             while ( FG.cTable[*s] == 0 ) s++;
02372             ss = s; c = *s; *s = 0;
02373             if ( c != '=' ) {
02374 Proper:
02375                 MesPrint("&Proper syntax for value-set is `Set name = value'");
02376                 return(1);
02377             }
02378             x = 0; s++;

```

```

02379         while ( *s >= '0' && *s <= '9' ) x = 10*x + (*s++-'0');
02380         if ( *s ) goto Proper;
02381         if ( StrICmp(name,(UBYTE *)"maxweight") == 0 ) {
02382             AC.tMaxWeight = x; /* Temporary. Made permanent later */
02383         }
02384         else if ( StrICmp(name,(UBYTE *)"defaultprecision") == 0 ) {
02385             AC.tDefaultPrecision = x; /* Temporary. Made permanent later */
02386         }
02387         else {
02388             MesPrint("&Illegal subkey in value set: %s",name);
02389             return(1);
02390         }
02391     }
02392 }
02393 /*-----*/
02394 #endif
02395 if ( ( s = SkipAName(s) ) == 0 ) {
02396     IllForm:MesPrint("&Illegal name for set");
02397     return(1);
02398 }
02399 c = *s; *s = 0;
02400 if ( TestName(name) ) goto IllForm;
02401 if ( ( ( type = GetName(AC.exprnames,name,&numberofset,NOAUTO) ) != NAMENOTFOUND )
02402 || ( ( type = GetName(AC.varnames,name,&numberofset,NOAUTO) ) != NAMENOTFOUND ) ) {
02403     if ( type != CSET ) NameConflict(type,name);
02404     else {
02405         MesPrint("&There is already a set with the name %s",name);
02406     }
02407     return(1);
02408 }
02409 if ( c == 0 ) {
02410     numberofset = AddSet(name,0);
02411     set = Sets + numberofset;
02412     return(0); /* empty set */
02413 }
02414 *s = c; ss = s; /* ss marks the end of the name */
02415 if ( *s == '(' ) {
02416     UBYTE *sss, cc;
02417     s++; sss = s; /* Beginning of option */
02418     while ( *s != ',' && *s != ')' && *s ) s++;
02419     cc = *s; *s = 0;
02420     if ( StrICont(sss,(UBYTE *)"ordered") == 0 ) {
02421         ordered = ORDEREDSET;
02422     }
02423     else {
02424         MesPrint("&Error: Illegal option in set definition: %s",sss);
02425         error = 1;
02426     }
02427     *s = cc;
02428     if ( *s != ')' ) {
02429         MesPrint("&Error: Currently only one option allowed in set definition.");
02430         error = 1;
02431         while ( *s && *s != ')' ) s++;
02432     }
02433     s++;
02434 }
02435 if ( *s == '{' ) {
02436     s++;
02437     if ( ( *s == 'd' || *s == 'D' ) && s[1] == '=' ) {
02438         s += 2;
02439         if ( *s == '-' || *s == '+' || FG.cTable[*s] == 1 ) {
02440             ParseSignedNumber(dim,s)
02441             if ( dim < -HALFMAX || dim > HALFMAX ) {
02442                 MesPrint("&Warning: dimension of %s (%d) out of range",
02443                     name,dim);
02444             }
02445         }
02446         if ( *s != ')' ) goto IllDim;
02447         else s++;
02448     }
02449     else {
02450     IllDim: MesPrint("&Error: Illegal dimension field for set %s",name);
02451         error = 1;
02452         s = SkipField(s,0);
02453     }
02454     while ( *s == ',' ) s++;
02455 }
02456 c = *ss; *ss = 0;
02457 numberofset = AddSet(name,dim);
02458 *ss = c;
02459 set = Sets + numberofset;
02460 set->flags |= ordered;
02461 if ( *s != ':' ) {
02462     MesPrint("&Proper syntax is `Set name:elements'");
02463     return(1);
02464 }
02465 s++;

```



```

02466     error = DoElements(s,set,name);
02467     AC.SetList.numtemp = AC.SetList.num;
02468     AC.SetElementList.numtemp = AC.SetElementList.num;
02469     return(error);
02470 }
02471
02472 /*
02473  #] CoSet :
02474  #[ DoTempSet :
02475
02476      Gets a {} set definition and returns a set number if the set is
02477      properly structured. This number refers either to an already
02478      existing set, or to a set that is defined here.
02479      From and to refer to the contents. They exclude the {}.
02480 */
02481
02482 int DoTempSet(UBYTE *from, UBYTE *to)
02483 {
02484     int i, num, j, sgn;
02485     WORD *e, *ep;
02486     UBYTE c;
02487     int setnum = AddSet(0,MAXPOSITIVE);
02488     SETS set = Sets + setnum, setp;
02489     set->name = -1;
02490     set->type = -1;
02491     c = *to; *to = 0;
02492     AC.wildflag = 0;
02493     while ( *from == ',' ) from++;
02494     if ( *from == '<' || *from == '>' ) {
02495         set->type = CRANGE;
02496         set->first = 3*MAXPOWER;
02497         set->last = -3*MAXPOWER;
02498         while ( *from == '<' || *from == '>' ) {
02499             if ( *from == '<' ) {
02500                 j = 1; from++;
02501                 if ( *from == '=' ) { from++; j++; }
02502             }
02503             else {
02504                 j = -1; from++;
02505                 if ( *from == '=' ) { from++; j--; }
02506             }
02507             sgn = 1;
02508             while ( *from == '-' || *from == '+' ) {
02509                 if ( *from == '-' ) sgn = -sgn;
02510                 from++;
02511             }
02512             ParseNumber(num,from)
02513             if ( *from && *from != ',' ) {
02514                 MesPrint("&Illegal number in ranged set definition");
02515                 return(-1);
02516             }
02517             if ( sgn < 0 ) num = -num;
02518             if ( num >= MAXPOWER || num <= -MAXPOWER ) {
02519                 Warning("Value in ranged set too big. Adjusted to infinity.");
02520                 if ( num > 0 ) num = 3*MAXPOWER;
02521                 else num = -3*MAXPOWER;
02522             }
02523             else if ( j == 2 ) num += 2*MAXPOWER;
02524             else if ( j == -2 ) num -= 2*MAXPOWER;
02525             if ( j > 0 ) set->first = num;
02526             else set->last = num;
02527             while ( *from == ',' ) from++;
02528         }
02529         if ( *from ) {
02530             MesPrint("&Definition of ranged set contains illegal objects");
02531             return(-1);
02532         }
02533     }
02534     else if ( DoElements(from,set,(UBYTE *)0) != 0 ) {
02535         AC.SetElementList.num = set->first;
02536         AC.SetList.num--; *to = c;
02537         return(-1);
02538     }
02539     *to = c;
02540 /*
02541     Now we have to test whether this set exists already.
02542 */
02543     num = set->last - set->first;
02544     for ( setp = Sets, i = 0; i < AC.SetList.num-1; i++, setp++ ) {
02545         if ( num != setp->last - setp->first ) continue;
02546         if ( set->type != setp->type ) continue;
02547         if ( set->type == CRANGE ) {
02548             if ( set->first == setp->first ) return(setp-Sets);
02549         }
02550         else {
02551             e = SetElements + set->first;
02552             ep = SetElements + setp->first;

```

```

02553         j = num;
02554         while ( --j >= 0 ) if ( *ep++ != *ep++ ) break;
02555         if ( j < 0 ) {
02556             AC.SetElementList.num = set->first;
02557             AC.SetList.num--;
02558             return(setp - Sets);
02559         }
02560     }
02561 }
02562 return(setnum);
02563 }
02564
02565 /*
02566 #] DoTempSet :
02567 #[ CoAuto :
02568
02569 To prepare first:
02570     Use of the proper pointers in the various declaration routines
02571     Proper action in .store and .clear
02572 */
02573
02574 int CoAuto(UBYTE *inp)
02575 {
02576     int retval;
02577
02578     AC.Symbols = &(AC.AutoSymbolList);
02579     AC.Vectors = &(AC.AutoVectorList);
02580     AC.Indices = &(AC.AutoIndexList);
02581     AC.Functions = &(AC.AutoFunctionList);
02582     AC.activenames = &(AC.autonames);
02583     AC.AutoDeclareFlag = WITHAUTO;
02584
02585     while ( *inp == ',' ) inp++;
02586     retval = CompileStatement(inp);
02587
02588     AC.AutoDeclareFlag = 0;
02589     AC.Symbols = &(AC.SymbolList);
02590     AC.Vectors = &(AC.VectorList);
02591     AC.Indices = &(AC.IndexList);
02592     AC.Functions = &(AC.FunctionList);
02593     AC.activenames = &(AC.varnames);
02594     return(retval);
02595 }
02596
02597 /*
02598 #] CoAuto :
02599 #[ AddDollar :
02600
02601 The actual addition. Special routine for additions 'on the fly'
02602 */
02603
02604 int AddDollar(UBYTE *name, WORD type, WORD *start, LONG size)
02605 {
02606     int nodenum, numdollar = AP.DollarList.num;
02607     WORD *s, *t;
02608     DOLLARS dol = (DOLLARS)FromVarList(&AP.DollarList);
02609     dol->name = AddName(AC.dollarnames, name, CDOLLAR, numdollar, &nodenum);
02610     dol->type = type;
02611     dol->node = nodenum;
02612     dol->zero = 0;
02613     dol->numdummies = 0;
02614 #ifdef WITHPTHREADS
02615     dol->pthreadslckread = dummylock;
02616     dol->pthreadslckwrite = dummylock;
02617 #endif
02618     dol->nfactors = 0;
02619     dol->factors = 0;
02620     AddRHS(AM.dbufnum, 1);
02621     AddLHS(AM.dbufnum);
02622     if ( start && size > 0 ) {
02623         dol->size = size;
02624         dol->where =
02625             s = (WORD *)Malloc1((size+1)*sizeof(WORD), "$-variable contents");
02626         t = start;
02627         while ( --size >= 0 ) *s++ = *t++;
02628         *s = 0;
02629     }
02630     else { dol->where = &(AM.dollarzero); dol->size = 0; }
02631     cbuf[AM.dbufnum].rhs[numdollar] = dol->where;
02632     cbuf[AM.dbufnum].CanCommu[numdollar] = 0;
02633     cbuf[AM.dbufnum].NumTerms[numdollar] = 0;
02634
02635     return(numdollar);
02636 }
02637
02638 /*
02639 #] AddDollar :

```

```

02640     #[ ReplaceDollar :
02641
02642     Replacements of dollar variables can happen at any time.
02643     For debugging purposes we should have a tracing facility.
02644
02645     Not in use????
02646 */
02647
02648 int ReplaceDollar(WORD number, WORD newtype, WORD *newstart, LONG newsize)
02649 {
02650     int error = 0;
02651     DOLLARS dol = Dollars + number;
02652     WORD *s, *t;
02653     LONG i;
02654     dol->type = newtype;
02655     if ( dol->size == newsize && newsize > 0 && newstart ) {
02656         s = dol->where; t = newstart; i = newsize;
02657         while ( --i >= 0 ) { if ( *s++ != *t++ ) break; }
02658         if ( i < 0 ) return(0);
02659     }
02660     if ( dol->where && dol->where != &(dol->zero) ) {
02661         M_free(dol->where,"dollar->where"); dol->where = &(dol->zero); dol->size = 0;
02662     }
02663     if ( newstart && newsize > 0 ) {
02664         dol->size = newsize;
02665         dol->where =
02666         s = (WORD *)Malloc1((newsize+1)*sizeof(WORD),"$-variable contents");
02667         t = newstart; i = newsize;
02668         while ( --i >= 0 ) *s++ = *t++;
02669         *s = 0;
02670     }
02671     return(error);
02672 }
02673
02674 /*
02675     #[ ReplaceDollar :
02676     #[ AddDubious :
02677
02678     This adds a variable of which we do not know the proper type.
02679 */
02680
02681 int AddDubious(UBYTE *name)
02682 {
02683     int nodenum, numdubious = AC.DubiousList.num;
02684     DUBIOUSV dub = (DUBIOUSV)FromVarList(&AC.DubiousList);
02685     dub->name = AddName(AC.varnames,name,CDUBIOUS,numdubious,&nodenum);
02686     dub->node = nodenum;
02687     return(numdubious);
02688 }
02689
02690 /*
02691     #[ AddDubious :
02692     #[ MakeDubious :
02693 */
02694
02695 int MakeDubious(NAMETREE *nametree, UBYTE *name, WORD *number)
02696 {
02697     NAMENODE *n;
02698     int node, newnode, i;
02699     if ( nametree->namenode == 0 ) return(-1);
02700     newnode = nametree->headnode;
02701     do {
02702         node = newnode;
02703         n = nametree->namenode+node;
02704         if ( ( i = StrCmp(name,nametree->namebuffer+n->name) ) < 0 )
02705             newnode = n->left;
02706         else if ( i > 0 ) newnode = n->right;
02707         else {
02708             if ( n->type != CDUBIOUS ) {
02709                 int numdubious = AC.DubiousList.num;
02710                 FUNCTIONS dub = (FUNCTIONS)FromVarList(&AC.DubiousList);
02711                 dub->name = n->name;
02712                 n->number = numdubious;
02713             }
02714             *number = n->number;
02715             return(CDUBIOUS);
02716         }
02717     } while ( newnode >= 0 );
02718     return(-1);
02719 }
02720
02721 /*
02722     #[ MakeDubious :
02723     #[ NameConflict :
02724 */
02725
02726 static char *nametype[] = { "symbol", "index", "vector", "function",

```

```

02727         "set", "expression" );
02728 static char *plural[] = { "", "n", "", "", "", "n" };
02729
02730 int NameConflict(int type, UBYTE *name)
02731 {
02732     if ( type == NAMENOTFOUND ) {
02733         MesPrint("%s has not been declared",name);
02734     }
02735     else if ( type != CDUBIOUS )
02736         MesPrint("%s has been declared as a%s %s already"
02737             ,name,plural[type],nametype[type]);
02738     return(1);
02739 }
02740
02741 /*
02742 #] NameConflict :
02743 #[ AddExpression :
02744 */
02745
02746 int AddExpression(UBYTE *name, int x, int y)
02747 {
02748     int nodenum, numexpr = AC.ExpressionList.num;
02749     EXPRESSIONS expr = (EXPRESSIONS)FromVarList(&AC.ExpressionList);
02750     UBYTE *s;
02751     expr->status = x;
02752     expr->printflag = y;
02753     PUTZERO(expr->onfile);
02754     PUTZERO(expr->size);
02755     expr->renum = 0;
02756     expr->renumlists = 0;
02757     expr->hidelevel = 0;
02758     expr->inmem = 0;
02759     expr->bracketinfo = expr->newbracketinfo = 0;
02760     if ( name ) {
02761         expr->name = AddName(AC.exprnames,name,CEXPRESSION,numexpr,&nodenum);
02762         expr->node = nodenum;
02763         expr->replace = NEWLYDEFINEDEXPRESSION ;
02764         s = name;
02765         while ( *s ) s++;
02766         expr->namesize = (s-name)+1;
02767     }
02768     else {
02769         expr->replace = REDEFINEDEXPRESSION;
02770         expr->name = AC.TransEname;
02771         expr->node = -1;
02772         expr->namesize = 0;
02773     }
02774     expr->vflags = 0;
02775     expr->numdummies = 0;
02776     expr->numfactors = 0;
02777     #ifdef PARALLELCODE
02778     expr->partodo = 0;
02779     #endif
02780     expr->uflags = 0;
02781     return(numexpr);
02782 }
02783
02784 /*
02785 #] AddExpression :
02786 #[ GetLabel :
02787 */
02788
02789 int GetLabel(UBYTE *name)
02790 {
02791     int i;
02792     LONG newnum;
02793     UBYTE **NewLabelNames;
02794     int *NewLabel;
02795     for ( i = 0; i < AC.NumLabels; i++ ) {
02796         if ( StrCmp(name,AC.LabelNames[i]) == 0 ) return(i);
02797     }
02798     if ( AC.NumLabels >= AC.MaxLabels ) {
02799         newnum = 2*AC.MaxLabels;
02800         if ( newnum == 0 ) newnum = 10;
02801         if ( newnum > 32765 ) newnum = 32765;
02802         if ( newnum == AC.MaxLabels ) {
02803             MesPrint("&More than 32765 labels in one module. Please simplify.");
02804             Terminate(-1);
02805         }
02806         NewLabelNames = (UBYTE **)Malloc1((sizeof(UBYTE *)+sizeof(int))
02807             *newnum,"Labels");
02808         NewLabel = (int *) (NewLabelNames+newnum);
02809         for ( i = 0; i < AC.MaxLabels; i++ ) {
02810             NewLabelNames[i] = AC.LabelNames[i];
02811             NewLabel[i] = AC.Labels[i];
02812         }
02813         if ( AC.LabelNames ) M_free(AC.LabelNames,"Labels");

```

```

02814     AC.LabelNames = NewLabelNames;
02815     AC.Labels = NewLabel;
02816     AC.MaxLabels = newnum;
02817 }
02818 i = AC.NumLabels++;
02819 AC.LabelNames[i] = strDup1(name,"Labels");
02820 AC.Labels[i] = -1;
02821 return(i);
02822 }
02823
02824 /*
02825 #] GetLabel :
02826 #[ ResetVariables :
02827
02828 Resets the variables.
02829 par = 0 The list of temporary sets (after each .sort)
02830 par = 1 The list of local variables (after each .store)
02831 par = 2 All variables (after each .clear)
02832 */
02833
02834 void ResetVariables(int par)
02835 {
02836     int i, j;
02837     TABLES T;
02838     switch ( par ) {
02839     case 0 : /* Only the sets without a name */
02840         AC.SetList.num = AC.SetList.numtemp;
02841         AC.SetElementList.num = AC.SetElementList.numtemp;
02842         break;
02843     case 2 :
02844         for ( i = AC.SymbolList.numclear; i < AC.SymbolList.num; i++ )
02845             AC.varnames->namenode[symbols[i].node].type = CDELETE;
02846         AC.SymbolList.num = AC.SymbolList.numglobal = AC.SymbolList.numclear;
02847         for ( i = AC.VectorList.numclear; i < AC.VectorList.num; i++ )
02848             AC.varnames->namenode[vectors[i].node].type = CDELETE;
02849         AC.VectorList.num = AC.VectorList.numglobal = AC.VectorList.numclear;
02850         for ( i = AC.IndexList.numclear; i < AC.IndexList.num; i++ )
02851             AC.varnames->namenode[indices[i].node].type = CDELETE;
02852         AC.IndexList.num = AC.IndexList.numglobal = AC.IndexList.numclear;
02853         for ( i = AC.FunctionList.numclear; i < AC.FunctionList.num; i++ ) {
02854             AC.varnames->namenode[functions[i].node].type = CDELETE;
02855             if ( ( T = functions[i].tabl ) != 0 ) {
02856                 if ( T->tablepointers ) M_free(T->tablepointers,"tablepointers");
02857                 if ( T->prototype ) M_free(T->prototype,"tableprototype");
02858                 if ( T->mm ) M_free(T->mm,"tableminmax");
02859                 if ( T->flags ) M_free(T->flags,"tableflags");
02860                 if ( T->argtail ) M_free(T->argtail,"table arguments");
02861                 if ( T->boomlijst ) M_free(T->boomlijst,"TableTree");
02862                 for ( j = 0; j < T->buffersfill; j++ ) { /* was <= */
02863                     finishcbuf(T->buffers[j]);
02864                 }
02865                 /*[07apr2004 mt]:*/ /*memory leak*/
02866                 if ( T->buffers ) M_free(T->buffers,"Table buffers");
02867                 /*:[07apr2004 mt]*/
02868                 finishcbuf(T->bufnum);
02869                 if ( T->spare ) {
02870                     TABLES TT = T->spare;
02871                     if ( TT->mm ) M_free(TT->mm,"tableminmax");
02872                     if ( TT->flags ) M_free(TT->flags,"tableflags");
02873                     if ( TT->tablepointers ) M_free(TT->tablepointers,"tablepointers");
02874                     for ( j = 0; j < TT->buffersfill; j++ ) { /* was <= */
02875                         finishcbuf(TT->buffers[j]);
02876                     }
02877                     if ( TT->boomlijst ) M_free(TT->boomlijst,"TableTree");
02878                     /*[07apr2004 mt]:*/ /*memory leak*/
02879                     if ( TT->buffers ) M_free(TT->buffers,"Table buffers");
02880                     /*:[07apr2004 mt]*/
02881                     M_free(TT,"table");
02882                 }
02883                 M_free(T,"table");
02884             }
02885         }
02886         AC.FunctionList.num = AC.FunctionList.numglobal = AC.FunctionList.numclear;
02887         for ( i = AC.SetList.numclear; i < AC.SetList.num; i++ ) {
02888             if ( Sets[i].node >= 0 )
02889                 AC.varnames->namenode[Sets[i].node].type = CDELETE;
02890         }
02891         AC.SetList.numtemp = AC.SetList.num = AC.SetList.numglobal = AC.SetList.numclear;
02892         for ( i = AC.DubiousList.numclear; i < AC.DubiousList.num; i++ )
02893             AC.varnames->namenode[Dubious[i].node].type = CDELETE;
02894         AC.DubiousList.num = AC.DubiousList.numglobal = AC.DubiousList.numclear;
02895         AC.SetElementList.numtemp = AC.SetElementList.num =
02896             AC.SetElementList.numglobal = AC.SetElementList.numclear;
02897         CompactifyTree(AC.varnames,VARNAMES);
02898         AC.varnames->namefill = AC.varnames->globalnamefill = AC.varnames->clearnamefill;
02899         AC.varnames->nodefill = AC.varnames->globalnodefill = AC.varnames->clearnodefill;
02900

```

```

02901     for ( i = AC.AutoSymbolList.numclear; i < AC.AutoSymbolList.num; i++ )
02902         AC.autonames->namenode[
02903             ((SYMBOLS) (AC.AutoSymbolList.lijst))[i].node].type = CDELETE;
02904     AC.AutoSymbolList.num = AC.AutoSymbolList.numglobal
02905         = AC.AutoSymbolList.numclear;
02906     for ( i = AC.AutoVectorList.numclear; i < AC.AutoVectorList.num; i++ )
02907         AC.autonames->namenode[
02908             ((VECTORS) (AC.AutoVectorList.lijst))[i].node].type = CDELETE;
02909     AC.AutoVectorList.num = AC.AutoVectorList.numglobal
02910         = AC.AutoVectorList.numclear;
02911     for ( i = AC.AutoIndexList.numclear; i < AC.AutoIndexList.num; i++ )
02912         AC.autonames->namenode[
02913             ((INDICES) (AC.AutoIndexList.lijst))[i].node].type = CDELETE;
02914     AC.AutoIndexList.num = AC.AutoIndexList.numglobal
02915         = AC.AutoIndexList.numclear;
02916     for ( i = AC.AutoFunctionList.numclear; i < AC.AutoFunctionList.num; i++ ) {
02917         AC.autonames->namenode[
02918             ((FUNCTIONS) (AC.AutoFunctionList.lijst))[i].node].type = CDELETE;
02919         if ( ( T = ((FUNCTIONS) (AC.AutoFunctionList.lijst))[i].tabl ) != 0 ) {
02920             if ( T->tablepointers ) M_free(T->tablepointers, "tablepointers");
02921             if ( T->prototype ) M_free(T->prototype, "tableprototype");
02922             if ( T->mm ) M_free(T->mm, "tableminmax");
02923             if ( T->flags ) M_free(T->flags, "tableflags");
02924             if ( T->argtail ) M_free(T->argtail, "table arguments");
02925             if ( T->boomlijst ) M_free(T->boomlijst, "TableTree");
02926             for ( j = 0; j < T->buffersfill; j++ ) { /* was <= */
02927                 finishcbuf(T->buffers[j]);
02928             }
02929             if ( T->spare ) {
02930                 TABLES TT = T->spare;
02931                 if ( TT->mm ) M_free(TT->mm, "tableminmax");
02932                 if ( TT->flags ) M_free(TT->flags, "tableflags");
02933                 if ( TT->tablepointers ) M_free(TT->tablepointers, "tablepointers");
02934                 for ( j = 0; j < TT->buffersfill; j++ ) { /* was <= */
02935                     finishcbuf(TT->buffers[j]);
02936                 }
02937                 if ( TT->boomlijst ) M_free(TT->boomlijst, "TableTree");
02938                 M_free(TT, "table");
02939             }
02940             M_free(T, "table");
02941         }
02942     }
02943     AC.AutoFunctionList.num = AC.AutoFunctionList.numglobal
02944         = AC.AutoFunctionList.numclear;
02945     CompactifyTree(AC.autonames, AUTONAMES);
02946     AC.autonames->namefill = AC.autonames->globalnamefill
02947         = AC.autonames->clearnamefill;
02948     AC.autonames->nodefill = AC.autonames->globalnodefill
02949         = AC.autonames->clearnodefill;
02950     ReleaseTB();
02951     break;
02952 case 1 :
02953     for ( i = AC.SymbolList.numglobal; i < AC.SymbolList.num; i++ )
02954         AC.varnames->namenode[symbols[i].node].type = CDELETE;
02955     AC.SymbolList.num = AC.SymbolList.numglobal;
02956     for ( i = AC.VectorList.numglobal; i < AC.VectorList.num; i++ )
02957         AC.varnames->namenode[vectors[i].node].type = CDELETE;
02958     AC.VectorList.num = AC.VectorList.numglobal;
02959     for ( i = AC.IndexList.numglobal; i < AC.IndexList.num; i++ )
02960         AC.varnames->namenode[indices[i].node].type = CDELETE;
02961     AC.IndexList.num = AC.IndexList.numglobal;
02962     for ( i = AC.FunctionList.numglobal; i < AC.FunctionList.num; i++ ) {
02963         AC.varnames->namenode[functions[i].node].type = CDELETE;
02964         if ( ( T = functions[i].tabl ) != 0 ) {
02965             if ( T->tablepointers ) M_free(T->tablepointers, "tablepointers");
02966             if ( T->prototype ) M_free(T->prototype, "tableprototype");
02967             if ( T->mm ) M_free(T->mm, "tableminmax");
02968             if ( T->flags ) M_free(T->flags, "tableflags");
02969             if ( T->argtail ) M_free(T->argtail, "table arguments");
02970             if ( T->boomlijst ) M_free(T->boomlijst, "TableTree");
02971             for ( j = 0; j < T->buffersfill; j++ ) { /* was <= */
02972                 finishcbuf(T->buffers[j]);
02973             }
02974             /*[07apr2004 mt]:*/ /*memory leak*/
02975             if ( T->buffers ) M_free(T->buffers, "Table buffers");
02976             /*:[07apr2004 mt]:*/
02977             finishcbuf(T->bufnum);
02978             if ( T->spare ) {
02979                 TABLES TT = T->spare;
02980                 if ( TT->mm ) M_free(TT->mm, "tableminmax");
02981                 if ( TT->flags ) M_free(TT->flags, "tableflags");
02982                 if ( TT->tablepointers ) M_free(TT->tablepointers, "tablepointers");
02983                 for ( j = 0; j < TT->buffersfill; j++ ) { /* was <= */
02984                     finishcbuf(TT->buffers[j]);
02985                 }
02986                 if ( TT->boomlijst ) M_free(TT->boomlijst, "TableTree");
02987                 /*[07apr2004 mt]:*/ /*memory leak*/

```

```

02988             if ( TT->buffers ) M_free(TT->buffers,"Table buffers");
02989             /*:[07apr2004 mt]*/
02990             M_free(TT,"table");
02991         }
02992         M_free(T,"table");
02993     }
02994 }
02995 #ifdef TABLECLEANUP
02996 {
02997     int j;
02998     WORD *tp;
02999     for ( i = 0; i < AC.FunctionList.numglobal; i++ ) {
03000 /*
03001         Now, if the table definition is from after the .global
03002         while the function is from before, there is a problem.
03003         This could be resolved by defining CTable (=Table), Ntable
03004         and do away with the previous function definition.
03005 */
03006         if ( ( T = functions[i].tabl ) != 0 ) {
03007 /*
03008             First restore overwritten definitions.
03009 */
03010             if ( T->sparse ) {
03011                 T->totind = T->mdefined;
03012                 for ( j = 0, tp = T->tablepointers; j < T->totind; j++ ) {
03013                     tp += ABS(T->numind);
03014 #if TABLEEXTENSION == 2
03015                     tp[0] = tp[1];
03016 #else
03017                     tp[0] = tp[2];
03018                     tp[1] = tp[3];
03019                     tp[4] = tp[5];
03020 #endif
03021                     tp += TABLEEXTENSION;
03022                 }
03023                 RedoTableTree(T,T->totind);
03024                 if ( T->sparse ) {
03025                     TABLES TT = T->sparse;
03026                     TT->totind = TT->mdefined;
03027                     for ( j = 0, tp = TT->tablepointers; j < TT->totind; j++ ) {
03028                         tp += ABS(TT->numind);
03029 #if TABLEEXTENSION == 2
03030                         tp[0] = tp[1];
03031 #else
03032                         tp[0] = tp[2];
03033                         tp[1] = tp[3];
03034                         tp[4] = tp[5];
03035 #endif
03036                         tp += TABLEEXTENSION;
03037                     }
03038                     RedoTableTree(TT,TT->totind);
03039                     cbuf[TT->bufnum].numlhs = cbuf[TT->bufnum].mnumlhs;
03040                     cbuf[TT->bufnum].numrhs = cbuf[TT->bufnum].mnumrhs;
03041                 }
03042             }
03043             else {
03044                 for ( j = 0, tp = T->tablepointers; j < T->totind; j++ ) {
03045 #if TABLEEXTENSION == 2
03046                     tp[0] = tp[1];
03047 #else
03048                     tp[0] = tp[2];
03049                     tp[1] = tp[3];
03050                     tp[4] = tp[5];
03051 #endif
03052                 }
03053                 T->defined = T->mdefined;
03054             }
03055             cbuf[T->bufnum].numlhs = cbuf[T->bufnum].mnumlhs;
03056             cbuf[T->bufnum].numrhs = cbuf[T->bufnum].mnumrhs;
03057         }
03058     }
03059 }
03060 #endif
03061 AC.FunctionList.num = AC.FunctionList.numglobal;
03062 for ( i = AC.SetList.numglobal; i < AC.SetList.num; i++ ) {
03063     if ( Sets[i].node >= 0 )
03064         AC.varnames->namenode[Sets[i].node].type = CDELETE;
03065 }
03066 AC.SetList.numtemp = AC.SetList.num = AC.SetList.numglobal;
03067 for ( i = AC.DubiousList.numglobal; i < AC.DubiousList.num; i++ )
03068     AC.varnames->namenode[Dubious[i].node].type = CDELETE;
03069 AC.DubiousList.num = AC.DubiousList.numglobal;
03070 AC.SetElementList.numtemp = AC.SetElementList.num =
03071     AC.SetElementList.numglobal;
03072 CompactifyTree(AC.varnames,VARNAMES);
03073 AC.varnames->namefill = AC.varnames->globalnamefill;
03074 AC.varnames->nodefill = AC.varnames->globalnodefill;

```

```

03075
03076     for ( i = AC.AutoSymbolList.numglobal; i < AC.AutoSymbolList.num; i++ )
03077         AC.autonames->namenode[
03078             ((SYMBOLS) (AC.AutoSymbolList.lijst))[i].node].type = CDELETE;
03079     AC.AutoSymbolList.num = AC.AutoSymbolList.numglobal;
03080     for ( i = AC.AutoVectorList.numglobal; i < AC.AutoVectorList.num; i++ )
03081         AC.autonames->namenode[
03082             ((VECTORS) (AC.AutoVectorList.lijst))[i].node].type = CDELETE;
03083     AC.AutoVectorList.num = AC.AutoVectorList.numglobal;
03084     for ( i = AC.AutoIndexList.numglobal; i < AC.AutoIndexList.num; i++ )
03085         AC.autonames->namenode[
03086             ((INDICES) (AC.AutoIndexList.lijst))[i].node].type = CDELETE;
03087     AC.AutoIndexList.num = AC.AutoIndexList.numglobal;
03088     for ( i = AC.AutoFunctionList.numglobal; i < AC.AutoFunctionList.num; i++ ) {
03089         AC.autonames->namenode[
03090             ((FUNCTIONS) (AC.AutoFunctionList.lijst))[i].node].type = CDELETE;
03091         if ( ( T = ((FUNCTIONS) (AC.AutoFunctionList.lijst))[i].tabl ) != 0 ) {
03092             if ( T->tablepointers ) M_free(T->tablepointers, "tablepointers");
03093             if ( T->prototype ) M_free(T->prototype, "tableprototype");
03094             if ( T->mm ) M_free(T->mm, "tableminmax");
03095             if ( T->flags ) M_free(T->flags, "tableflags");
03096             if ( T->argtail ) M_free(T->argtail, "table arguments");
03097             if ( T->boomlijst ) M_free(T->boomlijst, "TableTree");
03098             for ( j = 0; j < T->buffersfill; j++ ) { /* was <= */
03099                 finishcbuf(T->buffers[j]);
03100             }
03101             if ( T->spare ) {
03102                 TABLES TT = T->spare;
03103                 if ( TT->mm ) M_free(TT->mm, "tableminmax");
03104                 if ( TT->flags ) M_free(TT->flags, "tableflags");
03105                 if ( TT->tablepointers ) M_free(TT->tablepointers, "tablepointers");
03106                 for ( j = 0; j < TT->buffersfill; j++ ) { /* was <= */
03107                     finishcbuf(TT->buffers[j]);
03108                 }
03109                 if ( TT->boomlijst ) M_free(TT->boomlijst, "TableTree");
03110                 M_free(TT, "table");
03111             }
03112             M_free(T, "table");
03113         }
03114     }
03115     AC.AutoFunctionList.num = AC.AutoFunctionList.numglobal;
03116
03117     CompactifyTree(AC.autonames, AUTONAMES);
03118
03119     AC.autonames->namefill = AC.autonames->globalnamefill;
03120     AC.autonames->nodefill = AC.autonames->globalnodefill;
03121     break;
03122 }
03123 }
03124
03125 /*
03126 #] ResetVariables :
03127 #[ RemoveDollars :
03128 */
03129
03130 void RemoveDollars(void)
03131 {
03132     DOLLARS d;
03133     CBUF *C = cbuf + AM.dbufnum;
03134     int numdollar = AP.DollarList.num;
03135     if ( numdollar > 0 ) {
03136         while ( numdollar > AM.gcNumDollars ) {
03137             numdollar--;
03138             d = Dollars + numdollar;
03139             if ( d->where && d->where != &(d->zero) && d->where != &(AM.dollarzero) ) {
03140                 M_free(d->where, "dollar->where"); d->where = &(d->zero); d->size = 0;
03141             }
03142             AC.dollarnames->namenode[d->node].type = CDELETE;
03143         }
03144         AP.DollarList.num = AM.gcNumDollars;
03145         CompactifyTree(AC.dollarnames, DOLLARNAMES);
03146
03147         C->numrhs = C->mnumrhs;
03148         C->numlhs = C->mnumlhs;
03149     }
03150 }
03151
03152 /*
03153 #] RemoveDollars :
03154 #[ Globalize :
03155 */
03156
03157 void Globalize(int par)
03158 {
03159     int i, j;
03160     WORD *tp;
03161     if ( par == 1 ) {

```



```

03162     AC.SymbolList.numclear      = AC.SymbolList.num;
03163     AC.VectorList.numclear      = AC.VectorList.num;
03164     AC.IndexList.numclear       = AC.IndexList.num;
03165     AC.FunctionList.numclear    = AC.FunctionList.num;
03166     AC.SetList.numclear         = AC.SetList.num;
03167     AC.DubiousList.numclear     = AC.DubiousList.num;
03168     AC.SetElementList.numclear  = AC.SetElementList.num;
03169     AC.varnames->clearnamefill  = AC.varnames->namefill;
03170     AC.varnames->clearnodefill  = AC.varnames->nodefill;
03171
03172     AC.AutoSymbolList.numclear  = AC.AutoSymbolList.num;
03173     AC.AutoVectorList.numclear  = AC.AutoVectorList.num;
03174     AC.AutoIndexList.numclear   = AC.AutoIndexList.num;
03175     AC.AutoFunctionList.numclear = AC.AutoFunctionList.num;
03176     AC.autonames->clearnamefill = AC.autonames->namefill;
03177     AC.autonames->clearnodefill = AC.autonames->nodefill;
03178 }
03179 /* for ( i = AC.FunctionList.numglobal; i < AC.FunctionList.num; i++ ) { */
03180 for ( i = MAXBUILTINFUNCTION-FUNCTION; i < AC.FunctionList.num; i++ ) {
03181 /*
03182     We need here not only the not-yet-global functions. The already
03183     global ones may have obtained extra elements.
03184 */
03185     if ( functions[i].tabl ) {
03186         TABLES T = functions[i].tabl;
03187         if ( T->sparse ) {
03188             T->mdefined = T->totind;
03189             for ( j = 0, tp = T->tablepointers; j < T->totind; j++ ) {
03190                 tp += ABS(T->numind);
03191 #if TABLEEXTENSION == 2
03192                 tp[1] = tp[0];
03193 #else
03194                 tp[2] = tp[0]; tp[3] = tp[1]; tp[5] = tp[4] & (~ELEMENTUSED);
03195 #endif
03196                 tp += TABLEEXTENSION;
03197             }
03198             if ( T->spare ) {
03199                 TABLES TT = T->sparse;
03200                 TT->mdefined = TT->totind;
03201                 for ( j = 0, tp = TT->tablepointers; j < TT->totind; j++ ) {
03202                     tp += ABS(TT->numind);
03203 #if TABLEEXTENSION == 2
03204                     tp[1] = tp[0];
03205 #else
03206                     tp[2] = tp[0]; tp[3] = tp[1]; tp[5] = tp[4] & (~ELEMENTUSED);
03207 #endif
03208                     tp += TABLEEXTENSION;
03209                 }
03210                 cbuf[TT->bufnum].mnumlhs = cbuf[TT->bufnum].numlhs;
03211                 cbuf[TT->bufnum].mnumrhs = cbuf[TT->bufnum].numrhs;
03212             }
03213         }
03214         else {
03215             T->mdefined = T->defined;
03216             for ( j = 0, tp = T->tablepointers; j < T->totind; j++ ) {
03217 #if TABLEEXTENSION == 2
03218                 tp[1] = tp[0];
03219 #else
03220                 tp[2] = tp[0]; tp[3] = tp[1]; tp[5] = tp[4] & (~ELEMENTUSED);
03221 #endif
03222             }
03223             cbuf[T->bufnum].mnumlhs = cbuf[T->bufnum].numlhs;
03224             cbuf[T->bufnum].mnumrhs = cbuf[T->bufnum].numrhs;
03225         }
03226     }
03227 }
03228 for ( i = AC.AutoFunctionList.numglobal; i < AC.AutoFunctionList.num; i++ ) {
03229     if ( ((FUNCTIONS) (AC.AutoFunctionList.lijst))[i].tabl )
03230         ((FUNCTIONS) (AC.AutoFunctionList.lijst))[i].tabl->mdefined =
03231             ((FUNCTIONS) (AC.AutoFunctionList.lijst))[i].tabl->defined;
03232 }
03233 AC.SymbolList.numglobal      = AC.SymbolList.num;
03234 AC.VectorList.numglobal      = AC.VectorList.num;
03235 AC.IndexList.numglobal       = AC.IndexList.num;
03236 AC.FunctionList.numglobal    = AC.FunctionList.num;
03237 AC.SetList.numglobal         = AC.SetList.num;
03238 AC.DubiousList.numglobal     = AC.DubiousList.num;
03239 AC.SetElementList.numglobal  = AC.SetElementList.num;
03240 AC.varnames->globalnamefill  = AC.varnames->namefill;
03241 AC.varnames->globalnodefill  = AC.varnames->nodefill;
03242
03243 AC.AutoSymbolList.numglobal  = AC.AutoSymbolList.num;
03244 AC.AutoVectorList.numglobal  = AC.AutoVectorList.num;
03245 AC.AutoIndexList.numglobal   = AC.AutoIndexList.num;
03246 AC.AutoFunctionList.numglobal = AC.AutoFunctionList.num;
03247 AC.autonames->globalnamefill = AC.autonames->namefill;
03248 AC.autonames->globalnodefill = AC.autonames->nodefill;

```

```

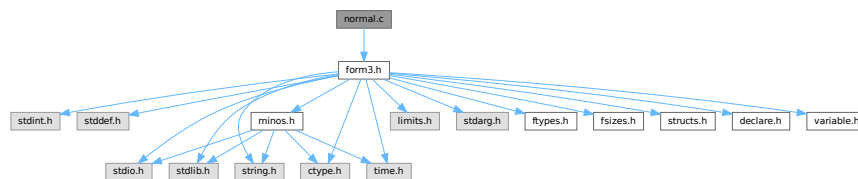
03249 }
03250
03251 /*
03252 #] Globalize :
03253 #[ TestName :
03254 */
03255
03256 int TestName(UBYTE *name)
03257 {
03258     if ( *name == '[' ) {
03259         while ( *name ) name++;
03260         if ( name[-1] == ']' ) return(0);
03261         return(-1);
03262     }
03263     while ( *name ) {
03264         if ( *name == '_' ) return(-1);
03265         name++;
03266     }
03267     return(0);
03268 }
03269
03270 /*
03271 #] TestName :
03272 */

```

4.87 normal.c File Reference

```
#include "form3.h"
```

Include dependency graph for normal.c:



Macros

- `#define` [MAXNUMBEROFNONCOMTERMS](#) 2

Functions

- int [CompareFunctions](#) (WORD *fleft, WORD *fright)
- int [Commute](#) (WORD *fleft, WORD *fright)
- int [Normalize](#) (PHEAD WORD *term)
- int [ExtraSymbol](#) (WORD sym, WORD pow, WORD nsym, WORD *ppsym, WORD *ncoef)
- int [DoTheta](#) (PHEAD WORD *t)
- int [DoDelta](#) (WORD *t)
- void [DoRevert](#) (WORD *fun, WORD *tmp)
- WORD [DetCommu](#) (WORD *terms)
- WORD [DoesCommu](#) (WORD *term)
- int [TreatPolyRatFun](#) (PHEAD WORD *prf)
- void [DropCoefficient](#) (PHEAD WORD *term)
- void [DropSymbols](#) (PHEAD WORD *term)
- int [SymbolNormalize](#) (WORD *term)
- int [TestFunFlag](#) (PHEAD WORD *tfun)
- int [BracketNormalize](#) (PHEAD WORD *term)

4.87.1 Detailed Description

Mainly the routine Normalize. This routine brings terms to standard FORM. Currently it has one serious drawback. Its buffers are all in the stack. This means these buffers have a fixed size (NORMSIZE). In the past this has caused problems and NORMSIZE had to be increased.

It is not clear whether Normalize can be called recursively.

Definition in file [normal.c](#).

4.87.2 Macro Definition Documentation

4.87.2.1 MAXNUMBEROFNONCOMTERMS

```
#define MAXNUMBEROFNONCOMTERMS 2
```

Definition at line [4528](#) of file [normal.c](#).

4.87.3 Function Documentation

4.87.3.1 CompareFunctions()

```
int CompareFunctions (
    WORD * fleft,
    WORD * fright )
```

Definition at line [54](#) of file [normal.c](#).

4.87.3.2 Commute()

```
int Commute (
    WORD * fleft,
    WORD * fright )
```

Definition at line [118](#) of file [normal.c](#).

4.87.3.3 Normalize()

```
int Normalize (
    PHEAD WORD * term )
```

Definition at line [193](#) of file [normal.c](#).

4.87.3.4 ExtraSymbol()

```
int ExtraSymbol (
    WORD sym,
    WORD pow,
    WORD nsym,
    WORD * ppsym,
    WORD * ncoef )
```

Definition at line [4216](#) of file [normal.c](#).

4.87.3.5 DoTheta()

```
int DoTheta (
    PHEAD WORD * t )
```

Definition at line [4278](#) of file [normal.c](#).

4.87.3.6 DoDelta()

```
int DoDelta (
    WORD * t )
```

Definition at line [4375](#) of file [normal.c](#).

4.87.3.7 DoRevert()

```
void DoRevert (
    WORD * fun,
    WORD * tmp )
```

Definition at line [4442](#) of file [normal.c](#).

4.87.3.8 DetCommu()

```
WORD DetCommu (
    WORD * terms )
```

Definition at line [4530](#) of file [normal.c](#).

4.87.3.9 DoesCommu()

```
WORD DoesCommu (
    WORD * term )
```

Definition at line [4589](#) of file [normal.c](#).

4.87.3.10 TreatPolyRatFun()

```
int TreatPolyRatFun (
    PHEAD WORD * prf )
```

Definition at line [4980](#) of file [normal.c](#).

4.87.3.11 DropCoefficient()

```
void DropCoefficient (
    PHEAD WORD * term )
```

Definition at line [5116](#) of file [normal.c](#).

4.87.3.12 DropSymbols()

```
void DropSymbols (
    PHEAD WORD * term )
```

Definition at line [5134](#) of file [normal.c](#).

4.87.3.13 SymbolNormalize()

```
int SymbolNormalize (
    WORD * term )
```

Routine normalizes terms that contain only symbols. Regular minimum and maximum properties are ignored.

We check whether there are negative powers in the output. This is not allowed.

Definition at line [5164](#) of file [normal.c](#).

Referenced by [InFunction\(\)](#), and [poly_sort\(\)](#).

4.87.3.14 TestFunFlag()

```
int TestFunFlag (
    PHEAD WORD * tfun )
```

Definition at line [5260](#) of file [normal.c](#).

4.87.3.15 BracketNormalize()

```
int BracketNormalize (
    PHEAD WORD * term )
```

Definition at line [5291](#) of file [normal.c](#).

4.88 normal.c

[Go to the documentation of this file.](#)

```

00001
00010 /* #[] License : */
00011 /*
00012 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00013 *   When using this file you are requested to refer to the publication
00014 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00015 *   This is considered a matter of courtesy as the development was paid
00016 *   for by FOM the Dutch physics granting agency and we would like to
00017 *   be able to track its scientific use to convince FOM of its value
00018 *   for the community.
00019 *
00020 *   This file is part of FORM.
00021 *
00022 *   FORM is free software: you can redistribute it and/or modify it under the
00023 *   terms of the GNU General Public License as published by the Free Software
00024 *   Foundation, either version 3 of the License, or (at your option) any later
00025 *   version.
00026 *
00027 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00028 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00029 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00030 *   details.
00031 *
00032 *   You should have received a copy of the GNU General Public License along
00033 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00034 */
00035 /* #[] License : */
00036 /*
00037 *   #[] Includes : normal.c
00038 */
00039
00040 #include "form3.h"
00041 #ifdef WITHFLOAT
00042 #include <gmp.h>
00043
00044 int PackFloat(WORD *,mpf_t);
00045 int UnpackFloat(mpf_t, WORD *);
00046 void RatToFloat(mpf_t result, UWORD *format, int ratsize);
00047 #endif
00048 /*
00049 *   #[] Includes :
00050 *   #[] Normalize :
00051 *   #[] CompareFunctions :
00052 */
00053
00054 int CompareFunctions(WORD *fleft,WORD *fright)
00055 {
00056     WORD k, kk;
00057     if ( AC.properorderflag ) {
00058         if ( ( *fleft >= (FUNCTION+WILDOFFSET)
00059             && functions[*fleft-FUNCTION-WILDOFFSET].spec >= TENSORFUNCTION )
00060             || ( *fleft >= FUNCTION && *fleft < (FUNCTION + WILDOFFSET)
00061             && functions[*fleft-FUNCTION].spec >= TENSORFUNCTION ) ) {}
00062         else {
00063             WORD *s1, *s2, *ss1, *ss2;
00064             s1 = fleft+FUNHEAD; s2 = fright+FUNHEAD;
00065             ss1 = fleft + fleft[1]; ss2 = fright + fright[1];
00066             while ( s1 < ss1 && s2 < ss2 ) {
00067                 k = CompArg(s1,s2);
00068                 if ( k > 0 ) return(1);
00069                 if ( k < 0 ) return(0);
00070                 NEXTARG(s1)
00071                 NEXTARG(s2)
00072             }
00073             if ( s1 < ss1 ) return(1);
00074             return(0);
00075         }
00076         k = fleft[1] - FUNHEAD;
00077         kk = fright[1] - FUNHEAD;
00078         fleft += FUNHEAD;
00079         fright += FUNHEAD;
00080         while ( k > 0 && kk > 0 ) {
00081             if ( *fleft < *fright ) return(0);
00082             else if ( *fleft++ > *fright++ ) return(1);
00083             k--; kk--;
00084         }
00085         if ( k > 0 ) return(1);
00086         return(0);
00087     }
00088     else {
00089         k = fleft[1] - FUNHEAD;
00090         kk = fright[1] - FUNHEAD;

```

```

00091     fleft += FUNHEAD;
00092     fright += FUNHEAD;
00093     while ( k > 0 && kk > 0 ) {
00094         if ( *fleft < *fright ) return(0);
00095         else if ( *fleft++ > *fright++ ) return(1);
00096         k--; kk--;
00097     }
00098     if ( k > 0 ) return(1);
00099     return(0);
00100 }
00101 }
00102
00103 /*
00104     #] CompareFunctions :
00105     #[ Commute :
00106
00107     This function gets two adjacent function pointers and decides
00108     whether these two functions should be exchanged to obtain a
00109     natural ordering.
00110
00111     Currently there is only an ordering of gamma matrices belonging
00112     to different spin lines.
00113
00114     Note that we skip for now the cases of (F)^(3/2) or 1/F and a few more
00115     of such funny functions.
00116 */
00117
00118 int Commute(WORD *fleft, WORD *fright)
00119 {
00120     WORD fun1, fun2;
00121     if ( *fleft == DOLLAREXPRESSION || *fright == DOLLAREXPRESSION ) return(0);
00122     fun1 = ABS(*fleft); fun2 = ABS(*fright);
00123     if ( *fleft >= GAMMA && *fleft <= GAMMASEVEN
00124         && *fright >= GAMMA && *fright <= GAMMASEVEN ) {
00125         if ( fleft[FUNHEAD] < AM.OffsetIndex && fleft[FUNHEAD] > fright[FUNHEAD] )
00126             return(1);
00127         return(0);
00128     }
00129     if ( fun1 >= WILDOFFSET ) fun1 -= WILDOFFSET;
00130     if ( fun2 >= WILDOFFSET ) fun2 -= WILDOFFSET;
00131     if ( ( ( functions[fun1-FUNCTION].flags & COULDCOMMUTE ) == 0 )
00132         || ( ( functions[fun2-FUNCTION].flags & COULDCOMMUTE ) == 0 ) ) return(0);
00133 /*
00134     if other conditions will come here, keep in mind that if *fleft < 0
00135     or *fright < 0 they are arguments in the exponent function as in f^(3/2)
00136 */
00137     if ( AC.CommuteInSet == 0 ) return(0);
00138 /*
00139     The code for CompareFunctions can be stolen from the commuting case.
00140
00141     We need the syntax:
00142         Commute Fun1,Fun2,...,Fun'n';
00143     For this Fun1,...,Fun'n' need to be noncommuting functions.
00144     These functions will commute with all members of the group.
00145     In the AC.paircommute buffer the representation is
00146         'n'+1,element1,...,element'n', 'm'+1,element1,...,element'm',0
00147     A function can belong to more than one group.
00148     If a function commutes with itself, it is most efficient to make a separate
00149     group of two elements for it as in
00150         Commute T,T;    -> 3,T,T
00151 */
00152     if ( fun1 >= fun2 ) {
00153         WORD *group = AC.CommuteInSet, *g1, *g2, *g3;
00154         while ( *group > 0 ) {
00155             g3 = group + *group;
00156             g1 = group+1;
00157             while ( g1 < g3 ) {
00158                 if ( *g1 == fun1 || ( fun1 <= GAMMASEVEN && fun1 >= GAMMA
00159                     && *g1 <= GAMMASEVEN && *g1 >= GAMMA ) ) {
00160                     g2 = group+1;
00161                     while ( g2 < g3 ) {
00162                         if ( g1 != g2 && ( *g2 == fun2 ||
00163                             ( fun2 <= GAMMASEVEN && fun2 >= GAMMA
00164                             && *g2 <= GAMMASEVEN && *g2 >= GAMMA ) ) ) {
00165                             if ( fun1 != fun2 ) return(1);
00166                             if ( *fleft < 0 ) return(0);
00167                             if ( *fright < 0 ) return(1);
00168                             return(CompareFunctions(fleft,fright));
00169                         }
00170                         g2++;
00171                     }
00172                     break;
00173                 }
00174                 g1++;
00175             }
00176             group = g3;
00177         }

```

```

00178     }
00179     return(0);
00180 }
00181
00182 /*
00183     #] Commute :
00184     #[ Normalize :
00185
00186     This is the big normalization routine. It has a great need
00187     to be economical.
00188     There is a fixed limit to the number of objects coming in.
00189     Something should be done about it.
00190
00191 */
00192
00193 int Normalize(PHEAD WORD *term)
00194 {
00195     /*
00196         #[ Declarations :
00197     */
00198     GETBIDENTITY
00199     WORD *t, *m, *r, i, j, k, l, nsym, *ss, *tt, *u;
00200     WORD shortnum, stype;
00201     WORD *stop, *to = 0, *from = 0;
00202     /*
00203         The next variables could be better off in the AT.WorkSpace (?)
00204         Now they make stackallocations rather bothersome.
00205     */
00206     WORD psym[7*NORMSIZE], *ppsym;
00207     WORD pvec[NORMSIZE], *ppvec, nvec;
00208     WORD pdot[3*NORMSIZE], *ppdot, ndot;
00209     WORD pdel[2*NORMSIZE], *ppdel, ndel;
00210     WORD pind[NORMSIZE], nind;
00211     WORD *peps[NORMSIZE/3], neps;
00212     WORD *pden[NORMSIZE/3], nden;
00213     WORD *pcom[NORMSIZE], ncom;
00214     WORD *pnco[NORMSIZE], nnco;
00215     WORD *pcon[2*NORMSIZE], ncon; /* Pointer to contractable indices */
00216     WORD *n_coef, ncoef; /* Accumulator for the coefficient */
00217     WORD *n_llnum, *lnum, nnum;
00218     WORD *termout, oldtoprhs = 0, subtype;
00219     WORD ReplaceType, ReplaceVeto = 0, didcontr;
00220     int regval = 0;
00221     WORD *ReplaceSub;
00222     WORD *fillsetexp;
00223     CBUF *C = cbuf+AT.ebufnum;
00224     WORD *ANsc = 0, *ANsm = 0, *ANsr = 0, PolyFunMode;
00225     #ifdef WITHFLOAT
00226     WORD withfloat = 0;
00227     WORD *firstfloat = 0;
00228     #endif
00229     LONG oldcpointer = 0, x;
00230     n_coef = TermMalloc("NormCoef");
00231     n_llnum = TermMalloc("n_llnum");
00232     lnum = n_llnum+1;
00233     /*
00234         int termflag;
00235     */
00236     /*
00237         #] Declarations :
00238         #[ Setup :
00239     PrintTerm(term, "Normalize");
00240     */
00241
00242     Restart:
00243     didcontr = 0;
00244     ReplaceType = -1;
00245     t = term;
00246     if ( !*t ) { TermFree(n_coef, "NormCoef"); TermFree(n_llnum, "n_llnum"); return(regval); }
00247     r = t + *t;
00248     ncoef = r[-1];
00249     i = ABS(ncoef);
00250     r -= i;
00251     m = r;
00252     t = n_coef;
00253     NCOPY(t, r, i);
00254     termout = AT.WorkPointer;
00255     AT.WorkPointer = (WORD *)(((UBYTE *) (AT.WorkPointer)) + AM.MaxTer);
00256     fillsetexp = termout+1;
00257     AN.PolyNormFlag = 0; PolyFunMode = AN.PolyFunTodo;
00258     /*
00259         termflag = 0;
00260     */
00261     /*
00262         #] Setup :
00263         #[ First scan :
00264     */

```



```

00265     nsym = nvec = ndot = ndel = neps = nden =
00266     nind = ncom = nnco = ncon = 0;
00267     ppsym = psym;
00268     ppvec = pvec;
00269     ppdot = pdot;
00270     ppdel = pdel;
00271     t = term + 1;
00272     conscan:;
00273     if ( t < m ) do {
00274         r = t + t[1];
00275         switch ( *t ) {
00276             case SYMBOL :
00277                 t += 2;
00278                 from = m;
00279                 do {
00280                     if ( t[1] == 0 ) {
00281                         /* if ( *t == 0 || *t == MAXPOWER ) goto NormZZ; */
00282                         t += 2;
00283                         goto NextSymbol;
00284                     }
00285                     if ( *t <= DENOMINATORSYMBOL && *t >= COEFFSYMBOL ) {
00286                         /*
00287                             if ( AN.NoScrat2 == 0 ) {
00288                                 AN.NoScrat2 = (UWORD *)Malloc1((AM.MaxTal+2)*sizeof(UWORD), "Normalize");
00289                             }
00290                         */
00291                         if ( AN.cTerm ) m = AN.cTerm;
00292                         else m = term;
00293                         m += *m;
00294                         ncoef = REDLENG(ncoef);
00295                         if ( *t == COEFFSYMBOL ) {
00296                             i = t[1];
00297                             nnum = REDLENG(m[-1]);
00298                             m -= ABS(m[-1]);
00299                             if ( i > 0 ) {
00300                                 while ( i > 0 ) {
00301                                     if ( MulRat(BHEAD (UWORD *)n_coef, ncoef, (UWORD *)m, nnum,
00302                                         (UWORD *)n_coef, &ncoef) ) goto FromNorm;
00303                                     i--;
00304                                 }
00305                             }
00306                             else if ( i < 0 ) {
00307                                 while ( i < 0 ) {
00308                                     if ( DivRat(BHEAD (UWORD *)n_coef, ncoef, (UWORD *)m, nnum,
00309                                         (UWORD *)n_coef, &ncoef) ) goto FromNorm;
00310                                     i++;
00311                                 }
00312                             }
00313                         }
00314                         else {
00315                             i = m[-1];
00316                             nnum = (ABS(i)-1)/2;
00317                             if ( *t == NUMERATORSYMBOL ) { m -= nnum + 1; }
00318                             else { m--; }
00319                             while ( *m == 0 && nnum > 1 ) { m--; nnum--; }
00320                             m -= nnum;
00321                             if ( i < 0 && *t == NUMERATORSYMBOL ) nnum = -nnum;
00322                             i = t[1];
00323                             if ( i > 0 ) {
00324                                 while ( i > 0 ) {
00325                                     if ( Mully(BHEAD (UWORD *)n_coef, &ncoef, (UWORD *)m, nnum) )
00326                                         goto FromNorm;
00327                                     i--;
00328                                 }
00329                             }
00330                             else if ( i < 0 ) {
00331                                 while ( i < 0 ) {
00332                                     if ( Divvy(BHEAD (UWORD *)n_coef, &ncoef, (UWORD *)m, nnum) )
00333                                         goto FromNorm;
00334                                     i++;
00335                                 }
00336                             }
00337                         }
00338                         ncoef = INCLENG(ncoef);
00339                         t += 2;
00340                         goto NextSymbol;
00341                     }
00342                     else if ( *t == DIMENSIONSYMBOL ) {
00343                         if ( AN.cTerm ) m = AN.cTerm;
00344                         else m = term;
00345                         k = DimensionTerm(m);
00346                         if ( k == 0 ) goto NormZero;
00347                         if ( k == MAXPOSITIVE ) {
00348                             MLOCK(ErrorMessageLock);
00349                             MesPrint("Dimension_ is undefined in term %t");
00350                             MUNLOCK(ErrorMessageLock);
00351                             goto NormMin;

```

```

00352     }
00353     if ( k == -MAXPOSITIVE ) {
00354         MLOCK(ErrorMessageLock);
00355         MesPrint("Dimension_ out of range in term %t");
00356         MUNLOCK(ErrorMessageLock);
00357         goto NormMin;
00358     }
00359     if ( k > 0 ) { *((UWORD *)lnum) = k; nnum = 1; }
00360     else { *((UWORD *)lnum) = -k; nnum = -1; }
00361     ncoef = REDLENG(ncoef);
00362     if ( Mully(BHEAD (UWORD *)n_coef,&ncoef, (UWORD *)lnum,nnum) ) goto FromNorm;
00363     ncoef = INCLENG(ncoef);
00364     t += 2;
00365     goto NextSymbol;
00366 }
00367 if ( ( *t >= MAXPOWER && *t < 2*MAXPOWER )
00368     || ( *t < -MAXPOWER && *t > -2*MAXPOWER ) ) {
00369 /*
00370 #] TO SNUMBER :
00371 */
00372     if ( *t < 0 ) {
00373         *t += MAXPOWER;
00374         *t = -*t;
00375         if ( t[1] & 1 ) ncoef = -ncoef;
00376     }
00377     else if ( *t == MAXPOWER ) {
00378         if ( t[1] > 0 ) goto NormZero;
00379         goto NormInf;
00380     }
00381     else {
00382         *t -= MAXPOWER;
00383     }
00384     lnum[0] = *t;
00385     nnum = 1;
00386     if ( t[1] && RaisPow(BHEAD (UWORD *)lnum,&nnum, (UWORD) (ABS(t[1]))) )
00387         goto FromNorm;
00388     ncoef = REDLENG(ncoef);
00389     if ( t[1] < 0 ) {
00390         if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef, (UWORD *)lnum,nnum) )
00391             goto FromNorm;
00392     }
00393     else if ( t[1] > 0 ) {
00394         if ( Mully(BHEAD (UWORD *)n_coef,&ncoef, (UWORD *)lnum,nnum) )
00395             goto FromNorm;
00396     }
00397     ncoef = INCLENG(ncoef);
00398 /*
00399 #] TO SNUMBER :
00400 */
00401     t += 2;
00402     goto NextSymbol;
00403 }
00404 if ( ( *t <= NumSymbols && *t > -MAXPOWER )
00405     && ( symbols[*t].complex & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY ) {
00406     if ( t[1] <= 2*MAXPOWER && t[1] >= -2*MAXPOWER ) {
00407         t[1] %= symbols[*t].maxpower;
00408         if ( t[1] < 0 ) t[1] += symbols[*t].maxpower;
00409         if ( ( symbols[*t].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
00410             if ( ( ( symbols[*t].maxpower & 1 ) == 0 ) &&
00411                 ( t[1] >= symbols[*t].maxpower/2 ) ) {
00412                 t[1] -= symbols[*t].maxpower/2; ncoef = -ncoef;
00413             }
00414         }
00415         if ( t[1] == 0 ) { t += 2; goto NextSymbol; }
00416     }
00417 }
00418 i = nsym;
00419 m = ppsym;
00420 if ( i > 0 ) do {
00421     m -= 2;
00422     if ( *t == *m ) {
00423         t++; m++;
00424         if ( *t > 2*MAXPOWER || *t < -2*MAXPOWER
00425             || *m > 2*MAXPOWER || *m < -2*MAXPOWER ) {
00426             MLOCK(ErrorMessageLock);
00427             MesPrint("Illegal wildcard power combination.");
00428             MUNLOCK(ErrorMessageLock);
00429             goto NormMin;
00430         }
00431         *m += *t;
00432         if ( ( t[-1] <= NumSymbols && t[-1] > -MAXPOWER )
00433             && ( symbols[t[-1]].complex & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY
00434         ) {
00435             *m %= symbols[t[-1]].maxpower;
00436             if ( *m < 0 ) *m += symbols[t[-1]].maxpower;
00437             if ( ( symbols[t[-1]].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
00438                 if ( ( ( symbols[t[-1]].maxpower & 1 ) == 0 ) &&

```

```

00438             ( *m >= symbols[t[-1]].maxpower/2 ) ) {
00439                 *m -= symbols[t[-1]].maxpower/2; ncoef = -ncoef;
00440             }
00441         }
00442     }
00443     if ( *m >= 2*MAXPOWER || *m <= -2*MAXPOWER ) {
00444         MLOCK(ErrorMessageLock);
00445         MesPrint("Power overflow during normalization");
00446         MUNLOCK(ErrorMessageLock);
00447         goto NormMin;
00448     }
00449     if ( !*m ) {
00450         m--;
00451         while ( i < nsym )
00452             { *m = m[2]; m++; *m = m[2]; m++; i++; }
00453         ppsym -= 2;
00454         nsym--;
00455     }
00456     t++;
00457     goto NextSymbol;
00458 }
00459 } while ( *t < *m && --i > 0 );
00460 m = ppsym;
00461 while ( i < nsym )
00462     { m--; m[2] = *m; m--; m[2] = *m; i++; }
00463 *m++ = *t++;
00464 *m = *t++;
00465 ppsym += 2;
00466 nsym++;
00467 NextSymbol;;
00468     } while ( t < r );
00469     m = from;
00470     break;
00471 case VECTOR :
00472     t += 2;
00473     do {
00474         if ( t[0] == AM.vectorzero ) goto NormZero;
00475         if ( t[1] == FUNNYVEC ) {
00476             pind[nind++] = *t;
00477             t += 2;
00478         }
00479         else if ( t[1] < 0 ) {
00480             if ( *t == NOINDEX && t[1] == NOINDEX ) t += 2;
00481             else {
00482                 if ( t[1] == AM.vectorzero ) goto NormZero;
00483                 *ppdot++ = *t++; *ppdot++ = *t++; *ppdot++ = 1; ndot++;
00484             }
00485         }
00486         else { *ppvec++ = *t++; *ppvec++ = *t++; nvec += 2; }
00487     } while ( t < r );
00488     break;
00489 case DOTPRODUCT :
00490     t += 2;
00491     do {
00492         if ( t[2] == 0 ) t += 3;
00493         else if ( ndot > 0 && t[0] == ppdot[-3]
00494             && t[1] == ppdot[-2] ) {
00495             ppdot[-1] += t[2];
00496             t += 3;
00497             if ( ppdot[-1] == 0 ) { ppdot -= 3; ndot--; }
00498         }
00499         else if ( t[0] == AM.vectorzero || t[1] == AM.vectorzero ) {
00500             if ( t[2] > 0 ) goto NormZero;
00501             goto NormInf;
00502         }
00503         else {
00504             *ppdot++ = *t++; *ppdot++ = *t++;
00505             *ppdot++ = *t++; ndot++;
00506         }
00507     } while ( t < r );
00508     break;
00509 case HAAKJE :
00510     break;
00511 case SETSET:
00512     if ( WildFill(BHEAD termout,term,AT.dummysubexp) < 0 ) goto FromNorm;
00513     i = *termout;
00514     t = termout; m = term;
00515     NCOPY(m,t,i);
00516     goto Restart;
00517 case DOLLAREXPRESSION :
00518 /*
00519     We have DOLLAREXPRESSION,4,number,power
00520     Replace by SUBEXPRESSION and exit elegantly to let
00521     TestSub pick it up. Of course look for special cases first.
00522     Note that we have a special compiler buffer for the values.
00523 */
00524     if ( AR.Eside != LHSIDE ) {

```

```

00525         DOLLARS d = Dollars + t[2];
00526 #ifdef WITHPTHREADS
00527         int nummodopt, ptype = -1;
00528         if ( AS.MultiThreaded ) {
00529             for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
00530                 if ( t[2] == ModOptdollars[nummodopt].number ) break;
00531             }
00532             if ( nummodopt < NumModOptdollars ) {
00533                 ptype = ModOptdollars[nummodopt].type;
00534                 if ( ptype == MODLOCAL ) {
00535                     d = ModOptdollars[nummodopt].dstruct+AT.identity;
00536                 }
00537                 else {
00538                     LOCK(d->pthreadlockread);
00539                 }
00540             }
00541         }
00542 #endif
00543         if ( d->type == DOLZERO ) {
00544 #ifdef WITHPTHREADS
00545             if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
00546 #endif
00547             if ( t[3] == 0 ) goto NormZZ;
00548             if ( t[3] < 0 ) goto NormInf;
00549             goto NormZero;
00550         }
00551         else if ( d->type == DOLNUMBER ) {
00552             nnum = d->where[0];
00553             if ( nnum > 0 ) {
00554                 nnum = d->where[nnum-1];
00555                 if ( nnum < 0 ) { ncoef = -ncoef; nnum = -nnum; }
00556                 nnum = (nnum-1)/2;
00557                 for ( i = 1; i <= nnum; i++ ) lnum[i-1] = d->where[i];
00558             }
00559             if ( nnum == 0 || ( nnum == 1 && lnum[0] == 0 ) ) {
00560 #ifdef WITHPTHREADS
00561                 if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
00562 #endif
00563                 if ( t[3] < 0 ) goto NormInf;
00564                 else if ( t[3] == 0 ) goto NormZZ;
00565                 goto NormZero;
00566             }
00567             if ( t[3] && RaisPow(BHEAD (UWORD *)lnum,&nnum,(UWORD) (ABS(t[3]))) ) goto
FromNorm;
00568             ncoef = REDLENG(ncoef);
00569             if ( t[3] < 0 ) {
00570                 if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) ) {
00571 #ifdef WITHPTHREADS
00572                     if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
00573 #endif
00574                     goto FromNorm;
00575                 }
00576             }
00577             else if ( t[3] > 0 ) {
00578                 if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) ) {
00579 #ifdef WITHPTHREADS
00580                     if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
00581 #endif
00582                     goto FromNorm;
00583                 }
00584             }
00585             ncoef = INCLENG(ncoef);
00586         }
00587         else if ( d->type == DOLINDEX ) {
00588             if ( d->index == 0 ) {
00589 #ifdef WITHPTHREADS
00590                 if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
00591 #endif
00592                 goto NormZero;
00593             }
00594             if ( d->index != NOINDEX ) pind[nind++] = d->index;
00595         }
00596         else if ( d->type == DOLTERMS ) {
00597             if ( t[3] >= MAXPOWER || t[3] <= -MAXPOWER ) {
00598                 if ( d->where[0] == 0 ) goto NormZero;
00599                 if ( d->where[d->where[0]] != 0 ) {
00600 IllDollarExp:
00601                     MLOCK(ErrorMessageLock);
00602                     MesPrint("!!!Illegal $ expansion with wildcard power!!!");
00603                     MUNLOCK(ErrorMessageLock);
00604                     goto FromNorm;
00605                 }
00606 /*
00607 At this point we should only admit symbols and dotproducts
00608 We expand the dollar directly and do not send it back.
00609 */
00610 { WORD *td, *tdstop, dj;

```

```

00611         td = d->where+1;
00612         tdstop = d->where+d->where[0];
00613         if ( tdstop[-1] != 3 || tdstop[-2] != 1
00614             || tdstop[-3] != 1 ) goto IllDollarExp;
00615         tdstop -= 3;
00616         if ( td >= tdstop ) goto IllDollarExp;
00617         while ( td < tdstop ) {
00618             if ( *td == SYMBOL ) {
00619                 for ( dj = 2; dj < td[1]; dj += 2 ) {
00620                     if ( td[dj+1] == 1 ) {
00621                         *ppsym++ = td[dj];
00622                         *ppsym++ = t[3];
00623                         nsym++;
00624                     }
00625                     else if ( td[dj+1] == -1 ) {
00626                         *ppsym++ = td[dj];
00627                         *ppsym++ = -t[3];
00628                         nsym++;
00629                     }
00630                     else goto IllDollarExp;
00631                 }
00632             }
00633             else if ( *td == DOTPRODUCT ) {
00634                 for ( dj = 2; dj < td[1]; dj += 3 ) {
00635                     if ( td[dj+2] == 1 ) {
00636                         *ppdot++ = td[dj];
00637                         *ppdot++ = td[dj+1];
00638                         *ppdot++ = t[3];
00639                         ndot++;
00640                     }
00641                     else if ( td[dj+2] == -1 ) {
00642                         *ppdot++ = td[dj];
00643                         *ppdot++ = td[dj+1];
00644                         *ppdot++ = -t[3];
00645                         ndot++;
00646                     }
00647                     else goto IllDollarExp;
00648                 }
00649             }
00650             else goto IllDollarExp;
00651             td += td[1];
00652         }
00653         regval = 2;
00654         break;
00655     }
00656 }
00657 t[0] = SUBEXPRESSION;
00658 t[4] = AM.dbufnum;
00659 if ( t[3] == 0 ) {
00660 #ifdef WITHPTHREADS
00661     if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
00662 #endif
00663     break;
00664 }
00665 regval = 2;
00666 t = r;
00667 while ( t < m ) {
00668     if ( *t == DOLLAREXPRESSION ) {
00669 #ifdef WITHPTHREADS
00670         if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
00671 #endif
00672         d = Dollars + t[2];
00673 #ifdef WITHPTHREADS
00674         if ( AS.MultiThreaded ) {
00675             for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
00676                 if ( t[2] == ModOptdollars[nummodopt].number ) break;
00677             }
00678             if ( nummodopt < NumModOptdollars ) {
00679                 ptype = ModOptdollars[nummodopt].type;
00680                 if ( ptype == MODLOCAL ) {
00681                     d = ModOptdollars[nummodopt].dstruct+AT.identity;
00682                 }
00683                 else {
00684                     LOCK(d->pthreadlockread);
00685                 }
00686             }
00687         }
00688 #endif
00689         if ( d->type == DOLTERMS ) {
00690             t[0] = SUBEXPRESSION;
00691             t[4] = AM.dbufnum;
00692         }
00693     }
00694     t += t[1];
00695 }
00696 #ifdef WITHPTHREADS
00697     if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }

```

```

00698 #endif
00699         goto RegEnd;
00700     }
00701     else {
00702 #ifdef WITHPTHREADS
00703         if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
00704 #endif
00705         MLOCK(ErrorMessageLock);
00706         MesPrint("!!!This $ variation has not been implemented yet!!!");
00707         MUNLOCK(ErrorMessageLock);
00708         goto NormMin;
00709     }
00710 #ifdef WITHPTHREADS
00711     if ( ptype > 0 && ptype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
00712 #endif
00713     }
00714     else {
00715         pnco[ncco++] = t;
00716 /*
00717         The next statement should be safe as the value is used
00718         only by the compiler (ie the master).
00719 */
00720         AC.lhdollarflag = 1;
00721     }
00722     break;
00723 case DELTA :
00724     t += 2;
00725     do {
00726         if ( *t < 0 ) {
00727             if ( *t == SUMMEDIND ) {
00728                 if ( t[1] < -NMIN4SHIFT ) {
00729                     k = -t[1]-NMIN4SHIFT;
00730                     k = ExtraSymbol(k,1,nsym,ppsym,&ncoef);
00731                     nsym += k;
00732                     ppsym += (k * 2);
00733                 }
00734                 else if ( t[1] == 0 ) goto NormZero;
00735                 else {
00736                     if ( t[1] < 0 ) {
00737                         lnum[0] = -t[1];
00738                         nnum = -1;
00739                     }
00740                     else {
00741                         lnum[0] = t[1];
00742                         nnum = 1;
00743                     }
00744                     ncoef = REDLENG(ncoef);
00745                     if ( Mully(BHEAD (UWORD *)n_coef,&ncoef, (UWORD *)lnum,nnum) )
00746                         goto FromNorm;
00747                     ncoef = INCLENG(ncoef);
00748                 }
00749                 t += 2;
00750             }
00751             else if ( *t == NOINDEX && t[1] == NOINDEX ) t += 2;
00752             else if ( *t == EMPTYINDEX && t[1] == EMPTYINDEX ) {
00753                 *ppdel++ = *t++; *ppdel++ = *t++; ndel += 2;
00754             }
00755             else
00756                 if ( t[1] < 0 ) {
00757                     *ppdot++ = *t++; *ppdot++ = *t++; *ppdot++ = 1; ndot++;
00758                 }
00759             else {
00760                 *ppvec++ = *t++; *ppvec++ = *t++; nvec += 2;
00761             }
00762         }
00763         else {
00764             if ( t[1] < 0 ) {
00765                 *ppvec++ = t[1]; *ppvec++ = *t; t+=2; nvec += 2;
00766             }
00767             else { *ppdel++ = *t++; *ppdel++ = *t++; ndel += 2; }
00768         }
00769     } while ( t < r );
00770     break;
00771 case FACTORIAL :
00772 /*
00773     (FACTORIAL,FUNHEAD+2,..,-SNUMBER,number)
00774 */
00775     if ( t[FUNHEAD] == -SNUMBER && t[1] == FUNHEAD+2
00776         && t[FUNHEAD+1] >= 0 ) {
00777         if ( Factorial(BHEAD t[FUNHEAD+1],(UWORD *)lnum,&nnum) )
00778             goto FromNorm;
00779 MulIn:
00780         ncoef = REDLENG(ncoef);
00781         if ( Mully(BHEAD (UWORD *)n_coef,&ncoef, (UWORD *)lnum,nnum) ) goto FromNorm;
00782         ncoef = INCLENG(ncoef);
00783     }
00784     else pcom[ncom++] = t;

```

```

00785         break;
00786     case BERNOULLIFUNCTION :
00787     /*
00788         (BERNOULLIFUNCTION, FUNHEAD+2, ..., -SNUMBER, number)
00789     */
00790         if ( ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] >= 0 )
00791             && ( t[1] == FUNHEAD+2 || ( t[1] == FUNHEAD+4 &&
00792 t[FUNHEAD+2] == -SNUMBER && ABS(t[FUNHEAD+3]) == 1 ) ) ) {
00793             WORD inum, mnum;
00794             if ( Bernoulli(t[FUNHEAD+1], (UWORD *)lnum, &nnum) )
00795                 goto FromNorm;
00796             if ( nnum == 0 ) goto NormZero;
00797             inum = nnum; if ( inum < 0 ) inum = -inum;
00798             inum--; inum /= 2;
00799             mnum = inum;
00800             while ( lnum[mnum-1] == 0 ) mnum--;
00801             if ( nnum < 0 ) mnum = -mnum;
00802             ncoef = REDLENG(ncoef);
00803             if ( Mully(BHEAD (UWORD *)n_coef, &ncoef, (UWORD *)lnum, mnum) ) goto FromNorm;
00804             mnum = inum;
00805             while ( lnum[inum+mnum-1] == 0 ) mnum--;
00806             if ( Divvy(BHEAD (UWORD *)n_coef, &ncoef, (UWORD *) (lnum+inum), mnum) ) goto
FromNorm;
00807             ncoef = INCLENG(ncoef);
00808             if ( t[1] == FUNHEAD+4 && t[FUNHEAD+1] == 1
00809                 && t[FUNHEAD+3] == -1 ) ncoef = -ncoef;
00810         }
00811         else pcom[ncom++] = t;
00812         break;
00813     case NUMARGSFUN:
00814     /*
00815         Numerical function giving the number of arguments.
00816     */
00817         k = 0;
00818         t += FUNHEAD;
00819         while ( t < r ) {
00820             k++;
00821             NEXTARG(t);
00822         }
00823         if ( k == 0 ) goto NormZero;
00824         *((UWORD *)lnum) = k;
00825         nnum = 1;
00826         goto MulIn;
00827     case NUMFACTORS:
00828     /*
00829         Numerical function giving the number of factors in an expression.
00830     */
00831         t += FUNHEAD;
00832         if ( *t == -EXPRESSION ) {
00833             k = AS.OldNumFactors[t[1]];
00834         }
00835         else if ( *t == -DOLLAREXPRESSION ) {
00836             k = Dollars[t[1]].nfactors;
00837         }
00838         else {
00839             pcom[ncom++] = t;
00840             break;
00841         }
00842         if ( k == 0 ) goto NormZero;
00843         *((UWORD *)lnum) = k;
00844         nnum = 1;
00845         goto MulIn;
00846     case NUMTERMSFUN:
00847     /*
00848         Numerical function giving the number of terms in the single argument.
00849     */
00850         if ( t[FUNHEAD] < 0 ) {
00851             if ( t[FUNHEAD] <= -FUNCTION && t[1] == FUNHEAD+1 ) break;
00852             if ( t[FUNHEAD] > -FUNCTION && t[1] == FUNHEAD+2 ) {
00853                 if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] == 0 ) goto NormZero;
00854                 break;
00855             }
00856             pcom[ncom++] = t;
00857             break;
00858         }
00859         if ( t[FUNHEAD] > 0 && t[FUNHEAD] == t[1]-FUNHEAD ) {
00860             k = 0;
00861             t += FUNHEAD+ARGHEAD;
00862             while ( t < r ) {
00863                 k++;
00864                 t += *t;
00865             }
00866             if ( k == 0 ) goto NormZero;
00867             *((UWORD *)lnum) = k;
00868             nnum = 1;
00869             goto MulIn;
00870         }

```

```

00871         else pcom[ncom++] = t;
00872         break;
00873     case COUNTFUNCTION:
00874         if ( AN.cTerm ) {
00875             k = CountFun(AN.cTerm,t);
00876         }
00877         else {
00878             k = CountFun(term,t);
00879         }
00880         if ( k == 0 ) goto NormZero;
00881         if ( k > 0 ) { *((UWORD *)lnum) = k; nnum = 1; }
00882         else { *((UWORD *)lnum) = -k; nnum = -1; }
00883         goto MulIn;
00884         break;
00885     case MAKERATIONAL:
00886         if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+2] == -SNUMBER
00887             && t[1] == FUNHEAD+4 && t[FUNHEAD+3] > 1 ) {
00888             WORD x1[2], sgn;
00889             if ( t[FUNHEAD+1] == 0 ) goto NormZero;
00890             if ( t[FUNHEAD+1] < 0 ) { t[FUNHEAD+1] = -t[FUNHEAD+1]; sgn = -1; }
00891             else sgn = 1;
00892             if ( MakeRational(t[FUNHEAD+1],t[FUNHEAD+3],x1,x1+1) ) {
00893                 static int warnflag = 1;
00894                 if ( warnflag ) {
00895                     MesPrint("%w Warning: fraction could not be reconstructed in
MakeRational_");
00896                     warnflag = 0;
00897                 }
00898                 x1[0] = t[FUNHEAD+1]; x1[1] = 1;
00899             }
00900             if ( sgn < 0 ) { t[FUNHEAD+1] = -t[FUNHEAD+1]; x1[0] = -x1[0]; }
00901             if ( x1[0] < 0 ) { sgn = -1; x1[0] = -x1[0]; }
00902             else sgn = 1;
00903             ncoef = REDLENG(ncoef);
00904             if ( MulRat(BHEAD (UWORD *)n_coef,ncoef, (UWORD *)x1,sgn,
00905                 (UWORD *)n_coef,&ncoef) ) goto FromNorm;
00906             ncoef = INCLENG(ncoef);
00907         }
00908         else {
00909             WORD nargs = 0, *tt, *ttstop, *arg1 = 0, *arg2 = 0;
00910             UWORD *x1, *x2, *xx;
00911             WORD nx1,nx2,nxx;
00912             ttstop = t + t[1]; tt = t+FUNHEAD;
00913             while ( tt < ttstop ) {
00914                 nargs++;
00915                 if ( nargs == 1 ) arg1 = tt;
00916                 else arg2 = tt;
00917                 NEXTARG(tt);
00918             }
00919             if ( nargs != 2 ) goto defaultcase;
00920             if ( *arg2 == -SNUMBER && arg2[1] <= 1 ) goto defaultcase;
00921             else if ( *arg2 > 0 && ttstop[-1] < 0 ) goto defaultcase;
00922             x1 = NumberMalloc("Norm-MakeRational");
00923             if ( *arg1 == -SNUMBER ) {
00924                 if ( arg1[1] == 0 ) {
00925                     NumberFree(x1,"Norm-MakeRational");
00926                     goto NormZero;
00927                 }
00928                 if ( arg1[1] < 0 ) { x1[0] = -arg1[1]; nx1 = -1; }
00929                 else { x1[0] = arg1[1]; nx1 = 1; }
00930             }
00931             else if ( *arg1 > 0 ) {
00932                 WORD *tc;
00933                 nx1 = (ABS(arg2[-1])-1)/2;
00934                 tc = arg1+ARGHEAD+1+nx1;
00935                 if ( tc[0] != 1 ) {
00936                     NumberFree(x1,"Norm-MakeRational");
00937                     goto defaultcase;
00938                 }
00939                 for ( i = 1; i < nx1; i++ ) if ( tc[i] != 0 ) {
00940                     NumberFree(x1,"Norm-MakeRational");
00941                     goto defaultcase;
00942                 }
00943                 tc = arg1+ARGHEAD+1;
00944                 for ( i = 0; i < nx1; i++ ) x1[i] = tc[i];
00945                 if ( arg2[-1] < 0 ) nx1 = -nx1;
00946             }
00947             else {
00948                 NumberFree(x1,"Norm-MakeRational");
00949                 goto defaultcase;
00950             }
00951             x2 = NumberMalloc("Norm-MakeRational");
00952             if ( *arg2 == -SNUMBER ) {
00953                 if ( arg2[1] <= 1 ) {
00954                     NumberFree(x2,"Norm-MakeRational");
00955                     NumberFree(x1,"Norm-MakeRational");
00956                     goto defaultcase;

```



```

00957         }
00958         else { x2[0] = arg2[1]; nx2 = 1; }
00959     }
00960     else if ( *arg2 > 0 ) {
00961         WORD *tc;
00962         nx2 = (ttstop[-1]-1)/2;
00963         tc = arg2+ARGHEAD+1+nx2;
00964         if ( tc[0] != 1 ) {
00965             NumberFree(x2,"Norm-MakeRational");
00966             NumberFree(x1,"Norm-MakeRational");
00967             goto defaultcase;
00968         }
00969         for ( i = 1; i < nx2; i++ ) if ( tc[i] != 0 ) {
00970             NumberFree(x2,"Norm-MakeRational");
00971             NumberFree(x1,"Norm-MakeRational");
00972             goto defaultcase;
00973         }
00974         tc = arg2+ARGHEAD+1;
00975         for ( i = 0; i < nx2; i++ ) x2[i] = tc[i];
00976     }
00977     else {
00978         NumberFree(x2,"Norm-MakeRational");
00979         NumberFree(x1,"Norm-MakeRational");
00980         goto defaultcase;
00981     }
00982     if ( BigLong(x1,ABS(nx1),x2,nx2) >= 0 ) {
00983         UWORD *x3 = NumberMalloc("Norm-MakeRational");
00984         UWORD *x4 = NumberMalloc("Norm-MakeRational");
00985         WORD nx3, nx4;
00986         DivLong(x1,nx1,x2,nx2,x3,&nx3,x4,&nx4);
00987         for ( i = 0; i < ABS(nx4); i++ ) x1[i] = x4[i];
00988         nx1 = nx4;
00989         NumberFree(x4,"Norm-MakeRational");
00990         NumberFree(x3,"Norm-MakeRational");
00991     }
00992     xx = (UWORD *) (TermMalloc("Norm-MakeRational"));
00993     if ( MakeLongRational(BHEAD x1,nx1,x2,nx2,xx,&nxx) ) {
00994         static int warnflag = 1;
00995         if ( warnflag ) {
00996             MesPrint("%w Warning: fraction could not be reconstructed in
MakeRational_");
00997             warnflag = 0;
00998         }
00999         ncoef = REDLENG(ncoef);
01000         if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,x1,nx1) )
01001             goto FromNorm;
01002     }
01003     else {
01004         ncoef = REDLENG(ncoef);
01005         if ( MulRat(BHEAD (UWORD *)n_coef,ncoef,xx,nxx,
01006             (UWORD *)n_coef,&ncoef) ) goto FromNorm;
01007     }
01008     ncoef = INCLENG(ncoef);
01009     TermFree(xx,"Norm-MakeRational");
01010     NumberFree(x2,"Norm-MakeRational");
01011     NumberFree(x1,"Norm-MakeRational");
01012 }
01013 break;
01014 case TERMFUNCTION:
01015     if ( t[1] == FUNHEAD && AN.cTerm ) {
01016         ANsr = r; ANsm = m; ANsc = AN.cTerm;
01017         AN.cTerm = 0;
01018         t = ANsc + 1;
01019         m = ANsc + *ANsc;
01020         ncoef = REDLENG(ncoef);
01021         nnum = REDLENG(m[-1]);
01022         m -= ABS(m[-1]);
01023         if ( MulRat(BHEAD (UWORD *)n_coef,ncoef, (UWORD *)m,nnum,
01024             (UWORD *)n_coef,&ncoef) ) goto FromNorm;
01025         ncoef = INCLENG(ncoef);
01026         r = t;
01027     }
01028     break;
01029 case FIRSTBRACKET:
01030     if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -EXPRESSION ) {
01031         if ( GetFirstBracket(termout,t[FUNHEAD+1]) < 0 ) goto FromNorm;
01032         if ( *termout == 0 ) goto NormZero;
01033         if ( *termout > 4 ) {
01034             WORD *r1, *r2, *r3;
01035             while ( r < m ) *t++ = *r++;
01036             r1 = term + *term;
01037             r2 = termout + *termout; r2 -= ABS(r2[-1]);
01038             while ( r < r1 ) *r2++ = *r++;
01039             r3 = termout + 1;
01040             while ( r3 < r2 ) *t++ = *r3++;
01041             *term = t - term;
01042             if ( AT.WorkPointer > term && AT.WorkPointer < t )

```

```

01043             AT.WorkPointer = t;
01044             goto Restart;
01045         }
01046     }
01047     break;
01048 case FIRSTTERM:
01049 case CONTENTTERM:
01050     if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -EXPRESSION ) {
01051         {
01052             EXPRESSIONS e = Expressions+t[FUNHEAD+1];
01053             POSITION oldondisk = AS.OldOnFile[t[FUNHEAD+1]];
01054             if ( e->replace == NEWLYDEFINEDEXPRESSION ) {
01055                 AS.OldOnFile[t[FUNHEAD+1]] = e->onfile;
01056             }
01057             if ( *t == FIRSTTERM ) {
01058                 if ( GetFirstTerm(termout,t[FUNHEAD+1],0) < 0 ) goto FromNorm;
01059             }
01060             else if ( *t == CONTENTTERM ) {
01061                 if ( GetContent(termout,t[FUNHEAD+1]) < 0 ) goto FromNorm;
01062             }
01063             AS.OldOnFile[t[FUNHEAD+1]] = oldondisk;
01064             if ( *termout == 0 ) goto NormZero;
01065         }
01066 PasteIn;;
01067     {
01068         WORD *r1, *r2, *r3, *r4, *r5, nrl, *rterm;
01069         r2 = termout + *termout; lnum = r2 - ABS(r2[-1]);
01070         nnum = REDLENG(r2[-1]);
01071
01072         r1 = term + *term; r3 = r1 - ABS(r1[-1]);
01073         nrl = REDLENG(r1[-1]);
01074         if ( Mully(BHEAD (UWORD *)lnum,&nnum,(UWORD *)r3,nrl) ) goto FromNorm;
01075         nnum = INCLENG(nnum); nrl = ABS(nnum); lnum[nrl-1] = nnum;
01076         rterm = TermMalloc("FirstTerm/ContentTerm");
01077         r4 = rterm+1; r5 = term+1; while ( r5 < t ) *r4++ = *r5++;
01078         r5 = termout+1; while ( r5 < lnum ) *r4++ = *r5++;
01079         r5 = r; while ( r5 < r3 ) *r4++ = *r5++;
01080         r5 = lnum; NCOPY(r4,r5,nrl);
01081         *rterm = r4-rterm;
01082         nrl = *rterm; r1 = term; r2 = rterm; NCOPY(r1,r2,nrl);
01083         TermFree(rterm,"FirstTerm/ContentTerm");
01084         if ( AT.WorkPointer > term && AT.WorkPointer < r1 )
01085             AT.WorkPointer = r1;
01086         goto Restart;
01087     }
01088 }
01089 else if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -DOLLAREXPRESSION ) {
01090     DOLLARS d = Dollars + t[FUNHEAD+1], newd = 0;
01091     int idol, ido;
01092 #ifdef WITHPTHREADS
01093     int nummodopt, dtype = -1;
01094     if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
01095         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01096             if ( t[FUNHEAD+1] == ModOptdollars[nummodopt].number ) break;
01097         }
01098         if ( nummodopt < NumModOptdollars ) {
01099             dtype = ModOptdollars[nummodopt].type;
01100             if ( dtype == MODLOCAL ) {
01101                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
01102             }
01103         }
01104     }
01105 #endif
01106     if ( d->where && ( d->type == DOLTERMS || d->type == DOLNUMBER ) ) {
01107         newd = d;
01108     }
01109     else {
01110         if ( ( newd = DolToTerms(BHEAD t[FUNHEAD+1]) ) == 0 )
01111             goto NormZero;
01112     }
01113     if ( newd->where[0] == 0 ) {
01114         M_free(newd,"Copy of dollar variable");
01115         goto NormZero;
01116     }
01117     if ( *t == FIRSTTERM ) {
01118         idol = newd->where[0];
01119         for ( ido = 0; ido < idol; ido++ ) termout[idol] = newd->where[idol];
01120     }
01121     else if ( *t == CONTENTTERM ) {
01122         WORD *tterm;
01123         tterm = newd->where;
01124         idol = tterm[0];
01125         for ( ido = 0; ido < idol; ido++ ) termout[idol] = tterm[idol];
01126         tterm += *tterm;
01127         while ( *tterm ) {
01128             if ( ContentMerge(BHEAD termout,tterm) < 0 ) goto FromNorm;
01129             tterm += *tterm;

```

```

01130     }
01131     }
01132     if ( newd != d ) {
01133         if ( newd->factors ) M_free(newd->factors,"Dollar factors");
01134         M_free(newd,"Copy of dollar variable");
01135         newd = 0;
01136     }
01137     goto PasteIn;
01138 }
01139 break;
01140 case TERMSINEXPR:
01141 {
01142     if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -EXPRESSION ) {
01143         x = TermsInExpression(t[FUNHEAD+1]);
01144 multermnum:
01145         if ( x == 0 ) goto NormZero;
01146         if ( x < 0 ) {
01147             x = -x;
01148             if ( x > (LONG)WORDMASK ) { lnum[0] = x & WORDMASK;
01149                 lnum[1] = x » BITSINWORD; nnum = -2;
01150             }
01151             else { lnum[0] = x; nnum = -1; }
01152         }
01153         else if ( x > (LONG)WORDMASK ) {
01154             lnum[0] = x & WORDMASK;
01155             lnum[1] = x » BITSINWORD;
01156             nnum = 2;
01157         }
01158         else { lnum[0] = x; nnum = 1; }
01159         ncoef = REDLENG(ncoef);
01160         if ( Mully(BHEAD (UWORD *)n_coef,&ncoef, (UWORD *)lnum,nnum) )
01161             goto FromNorm;
01162         ncoef = INCLENG(ncoef);
01163     }
01164     else if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -DOLLAREXPRESSION ) {
01165         x = TermsInDollar(t[FUNHEAD+1]);
01166         goto multermnum;
01167     }
01168     else { pcom[ncom++] = t; }
01169 }
01170 break;
01171 case SIZEOFFUNCTION:
01172 {
01173     if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -EXPRESSION ) {
01174         x = SizeOfExpression(t[FUNHEAD+1]);
01175         goto multermnum;
01176     }
01177     else if ( ( t[1] == FUNHEAD+2 ) && t[FUNHEAD] == -DOLLAREXPRESSION ) {
01178         x = SizeOfDollar(t[FUNHEAD+1]);
01179         goto multermnum;
01180     }
01181     else { pcom[ncom++] = t; }
01182 }
01183 break;
01184 case MATCHFUNCTION:
01185 case PATTERNFUNCTION:
01186     break;
01187 case BINOMIAL:
01188 /*
01189     Binomial function for internal use for the moment.
01190     The routine in reken.c should be more efficient.
01191 */
01192 if ( t[1] == FUNHEAD+4 && t[FUNHEAD] == -SNUMBER
01193     && t[FUNHEAD+1] >= 0 && t[FUNHEAD+2] == -SNUMBER
01194     && t[FUNHEAD+3] >= 0 && t[FUNHEAD+1] >= t[FUNHEAD+3] ) {
01195     if ( t[FUNHEAD+1] > t[FUNHEAD+3] ) {
01196         if ( GetBinom((UWORD *)lnum,&nnum,
01197             t[FUNHEAD+1],t[FUNHEAD+3]) ) goto FromNorm;
01198         if ( nnum == 0 ) goto NormZero;
01199         goto MulIn;
01200     }
01201     else pcom[ncom++] = t;
01202     break;
01203 case SIGNFUN:
01204 /*
01205     Numerical function giving (-1)^arg
01206 */
01207 if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER ) {
01208     if ( ( t[FUNHEAD+1] & 1 ) != 0 ) ncoef = -ncoef;
01209 }
01210 else if ( ( t[FUNHEAD] > 0 ) && ( t[1] == FUNHEAD+t[FUNHEAD] ) )
01211 && ( t[FUNHEAD] == ARGHEAD+1+abs(t[t[1]-1]) ) ) {
01212     UWORD *numer1,*denom1;
01213     WORD nsize = abs(t[t[1]-1]), nnsiz, isize;
01214     nnsiz = (nsize-1)/2;
01215     numer1 = (UWORD *)t + FUNHEAD+ARGHEAD+1;
01216     denom1 = numer1 + nnsiz;

```

```

01217         for ( isize = 1; isize < nnsz; isize++ ) {
01218             if ( denom1[isize] ) break;
01219         }
01220         if ( ( denom1[0] != 1 ) || isize < nnsz ) {
01221             pcom[ncom++] = t;
01222         }
01223         else {
01224             if ( ( numer1[0] & 1 ) != 0 ) ncoef = -ncoef;
01225         }
01226     }
01227     else {
01228         goto doflags;
01229     /*
01230         pcom[ncom++] = t; */
01231     }
01232     break;
01233 case SIGFUNCTION:
01234     /*
01235     Numerical function giving the sign of the numerical argument
01236     The sign of zero is 1.
01237     If there are roots of unity they are part of the sign.
01238     */
01239     if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER ) {
01240         if ( t[FUNHEAD+1] < 0 ) ncoef = -ncoef;
01241     }
01242     else if ( ( t[1] == FUNHEAD+2 ) && ( t[FUNHEAD] == -SYMBOL )
01243         && ( ( t[FUNHEAD+1] <= NumSymbols && t[FUNHEAD+1] > -MAXPOWER )
01244         && ( symbols[t[FUNHEAD+1]].complex & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY )
01245     ) {
01246         k = t[FUNHEAD+1];
01247         from = m;
01248         i = nsym;
01249         m = ppsym;
01250         if ( i > 0 ) do {
01251             m -= 2;
01252             if ( k == *m ) {
01253                 m++;
01254                 *m = *m + 1;
01255                 *m %= symbols[k].maxpower;
01256                 if ( ( symbols[k].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
01257                     if ( ( ( symbols[k].maxpower & 1 ) == 0 ) &&
01258                         ( *m >= symbols[k].maxpower/2 ) ) {
01259                         *m -= symbols[k].maxpower/2; ncoef = -ncoef;
01260                     }
01261                 }
01262                 if ( !*m ) {
01263                     m--;
01264                     while ( i < nsym )
01265                         { *m = m[2]; m++; *m = m[2]; m++; i++; }
01266                     ppsym -= 2;
01267                     nsym--;
01268                     goto sigDoneSymbol;
01269                 }
01270             } while ( k < *m && --i > 0 );
01271             m = ppsym;
01272             while ( i < nsym )
01273                 { m--; m[2] = *m; m--; m[2] = *m; i++; }
01274             *m++ = k;
01275             *m = 1;
01276             ppsym += 2;
01277             nsym++;
01278             sigDoneSymbol:;
01279             m = from;
01280         }
01281         else if ( ( t[FUNHEAD] > 0 ) && ( t[1] == FUNHEAD+t[FUNHEAD] ) ) {
01282             if ( t[FUNHEAD] == ARGHEAD+1+abs(t[t[1]-1]) ) {
01283                 if ( t[t[1]-1] < 0 ) ncoef = -ncoef;
01284             }
01285         }
01286         /*
01287         Now we should fish out the roots of unity
01288         */
01289         else if ( ( t[FUNHEAD+ARGHEAD]+FUNHEAD+ARGHEAD == t[1] )
01290             && ( t[FUNHEAD+ARGHEAD+1] == SYMBOL ) ) {
01291             WORD *ts = t + FUNHEAD+ARGHEAD+3;
01292             WORD its = ts[-1]-2;
01293             while ( its > 0 ) {
01294                 if ( ( *ts != 0 ) && ( ( *ts > NumSymbols || *ts <= -MAXPOWER )
01295                     || ( symbols[*ts].complex & VARTYPEROOTOFUNITY ) != VARTYPEROOTOFUNITY )
01296                 ) {
01297                     goto signogood;
01298                 }
01299                 ts += 2; its -= 2;
01300             }
01301         }
01302         /*
01303         Now we have only roots of unity which should be
01304         registered in the list of symbols.
01305         */

```

```

01302         if ( t[t[1]-1] < 0 ) ncoef = -ncoef;
01303         ts = t + FUNHEAD+ARGHEAD+3;
01304         its = ts[-1]-2;
01305         from = m;
01306         while ( its > 0 ) {
01307             i = nsym;
01308             m = ppsym;
01309             if ( i > 0 ) do {
01310                 m -= 2;
01311                 if ( *ts == *m ) {
01312                     ts++; m++;
01313                     *m += *ts;
01314                     if ( ( ts[-1] <= NumSymbols && ts[-1] > -MAXPOWER ) &&
01315                         ( symbols[ts[-1]].complex & VARTYPEROOTOFUNITY ) ==
VARTYPEROOTOFUNITY ) {
01316                         *m %= symbols[ts[-1]].maxpower;
01317                         if ( *m < 0 ) *m += symbols[ts[-1]].maxpower;
01318                         if ( ( symbols[ts[-1]].complex & VARTYPEMINUS ) ==
VARTYPEMINUS ) {
01319                             if ( ( ( symbols[ts[-1]].maxpower & 1 ) == 0 ) &&
01320                                 ( *m >= symbols[ts[-1]].maxpower/2 ) ) {
01321                                 *m -= symbols[ts[-1]].maxpower/2; ncoef = -ncoef;
01322                             }
01323                         }
01324                     }
01325                     if ( !*m ) {
01326                         m--;
01327                         while ( i < nsym )
01328                             { *m = m[2]; m++; *m = m[2]; m++; i++; }
01329                         ppsym -= 2;
01330                         nsym--;
01331                     }
01332                     ts++; its -= 2;
01333                     goto sigNextSymbol;
01334                 }
01335             } while ( *ts < *m && --i > 0 );
01336             m = ppsym;
01337             while ( i < nsym )
01338                 { m--; m[2] = *m; m--; m[2] = *m; i++; }
01339             *m++ = *ts++;
01340             *m = *ts++;
01341             ppsym += 2;
01342             nsym++; its -= 2;
01343 sigNextSymbol:;
01344         }
01345         m = from;
01346     }
01347     else {
01348 signogood:         pcom[ncom++] = t;
01349     }
01350 }
01351 else pcom[ncom++] = t;
01352 break;
01353 case ABSFUNCTION:
01354 /*
01355     Numerical function giving the absolute value of the
01356     numerical argument. Or roots of unity.
01357 */
01358     if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER ) {
01359         k = t[FUNHEAD+1];
01360         if ( k < 0 ) k = -k;
01361         if ( k == 0 ) goto NormZero;
01362         *((UWORD *)lnum) = k; nnum = 1;
01363         goto MulIn;
01364     }
01365 }
01366 else if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SYMBOL ) {
01367     k = t[FUNHEAD+1];
01368     if ( ( k > NumSymbols || k <= -MAXPOWER )
01369         || ( symbols[k].complex & VARTYPEROOTOFUNITY ) != VARTYPEROOTOFUNITY )
01370         goto absnogood;
01371 }
01372 else if ( ( t[FUNHEAD] > 0 ) && ( t[1] == FUNHEAD+t[FUNHEAD] )
01373     && ( t[1] == FUNHEAD+ARGHEAD+t[FUNHEAD+ARGHEAD] ) ) {
01374     if ( t[FUNHEAD] == ARGHEAD+1+abs(t[t[1]-1]) ) {
01375         WORD *ts;
01376 absnosymbols:         ts = t + t[1] -1;
01377                         ncoef = REDLENG(ncoef);
01378                         nnum = REDLENG(*ts);
01379                         if ( nnum < 0 ) nnum = -nnum;
01380                         if ( MulRat(BHEAD (UWORD *)n_coef,ncoef,
01381                             (UWORD *) (ts-ABS(*ts)+1),nnum,
01382                             (UWORD *)n_coef,&ncoef) ) goto FromNorm;
01383                         ncoef = INCLENG(ncoef);
01384                     }
01385 /*
01386     Now get rid of the roots of unity. This includes i_

```

```

01387 */
01388
01389         else if ( t[FUNHEAD+ARGHEAD+1] == SYMBOL ) {
01390             WORD *ts = t+FUNHEAD+ARGHEAD+1;
01391             WORD its = ts[1] - 2;
01392             ts += 2;
01393             while ( its > 0 ) {
01394                 if ( *ts == 0 ) { }
01395                 else if ( ( *ts > NumSymbols || *ts <= -MAXPOWER )
01396                     || ( symbols[*ts].complex & VARTYPEROOTOFUNITY )
01397                     != VARTYPEROOTOFUNITY ) goto absnogood;
01398                 its -= 2; ts += 2;
01399             }
01400             goto absnosymbols;
01401         }
01402     else {
01403         pcom[ncom++] = t;
01404     }
01405     else pcom[ncom++] = t;
01406     break;
01407 case MODFUNCTION:
01408 case MOD2FUNCTION:
01409     /*
01410     Mod function. Does work if two arguments and the
01411     second argument is a positive short number
01412     */
01413     if ( t[1] == FUNHEAD+4 && t[FUNHEAD] == -SNUMBER
01414         && t[FUNHEAD+2] == -SNUMBER && t[FUNHEAD+3] > 1 ) {
01415         WORD tmod;
01416         tmod = (t[FUNHEAD+1]*t[FUNHEAD+3]);
01417         if ( tmod < 0 ) tmod += t[FUNHEAD+3];
01418         if ( *t == MOD2FUNCTION && tmod > t[FUNHEAD+3]/2 )
01419             tmod -= t[FUNHEAD+3];
01420         if ( tmod < 0 ) {
01421             *((UWORD *)lnum) = -tmod;
01422             nnum = -1;
01423         }
01424         else if ( tmod > 0 ) {
01425             *((UWORD *)lnum) = tmod;
01426             nnum = 1;
01427         }
01428         else goto NormZero;
01429         goto MulIn;
01430     }
01431     else if ( t[1] > t[FUNHEAD+2] && t[FUNHEAD] > 0
01432         && t[FUNHEAD+t[FUNHEAD]] == -SNUMBER
01433         && t[FUNHEAD+t[FUNHEAD]+1] > 1
01434         && t[1] == FUNHEAD+2+t[FUNHEAD] ) {
01435         WORD *ttt = t+FUNHEAD, iii;
01436         iii = ttt[*ttt-1];
01437         if ( *ttt == ttt[ARGHEAD]+ARGHEAD &&
01438             ttt[ARGHEAD] == ABS(iii)+1 ) {
01439             WORD ncmmod = 1;
01440             WORD cmod = ttt[*ttt+1];
01441             iii = REDLENG(iii);
01442             if ( *t == MODFUNCTION ) {
01443                 if ( TakeModulus((UWORD *) (ttt+ARGHEAD+1)
01444                     , &iii, (UWORD *) (&cmod), ncmmod, UNPACK|NOINVERSES) )
01445                     goto FromNorm;
01446             }
01447             else {
01448                 if ( TakeModulus((UWORD *) (ttt+ARGHEAD+1)
01449                     , &iii, (UWORD *) (&cmod), ncmmod, UNPACK|POSNEG|NOINVERSES) )
01450                     goto FromNorm;
01451             }
01452             if ( *t == MOD2FUNCTION && ttt[ARGHEAD+1] > cmod/2 && iii > 0 ) {
01453                 ttt[ARGHEAD+1] -= cmod;
01454             }
01455             if ( ttt[ARGHEAD+1] < 0 ) {
01456                 *((UWORD *)lnum) = -ttt[ARGHEAD+1];
01457                 nnum = -1;
01458             }
01459             else if ( ttt[ARGHEAD+1] > 0 ) {
01460                 *((UWORD *)lnum) = ttt[ARGHEAD+1];
01461                 nnum = 1;
01462             }
01463             else goto NormZero;
01464             goto MulIn;
01465         }
01466     }
01467     else if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER ) {
01468         *((UWORD *)lnum) = t[FUNHEAD+1];
01469         if ( *lnum == 0 ) goto NormZero;
01470         nnum = 1;
01471         goto MulIn;
01472     }
01473     else if ( ( ( t[FUNHEAD] < 0 ) && ( t[FUNHEAD] == -SNUMBER )

```

```

01474      && ( t[1] >= ( FUNHEAD+6+ARGHEAD ) )
01475      && ( t[FUNHEAD+2] >= 4+ARGHEAD )
01476      && ( t[t[1]-1] == t[FUNHEAD+2+ARGHEAD]-1 ) ) ||
01477      ( ( t[FUNHEAD] > 0 )
01478      && ( t[FUNHEAD]-ARGHEAD-1 == ABS(t[FUNHEAD+t[FUNHEAD]-1]) ) )
01479      && ( t[FUNHEAD+t[FUNHEAD]]-ARGHEAD-1 == t[t[1]-1] ) ) ) {
01480  /*
01481      Check that the last (long) number is integer
01482  */
01483      WORD *ttt = t + t[1], iii, iiil;
01484      UWORD coefbuf[2], *coef2, ncoef2;
01485      iii = (ttt[-1]-1)/2;
01486      ttt -= iii;
01487      if ( ttt[-1] != 1 ) {
01488  exitfromhere:
01489          pcom[ncom++] = t;
01490          break;
01491      }
01492      iii--;
01493      for ( iiil = 0; iiil < iii; iiil++ ) {
01494          if ( ttt[iiil] != 0 ) goto exitfromhere;
01495      }
01496  /*
01497      Now we have a hit!
01498      The first argument will be put in lnum.
01499      It will be a rational.
01500      The second argument will be a long integer in coef2.
01501  */
01502      ttt = t + FUNHEAD;
01503      if ( *ttt < 0 ) {
01504          if ( ttt[1] < 0 ) {
01505              nnum = -1; lnum[0] = -ttt[1]; lnum[1] = 1;
01506          }
01507          else {
01508              nnum = 1; lnum[0] = ttt[1]; lnum[1] = 1;
01509          }
01510      }
01511      else {
01512          nnum = ABS(ttt[ttt[0]-1] - 1);
01513          for ( iii = 0; iii < nnum; iii++ ) {
01514              lnum[iii] = ttt[ARGHEAD+1+iii];
01515          }
01516          nnum = nnum/2;
01517          if ( ttt[ttt[0]-1] < 0 ) nnum = -nnum;
01518      }
01519      NEXTARG(ttt);
01520      if ( *ttt < 0 ) {
01521          coef2 = coefbuf;
01522          ncoef2 = 3; *coef2 = (UWORD) (ttt[1]);
01523          coef2[1] = 1;
01524      }
01525      else {
01526          coef2 = (UWORD *) (ttt+ARGHEAD+1);
01527          ncoef2 = (ttt[ttt[0]-1]-1)/2;
01528      }
01529      if ( TakeModulus((UWORD *)lnum,&nnum,(UWORD *)coef2,ncoef2,
01530                      UNPACK|NOINVERSES|FROMFUNCTION) ) {
01531          goto FromNorm;
01532      }
01533      if ( *t == MOD2FUNCTION && nnum > 0 ) {
01534          UWORD *coef3 = NumberMalloc("Mod2Function"), two = 2;
01535          WORD ncoef3;
01536          if ( MulLong((UWORD *)lnum,nnum,&two,1,coef3,&ncoef3) )
01537              goto FromNorm;
01538          if ( BigLong(coef3,ncoef3,(UWORD *)coef2,ncoef2) > 0 ) {
01539              nnum = -nnum;
01540              AddLong((UWORD *)lnum,nnum,(UWORD *)coef2,ncoef2
01541                      , (UWORD *)lnum,&nnum);
01542              nnum = -nnum;
01543          }
01544          NumberFree(coef3,"Mod2Function");
01545      }
01546  /*
01547      Do we have to pack? No, because the answer is not a fraction
01548  */
01549      ncoef = REDLENG(ncoef);
01550      if ( Mully(BHEAD,(UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum) )
01551          goto FromNorm;
01552      ncoef = INCLENG(ncoef);
01553  }
01554      else pcom[ncom++] = t;
01555      break;
01556  case EXTEUCLIDEAN:
01557      {
01558          WORD argcount = 0, *tc, *ts, xc, xs, *tcc;
01559          UWORD *Num1, *Num2, *Num3, *Num4;
01560          WORD size1, size2, size3, size4, space;

```

```

01561 tc = t+FUNHEAD; ts = t + t[1];
01562 while ( argcount < 3 && tc < ts ) { NEXTARG(tc); argcount++; }
01563 if ( argcount != 2 ) goto defaultcase;
01564 if ( t[FUNHEAD] == -SNUMBER ) {
01565     if ( t[FUNHEAD+1] <= 1 ) goto defaultcase;
01566     if ( t[FUNHEAD+2] == -SNUMBER ) {
01567         if ( t[FUNHEAD+3] <= 1 ) goto defaultcase;
01568         Num2 = NumberMalloc("modinverses");
01569         *Num2 = t[FUNHEAD+3]; size2 = 1;
01570     }
01571     else {
01572         if ( ts[-1] < 0 ) goto defaultcase;
01573         if ( ts[-1] != t[FUNHEAD+2]-ARGHEAD-1 ) goto defaultcase;
01574         xs = (ts[-1]-1)/2;
01575         tcc = ts-xs-1;
01576         if ( *tcc != 1 ) goto defaultcase;
01577         for ( i = 1; i < xs; i++ ) {
01578             if ( tcc[i] != 0 ) goto defaultcase;
01579         }
01580         Num2 = NumberMalloc("modinverses");
01581         size2 = xs;
01582         for ( i = 0; i < xs; i++ ) Num2[i] = t[FUNHEAD+ARGHEAD+3+i];
01583     }
01584     Num1 = NumberMalloc("modinverses");
01585     *Num1 = t[FUNHEAD+1]; size1 = 1;
01586 }
01587 else {
01588     tc = t + FUNHEAD + t[FUNHEAD];
01589     if ( tc[-1] < 0 ) goto defaultcase;
01590     if ( tc[-1] != t[FUNHEAD]-ARGHEAD-1 ) goto defaultcase;
01591     xc = (tc[-1]-1)/2;
01592     tcc = tc-xc-1;
01593     if ( *tcc != 1 ) goto defaultcase;
01594     for ( i = 1; i < xc; i++ ) {
01595         if ( tcc[i] != 0 ) goto defaultcase;
01596     }
01597     if ( *tc == -SNUMBER ) {
01598         if ( tc[1] <= 1 ) goto defaultcase;
01599         Num2 = NumberMalloc("modinverses");
01600         *Num2 = tc[1]; size2 = 1;
01601     }
01602     else {
01603         if ( ts[-1] < 0 ) goto defaultcase;
01604         if ( ts[-1] != t[FUNHEAD+2]-ARGHEAD-1 ) goto defaultcase;
01605         xs = (ts[-1]-1)/2;
01606         tcc = ts-xs-1;
01607         if ( *tcc != 1 ) goto defaultcase;
01608         for ( i = 1; i < xs; i++ ) {
01609             if ( tcc[i] != 0 ) goto defaultcase;
01610         }
01611         Num2 = NumberMalloc("modinverses");
01612         size2 = xs;
01613         for ( i = 0; i < xs; i++ ) Num2[i] = tc[ARGHEAD+1+i];
01614     }
01615     Num1 = NumberMalloc("modinverses");
01616     size1 = xc;
01617     for ( i = 0; i < xc; i++ ) Num1[i] = t[FUNHEAD+ARGHEAD+1+i];
01618 }
01619 Num3 = NumberMalloc("modinverses");
01620 Num4 = NumberMalloc("modinverses");
01621 GetLongModInverses(BHEAD Num1,size1,Num2,size2
01622     ,Num3,&size3,Num4,&size4);
01623 /*
01624 Now we have to compose the answer. This needs more space
01625 and hence we have to put this inside the term.
01626 Compute first how much extra space we need.
01627 Then move the trailing part of the term upwards.
01628 Do not forget relevant pointers!!! (r, m, termout, AT.WorkPointer)
01629 */
01630 space = 0;
01631 if ( ( size3 == 1 || size3 == -1 ) && (*Num3&TOPBITONLY) == 0 ) space += 2;
01632 else space += ARGHEAD + 2*ABS(size3) + 2;
01633 if ( ( size4 == 1 || size4 == -1 ) && (*Num4&TOPBITONLY) == 0 ) space += 2;
01634 else space += ARGHEAD + 2*ABS(size4) + 2;
01635 tt = term + *term; u = tt + space;
01636 while ( tt >= ts ) *--u = *--tt;
01637 m += space; r += space;
01638 *term += space;
01639 t[1] += space;
01640 if ( ( size3 == 1 || size3 == -1 ) && (*Num3&TOPBITONLY) == 0 ) {
01641     *ts++ = -SNUMBER; *ts = (WORD) (*Num3);
01642     if ( size3 < 0 ) *ts = -*ts;
01643     ts++;
01644 }
01645 else {
01646     *ts++ = 2*ABS(size3)+ARGHEAD+2;
01647     *ts++ = 0; FILLARG(ts)

```



```

01648         *ts++ = 2*ABS(size3)+1;
01649         for ( i = 0; i < ABS(size3); i++ ) *ts++ = Num3[i];
01650         *ts++ = 1;
01651         for ( i = 1; i < ABS(size3); i++ ) *ts++ = 0;
01652         if ( size3 < 0 ) *ts++ = 2*size3-1;
01653         else *ts++ = 2*size3+1;
01654     }
01655     if ( ( size4 == 1 || size4 == -1 ) && (*Num4&TOPBITONLY) == 0 ) {
01656         *ts++ = -SNUMBER; *ts = *Num4;
01657         if ( size4 < 0 ) *ts = -*ts;
01658         ts++;
01659     }
01660     else {
01661         *ts++ = 2*ABS(size4)+ARGHEAD+2;
01662         *ts++ = 0; FILLARG(ts)
01663         *ts++ = 2*ABS(size4)+2;
01664         for ( i = 0; i < ABS(size4); i++ ) *ts++ = Num4[i];
01665         *ts++ = 1;
01666         for ( i = 1; i < ABS(size4); i++ ) *ts++ = 0;
01667         if ( size4 < 0 ) *ts++ = 2*size4-1;
01668         else *ts++ = 2*size4+1;
01669     }
01670     NumberFree(Num4,"modinverses");
01671     NumberFree(Num3,"modinverses");
01672     NumberFree(Num1,"modinverses");
01673     NumberFree(Num2,"modinverses");
01674     t[2] = 0; /* mark function as clean. */
01675     goto Restart;
01676 }
01677 break;
01678 case GCDFUNCTION:
01679 #ifdef EVALUATEGCD
01680 #ifdef NEWGCDFUNCTION
01681 {
01682 /*
01683     Has two integer arguments
01684     Four cases: S,S, S,L, L,S, L,L
01685 */
01686     WORD *num1, *num2, size1, size2, stor1, stor2, *ttt, ti;
01687     if ( t[1] == FUNHEAD+4 && t[FUNHEAD] == -SNUMBER
01688         && t[FUNHEAD+2] == -SNUMBER && t[FUNHEAD+1] != 0
01689         && t[FUNHEAD+3] != 0 ) { /* Short,Short */
01690         stor1 = t[FUNHEAD+1];
01691         stor2 = t[FUNHEAD+3];
01692         if ( stor1 < 0 ) stor1 = -stor1;
01693         if ( stor2 < 0 ) stor2 = -stor2;
01694         num1 = &stor1; num2 = &stor2;
01695         size1 = size2 = 1;
01696         goto gcdcalc;
01697     }
01698     else if ( t[1] > FUNHEAD+4 ) {
01699         if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] != 0
01700             && t[FUNHEAD+2] == t[1]-FUNHEAD-2 &&
01701             ABS(t[t[1]-1]) == t[FUNHEAD+2]-1-ARGHEAD ) { /* Short,Long */
01702             num2 = t + t[1];
01703             size2 = ABS(num2[-1]);
01704             ttt = num2-1;
01705             num2 -= size2;
01706             size2 = (size2-1)/2;
01707             ti = size2;
01708             while ( ti > 1 && ttt[-1] == 0 ) { ttt--; ti--; }
01709             if ( ti == 1 && ttt[-1] == 1 ) {
01710                 stor1 = t[FUNHEAD+1];
01711                 if ( stor1 < 0 ) stor1 = -stor1;
01712                 num1 = &stor1;
01713                 size1 = 1;
01714                 goto gcdcalc;
01715             }
01716             else pcom[ncom++] = t;
01717         }
01718         else if ( t[FUNHEAD] > 0 &&
01719             t[FUNHEAD]-1-ARGHEAD == ABS(t[t[FUNHEAD]+FUNHEAD-1]) ) {
01720             num1 = t + FUNHEAD + t[FUNHEAD];
01721             size1 = ABS(num1[-1]);
01722             ttt = num1-1;
01723             num1 -= size1;
01724             size1 = (size1-1)/2;
01725             ti = size1;
01726             while ( ti > 1 && ttt[-1] == 0 ) { ttt--; ti--; }
01727             if ( ti == 1 && ttt[-1] == 1 ) {
01728                 if ( t[1]-FUNHEAD == t[FUNHEAD]+2 && t[t[1]-2] == -SNUMBER
01729                     && t[t[1]-1] != 0 ) { /* Long,Short */
01730                     stor2 = t[t[1]-1];
01731                     if ( stor2 < 0 ) stor2 = -stor2;
01732                     num2 = &stor2;
01733                     size2 = 1;
01734                     goto gcdcalc;

```

```

01735     }
01736     else if ( t[1]-FUNHEAD == t[FUNHEAD]+t[FUNHEAD+t[FUNHEAD]]
01737     && ABS(t[t[1]-1]) == t[FUNHEAD+t[FUNHEAD]] - ARGHEAD-1 ) {
01738         num2 = t + t[1];
01739         size2 = ABS(num2[-1]);
01740         ttt = num2-1;
01741         num2 -= size2;
01742         size2 = (size2-1)/2;
01743         ti = size2;
01744         while ( ti > 1 && ttt[-1] == 0 ) { ttt--; ti--; }
01745         if ( ti == 1 && ttt[-1] == 1 ) {
01746 gcdcalc:         if ( GcdLong(BHEAD (UWORD *)num1,size1,(UWORD *)num2,size2
01747             , (UWORD *)lnum,&nnum) ) goto FromNorm;
01748             goto MulIn;
01749         }
01750         else pcom[ncom++] = t;
01751     }
01752     else pcom[ncom++] = t;
01753 }
01754     else pcom[ncom++] = t;
01755 }
01756     else pcom[ncom++] = t;
01757 }
01758 else pcom[ncom++] = t;
01759 }
01760 #else
01761 {
01762     WORD *gcd = AT.WorkPointer;
01763     if ( ( gcd = EvaluateGcd(BHEAD t) ) == 0 ) goto FromNorm;
01764     if ( *gcd == 4 && gcd[1] == 1 && gcd[2] == 1 && gcd[4] == 0 ) {
01765         AT.WorkPointer = gcd;
01766     }
01767     else if ( gcd[*gcd] == 0 ) {
01768         WORD *t1, iii, change, *num, *den, numsize, densize;
01769         if ( gcd[*gcd-1] < *gcd-1 ) {
01770             t1 = gcd+1;
01771             for ( iii = 2; iii < t1[1]; iii += 2 ) {
01772                 change = ExtraSymbol(t1[iii],t1[iii+1],nsym,ppsym,&ncoef);
01773                 nsym += change;
01774                 ppsym += change * 2;
01775             }
01776         }
01777         t1 = gcd + *gcd;
01778         iii = t1[-1]; num = t1-iii; numsize = (iii-1)/2;
01779         den = num + numsize; densize = numsize;
01780         while ( numsize > 1 && num[numsize-1] == 0 ) numsize--;
01781         while ( densize > 1 && den[densize-1] == 0 ) densize--;
01782         if ( numsize > 1 || num[0] != 1 ) {
01783             ncoef = REDLENG(ncoef);
01784             if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)num,numsize) ) goto
FromNorm;
01785             ncoef = INCLENG(ncoef);
01786         }
01787         if ( densize > 1 || den[0] != 1 ) {
01788             ncoef = REDLENG(ncoef);
01789             if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)den,densize) ) goto
FromNorm;
01790             ncoef = INCLENG(ncoef);
01791         }
01792         AT.WorkPointer = gcd;
01793     }
01794     else { /* a whole expression */
01795 /*
01796         Action: Put the expression in a compiler buffer.
01797         Insert a SUBEXPRESSION subterm
01798         Set the return value of the routine such that in
01799         Generator the term gets sent again to TestSub.
01800
01801         1: put in C (ebufnum)
01802         2: after that the Workspace is free again.
01803         3: insert the SUBEXPRESSION
01804         4: copy the top part of the term down
01805 */
01806         LONG size = AT.WorkPointer - gcd;
01807
01808         ss = AddRHS(AT.ebufnum,1);
01809         while ( (ss + size + 10) > C->Top ) ss = DoubleCbuffer(AT.ebufnum,ss,13);
01810         tt = gcd;
01811         NCOPY(ss,tt,size);
01812         C->rhs[C->numrhs+1] = ss;
01813         C->Pointer = ss;
01814
01815         t[0] = SUBEXPRESSION;
01816         t[1] = SUBEXPSIZE;
01817         t[2] = C->numrhs;
01818         t[3] = 1;
01819         t[4] = AT.ebufnum;

```

```

01820         t += 5;
01821         tt = term + *term;
01822         while ( r < tt ) *t++ = *r++;
01823         *term = t - term;
01824
01825         regval = 1;
01826         goto RegEnd;
01827     }
01828 }
01829 #endif
01830 #else
01831     MesPrint(" Unexpected call to EvaluateGCD");
01832     Terminate(-1);
01833 #endif
01834
01835     break;
01836 case MINFUNCTION:
01837 case MAXFUNCTION:
01838     if ( t[1] == FUNHEAD ) break;
01839     {
01840         WORD *ttt = t + FUNHEAD;
01841         WORD *tttstop = t + t[1];
01842         WORD tterm[4], iiii;
01843         while ( ttt < tttstop ) {
01844             if ( *ttt > 0 ) {
01845                 if ( ttt[ARGHEAD]-1 > ABS(ttt[*ttt-1]) ) goto nospec;
01846                 ttt += *ttt;
01847             }
01848             else {
01849                 if ( *ttt != -SNUMBER ) goto nospec;
01850                 ttt += 2;
01851             }
01852         }
01853     }
01854     /*
01855     Function has only numerical arguments
01856     Pick up the first argument.
01857     */
01858     ttt = t + FUNHEAD;
01859     if ( *ttt > 0 ) {
01860         loadnew1:
01861         for ( iiii = 0; iiii < ttt[ARGHEAD]; iiii++ )
01862             n_llnum[iiii] = ttt[ARGHEAD+iiii];
01863         ttt += *ttt;
01864     }
01865     else {
01866         loadnew2:
01867         if ( ttt[1] == 0 ) {
01868             n_llnum[0] = n_llnum[1] = n_llnum[2] = n_llnum[3] = 0;
01869         }
01870         else {
01871             n_llnum[0] = 4;
01872             if ( ttt[1] > 0 ) { n_llnum[1] = ttt[1]; n_llnum[3] = 3; }
01873             else { n_llnum[1] = -ttt[1]; n_llnum[3] = -3; }
01874             n_llnum[2] = 1;
01875         }
01876         ttt += 2;
01877     }
01878     /*
01879     Now loop over the other arguments
01880     */
01881     while ( ttt < tttstop ) {
01882         if ( *ttt > 0 ) {
01883             if ( n_llnum[0] == 0 ) {
01884                 if ( ( *t == MINFUNCTION && ttt[*ttt-1] < 0 )
01885                     || ( *t == MAXFUNCTION && ttt[*ttt-1] > 0 ) )
01886                     goto loadnew1;
01887             }
01888             else {
01889                 ttt += ARGHEAD;
01890                 iiii = CompCoef(n_llnum,ttt);
01891                 if ( ( iiii > 0 && *t == MINFUNCTION )
01892                     || ( iiii < 0 && *t == MAXFUNCTION ) ) {
01893                     for ( iiii = 0; iiii < ttt[0]; iiii++ )
01894                         n_llnum[iiii] = ttt[iiii];
01895                 }
01896                 ttt += *ttt;
01897             }
01898         }
01899         else {
01900             if ( n_llnum[0] == 0 ) {
01901                 if ( ( *t == MINFUNCTION && ttt[1] < 0 )
01902                     || ( *t == MAXFUNCTION && ttt[1] > 0 ) )
01903                     goto loadnew2;
01904             }
01905             else if ( ttt[1] == 0 ) {
01906                 if ( ( *t == MINFUNCTION && n_llnum[*n_llnum-1] > 0 )
01907                     || ( *t == MAXFUNCTION && n_llnum[*n_llnum-1] < 0 ) ) {
01908                     n_llnum[0] = 0;

```

```

01907         }
01908     }
01909     else {
01910         tterm[0] = 4; tterm[2] = 1;
01911         if ( ttt[1] < 0 ) { tterm[1] = -ttt[1]; tterm[3] = -3; }
01912         else { tterm[1] = ttt[1]; tterm[3] = 3; }
01913         iii = CompCoeef(n_llnum,tterm);
01914         if ( ( iii > 0 && *t == MINFUNCTION )
01915             || ( iii < 0 && *t == MAXFUNCTION ) ) {
01916             for ( iii = 0; iii < 4; iii++ )
01917                 n_llnum[iii] = tterm[iii];
01918         }
01919     }
01920     ttt += 2;
01921 }
01922 }
01923 if ( n_llnum[0] == 0 ) goto NormZero;
01924 ncoef = REDLENG(ncoef);
01925 nnum = REDLENG(n_llnum[*n_llnum-1]);
01926 if ( MulRat(BHEAD (UWORD *)n_coef,ncoef, (UWORD *)lnum,nnum,
01927 (UWORD *)n_coef,&ncoef) ) goto FromNorm;
01928 ncoef = INCLENG(ncoef);
01929 }
01930 break;
01931 case INVERSEFACTORIAL:
01932 if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] >= 0 ) {
01933     if ( Factorial(BHEAD t[FUNHEAD+1],(UWORD *)lnum,&nnum) )
01934         goto FromNorm;
01935     ncoef = REDLENG(ncoef);
01936     if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef, (UWORD *)lnum,nnum) ) goto FromNorm;
01937     ncoef = INCLENG(ncoef);
01938 }
01939 else {
01940 nospec:
01941     pcom[ncom++] = t;
01942 }
01943 break;
01944 case MAXPOWEROF:
01945 if ( ( t[FUNHEAD] == -SYMBOL )
01946     && ( t[FUNHEAD+1] > 0 ) && ( t[1] == FUNHEAD+2 ) ) {
01947     *(UWORD *)lnum = symbols[t[FUNHEAD+1]].maxpower;
01948     nnum = 1;
01949     goto MullIn;
01950 }
01951 else { pcom[ncom++] = t; }
01952 break;
01953 case MINPOWEROF:
01954 if ( ( t[FUNHEAD] == -SYMBOL )
01955     && ( t[FUNHEAD+1] > 0 ) && ( t[1] == FUNHEAD+2 ) ) {
01956     *(UWORD *)lnum = symbols[t[FUNHEAD+1]].minpower;
01957     nnum = 1;
01958     goto MullIn;
01959 }
01960 else { pcom[ncom++] = t; }
01961 break;
01962 case PRIMENUMBER :
01963 if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER
01964     && t[FUNHEAD+1] > 0 ) {
01965     UWORD xp = (UWORD)(NextPrime(BHEAD t[FUNHEAD+1]));
01966     ncoef = REDLENG(ncoef);
01967     if ( Mullly(BHEAD (UWORD *)n_coef,&ncoef,&xp,1) ) goto FromNorm;
01968     ncoef = INCLENG(ncoef);
01969 }
01970 else goto defaultcase;
01971 break;
01972 case MOEBIUS:
01973 /*
01974 Numerical function giving -1,0,1 or no evaluation
01975 */
01976 if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER
01977     && t[FUNHEAD+1] > 0 ) {
01978     WORD val = Moebius(BHEAD t[FUNHEAD+1]);
01979     if ( val == 0 ) goto NormZero;
01980     if ( val < 0 ) ncoef = -ncoef;
01981 }
01982 else {
01983     pcom[ncom++] = t;
01984 }
01985 break;
01986 case LNUMBER :
01987 ncoef = REDLENG(ncoef);
01988 if ( Mullly(BHEAD (UWORD *)n_coef,&ncoef, (UWORD *) (t+3),t[2]) ) goto FromNorm;
01989 ncoef = INCLENG(ncoef);
01990 break;
01991 case SNUMBER :
01992 if ( t[2] < 0 ) {
01993     t[2] = -t[2];
01994     if ( t[3] & 1 ) ncoef = -ncoef;

```

```

01994         }
01995         else if ( t[2] == 0 ) {
01996             if ( t[3] < 0 ) goto NormInf;
01997             goto NormZero;
01998         }
01999         lnum[0] = t[2];
02000         nnum = 1;
02001         if ( t[3] && RaisPow(BHEAD (UWORD *)lnum,&nnum, (UWORD) (ABS(t[3]))) ) goto FromNorm;
02002         ncoef = REDLENG(ncoef);
02003         if ( t[3] < 0 ) {
02004             if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef, (UWORD *)lnum,nnum) )
02005                 goto FromNorm;
02006         }
02007         else if ( t[3] > 0 ) {
02008             if ( Mully(BHEAD (UWORD *)n_coef,&ncoef, (UWORD *)lnum,nnum) )
02009                 goto FromNorm;
02010         }
02011         ncoef = INCLENG(ncoef);
02012         break;
02013     case GAMMA :
02014     case GAMMAI :
02015     case GAMMAFIVE :
02016     case GAMMASIX :
02017     case GAMMASEVEN :
02018         if ( t[1] == FUNHEAD ) {
02019             MLOCK(ErrorMessageLock);
02020             MesPrint("Gamma matrix without spin line encountered.");
02021             MUNLOCK(ErrorMessageLock);
02022             goto NormMin;
02023         }
02024         pnco[nnco++] = t;
02025         t += FUNHEAD+1;
02026         goto ScanCont;
02027     case LEVICIVITA :
02028         peps[neps++] = t;
02029         if ( ( t[2] & DIRTYFLAG ) == DIRTYFLAG ) {
02030             t[2] &= ~DIRTYFLAG;
02031             t[2] |= DIRTYSYMFLAG;
02032         }
02033         t += FUNHEAD;
02034     ScanCont:
02035         while ( t < r ) {
02036             if ( *t >= AM.OffsetIndex &&
02037                 ( *t >= AM.DumInd || ( *t < AM.WilInd &&
02038                     indices[*t-AM.OffsetIndex].dimension ) ) )
02039                 pcon[ncon++] = t;
02040             t++;
02041         }
02042         break;
02043     case EXPONENT :
02044         {
02045             WORD *rr;
02046             k = 1;
02047             rr = t + FUNHEAD;
02048             if ( *rr == ARGHEAD || ( *rr == -SNUMBER && rr[1] == 0 ) )
02049                 k = 0;
02050             if ( *rr == -SNUMBER && rr[1] == 1 ) break;
02051             if ( *rr <= -FUNCTION ) k = *rr;
02052             NEXTARG(rr)
02053             if ( *rr == ARGHEAD || ( *rr == -SNUMBER && rr[1] == 0 ) ) {
02054                 if ( k == 0 ) goto NormZZ;
02055                 break;
02056             }
02057             if ( *rr == -SNUMBER && rr[1] > 0 && rr[1] < MAXPOWER && k < 0 ) {
02058                 k = -k;
02059                 if ( functions[k-FUNCTION].commute ) {
02060                     for ( i = 0; i < rr[1]; i++ ) pnco[nnco++] = rr-1;
02061                 }
02062                 else {
02063                     for ( i = 0; i < rr[1]; i++ ) pcom[ncom++] = rr-1;
02064                 }
02065                 break;
02066             }
02067             if ( k == 0 ) goto NormZero;
02068             if ( t[FUNHEAD] == -SYMBOL && *rr == -SNUMBER && t[1] == FUNHEAD+4 ) {
02069                 if ( rr[1] < MAXPOWER ) {
02070                     t[FUNHEAD+2] = t[FUNHEAD+1]; t += FUNHEAD+2;
02071                     from = m;
02072                     goto NextSymbol;
02073                 }
02074             }
02075             /*
02076             if ( ( t[FUNHEAD] > 0 && t[FUNHEAD+1] != 0 )
02077                 || ( *rr > 0 && rr[1] != 0 ) ) {}
02078             else
02079                 /*
02080                 t[2] &= ~DIRTYSYMFLAG;

```

```

02081         pnco[ncco++] = t;
02082     }
02083     break;
02084 case DENOMINATOR :
02085     t[2] &= ~DIRTYSYMFLAG;
02086     pden[nden++] = t;
02087     pnco[ncco++] = t;
02088     break;
02089 case INDEX :
02090     t += 2;
02091     do {
02092         if ( *t == 0 || *t == AM.vectorzero ) goto NormZero;
02093         if ( *t > 0 && *t < AM.OffsetIndex ) {
02094             lnum[0] = *t++;
02095             nnum = 1;
02096             ncoef = REDLENG(ncoef);
02097             if ( Mully(BHEAD (UWORD *)n_coef,&ncoef, (UWORD *)lnum,nnum) )
02098                 goto FromNorm;
02099             ncoef = INCLENG(ncoef);
02100         }
02101         else if ( *t == NOINDEX ) t++;
02102         else pind[nind++] = *t++;
02103     } while ( t < r );
02104     break;
02105 case SUBEXPRESSION :
02106     if ( t[3] == 0 ) break;
02107 case EXPRESSION :
02108     goto RegEnd;
02109 case ROOTFUNCTION :
02110 /*
02111     Tries to take the n-th root inside the rationals
02112     If this is not possible, it clears all flags and
02113     hence tries no more.
02114     Notation:
02115     root_(power(=integer),(rational)number)
02116 */
02117     { WORD nc;
02118     if ( t[2] == 0 ) goto defaultcase;
02119     if ( t[FUNHEAD] != -SNUMBER || t[FUNHEAD+1] < 0 ) goto defaultcase;
02120     if ( t[FUNHEAD+2] == -SNUMBER ) {
02121         if ( t[FUNHEAD+1] == 0 && t[FUNHEAD+3] == 0 ) goto NormZZ;
02122         if ( t[FUNHEAD+1] == 0 ) break;
02123         if ( t[FUNHEAD+3] < 0 ) {
02124             AT.WorkPointer[0] = -t[FUNHEAD+3];
02125             nc = -1;
02126         }
02127         else {
02128             AT.WorkPointer[0] = t[FUNHEAD+3];
02129             nc = 1;
02130         }
02131         AT.WorkPointer[1] = 1;
02132     }
02133     else if ( t[FUNHEAD+2] == t[1]-FUNHEAD-2
02134     && t[FUNHEAD+2] == t[FUNHEAD+2+ARGHEAD]+ARGHEAD
02135     && ABS(t[t[1]-1]) == t[FUNHEAD+2+ARGHEAD] - 1 ) {
02136         WORD *r1, *r2;
02137         if ( t[FUNHEAD+1] == 0 ) break;
02138         i = t[t[1]-1]; r1 = t + FUNHEAD+ARGHEAD+3;
02139         nc = REDLENG(i);
02140         i = ABS(i) - 1;
02141         r2 = AT.WorkPointer;
02142         while ( --i >= 0 ) *r2++ = *r1++;
02143     }
02144     else goto defaultcase;
02145     if ( TakeRatRoot((UWORD *)AT.WorkPointer,&nc,t[FUNHEAD+1]) ) {
02146         t[2] = 0;
02147         goto defaultcase;
02148     }
02149     ncoef = REDLENG(ncoef);
02150     if ( Mully(BHEAD (UWORD *)n_coef,&ncoef, (UWORD *)AT.WorkPointer,nc) )
02151         goto FromNorm;
02152     if ( nc < 0 ) nc = -nc;
02153     if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef, (UWORD *) (AT.WorkPointer+nc),nc) )
02154         goto FromNorm;
02155     ncoef = INCLENG(ncoef);
02156     }
02157     break;
02158 case RANDOMFUNCTION :
02159     {
02160     WORD nnc, nc, nca, nr;
02161     UWORD xx;
02162 /*
02163     Needs one positive integer argument.
02164     returns (wranf()%argument)+1.
02165     We may call wranf several times to paste UWORDS together
02166     when we need long numbers.
02167     We make little errors when taking the % operator

```

```

02168                (not 100% uniform). We correct for that by redoing the
02169                calculation in the (unlikely) case that we are in leftover area
02170 */
02171         if ( t[1] == FUNHEAD ) goto defaultcase;
02172         if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER &&
02173             t[FUNHEAD+1] > 0 ) {
02174             if ( t[FUNHEAD+1] == 1 ) break;
02175         redoshort:
02176             ((UWORD *)AT.WorkPointer)[0] = wranf(BHEAD0);
02177             ((UWORD *)AT.WorkPointer)[1] = wranf(BHEAD0);
02178             nr = 2;
02179             if ( ((UWORD *)AT.WorkPointer)[1] == 0 ) {
02180                 nr = 1;
02181                 if ( ((UWORD *)AT.WorkPointer)[0] == 0 ) {
02182                     nr = 0;
02183                 }
02184             }
02185             xx = (UWORD) (t[FUNHEAD+1]);
02186             if ( nr ) {
02187                 DivLong((UWORD *)AT.WorkPointer,nr
02188                     ,&xx,1
02189                     ,((UWORD *)AT.WorkPointer)+4,&nnc
02190                     ,((UWORD *)AT.WorkPointer)+2,&nc);
02191                 ((UWORD *)AT.WorkPointer)[4] = 0;
02192                 ((UWORD *)AT.WorkPointer)[5] = 0;
02193                 ((UWORD *)AT.WorkPointer)[6] = 1;
02194                 DivLong((UWORD *)AT.WorkPointer+4,3
02195                     ,&xx,1
02196                     ,((UWORD *)AT.WorkPointer)+9,&nnc
02197                     ,((UWORD *)AT.WorkPointer)+7,&nca);
02198                 AddLong((UWORD *)AT.WorkPointer+4,3
02199                     ,((UWORD *)AT.WorkPointer)+7,-nca
02200                     ,((UWORD *)AT.WorkPointer)+9,&nnc);
02201                 if ( BigLong((UWORD *)AT.WorkPointer,nr
02202                     ,((UWORD *)AT.WorkPointer)+9,nnc) >= 0 ) goto redoshort;
02203             }
02204             else nc = 0;
02205             if ( nc == 0 ) {
02206                 AT.WorkPointer[2] = (WORD)xx;
02207                 nc = 1;
02208             }
02209             ncoef = REDLENG(ncoef);
02210             if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,((UWORD *)AT.WorkPointer))+2,nc) )
02211                 goto FromNorm;
02212             ncoef = INCLENG(ncoef);
02213         }
02214         else if ( t[FUNHEAD] > 0 && t[1] == t[FUNHEAD]+FUNHEAD
02215             && ABS(t[t[1]-1]) == t[FUNHEAD]-1-ARGHEAD && t[t[1]-1] > 0 ) {
02216             WORD nna, nnb, nni, nnb2, nnb2a;
02217             UWORD *nnt;
02218             nna = t[t[1]-1];
02219             nnb2 = nna-1;
02220             nnb = nnb2/2;
02221             nnt = (UWORD *) (t+t[1]-1-nnb); /* start of denominator */
02222             if ( *nnt != 1 ) goto defaultcase;
02223             for ( nni = 1; nni < nnb; nni++ ) {
02224                 if ( nnt[nni] != 0 ) goto defaultcase;
02225             }
02226             nnt = (UWORD *) (t + FUNHEAD + ARGHEAD + 1);
02227
02228             for ( nni = 0; nni < nnb2; nni++ ) {
02229                 ((UWORD *)AT.WorkPointer)[nni] = wranf(BHEAD0);
02230             }
02231             nnb2a = nnb2;
02232             while ( nnb2a > 0 && ((UWORD *)AT.WorkPointer)[nnb2a-1] == 0 ) nnb2a--;
02233             if ( nnb2a > 0 ) {
02234                 DivLong((UWORD *)AT.WorkPointer,nnb2a
02235                     ,nnt,nnb
02236                     ,((UWORD *)AT.WorkPointer)+2*nnb2,&nnc
02237                     ,((UWORD *)AT.WorkPointer)+nnb2,&nc);
02238                 for ( nni = 0; nni < nnb2; nni++ ) {
02239                     ((UWORD *)AT.WorkPointer)[nni+2*nnb2] = 0;
02240                 }
02241                 ((UWORD *)AT.WorkPointer)[3*nnb2] = 1;
02242                 DivLong((UWORD *)AT.WorkPointer+2*nnb2,nnb2+1
02243                     ,nnt,nnb
02244                     ,((UWORD *)AT.WorkPointer)+4*nnb2+1,&nnc
02245                     ,((UWORD *)AT.WorkPointer)+3*nnb2+1,&nca);
02246                 AddLong((UWORD *)AT.WorkPointer+2*nnb2,nnb2+1
02247                     ,((UWORD *)AT.WorkPointer)+3*nnb2+1,-nca
02248                     ,((UWORD *)AT.WorkPointer)+4*nnb2+1,&nnc);
02249                 if ( BigLong((UWORD *)AT.WorkPointer,nnb2a
02250                     ,((UWORD *)AT.WorkPointer)+4*nnb2+1,nnc) >= 0 ) goto redoshort;
02251             }
02252             else nc = 0;
02253             if ( nc == 0 ) {
02254                 for ( nni = 0; nni < nnb; nni++ ) {

```

```

02255             ((UWORD *)AT.WorkPointer)[nnb2+nni] = nnt[nni];
02256         }
02257         nc = nnb;
02258     }
02259     ncoef = REDLENG(ncoef);
02260     if ( Mully(BHEAD (UWORD *)n_coef,&ncoef, ((UWORD *) (AT.WorkPointer))+nnb2,nc) )
02261         goto FromNorm;
02262     ncoef = INCLENG(ncoef);
02263 }
02264 else goto defaultcase;
02265 }
02266 break;
02267 case RANPERM :
02268     if ( *t == RANPERM && t[1] > FUNHEAD && t[FUNHEAD] <= -FUNCTION ) {
02269         WORD **pwork;
02270         WORD *mm, *ww, *ow = AT.WorkPointer;
02271         WORD *Array, *targ, *argstop, nargs = 0, itot;
02272         int ie;
02273         argstop = t+t[1];
02274         targ = t+FUNHEAD+1;
02275         while ( targ < argstop ) {
02276             nargs++; NEXTARG(targ);
02277         }
02278         WantAddPointers(nargs);
02279         pwork = AT.pWorkSpace + AT.pWorkPointer;
02280         targ = t+FUNHEAD+1; nargs = 0;
02281         while ( targ < argstop ) {
02282             pwork[nargs++] = targ;
02283             NEXTARG(targ);
02284         }
02285         /*
02286          Make a random permutation of the numbers 0,...,narg-1
02287          The following code works also for nargs == 0 and nargs == 1
02288         */
02289         ow = AT.WorkPointer;
02290         Array = AT.WorkPointer;
02291         AT.WorkPointer += nargs;
02292         for ( i = 0; i < nargs; i++ ) Array[i] = i;
02293         for ( i = 2; i <= nargs; i++ ) {
02294             itot = (WORD) (iranf(BHEAD i));
02295             for ( j = 0; j < itot; j++ ) CYCLE1(WORD,Array,i)
02296         }
02297         mm = AT.WorkPointer;
02298         *mm++ = -t[FUNHEAD];
02299         *mm++ = t[1] - 1;
02300         for ( ie = 2; ie < FUNHEAD; ie++ ) *mm++ = t[ie];
02301         for ( i = 0; i < nargs; i++ ) {
02302             ww = pwork[Array[i]];
02303             CopyArg(mm,ww);
02304         }
02305         mm = AT.WorkPointer; t++; ww = t;
02306         i = mm[1]; NCOPY(ww,mm,i)
02307         AT.WorkPointer = ow;
02308         goto TryAgain;
02309     }
02310     pnco[nnco++] = t;
02311     break;
02312 case PUTFIRST : /* First argument should be a function, second a number */
02313     if ( ( t[2] & DIRTYFLAG ) != 0 && t[FUNHEAD] <= -FUNCTION
02314         && t[FUNHEAD+1] == -SNUMBER && t[FUNHEAD+2] > 0 ) {
02315         WORD *rr = t+t[1], *mm = t+FUNHEAD+3, *tt, *ttl, *tt2, num = 0;
02316         /*
02317          now count the arguments. If not enough: no action.
02318         */
02319         while ( mm < rr ) { num++; NEXTARG(mm); }
02320         if ( num < t[FUNHEAD+2] ) { pnco[nnco++] = t; break; }
02321         /* Replace "putfirst_" with the resulting function code. mm goes to arg start. */
02322         *t = -t[FUNHEAD]; mm = t+FUNHEAD+3;
02323         /* Set i to the arg number we are putting first, then move mm to its start. */
02324         i = t[FUNHEAD+2];
02325         while ( --i > 0 ) { NEXTARG(mm); }
02326         /* Keep a pointer to the arguments trailing the selected: */
02327         WORD *argTail = mm;
02328         NEXTARG(argTail);
02329         tt = TermMalloc("Select_"); /* Move selected out of the way into tmp space */
02330         ttl = tt;
02331         /* Normal argument: */
02332         if ( *mm > 0 ) {
02333             for ( i = 0; i < *mm; i++ ) *ttl++ = mm[i];
02334         }
02335         /* Fast-notation function: single word */
02336         else if ( *mm <= -FUNCTION ) { *ttl++ = *mm; }
02337         /* Fast-notation symbol, number etc: two words */
02338         else { *ttl++ = mm[0]; *ttl++ = mm[1]; }
02339         /* Put tt2 at the start of the original arguments */
02340         tt2 = t+FUNHEAD+3;
02341         /* Copy leading original arguments after the new first, in the tmp space. */

```



```

02342         while ( tt2 < mm ) *tt1++ = *tt2++;
02343         /* i contains the size so far. tt1 goes to the start of the tmp space.
02344            tt2 to the final argument location (we overwrite "putfirst_") */
02345         i = tt1-tt; tt1 = tt; tt2 = t+FUNHEAD;
02346         /* Copy everything so far to its final place */
02347         NCOPY(tt2,tt1,i);
02348         /* We are finished with the tmp space */
02349         TermFree(tt,"Select_");
02350         /* Now copy the trailing args. Use the stored pointer, *mm has been edited
02351            during the copy to tt2 above! NEXTARG(mm) would produce nonsense. */
02352         while ( argTail < rr ) *tt2++ = *argTail++;
02353         /* Set the size of all function args */
02354         t[1] = tt2 - t;
02355         /* Now copy the rest of the term, and set the final term size */
02356         rr = term + *term;
02357         while ( argTail < rr ) *tt2++ = *argTail++;
02358         *term = tt2-term;
02359         if ( functions[*t-FUNCTION].spec == TENSORFUNCTION ) {
02360             // If the output function is a tensor, we have one more job to do.
02361             // The args are formatted as single words, representing indices or vectors.
02362             // There is no ARGHEAD.
02363             WORD *dst = t + FUNHEAD;
02364             WORD *src = dst + 1;
02365             // Strip the type information from the args:
02366             while ( src < t + t[1] ) {
02367                 *dst = *src;
02368                 dst++; src++;
02369             }
02370             t[1] = dst - t;
02371             // Now copy the rest of the term. We've advanced src one too many above.
02372             src--;
02373             while ( src < term + *term ) {
02374                 *dst++ = *src++;
02375             }
02376             *term = dst - term;
02377         }
02378         goto Restart;
02379     }
02380     else pnco[nnco++] = t;
02381     break;
02382 #ifdef WITHFLOAT
02383     case FLOATFUN :
02384     /*
02385         If it is a proper float_ we give it special treatment.
02386         If it is not proper, we treat it as a regular commuting function.
02387     */
02388     if ( withfloat == 0 ) {
02389         if ( TestFloat(t) == 0 ) goto defaultcase;
02390         firstfloat = t;
02391         withfloat = 1;
02392     }
02393     else { /* There are now at least two float_'s. Time for action. */
02394         if ( TestFloat(t) == 0 ) goto defaultcase;
02395         if ( withfloat == 1 ) UnpackFloat(aux4,firstfloat);
02396         withfloat++;
02397         UnpackFloat(aux5,t);
02398         mpf_mul(aux4,aux4,aux5);
02399     }
02400     break;
02401 #endif
02402     case INTFUNCTION :
02403     /*
02404         Can be resolved if the first argument is a number
02405         and the second argument either doesn't exist or has
02406         the value +1, 0, -1
02407         +1 : rounding up
02408         0  : rounding towards zero
02409         -1 : rounding down (same as no argument)
02410     */
02411     if ( t[1] <= FUNHEAD ) break;
02412     {
02413         WORD *rr, den, num;
02414         to = t + FUNHEAD;
02415         if ( *to > 0 ) {
02416             if ( *to == ARGHEAD ) goto NormZero;
02417             rr = to + *to;
02418             i = rr[-1];
02419             j = ABS(i);
02420             if ( to[ARGHEAD] != j+1 ) goto NoInteg;
02421             if ( rr >= r ) k = -1;
02422             else if ( *rr == ARGHEAD ) { k = 0; rr += ARGHEAD; }
02423             else if ( *rr == -SNUMBER ) { k = rr[1]; rr += 2; }
02424             else goto NoInteg;
02425             if ( rr != r ) goto NoInteg;
02426             if ( k > 1 || k < -1 ) goto NoInteg;
02427             to += ARGHEAD+1;
02428             j = (j-1) >> 1;

```

```

02429         i = ( i < 0 ) ? -j: j;
02430         UnPack((UWORD *)to,i,&den,&num);
02431     /*
02432     Potentially the use of NoScrat2 is unsafe.
02433     It makes the routine not reentrant, but because it is
02434     used only locally and because we only call the
02435     low level routines DivLong and AddLong which never
02436     make calls involving Normalize, things are OK after all
02437 */
02438     if ( AN.NoScrat2 == 0 ) {
02439         AN.NoScrat2 = (UWORD *)Malloc1((AM.MaxTal+2)*sizeof(UWORD),"Normalize");
02440     }
02441     if ( DivLong((UWORD *)to,num,(UWORD *) (to+j),den
02442     ,(UWORD *)AT.WorkPointer,&num,AN.NoScrat2,&den) ) goto FromNorm;
02443     if ( k < 0 && den < 0 ) {
02444         *AN.NoScrat2 = 1;
02445         den = -1;
02446         if ( AddLong((UWORD *)AT.WorkPointer,num
02447         ,AN.NoScrat2,den,(UWORD *)AT.WorkPointer,&num) )
02448             goto FromNorm;
02449     }
02450     else if ( k > 0 && den > 0 ) {
02451         *AN.NoScrat2 = 1;
02452         den = 1;
02453         if ( AddLong((UWORD *)AT.WorkPointer,num,
02454         AN.NoScrat2,den,(UWORD *)AT.WorkPointer,&num) )
02455             goto FromNorm;
02456     }
02457 }
02458 else if ( *to == -SNUMBER ) { /* No rounding needed */
02459     if ( to[1] < 0 ) { *AT.WorkPointer = -to[1]; num = -1; }
02460     else if ( to[1] == 0 ) goto NormZero;
02461     else { *AT.WorkPointer = to[1]; num = 1; }
02462 }
02463 else goto NoInteg;
02464 if ( num == 0 ) goto NormZero;
02465 ncoef = REDLENG(ncoef);
02466 if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)AT.WorkPointer,num) )
02467     goto FromNorm;
02468 ncoef = INCLENG(ncoef);
02469 break;
02470 }
02471 }
02472 NoInteg;;
02473 /*
02474     Fall through if it cannot be resolved
02475 */
02476     default :
02477 defaultcase;;
02478     if ( *t < FUNCTION ) {
02479         MLOCK(ErrorMessageLock);
02480         MesPrint("Illegal code in Norm");
02481 #ifdef DEBUGON
02482     {
02483         UBYTE OutBuf[140];
02484         AO.OutFill = AO.OutputLine = OutBuf;
02485         t = term;
02486         AO.OutSkip = 3;
02487         FiniLine();
02488         i = *t;
02489         while ( --i >= 0 ) {
02490             TalToLine((UWORD)(t++));
02491             TokenToLine((UBYTE *)" ");
02492         }
02493         AO.OutSkip = 0;
02494         FiniLine();
02495     }
02496 #endif
02497     MUNLOCK(ErrorMessageLock);
02498     goto NormMin;
02499 }
02500 if ( *t == REPLACEMENT ) {
02501     if ( AR.Eside != LHSIDE ) ReplaceVeto--;
02502     pcom[ncom++] = t;
02503     break;
02504 }
02505 /*
02506 if ( *t == AM.termfunnum && t[1] == FUNHEAD+2
02507 && t[FUNHEAD] == -DOLLAREXPRESSION ) termflag++;
02508 */
02509 if ( *t == DUMMYFUN || *t == DUMMYTEN ) {}
02510 else {
02511     if ( *t < (FUNCTION + WILDOFFSET) ) {
02512         if ( ( functions[*t-FUNCTION].maxnumargs > 0 )
02513             || ( functions[*t-FUNCTION].minnumargs > 0 ) )
02514             && ( t[2] & DIRTYFLAG ) != 0 ) {
02515             /*

```

```

02516                                     Number of arguments is bounded. And we have not checked.
02517 */
02518                                     WORD *ta = t + FUNHEAD, *tb = t + t[1];
02519                                     int numarg = 0;
02520                                     while ( ta < tb ) { numarg++; NEXTARG(ta) }
02521                                     if ( ( functions[*t-FUNCTION].maxnumargs > 0 )
02522                                         && ( numarg >= functions[*t-FUNCTION].maxnumargs ) )
02523                                         goto NormZero;
02524                                     if ( ( functions[*t-FUNCTION].minnumargs > 0 )
02525                                         && ( numarg < functions[*t-FUNCTION].minnumargs ) )
02526                                         goto NormZero;
02527                                     }
02528 doflags:
02529                                     if ( ( ( t[2] & DIRTYFLAG ) != 0 ) && ( functions[*t-FUNCTION].tabl == 0 ) ) {
02530                                         t[2] &= ~DIRTYFLAG;
02531                                         t[2] |= DIRTYSYMFLAG;
02532                                     }
02533                                     if ( functions[*t-FUNCTION].commute ) { pnco[ncco++] = t; }
02534                                     else { pcom[ncom++] = t; }
02535                                     }
02536                                     else {
02537                                         if ( ( ( t[2] & DIRTYFLAG ) != 0 ) && ( functions[*t-FUNCTION-WILDOFFSET].tabl
02538 == 0 ) ) {
02539                                             t[2] &= ~DIRTYFLAG;
02540                                             t[2] |= DIRTYSYMFLAG;
02541                                         }
02542                                         if ( functions[*t-FUNCTION-WILDOFFSET].commute ) {
02543                                             pnco[ncco++] = t;
02544                                         }
02545                                         else { pcom[ncom++] = t; }
02546                                     }
02547                                     }
02548                                     /* Now hunt for contractible indices */
02549
02550                                     if ( ( *t < (FUNCTION + WILDOFFSET)
02551                                         && functions[*t-FUNCTION].spec >= TENSORFUNCTION ) || (
02552                                         *t >= (FUNCTION + WILDOFFSET)
02553                                         && functions[*t-FUNCTION-WILDOFFSET].spec >= TENSORFUNCTION ) ) {
02554                                         if ( *t >= GAMMA && *t <= GAMMASEVEN ) t++;
02555                                         t += FUNHEAD;
02556                                         while ( t < r ) {
02557                                             if ( *t == AM.vectorzero ) goto NormZero;
02558                                             if ( *t >= AM.OffsetIndex && ( *t >= AM.DumInd
02559 || ( *t < AM.WilInd && indices[*t-AM.OffsetIndex].dimension ) ) ) {
02560                                                 pcon[ncon++] = t;
02561                                             }
02562                                             else if ( *t == FUNNYWILD ) { t++; }
02563                                             t++;
02564                                         }
02565                                     }
02566                                     else {
02567                                         t += FUNHEAD;
02568                                         while ( t < r ) {
02569                                             if ( *t > 0 ) {
02570 /*
02571                                     Here we should worry about a recursion
02572                                     A problem is the possibility of a construct
02573                                     like f(mu+nu)
02574 */
02575                                             t += *t;
02576                                         }
02577                                         else if ( *t <= -FUNCTION ) t++;
02578                                         else if ( *t == -INDEX ) {
02579                                             if ( t[1] >= AM.OffsetIndex &&
02580                                                 ( t[1] >= AM.DumInd || ( t[1] < AM.WilInd
02581                                                 && indices[t[1]-AM.OffsetIndex].dimension ) ) )
02582                                                 pcon[ncon++] = t+1;
02583                                             t += 2;
02584                                         }
02585                                         else if ( *t == -SYMBOL ) {
02586                                             if ( t[1] >= MAXPOWER && t[1] < 2*MAXPOWER ) {
02587                                                 *t = -SNUMBER;
02588                                                 t[1] -= MAXPOWER;
02589                                             }
02590                                             else if ( t[1] < -MAXPOWER && t[1] > -2*MAXPOWER ) {
02591                                                 *t = -SNUMBER;
02592                                                 t[1] += MAXPOWER;
02593                                             }
02594                                             else t += 2;
02595                                         }
02596                                         else t += 2;
02597                                     }
02598                                     }
02599                                     break;
02600                                     }
02601                                     t = r;

```

```

02602 TryAgain;;
02603     } while ( t < m );
02604     if ( ANsc ) {
02605         AN.cTerm = ANsc;
02606         r = t = ANsr; m = ANsm;
02607         ANsc = ANsm = ANsr = 0;
02608         goto conscan;
02609     }
02610 /*
02611 #] First scan :
02612 #[ Easy denominators :
02613
02614 Easy denominators are denominators that can be replaced by
02615 negative powers of individual subterms. This may add to all
02616 our sublists.
02617
02618 */
02619     if ( nden ) {
02620         for ( k = 0, i = 0; i < nden; i++ ) {
02621             t = pden[i];
02622             if ( ( t[2] & DIRTYFLAG ) == 0 ) continue;
02623             r = t + t[1]; m = t + FUNHEAD;
02624             if ( m >= r ) {
02625                 for ( j = i+1; j < nden; j++ ) pden[j-1] = pden[j];
02626                 nden--;
02627                 for ( j = 0; j < nnco; j++ ) if ( pnco[j] == t ) break;
02628                 for ( j++; j < nnco; j++ ) pnco[j-1] = pnco[j];
02629                 nnco--;
02630                 i--;
02631             }
02632             else {
02633                 NEXTARG(m);
02634                 if ( m >= r ) continue;
02635             /*
02636                 We have more than one argument. Split the function.
02637             */
02638                 if ( k == 0 ) {
02639                     k = 1; to = termout; from = term;
02640                 }
02641                 while ( from < t ) *to++ = *from++;
02642                 m = t + FUNHEAD;
02643                 while ( m < r ) {
02644                     stop = to;
02645                     *to++ = DENOMINATOR;
02646                     for ( j = 1; j < FUNHEAD; j++ ) *to++ = 0;
02647                     if ( *m < -FUNCTION ) *to++ = *m++;
02648                     else if ( *m < 0 ) { *to++ = *m++; *to++ = *m++; }
02649                     else {
02650                         j = *m; while ( --j >= 0 ) *to++ = *m++;
02651                     }
02652                     stop[1] = WORDDIF(to,stop);
02653                 }
02654                 from = r;
02655                 if ( i == nden - 1 ) {
02656                     stop = term + *term;
02657                     while ( from < stop ) *to++ = *from++;
02658                     i = *termout = WORDDIF(to,termout);
02659                     to = term; from = termout;
02660                     while ( --i >= 0 ) *to++ = *from++;
02661                     goto Restart;
02662                 }
02663             }
02664         }
02665         for ( i = 0; i < nden; i++ ) {
02666             t = pden[i];
02667             if ( ( t[2] & DIRTYFLAG ) == 0 ) continue;
02668             t[2] = 0;
02669             if ( t[FUNHEAD] == -SYMBOL ) {
02670                 WORD change;
02671                 t += FUNHEAD+1;
02672                 change = ExtraSymbol(*t,-1,nsym,ppsym,&ncoef);
02673                 nsym += change;
02674                 ppsym += change * 2;
02675                 goto DropDen;
02676             }
02677             else if ( t[FUNHEAD] == -SNUMBER ) {
02678                 t += FUNHEAD+1;
02679                 if ( *t == 0 ) goto NormInf;
02680                 if ( *t < 0 ) { *AT.WorkPointer = -*t; j = -1; }
02681                 else { *AT.WorkPointer = *t; j = 1; }
02682                 ncoef = REDLENG(ncoef);
02683                 if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)AT.WorkPointer,j) )
02684                     goto FromNorm;
02685                 ncoef = INCLENG(ncoef);
02686                 goto DropDen;
02687             }
02688             else if ( t[FUNHEAD] == ARGHEAD ) goto NormInf;

```

```

02689         else if ( t[FUNHEAD] > 0 && t[FUNHEAD+ARGHEAD] ==
02690 t[FUNHEAD]-ARGHEAD ) {
02691             /* Only one term */
02692             r = t + t[1] - 1;
02693             t += FUNHEAD + ARGHEAD + 1;
02694             j = *r;
02695             m = r - ABS(*r) + 1;
02696             if ( j != 3 || ( ( *m != 1 ) || ( m[1] != 1 ) ) ) {
02697                 ncoef = REDLENG(ncoef);
02698                 if ( DivRat(BHEAD (UWORD *)n_coef,ncoef, (UWORD *)m,REDLENG(j), (UWORD
*)n_coef,&ncoef) ) goto FromNorm;
02699                 ncoef = INCLENG(ncoef);
02700                 j = ABS(j) - 3;
02701                 t[-FUNHEAD-ARGHEAD] -= j;
02702                 t[-ARGHEAD-1] -= j;
02703                 t[-1] -= j;
02704                 m[0] = m[1] = 1;
02705                 m[2] = 3;
02706             }
02707             while ( t < m ) {
02708                 r = t + t[1];
02709                 if ( *t == SYMBOL || *t == DOTPRODUCT ) {
02710                     k = t[1];
02711                     pden[i][1] -= k;
02712                     pden[i][FUNHEAD] -= k;
02713                     pden[i][FUNHEAD+ARGHEAD] -= k;
02714                     m -= k;
02715                     stop = m + 3;
02716                     tt = to = t;
02717                     from = r;
02718                     if ( *t == SYMBOL ) {
02719                         t += 2;
02720                         while ( t < r ) {
02721                             WORD change;
02722                             change = ExtraSymbol(*t,-t[1],nsym,ppsym,&ncoef);
02723                             nsym += change;
02724                             ppsym += change * 2;
02725                             t += 2;
02726                         }
02727                     }
02728                     else {
02729                         t += 2;
02730                         while ( t < r ) {
02731                             *ppdot++ = *t++;
02732                             *ppdot++ = *t++;
02733                             *ppdot++ = -*t++;
02734                             ndot++;
02735                         }
02736                     }
02737                     while ( to < stop ) *to++ = *from++;
02738                     r = tt;
02739                 }
02740 #ifdef WITHFLOAT
02741                 else if ( *t == FLOATFUN && TestFloat(t) ) {
02742                     k = t[1];
02743                     pden[i][1] -= k;
02744                     pden[i][FUNHEAD] -= k;
02745                     pden[i][FUNHEAD+ARGHEAD] -= k;
02746                     UnpackFloat(aux5,t);
02747                     if ( withfloat == 0 ) {
02748                         mpf_ui_div(aux4,1,aux5);
02749                         withfloat = 2;
02750                     }
02751                     else {
02752                         if ( withfloat == 1 ) UnpackFloat(aux4,firstfloat);
02753                         mpf_div(aux4,aux4,aux5);
02754                         withfloat++;
02755                     }
02756                     tt = to = t;
02757                     while ( r < m ) *to++ = *r++;
02758                     *to++ = 1; *to++ = 1; *to++ = 3;
02759                     if ( ncoef < 0 ) t[-1] = -t[-1];
02760                     m -= k;
02761                     stop = m + 3;
02762                     r = tt;
02763                 }
02764 #endif
02765                 t = r;
02766             }
02767             if ( pden[i][1] == 4+FUNHEAD+ARGHEAD ) {
02768 DropDen:
02769                 for ( j = 0; j < nnco; j++ ) {
02770                     if ( pden[i] == pnco[j] ) {
02771                         --nnco;
02772                         while ( j < nnco ) {
02773                             pnco[j] = pnco[j+1];
02774                             j++;

```

```

02775             }
02776             break;
02777         }
02778     }
02779     pden[i--] = pden[--nden];
02780 }
02781 }
02782 }
02783 }
02784 /*
02785 #] Easy denominators :
02786 #[ Index Contractions :
02787 */
02788 if ( ndel ) {
02789     t = pdel;
02790     for ( i = 0; i < ndel; i += 2 ) {
02791         if ( t[0] == t[1] ) {
02792             if ( t[0] == EMPTYINDEX ) {}
02793             else if ( *t < AM.OffsetIndex ) {
02794                 k = AC.FixIndices[*t];
02795                 if ( k < 0 ) { j = -1; k = -k; }
02796                 else if ( k > 0 ) j = 1;
02797                 else goto NormZero;
02798                 goto WithFix;
02799             }
02800             else if ( *t >= AM.DumInd ) {
02801                 k = AC.lDefDim;
02802                 if ( k ) goto docontract;
02803             }
02804             else if ( *t >= AM.WilInd ) {
02805                 k = indices[*t-AM.OffsetIndex-WILDOFFSET].dimension;
02806                 if ( k ) goto docontract;
02807             }
02808             else if ( ( k = indices[*t-AM.OffsetIndex].dimension ) != 0 ) {
02809 docontract:
02810                 if ( k > 0 ) {
02811                     j = 1;
02812                     shortnum = k;
02813                     ncoef = REDLENG(ncoef);
02814                     if ( Mully(BHEAD (UWORD *)n_coef,&ncoef, (UWORD *)(&shortnum),j) )
02815                         goto FromNorm;
02816                     ncoef = INCLENG(ncoef);
02817                 }
02818                 else {
02819                     WORD change;
02820                     change = ExtraSymbol((WORD)(-k), (WORD)1, nsym, ppsym, &ncoef);
02821                     nsym += change;
02822                     ppsym += change * 2;
02823                 }
02824                 t[1] = pdel[ndel-1];
02825                 t[0] = pdel[ndel-2];
02826 HaveCon:
02827                 ndel -= 2;
02828                 i -= 2;
02829             }
02830         }
02831         else {
02832             if ( *t < AM.OffsetIndex && t[1] < AM.OffsetIndex ) goto NormZero;
02833             j = *t - AM.OffsetIndex;
02834             if ( j >= 0 && ( ( *t >= AM.DumInd && AC.lDefDim )
02835 || ( *t < AM.WilInd && indices[j].dimension ) ) ) {
02836                 for ( j = i + 2, m = pdel+j; j < ndel; j += 2, m += 2 ) {
02837                     if ( *t == *m ) {
02838                         *t = m[1];
02839                         *m++ = pdel[ndel-2];
02840                         *m = pdel[ndel-1];
02841                         goto HaveCon;
02842                     }
02843                     else if ( *t == m[1] ) {
02844                         *t = *m;
02845                         *m++ = pdel[ndel-2];
02846                         *m = pdel[ndel-1];
02847                         goto HaveCon;
02848                     }
02849                 }
02850             }
02851             j = t[1]-AM.OffsetIndex;
02852             if ( j >= 0 && ( ( t[1] >= AM.DumInd && AC.lDefDim )
02853 || ( t[1] < AM.WilInd && indices[j].dimension ) ) ) {
02854                 for ( j = i + 2, m = pdel+j; j < ndel; j += 2, m += 2 ) {
02855                     if ( t[1] == *m ) {
02856                         t[1] = m[1];
02857                         *m++ = pdel[ndel-2];
02858                         *m = pdel[ndel-1];
02859                         goto HaveCon;
02860                     }
02861                     else if ( t[1] == m[1] ) {

```

```

02862             t[1] = *m;
02863             *m++ = pdel[ndel-2];
02864             *m = pdel[ndel-1];
02865             goto HaveCon;
02866         }
02867     }
02868 }
02869     t += 2;
02870 }
02871 }
02872 if ( ndel > 0 ) {
02873     if ( nvec ) {
02874         t = pdel;
02875         for ( i = 0; i < ndel; i++ ) {
02876             if ( *t >= AM.OffsetIndex && ( ( *t >= AM.DumInd && AC.lDefDim ) ||
02877                 ( *t < AM.WilInd && indices[*t-AM.OffsetIndex].dimension ) ) ) {
02878                 r = pvec + 1;
02879                 for ( j = 1; j < nvec; j += 2 ) {
02880                     if ( *r == *t ) {
02881                         if ( i & 1 ) {
02882                             *r = t[-1];
02883                             *t-- = pdel[--ndel];
02884                             i -= 2;
02885                         }
02886                         else {
02887                             *r = t[1];
02888                             t[1] = pdel[--ndel];
02889                             i--;
02890                         }
02891                         *t-- = pdel[--ndel];
02892                         break;
02893                     }
02894                     r += 2;
02895                 }
02896             }
02897             t++;
02898         }
02899     }
02900     if ( ndel > 0 && ncon ) {
02901         t = pdel;
02902         for ( i = 0; i < ndel; i++ ) {
02903             if ( *t >= AM.OffsetIndex && ( ( *t >= AM.DumInd && AC.lDefDim ) ||
02904                 ( *t < AM.WilInd && indices[*t-AM.OffsetIndex].dimension ) ) ) {
02905                 for ( j = 0; j < ncon; j++ ) {
02906                     if ( *pcon[j] == *t ) {
02907                         if ( i & 1 ) {
02908                             *pcon[j] = t[-1];
02909                             *t-- = pdel[--ndel];
02910                             i -= 2;
02911                         }
02912                         else {
02913                             *pcon[j] = t[1];
02914                             t[1] = pdel[--ndel];
02915                             i--;
02916                         }
02917                         *t-- = pdel[--ndel];
02918                         didcontr++;
02919                         r = pcon[j];
02920                         for ( j = 0; j < nnco; j++ ) {
02921                             m = pnco[j];
02922                             if ( r > m && r < m+m[1] ) {
02923                                 m[2] |= DIRTSYMFLAG;
02924                                 break;
02925                             }
02926                         }
02927                         for ( j = 0; j < ncom; j++ ) {
02928                             m = pcom[j];
02929                             if ( r > m && r < m+m[1] ) {
02930                                 m[2] |= DIRTSYMFLAG;
02931                                 break;
02932                             }
02933                         }
02934                         for ( j = 0; j < neps; j++ ) {
02935                             m = peps[j];
02936                             if ( r > m && r < m+m[1] ) {
02937                                 m[2] |= DIRTSYMFLAG;
02938                                 break;
02939                             }
02940                         }
02941                         break;
02942                     }
02943                 }
02944             }
02945             t++;
02946         }
02947     }
02948 }

```

```

02949     }
02950     if ( nvec ) {
02951         t = pvec + 1;
02952         for ( i = 3; i < nvec; i += 2 ) {
02953             k = *t - AM.OffsetIndex;
02954             if ( k >= 0 && ( ( *t > AM.DumInd && AC.lDefDim )
02955                 || ( *t < AM.WilInd && indices[k].dimension ) ) ) {
02956                 r = t + 2;
02957                 for ( j = i; j < nvec; j += 2 ) {
02958                     if ( *r == *t ) { /* Another dotproduct */
02959                         *ppdot++ = t[-1];
02960                         *ppdot++ = r[-1];
02961                         *ppdot++ = 1;
02962                         ndot++;
02963                         *r-- = pvec[--nvec];
02964                         *r = pvec[--nvec];
02965                         *t-- = pvec[--nvec];
02966                         *t-- = pvec[--nvec];
02967                         i -= 2;
02968                         break;
02969                     }
02970                     r += 2;
02971                 }
02972             }
02973             t += 2;
02974         }
02975         if ( nvec > 0 && ncon ) {
02976             t = pvec + 1;
02977             for ( i = 1; i < nvec; i += 2 ) {
02978                 k = *t - AM.OffsetIndex;
02979                 if ( k >= 0 && ( ( *t >= AM.DumInd && AC.lDefDim )
02980                     || ( *t < AM.WilInd && indices[k].dimension ) ) ) {
02981                     for ( j = 0; j < ncon; j++ ) {
02982                         if ( *pcon[j] == *t ) {
02983                             *pcon[j] = t[-1];
02984                             *t-- = pvec[--nvec];
02985                             *t-- = pvec[--nvec];
02986                             r = pcon[j];
02987                             pcon[j] = pcon[--ncon];
02988                             i -= 2;
02989                             for ( j = 0; j < nnco; j++ ) {
02990                                 m = pnco[j];
02991                                 if ( r > m && r < m+m[1] ) {
02992                                     m[2] |= DIRTYSYMFLAG;
02993                                     break;
02994                                 }
02995                             }
02996                             for ( j = 0; j < ncom; j++ ) {
02997                                 m = pcom[j];
02998                                 if ( r > m && r < m+m[1] ) {
02999                                     m[2] |= DIRTYSYMFLAG;
03000                                     break;
03001                                 }
03002                             }
03003                             for ( j = 0; j < neps; j++ ) {
03004                                 m = peps[j];
03005                                 if ( r > m && r < m+m[1] ) {
03006                                     m[2] |= DIRTYSYMFLAG;
03007                                     break;
03008                                 }
03009                             }
03010                             break;
03011                         }
03012                     }
03013                 }
03014                 t += 2;
03015             }
03016         }
03017     }
03018     /*
03019     #] Index Contractions :
03020     #[ NonCommuting Functions :
03021     */
03022     m = fillsetexp;
03023     if ( nnco ) {
03024         for ( i = 0; i < nnco; i++ ) {
03025             t = pnco[i];
03026             if ( ( *t >= (FUNCTION+WILDOFFSET)
03027                 && functions[*t-FUNCTION-WILDOFFSET].spec <= 0 )
03028                 || ( *t >= FUNCTION && *t < (FUNCTION + WILDOFFSET)
03029                 && functions[*t-FUNCTION].spec <= 0 ) ) {
03030                 DoRevert(t,m);
03031                 if ( didcontr ) {
03032                     r = t + FUNHEAD;
03033                     t += t[1];
03034                     while ( r < t ) {
03035                         if ( *r == -INDEX && r[1] >= 0 && r[1] < AM.OffsetIndex ) {

```



```

03036             *r = -SNUMBER;
03037             didcontr--;
03038             pnco[i][2] |= DIRTYSYMFLAG;
03039         }
03040         NEXTARG(r)
03041     }
03042 }
03043 }
03044 }
03045
03046 /* First should come the code for function properties. */
03047
03048 /* First we test for symmetric properties and the DIRTYSYMFLAG */
03049
03050 for ( i = 0; i < nnco; i++ ) {
03051     t = pnco[i];
03052     if ( *t > 0 && ( t[2] & DIRTYSYMFLAG ) && *t != DOLLAREXPRESSION ) {
03053         l = 0; /* to make the compiler happy */
03054         if ( ( *t >= (FUNCTION+WILDOFFSET)
03055             && ( l = functions[*t-FUNCTION-WILDOFFSET].symmetric ) > 0 )
03056             || ( *t >= FUNCTION && *t < (FUNCTION + WILDOFFSET)
03057             && ( l = functions[*t-FUNCTION].symmetric ) > 0 ) ) {
03058             if ( *t >= (FUNCTION+WILDOFFSET) ) {
03059                 *t -= WILDOFFSET;
03060                 j = FullSymmetrize(BHEAD t,l);
03061                 *t += WILDOFFSET;
03062             }
03063             else j = FullSymmetrize(BHEAD t,l);
03064             if ( ( l & ~REVERSEORDER ) == ANTISYMMETRIC ) {
03065                 if ( ( j & 2 ) != 0 ) goto NormZero;
03066                 if ( ( j & 1 ) != 0 ) ncoef = -ncoef;
03067             }
03068         }
03069         else t[2] &= ~DIRTYSYMFLAG;
03070     }
03071 }
03072
03073 /* Non commuting functions are then tested for commutation
03074 rules. If needed their order is exchanged. */
03075
03076 k = nnco - 1;
03077 for ( i = 0; i < k; i++ ) {
03078     j = i;
03079     while ( Commute(pnco[j],pnco[j+1]) ) {
03080         t = pnco[j]; pnco[j] = pnco[j+1]; pnco[j+1] = t;
03081         l = j+1;
03082         while ( l >= 0 && Commute(pnco[l],pnco[l+1]) ) {
03083             t = pnco[l]; pnco[l] = pnco[l+1]; pnco[l+1] = t;
03084             l--;
03085         }
03086         if ( ++j >= k ) break;
03087     }
03088 }
03089
03090 /* Finally they are written to output. gamma matrices
03091 are bundled if possible */
03092
03093 for ( i = 0; i < nnco; i++ ) {
03094     t = pnco[i];
03095     if ( *t == IDFUNCTION ) AN.idfunctionflag = 1;
03096     if ( *t >= GAMMA && *t <= GAMMASEVEN ) {
03097         WORD gtype;
03098         to = m;
03099         *m++ = GAMMA;
03100         m++;
03101         FILLFUN(m)
03102         *m++ = stype = t[FUNHEAD]; /* type of string */
03103         j = 0;
03104         nnum = 0;
03105         do {
03106             r = t + t[1];
03107             if ( *t == GAMMAFIVE ) {
03108                 gtype = GAMMA5; t += FUNHEAD; goto onegammamatrix; }
03109             else if ( *t == GAMMASIX ) {
03110                 gtype = GAMMA6; t += FUNHEAD; goto onegammamatrix; }
03111             else if ( *t == GAMMASEVEN ) {
03112                 gtype = GAMMA7; t += FUNHEAD; goto onegammamatrix; }
03113             t += FUNHEAD+1;
03114             while ( t < r ) {
03115                 gtype = *t;
03116 onegammamatrix:
03117                 if ( gtype == GAMMA5 ) {
03118                     if ( j == GAMMA1 ) j = GAMMA5;
03119                     else if ( j == GAMMA5 ) j = GAMMA1;
03120                     else if ( j == GAMMA7 ) ncoef = -ncoef;
03121                     if ( nnum & 1 ) ncoef = -ncoef;
03122                 }

```

```

03123         else if ( gtype == GAMMA6 || gtype == GAMMA7 ) {
03124             if ( nnum & 1 ) {
03125                 if ( gtype == GAMMA6 ) gtype = GAMMA7;
03126                 else gtype = GAMMA6;
03127             }
03128             if ( j == GAMMA1 ) j = gtype;
03129             else if ( j == GAMMA5 ) {
03130                 j = gtype;
03131                 if ( j == GAMMA7 ) ncoef = -ncoef;
03132             }
03133             else if ( j != gtype ) goto NormZero;
03134             else {
03135                 shortnum = 2;
03136                 ncoef = REDLENG(ncoef);
03137                 if ( Mully(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)(&shortnum),1) ) goto
FromNorm;
03138                 ncoef = INCLENG(ncoef);
03139             }
03140         }
03141         else {
03142             *m++ = gtype; nnum++;
03143         }
03144         t++;
03145     }
03146
03147     } while ( ( ++i < ncco ) && ( *(t = pcco[i]) >= GAMMA
&& *t <= GAMMASEVEN ) && ( t[FUNHEAD] == stype ) );
03148     i--;
03149     if ( j ) {
03150         k = WORDDIF(m,to) - FUNHEAD-1;
03151         r = m;
03152         from = m++;
03153         while ( --k >= 0 ) *from-- = *--r;
03154         *from = j;
03155     }
03156     to[1] = WORDDIF(m,to);
03157 }
03158
03159 else if ( *t < 0 ) {
03160     *m++ = -*t; *m++ = FUNHEAD; *m++ = 0;
03161     FILLFUN3(m)
03162 }
03163 else {
03164     if ( ( t[2] & DIRTYFLAG ) == DIRTYFLAG
&& *t != REPLACEMENT && *t != DOLLAREXPRESSSION
&& TestFunFlag(BHEAD t) ) ReplaceVeto = 1;
03165     k = t[1];
03166     NCOPY(m,t,k);
03167 }
03168 }
03169 }
03170 }
03171 }
03172 }
03173 /*
03174 #] NonCommuting Functions :
03175 #[ Commuting Functions :
03176 */
03177 if ( ncom ) {
03178     for ( i = 0; i < ncom; i++ ) {
03179         t = pcom[i];
03180         if ( ( *t >= (FUNCTION+WILDOFFSET)
&& functions[*t-FUNCTION-WILDOFFSET].spec <= 0 )
|| ( *t >= FUNCTION && *t < (FUNCTION + WILDOFFSET)
&& functions[*t-FUNCTION].spec <= 0 ) ) {
03181             DoRevert(t,m);
03182             if ( didcontr ) {
03183                 r = t + FUNHEAD;
03184                 t += t[1];
03185                 while ( r < t ) {
03186                     if ( *r == -INDEX && r[1] >= 0 && r[1] < AM.OffsetIndex ) {
03187                         *r = -SNUMBER;
03188                         didcontr--;
03189                         pcom[i][2] |= DIRTYSYMFLAG;
03190                     }
03191                     NEXTARG(r)
03192                 }
03193             }
03194         }
03195     }
03196 }
03197 }
03198 }
03199
03200 /* Now we test for symmetric properties and the DIRTYSYMFLAG */
03201
03202 for ( i = 0; i < ncom; i++ ) {
03203     t = pcom[i];
03204     if ( *t > 0 && ( t[2] & DIRTYSYMFLAG ) ) {
03205         l = 0; /* to make the compiler happy */
03206         if ( ( *t >= (FUNCTION+WILDOFFSET)
&& ( l = functions[*t-FUNCTION-WILDOFFSET].symmetric ) > 0 )
|| ( *t >= FUNCTION && *t < (FUNCTION + WILDOFFSET)

```

```

03209         && ( l = functions[*t-FUNCTION].symmetric ) > 0 ) ) {
03210             if ( *t >= (FUNCTION+WILDOFFSET) ) {
03211                 *t -= WILDOFFSET;
03212                 j = FullSymmetrize(BHEAD t,l);
03213                 *t += WILDOFFSET;
03214             }
03215             else j = FullSymmetrize(BHEAD t,l);
03216             if ( (l & ~REVERSEORDER) == ANTISYMMETRIC ) {
03217                 if ( ( j & 2 ) != 0 ) goto NormZero;
03218                 if ( ( j & 1 ) != 0 ) ncoef = -ncoef;
03219             }
03220         }
03221         else t[2] &= ~DIRTYSYMFLAG;
03222     }
03223 }
03224 /*
03225 Sort the functions
03226 From a purists point of view this can be improved.
03227 There are slow and fast arguments and no conversions are
03228 taken into account here.
03229 */
03230 for ( i = 1; i < ncom; i++ ) {
03231     for ( j = i; j > 0; j-- ) {
03232         WORD jj, kk;
03233         jj = j-1;
03234         t = pcom[jj];
03235         r = pcom[j];
03236         if ( *t < 0 ) {
03237             if ( *r < 0 ) { if ( *t >= *r ) goto NextI; }
03238             else { if ( -*t <= *r ) goto NextI; }
03239             goto jexch;
03240         }
03241         else if ( *r < 0 ) {
03242             if ( *t < -*r ) goto NextI;
03243             goto jexch;
03244         }
03245         else if ( *t != *r ) {
03246             if ( *t < *r ) goto NextI;
03247             t = pcom[j]; pcom[j] = pcom[jj]; pcom[jj] = t;
03248             continue;
03249         }
03250         if ( AC.properorderflag ) {
03251             if ( *t >= (FUNCTION+WILDOFFSET)
03252                 && functions[*t-FUNCTION-WILDOFFSET].spec >= TENSORFUNCTION )
03253                 || ( *t >= FUNCTION && *t < (FUNCTION + WILDOFFSET)
03254                     && functions[*t-FUNCTION].spec >= TENSORFUNCTION ) ) {}
03255             else {
03256                 WORD *s1, *s2, *ss1, *ss2;
03257                 s1 = t+FUNHEAD; s2 = r+FUNHEAD;
03258                 ss1 = t + t[1]; ss2 = r + r[1];
03259                 while ( s1 < ss1 && s2 < ss2 ) {
03260                     k = CompArg(s1,s2);
03261                     if ( k > 0 ) goto jexch;
03262                     if ( k < 0 ) goto NextI;
03263                     NEXTARG(s1)
03264                     NEXTARG(s2)
03265                 }
03266                 if ( s1 < ss1 ) goto jexch;
03267                 goto NextI;
03268             }
03269             k = t[1] - FUNHEAD;
03270             kk = r[1] - FUNHEAD;
03271             t += FUNHEAD;
03272             r += FUNHEAD;
03273             while ( k > 0 && kk > 0 ) {
03274                 if ( *t < *r ) goto NextI;
03275                 else if ( *t++ > *r++ ) goto jexch;
03276                 k--; kk--;
03277             }
03278             if ( k > 0 ) goto jexch;
03279             goto NextI;
03280         }
03281         else
03282         {
03283             k = t[1] - FUNHEAD;
03284             kk = r[1] - FUNHEAD;
03285             t += FUNHEAD;
03286             r += FUNHEAD;
03287             while ( k > 0 && kk > 0 ) {
03288                 if ( *t < *r ) goto NextI;
03289                 else if ( *t++ > *r++ ) goto jexch;
03290                 k--; kk--;
03291             }
03292             if ( k > 0 ) goto jexch;
03293             goto NextI;
03294         }
03295     }
}

```

```

03296 NextI;;
03297     }
03298     for ( i = 0; i < ncom; i++ ) {
03299         t = pcom[i];
03300         if ( *t == THETA || *t == THETA2 ) {
03301             if ( ( k = DoTheta(BHEAD t) ) == 0 ) goto NormZero;
03302             else if ( k < 0 ) {
03303                 k = t[1];
03304                 NCOPY(m,t,k);
03305             }
03306         }
03307         else if ( *t == DELTA2 || *t == DELTAP ) {
03308             if ( ( k = DoDelta(t) ) == 0 ) goto NormZero;
03309             else if ( k < 0 ) {
03310                 k = t[1];
03311                 NCOPY(m,t,k);
03312             }
03313         }
03314         else if ( *t == AR.PolyFunInv && AR.PolyFunType == 2 ) {
03315             /*
03316              If there are two arguments, exchange them, change the
03317              name of the function and go to dealing with PolyRatFun.
03318             */
03319             WORD *mm, *tt = t, numt = 0;
03320             tt += FUNHEAD;
03321             while ( tt < t+tt[1] ) { numt++; NEXTARG(tt) }
03322             if ( numt == 2 ) {
03323                 tt = t; mm = m; k = t[1];
03324                 NCOPY(mm,tt,k)
03325                 mm = m+FUNHEAD;
03326                 NEXTARG(mm);
03327                 tt = t+FUNHEAD;
03328                 if ( *mm < 0 ) {
03329                     if ( *mm <= -FUNCTION ) { *tt++ = *mm++; }
03330                     else { *tt++ = *mm++; *tt++ = *mm++; }
03331                 }
03332                 else {
03333                     k = *mm; NCOPY(tt,mm,k)
03334                 }
03335                 mm = m+FUNHEAD;
03336                 if ( *mm < 0 ) {
03337                     if ( *mm <= -FUNCTION ) { *tt++ = *mm++; }
03338                     else { *tt++ = *mm++; *tt++ = *mm++; }
03339                 }
03340                 else {
03341                     k = *mm; NCOPY(tt,mm,k)
03342                 }
03343                 *t = AR.PolyFun;
03344                 t[2] |= MUSTCLEANPRF;
03345                 goto regularratfun;
03346             }
03347         }
03348         else if ( *t == AR.PolyFun ) {
03349             if ( AR.PolyFunType == 1 ) { /* Regular PolyFun with one argument */
03350                 if ( t[FUNHEAD+1] == 0 && AR.Eside != LHSIDE &&
03351                     t[1] == FUNHEAD + 2 && t[FUNHEAD] == -SNUMBER ) goto NormZero;
03352                 if ( i > 0 && pcom[i-1][0] == AR.PolyFun ) {
03353                     if ( AN.PolyNormFlag == 0 ) {
03354                         AN.PolyNormFlag = 1;
03355                         AN.PolyFunTodo = 0;
03356                     }
03357                 }
03358                 k = t[1];
03359                 NCOPY(m,t,k);
03360             }
03361             else if ( AR.PolyFunType == 2 ) {
03362                 /*
03363                  PolyRatFun.
03364                  Regular type: Two arguments
03365                  Power expanded: One argument. Here to be treated as
03366                  AR.PolyFunType == 1, but with power cutoff.
03367                 */
03368                 regularratfun;
03369                 /*
03370                  First check for zeroes.
03371                 */
03372                 if ( t[FUNHEAD+1] == 0 && AR.Eside != LHSIDE &&
03373                     t[1] > FUNHEAD + 2 && t[FUNHEAD] == -SNUMBER ) {
03374                     u = t + FUNHEAD + 2;
03375                     if ( *u < 0 ) {
03376                         if ( *u <= -FUNCTION ) {}
03377                         else if ( t[1] == FUNHEAD+4 && t[FUNHEAD+2] == -SNUMBER
03378                             && t[FUNHEAD+3] == 0 ) goto NormPRF;
03379                         else if ( t[1] == FUNHEAD+4 ) goto NormZero;
03380                     }
03381                     else if ( t[1] == *u+FUNHEAD+2 ) goto NormZero;
03382                 }

```

```

03383         else {
03384             u = t+FUNHEAD; NEXTARG(u);
03385             if ( *u == -SNUMBER && u[1] == 0 ) goto NormInf;
03386         }
03387         if ( i > 0 && pcom[i-1][0] == AR.PolyFun ) AN.PolyNormFlag = 1;
03388         else if ( i < ncom-1 && pcom[i+1][0] == AR.PolyFun ) AN.PolyNormFlag = 1;
03389         k = t[1];
03390         if ( AN.PolyNormFlag ) {
03391             if ( AR.PolyFunExp == 0 ) {
03392                 AN.PolyFunTodo = 0;
03393                 NCOPY(m,t,k);
03394             }
03395             else if ( AR.PolyFunExp == 1 ) { /* get highest divergence */
03396                 if ( PolyFunMode == 0 ) {
03397                     NCOPY(m,t,k);
03398                     AN.PolyFunTodo = 1;
03399                 }
03400                 else {
03401                     WORD *mmm = m;
03402                     NCOPY(m,t,k);
03403                     if ( TreatPolyRatFun(BHEAD mmm) != 0 )
03404                         goto FromNorm;
03405                     m = mmm+mmm[1];
03406                 }
03407             }
03408             else {
03409                 if ( PolyFunMode == 0 ) {
03410                     NCOPY(m,t,k);
03411                     AN.PolyFunTodo = 1;
03412                 }
03413                 else {
03414                     WORD *mmm = m;
03415                     NCOPY(m,t,k);
03416                     if ( ExpandRat(BHEAD mmm) != 0 )
03417                         goto FromNorm;
03418                     m = mmm+mmm[1];
03419                 }
03420             }
03421         }
03422         else {
03423             if ( AR.PolyFunExp == 0 ) {
03424                 AN.PolyFunTodo = 0;
03425                 NCOPY(m,t,k);
03426             }
03427             else if ( AR.PolyFunExp == 1 ) { /* get highest divergence */
03428                 WORD *mmm = m;
03429                 NCOPY(m,t,k);
03430                 if ( TreatPolyRatFun(BHEAD mmm) != 0 )
03431                     goto FromNorm;
03432                 m = mmm+mmm[1];
03433             }
03434             else {
03435                 WORD *mmm = m;
03436                 NCOPY(m,t,k);
03437                 if ( ExpandRat(BHEAD mmm) != 0 )
03438                     goto FromNorm;
03439                 m = mmm+mmm[1];
03440             }
03441         }
03442     }
03443 }
03444 else if ( *t > 0 ) {
03445     if ( ( t[2] & DIRTYFLAG ) == DIRTYFLAG
03446         && *t != REPLACEMENT && TestFunFlag(BHEAD t) ) ReplaceVeto = 1;
03447     k = t[1];
03448     NCOPY(m,t,k);
03449 }
03450 else {
03451     *m++ = -*t; *m++ = FUNHEAD; *m++ = 0;
03452     FILLFUN3(m)
03453 }
03454 }
03455 }
03456 /*
03457 #] Commuting Functions :
03458 #[ Track Replace_ :
03459 */
03460 if ( ReplaceVeto < 0 ) {
03461 /*
03462     We found one (or more) replace_ functions and all other
03463     functions are 'clean' (no dirty flag).
03464     Now we check whether one of these functions can be used.
03465     Thus far the functions go from fillsetexp to m.
03466     Somewhere in there there are -ReplaceVeto occurrences of REPLACEMENT.
03467     Hunt for the first one that fits the bill.
03468     Note that replace_ is a commuting function.
03469 */

```

```

03470     WORD *ma = fillsetexp, *mb, *mc;
03471     while ( ma < m ) {
03472         mb = ma + ma[1];
03473         if ( *ma != REPLACEMENT ) {
03474             ma = mb;
03475             continue;
03476         }
03477         if ( *ma == REPLACEMENT && ReplaceType == -1 ) {
03478             mc = ma;
03479             ReplaceType = 0;
03480             if ( AN.RSsize < 2*ma[1]+SUBEXPSIZE ) {
03481                 if ( AN.ReplaceScrat ) M_free(AN.ReplaceScrat,"AN.ReplaceScrat");
03482                 AN.RSsize = 2*ma[1]+SUBEXPSIZE+40;
03483                 AN.ReplaceScrat = (WORD *)Malloc1((AN.RSsize+1)*sizeof(WORD),"AN.ReplaceScrat");
03484             }
03485             ma += FUNHEAD;
03486             ReplaceSub = AN.ReplaceScrat;
03487             ReplaceSub += SUBEXPSIZE;
03488             while ( ma < mb ) {
03489                 if ( *ma > 0 ) goto NoRep;
03490                 if ( *ma <= -FUNCTION ) {
03491                     *ReplaceSub++ = FUNTOFUN;
03492                     *ReplaceSub++ = 4;
03493                     *ReplaceSub++ = -*ma++;
03494                     if ( *ma > -FUNCTION ) goto NoRep;
03495                     *ReplaceSub++ = -*ma++;
03496                 }
03497                 else if ( ma+4 > mb ) goto NoRep;
03498                 else {
03499                     if ( *ma == -SYMBOL ) {
03500                         if ( ma[2] == -SYMBOL && ma+4 <= mb )
03501                             *ReplaceSub++ = SYMTOSYM;
03502                         else if ( ma[2] == -SNUMBER && ma+4 <= mb ) {
03503                             *ReplaceSub++ = SYMTONUM;
03504                             if ( ReplaceType == 0 ) {
03505                                 oldtoprhs = C->numrhs;
03506                                 oldcpointer = C->Pointer - C->Buffer;
03507                             }
03508                             ReplaceType = 1;
03509                         }
03510                         else if ( ma[2] == ARGHEAD && ma+2+ARGHEAD <= mb ) {
03511                             *ReplaceSub++ = SYMTONUM;
03512                             *ReplaceSub++ = 4;
03513                             *ReplaceSub++ = ma[1];
03514                             *ReplaceSub++ = 0;
03515                             ma += 2+ARGHEAD;
03516                             continue;
03517                         }
03518                     /*
03519                     Next is the subexpression. We have to test that
03520                     it isn't vector-like or index-like
03521                     */
03522                     else if ( ma[2] > 0 ) {
03523                         WORD *sstop, *ttstop, n;
03524                         ss = ma+2;
03525                         sstop = ss + *ss;
03526                         ss += ARGHEAD;
03527                         while ( ss < sstop ) {
03528                             tt = ss + *ss;
03529                             ttstop = tt - ABS(tt[-1]);
03530                             ss++;
03531                             while ( ss < ttstop ) {
03532                                 if ( *ss == INDEX ) goto NoRep;
03533                                 ss += ss[1];
03534                             }
03535                             ss = tt;
03536                         }
03537                         subtype = SYMTOSUB;
03538                         if ( ReplaceType == 0 ) {
03539                             oldtoprhs = C->numrhs;
03540                             oldcpointer = C->Pointer - C->Buffer;
03541                         }
03542                         ReplaceType = 1;
03543                         ss = AddRHS(AT.ebufnum,1);
03544                         tt = ma+2;
03545                         n = *tt - ARGHEAD;
03546                         tt += ARGHEAD;
03547                         while ( (ss + n + 10) > C->Top ) ss = DoubleCbuffer(AT.ebufnum,ss,14);
03548                         while ( --n >= 0 ) *ss++ = *tt++;
03549                         *ss++ = 0;
03550                         C->rhs[C->numrhs+1] = ss;
03551                         C->Pointer = ss;
03552                         *ReplaceSub++ = subtype;
03553                         *ReplaceSub++ = 4;
03554                         *ReplaceSub++ = ma[1];
03555                         *ReplaceSub++ = C->numrhs;
03556                         ma += 2 + ma[2];

```

```

03557         continue;
03558     }
03559     else goto NoRep;
03560 }
03561 else if ( ( *ma == -VECTOR || *ma == -MINVECTOR ) && ma+4 <= mb ) {
03562     if ( ma[2] == -VECTOR ) {
03563         if ( *ma == -VECTOR ) *ReplaceSub++ = VECTOVEC;
03564         else *ReplaceSub++ = VECTOMIN;
03565     }
03566     else if ( ma[2] == -MINVECTOR ) {
03567         if ( *ma == -VECTOR ) *ReplaceSub++ = VECTOMIN;
03568         else *ReplaceSub++ = VECTOVEC;
03569     }
03570 /*
03571     Next is a vector-like subexpression
03572     Search for vector nature first
03573 */
03574     else if ( ma[2] > 0 ) {
03575         WORD *sstop, *ttstop, *w, *mm, n, count;
03576         WORD *v1, *v2 = 0;
03577         if ( *ma == -MINVECTOR ) {
03578             ss = ma+2;
03579             sstop = ss + *ss;
03580             ss += ARGHEAD;
03581             while ( ss < sstop ) {
03582                 ss += *ss;
03583                 ss[-1] = -ss[-1];
03584             }
03585             *ma = -VECTOR;
03586         }
03587         ss = ma+2;
03588         sstop = ss + *ss;
03589         ss += ARGHEAD;
03590         while ( ss < sstop ) {
03591             tt = ss + *ss;
03592             ttstop = tt - ABS(tt[-1]);
03593             ss++;
03594             count = 0;
03595             while ( ss < ttstop ) {
03596                 if ( *ss == INDEX ) {
03597                     n = ss[1] - 2; ss += 2;
03598                     while ( --n >= 0 ) {
03599                         if ( *ss < MINSPEC ) count++;
03600                         ss++;
03601                     }
03602                 }
03603                 else ss += ss[1];
03604             }
03605             if ( count != 1 ) goto NoRep;
03606             ss = tt;
03607         }
03608         subtype = VECTOSUB;
03609         if ( ReplaceType == 0 ) {
03610             oldtoprhs = C->numrhs;
03611             oldcpointer = C->Pointer - C->Buffer;
03612         }
03613         ReplaceType = 1;
03614         mm = AddRHS(AT.ebufnum,1);
03615         *ReplaceSub++ = subtype;
03616         *ReplaceSub++ = 4;
03617         *ReplaceSub++ = ma[1];
03618         *ReplaceSub++ = C->numrhs;
03619         w = ma+2;
03620         n = *w - ARGHEAD;
03621         w += ARGHEAD;
03622         while ( (mm + n + 10) > C->Top )
03623             mm = DoubleCbuffer(AT.ebufnum,mm,15);
03624         while ( --n >= 0 ) *mm++ = *w++;
03625         *mm++ = 0;
03626         C->rhs[C->numrhs+1] = mm;
03627         C->Pointer = mm;
03628         mm = AddRHS(AT.ebufnum,1);
03629         w = ma+2;
03630         n = *w - ARGHEAD;
03631         w += ARGHEAD;
03632         while ( (mm + n + 13) > C->Top )
03633             mm = DoubleCbuffer(AT.ebufnum,mm,16);
03634         sstop = w + n;
03635         while ( w < sstop ) {
03636             tt = w + *w; ttstop = tt - ABS(tt[-1]);
03637             ss = mm; mm++; w++;
03638             while ( w < ttstop ) { /* Subterms */
03639                 if ( *w != INDEX ) {
03640                     n = w[1];
03641                     NCOPY(mm,w,n);
03642                 }
03643                 else {

```

```

03644         v1 = mm;
03645         *mm++ = *w++;
03646         *mm++ = n = *w++;
03647         n -= 2;
03648         while ( --n >= 0 ) {
03649             if ( *w >= MINSPEC ) *mm++ = *w++;
03650             else v2 = w++;
03651         }
03652         n = WORDDIFF(mm,v1);
03653         if ( n != v1[1] ) {
03654             if ( n <= 2 ) mm -= 2;
03655             else v1[1] = n;
03656             *mm++ = VECTOR;
03657             *mm++ = 4;
03658             *mm++ = *v2;
03659             *mm++ = FUNNYVEC;
03660         }
03661     }
03662 }
03663 while ( w < tt ) *mm++ = *w++;
03664 *ss = WORDDIFF(mm,ss);
03665 }
03666 *mm++ = 0;
03667 C->rhs[C->numrhs+1] = mm;
03668 C->Pointer = mm;
03669 if ( mm > C->Top ) {
03670     MLOCK(ErrorMessageLock);
03671     MesPrint("Internal error in Normalize with extra compiler
buffer");
03672     MUNLOCK(ErrorMessageLock);
03673     Terminate(-1);
03674 }
03675 ma += 2 + ma[2];
03676 continue;
03677 }
03678 else goto NoRep;
03679 }
03680 else if ( *ma == -INDEX ) {
03681     if ( ( ma[2] == -INDEX || ma[2] == -VECTOR )
03682         && ma+4 <= mb )
03683         *ReplaceSub++ = INDTOIND;
03684     else if ( ma[1] >= AM.OffsetIndex ) {
03685         if ( ma[2] == -SNUMBER && ma+4 <= mb
03686             && ma[3] >= 0 && ma[3] < AM.OffsetIndex )
03687             *ReplaceSub++ = INDTOIND;
03688         else if ( ma[2] == ARGHEAD && ma+2+ARGHEAD <= mb ) {
03689             *ReplaceSub++ = INDTOIND;
03690             *ReplaceSub++ = 4;
03691             *ReplaceSub++ = ma[1];
03692             *ReplaceSub++ = 0;
03693             ma += 2+ARGHEAD;
03694             continue;
03695         }
03696         else goto NoRep;
03697     }
03698     else goto NoRep;
03699 }
03700 else goto NoRep;
03701 *ReplaceSub++ = 4;
03702 *ReplaceSub++ = ma[1];
03703 *ReplaceSub++ = ma[3];
03704 ma += 4;
03705 }
03706 }
03707 }
03708 AN.ReplaceScrat[1] = ReplaceSub-AN.ReplaceScrat;
03709 /*
03710 Success. This means that we have to remove the replace_
03711 from the functions. It starts at mc and end at mb.
03712 */
03713 while ( mb < m ) *mc++ = *mb++;
03714 m = mc;
03715 break;
03716 NoRep:
03717 if ( ReplaceType > 0 ) {
03718     C->numrhs = oldtoprhs;
03719     C->Pointer = C->Buffer + oldcpointer;
03720 }
03721 ReplaceType = -1;
03722 if ( ++ReplaceVeto >= 0 ) break;
03723 }
03724 ma = mb;
03725 }
03726 }
03727 /*
03728 #] Track Replace_ :
03729 #[ LeviCivita tensors :

```



```

03730 */
03731     if ( neps ) {
03732         to = m;
03733         for ( i = 0; i < neps; i++ ) { /* Put the indices in order */
03734             t = peps[i];
03735             if ( ( t[2] & DIRTYSYMFLAG ) != DIRTYSYMFLAG ) continue;
03736             t[2] &= ~DIRTYSYMFLAG;
03737             if ( AR.Eside == LHSIDE || AR.Eside == LHSIDEX ) {
03738                 /* Potential problems with FUNNYWILD */
03739             }
03740             /*
03741              * First make sure all FUNNIES are at the end.
03742              * Then sort separately
03743             */
03744             r = t + FUNHEAD;
03745             m = tt = t + t[1];
03746             while ( r < m ) {
03747                 if ( *r != FUNNYWILD ) { r++; continue; }
03748                 k = r[1]; u = r + 2;
03749                 while ( u < tt ) {
03750                     u[-2] = *u;
03751                     if ( *u != FUNNYWILD ) ncoef = -ncoef;
03752                     u++;
03753                 }
03754                 tt[-2] = FUNNYWILD; tt[-1] = k; m -= 2;
03755             }
03756             t += FUNHEAD;
03757             do {
03758                 for ( r = t + 1; r < m; r++ ) {
03759                     if ( *r < *t ) { k = *r; *r = *t; *t = k; ncoef = -ncoef; }
03760                     else if ( *r == *t ) goto NormZero;
03761                 }
03762                 t++;
03763             } while ( t < m );
03764             do {
03765                 for ( r = t + 2; r < tt; r += 2 ) {
03766                     if ( r[1] < t[1] ) {
03767                         k = r[1]; r[1] = t[1]; t[1] = k; ncoef = -ncoef; }
03768                     else if ( r[1] == t[1] ) goto NormZero;
03769                 }
03770                 t += 2;
03771             } while ( t < tt );
03772             else {
03773                 m = t + t[1];
03774                 t += FUNHEAD;
03775                 do {
03776                     for ( r = t + 1; r < m; r++ ) {
03777                         if ( *r < *t ) { k = *r; *r = *t; *t = k; ncoef = -ncoef; }
03778                         else if ( *r == *t ) goto NormZero;
03779                     }
03780                     t++;
03781                 } while ( t < m );
03782             }
03783         }
03784         /* Sort the tensors */
03785         for ( i = 0; i < (neps-1); i++ ) {
03786             t = peps[i];
03787             for ( j = i+1; j < neps; j++ ) {
03788                 r = peps[j];
03789                 if ( t[1] > r[1] ) {
03790                     peps[i] = m = r; peps[j] = r = t; t = m;
03791                 }
03792                 else if ( t[1] == r[1] ) {
03793                     k = t[1] - FUNHEAD;
03794                     m = t + FUNHEAD;
03795                     r += FUNHEAD;
03796                     do {
03797                         if ( *r < *m ) {
03798                             m = peps[j]; peps[j] = t; peps[i] = t = m;
03799                             break;
03800                         }
03801                         else if ( *r++ > *m++ ) break;
03802                     } while ( --k > 0 );
03803                 }
03804             }
03805         }
03806         m = to;
03807         for ( i = 0; i < neps; i++ ) {
03808             t = peps[i];
03809             k = t[1];
03810             NCOPY(m,t,k);
03811         }
03812     }
03813 }
03814 */
03815 #] LeviCivita tensors :

```

```

03817     #[ Delta :
03818     /*
03819     if ( ndel ) {
03820         r = t = pdel;
03821         for ( i = 0; i < ndel; i += 2, r += 2 ) {
03822             if ( r[1] < r[0] ) { k = *r; *r = r[1]; r[1] = k; }
03823         }
03824         for ( i = 2; i < ndel; i += 2, t += 2 ) {
03825             r = t + 2;
03826             for ( j = i; j < ndel; j += 2 ) {
03827                 if ( *r > *t ) { r += 2; }
03828                 else if ( *r < *t ) {
03829                     k = *r; *r++ = *t; *t++ = k;
03830                     k = *r; *r++ = *t; *t-- = k;
03831                 }
03832                 else {
03833                     if ( *++r < t[1] ) {
03834                         k = *r; *r = t[1]; t[1] = k;
03835                     }
03836                     r++;
03837                 }
03838             }
03839         }
03840         t = pdel;
03841         *m++ = DELTA;
03842         *m++ = ndel + 2;
03843         i = ndel;
03844         NCOPY(m,t,i);
03845     }
03846     /*
03847     #[ Delta :
03848     #[ Loose Vectors/Indices :
03849     /*
03850     if ( nind ) {
03851         t = pind;
03852         for ( i = 0; i < nind; i++ ) {
03853             r = t + 1;
03854             for ( j = i+1; j < nind; j++ ) {
03855                 if ( *r < *t ) {
03856                     k = *r; *r = *t; *t = k;
03857                 }
03858                 r++;
03859             }
03860             t++;
03861         }
03862         t = pind;
03863         *m++ = INDEX;
03864         *m++ = nind + 2;
03865         i = nind;
03866         NCOPY(m,t,i);
03867     }
03868     /*
03869     #[ Loose Vectors/Indices :
03870     #[ Vectors :
03871     /*
03872     if ( nvec ) {
03873         t = pvec;
03874         for ( i = 2; i < nvec; i += 2 ) {
03875             r = t + 2;
03876             for ( j = i; j < nvec; j += 2 ) {
03877                 if ( *r == *t ) {
03878                     if ( *++r < t[1] ) {
03879                         k = *r; *r = t[1]; t[1] = k;
03880                     }
03881                     r++;
03882                 }
03883                 else if ( *r < *t ) {
03884                     k = *r; *r++ = *t; *t++ = k;
03885                     k = *r; *r++ = *t; *t-- = k;
03886                 }
03887                 else { r += 2; }
03888             }
03889             t += 2;
03890         }
03891         t = pvec;
03892         *m++ = VECTOR;
03893         *m++ = nvec + 2;
03894         i = nvec;
03895         NCOPY(m,t,i);
03896     }
03897     /*
03898     #[ Vectors :
03899     #[ Dotproducts :
03900     /*
03901     if ( ndot ) {
03902         to = m;
03903         m = t = pdot;

```

```

03904         i = ndot;
03905         while ( --i >= 0 ) {
03906             if ( *t > t[1] ) { j = *t; *t = t[1]; t[1] = j; }
03907             t += 3;
03908         }
03909         t = m;
03910         ndot *= 3;
03911         m += ndot;
03912         while ( t < (m-3) ) {
03913             r = t + 3;
03914             do {
03915                 if ( *r == *t ) {
03916                     if ( *++r == *++t ) {
03917                         r++;
03918                         if ( ( *r < MAXPOWER && t[1] < MAXPOWER )
03919                             || ( *r > -MAXPOWER && t[1] > -MAXPOWER ) ) {
03920                             t++;
03921                             *t += *r;
03922                             if ( *t > MAXPOWER || *t < -MAXPOWER ) {
03923                                 MLOCK(ErrorMessageLock);
03924                                 MesPrint("Exponent of dotproduct out of range: %d",*t);
03925                                 MUNLOCK(ErrorMessageLock);
03926                                 goto NormMin;
03927                             }
03928                             ndot -= 3;
03929                             *r-- = *--m;
03930                             *r-- = *--m;
03931                             *r = *--m;
03932                             if ( !*t ) {
03933                                 ndot -= 3;
03934                                 *t-- = *--m;
03935                                 *t-- = *--m;
03936                                 *t = *--m;
03937                                 t -= 3;
03938                                 break;
03939                             }
03940                         }
03941                         else if ( *r < *++t ) {
03942                             k = *r; *r++ = *t; *t = k;
03943                         }
03944                         else r++;
03945                         t -= 2;
03946                     }
03947                     else if ( *r < *t ) {
03948                         k = *r; *r++ = *t; *t++ = k;
03949                         k = *r; *r++ = *t; *t = k;
03950                         t -= 2;
03951                     }
03952                     else { r += 2; t--; }
03953                 }
03954                 else if ( *r < *t ) {
03955                     k = *r; *r++ = *t; *t++ = k;
03956                     k = *r; *r++ = *t; *t++ = k;
03957                     k = *r; *r++ = *t; *t = k;
03958                     t -= 2;
03959                 }
03960                 else { r += 3; }
03961             } while ( r < m );
03962             t += 3;
03963         }
03964         m = to;
03965         t = pdot;
03966         if ( ( i = ndot ) > 0 ) {
03967             *m++ = DOTPRODUCT;
03968             *m++ = i + 2;
03969             NCOPY(m,t,i);
03970         }
03971     }
03972     /*
03973     #] Dotproducts :
03974     #[ Symbols :
03975     */
03976     if ( nsym ) {
03977         nsym <= 1;
03978         t = psym;
03979         *m++ = SYMBOL;
03980         r = m;
03981         *m++ = ( i = nsym ) + 2;
03982         if ( i ) { do {
03983             if ( !*t ) {
03984                 if ( t[1] < (2*MAXPOWER) ) { /* powers of i */
03985                     if ( t[1] & 1 ) { *m++ = 0; *m++ = 1; }
03986                     else *r -= 2;
03987                     if ( *++t & 2 ) ncoef = -ncoef;
03988                     t++;
03989                 }
03990             }

```

```

03991     else if ( *t <= NumSymbols && *t > -2*MAXPOWER ) { /* Put powers in range */
03992         if ( ( ( t[1] > symbols[*t].maxpower ) && ( symbols[*t].maxpower < MAXPOWER ) ) ||
03993             ( ( t[1] < symbols[*t].minpower ) && ( symbols[*t].minpower > -MAXPOWER ) ) )
03994             &&
03995                 ( t[1] < 2*MAXPOWER ) && ( t[1] > -2*MAXPOWER ) ) {
03996             if ( i <= 2 || t[2] != *t ) goto NormZero;
03997         }
03998         if ( AN.ncmod == 1 && ( AC.modmode & ALSOPOWERS ) != 0 ) {
03999             if ( AC.cmod[0] == 1 ) t[1] = 0;
04000             else if ( t[1] >= 0 ) t[1] = 1 + (t[1]-1)%(AC.cmod[0]-1);
04001             else {
04002                 t[1] = -1 - (-t[1]-1)%(AC.cmod[0]-1);
04003                 if ( t[1] < 0 ) t[1] += (AC.cmod[0]-1);
04004             }
04005             if ( ( t[1] < (2*MAXPOWER) && t[1] >= MAXPOWER )
04006                 || ( t[1] > -(2*MAXPOWER) && t[1] <= -MAXPOWER ) ) {
04007                 MLOCK(ErrorMessageLock);
04008                 MesPrint("Exponent out of range: %d",t[1]);
04009                 MUNLOCK(ErrorMessageLock);
04010                 goto NormMin;
04011             }
04012             if ( AT.TrimPower && AR.PolyFunVar == *t && t[1] > AR.PolyFunPow ) {
04013                 goto NormZero;
04014             }
04015             else if ( t[1] ) {
04016                 *m++ = *t++;
04017                 *m++ = *t++;
04018             }
04019             else { *r -- 2; t += 2; }
04020         }
04021         else {
04022             *m++ = *t++; *m++ = *t++;
04023         }
04024     } while ( (i--2) > 0 ); }
04025     if ( *r <= 2 ) m = r-1;
04026 }
04027 /*
04028 #] Symbols :
04029 #[ float_ :
04030
04031 Here we treat float_ functions and combined them with the regular
04032 coefficient.
04033 */
04034
04035 #ifdef WITHFLOAT
04036     if ( withfloat ) {
04037         WORD floatsign = 3;
04038         /*
04039             First check whether the coefficient is already 1/1
04040         */
04041         if ( ABS(ncoef) == 3 && n_coef[0] == 1 && n_coef[1] == 1 ) {
04042             if ( withfloat == 1 ) {
04043                 t = firstfloat;
04044             }
04045             /*
04046                 We only interfere by making the sign positive.
04047                 The float_ function has already been tested.
04048             */
04049             if ( t[FUNHEAD+3] < 0 ) {
04050                 t[FUNHEAD+3] = -t[FUNHEAD+3];
04051                 floatsign = -floatsign;
04052             }
04053             if ( ncoef < 0 ) floatsign = -floatsign;
04054             /*
04055                 Copy to output;
04056             */
04057             AT.FloatPos = m-termout;
04058             i = t[1]; NCOPY(m,t,i)
04059         }
04060         else {
04061             PackFloat(m,aux4);
04062             if ( m[FUNHEAD+3] < 0 ) {
04063                 m[FUNHEAD+3] = -m[FUNHEAD+3];
04064                 floatsign = -floatsign;
04065             }
04066             if ( ncoef < 0 ) floatsign = -floatsign;
04067             AT.FloatPos = m-termout;
04068             m += m[1];
04069         }
04070     }
04071     else {
04072         if ( withfloat == 1 ) UnpackFloat(aux4,firstfloat);
04073         RatToFloat(aux5,(UWORD *)n_coef,ncoef);
04074         mpf_mul(aux4,aux4,aux5);
04075         PackFloat(m,aux4);
04076         AT.FloatPos = m-termout;
04077         if ( m[FUNHEAD+3] < 0 ) {

```

```

04077         m[FUNHEAD+3] = -m[FUNHEAD+3];
04078         floatsign = -floatsign;
04079     }
04080     m += m[1];
04081 }
04082     n_coef[0] = 1; n_coef[1] = 1; ncoef = floatsign;
04083 }
04084     else AT.FloatPos = 0;
04085 #endif
04086 /*
04087 #] float_ :
04088 #[ Do Replace_ :
04089 */
04091     stop = (WORD *) ((UBYTE *) (termout)) + AM.MaxTer;
04092     i = ABS(ncoef);
04093     if ( ( m + i ) > stop ) {
04094         MLOCK(ErrorMessageLock);
04095         MesPrint("Term too complex during normalization");
04096         MUNLOCK(ErrorMessageLock);
04097         goto NormMin;
04098     }
04099     if ( ReplaceType >= 0 ) {
04100         t = n_coef;
04101         i--;
04102         NCOPY(m,t,i);
04103         *m++ = ncoef;
04104         t = termout;
04105         *t = WORDDIF(m,t);
04106         if ( ReplaceType == 0 ) {
04107             AT.WorkPointer = termout+*termout;
04108             WildFill(BHEAD term,termout,AN.ReplaceScrat);
04109             termout = term + *term;
04110         }
04111         else {
04112             AT.WorkPointer = r = termout + *termout;
04113             WildFill(BHEAD r,termout,AN.ReplaceScrat);
04114             i = *r; m = term;
04115             NCOPY(m,r,i);
04116             termout = m;
04117
04118
04119             r = m = term;
04120             r += *term; r -= ABS(r[-1]);
04121             m++;
04122             while ( m < r ) {
04123                 if ( *m >= FUNCTION && m[1] > FUNHEAD &&
04124                     functions[*m-FUNCTION].spec != TENSORFUNCTION )
04125                     m[2] |= DIRTYFLAG;
04126                 m += m[1];
04127             }
04128         }
04129     /*
04130     The next 'reset' cannot be done. We still need the expression
04131     in the buffer. Note though that this may cause a runaway pointer
04132     if we are not very careful.
04133
04134     C->numrhs = oldtoprhs;
04135     C->Pointer = C->Buffer + oldcpointer;
04136 */
04137     TermFree(n_llnum,"n_llnum");
04138     TermFree(n_coef,"NormCoeef");
04139     return(1);
04140 }
04141     else {
04142         t = termout;
04143         k = WORDDIF(m,t);
04144         *t = k + i;
04145         m = term;
04146         NCOPY(m,t,k);
04147         i--;
04148         t = n_coef;
04149         NCOPY(m,t,i);
04150         *m++ = ncoef;
04151     }
04152 /*
04153 #] Do Replace_ :
04154 #[ Errors and Finish :
04155 */
04156 RegEnd:
04157     if ( termout < term + *term && termout >= term ) AT.WorkPointer = term + *term;
04158     else AT.WorkPointer = termout;
04159 /*
04160     if ( termflag ) { We have to assign the term to $variable(s)
04161         TermAssign(term);
04162     }
04163 */

```

```

04164     TermFree(n_llnum, "n_llnum");
04165     TermFree(n_coef, "NormCoef");
04166     return(regval);
04167
04168 NormInf:
04169     MLOCK(ErrorMessageLock);
04170     MesPrint("Division by zero during normalization");
04171     MUNLOCK(ErrorMessageLock);
04172     Terminate(-1);
04173
04174 NormZZ:
04175     MLOCK(ErrorMessageLock);
04176     MesPrint("0^0 during normalization of term");
04177     MUNLOCK(ErrorMessageLock);
04178     Terminate(-1);
04179
04180 NormPRF:
04181     MLOCK(ErrorMessageLock);
04182     MesPrint("0/0 in polyratfun during normalization of term");
04183     MUNLOCK(ErrorMessageLock);
04184     Terminate(-1);
04185
04186 NormZero:
04187     *term = 0;
04188     AT.WorkPointer = termout;
04189     TermFree(n_llnum, "n_llnum");
04190     TermFree(n_coef, "NormCoef");
04191     return(regval);
04192
04193 NormMin:
04194     TermFree(n_llnum, "n_llnum");
04195     TermFree(n_coef, "NormCoef");
04196     return(-1);
04197
04198 FromNorm:
04199     MLOCK(ErrorMessageLock);
04200     MesCall("Norm");
04201     MUNLOCK(ErrorMessageLock);
04202     TermFree(n_llnum, "n_llnum");
04203     TermFree(n_coef, "NormCoef");
04204     return(-1);
04205
04206 /*
04207 #] Errors and Finish :
04208 */
04209 }
04210
04211 /*
04212 #] Normalize :
04213 #[ ExtraSymbol :
04214 */
04215
04216 int ExtraSymbol(WORD sym, WORD pow, WORD nsym, WORD *ppsym, WORD *ncoef)
04217 {
04218     WORD *m, i;
04219     i = nsym;
04220     m = ppsym - 2;
04221     while ( i > 0 ) {
04222         if ( sym == *m ) {
04223             m++;
04224             if ( pow > 2*MAXPOWER || pow < -2*MAXPOWER
04225                 || *m > 2*MAXPOWER || *m < -2*MAXPOWER ) {
04226                 MLOCK(ErrorMessageLock);
04227                 MesPrint("Illegal wildcard power combination.");
04228                 MUNLOCK(ErrorMessageLock);
04229                 Terminate(-1);
04230             }
04231             *m += pow;
04232
04233             if ( ( sym <= NumSymbols && sym > -MAXPOWER )
04234                 && ( symbols[sym].complex & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY ) {
04235                 *m %= symbols[sym].maxpower;
04236                 if ( *m < 0 ) *m += symbols[sym].maxpower;
04237                 if ( ( symbols[sym].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
04238                     if ( ( ( symbols[sym].maxpower & 1 ) == 0 ) &&
04239                         ( *m >= symbols[sym].maxpower/2 ) ) {
04240                         *m -= symbols[sym].maxpower/2; *ncoef = -*ncoef;
04241                     }
04242                 }
04243             }
04244
04245             if ( *m >= 2*MAXPOWER || *m <= -2*MAXPOWER ) {
04246                 MLOCK(ErrorMessageLock);
04247                 MesPrint("Power overflow during normalization");
04248                 MUNLOCK(ErrorMessageLock);
04249                 return(-1);
04250             }

```

```

04251         if ( !*m ) {
04252             m--;
04253             while ( i < nsym )
04254                 { *m = m[2]; m++; *m = m[2]; m++; i++; }
04255             return(-1);
04256         }
04257         return(0);
04258     }
04259     else if ( sym < *m ) {
04260         m -= 2;
04261         i--;
04262     }
04263     else break;
04264 }
04265 m = ppsym;
04266 while ( i < nsym )
04267     { m--; m[2] = *m; m--; m[2] = *m; i++; }
04268 *m++ = sym;
04269 *m = pow;
04270 return(1);
04271 }
04272
04273 /*
04274     #] ExtraSymbol :
04275     #[ DoTheta :
04276 */
04277
04278 int DoTheta(PHEAD WORD *t)
04279 {
04280     GETBIDENTITY
04281     WORD k, *r1, *r2, *tstop, type;
04282     WORD ia, *ta, *tb, *stopa, *stopb;
04283     if ( AC.BraceNormalize ) return(-1);
04284     type = *t;
04285     k = t[1];
04286     tstop = t + k;
04287     t += FUNHEAD;
04288     if ( k <= FUNHEAD ) return(1);
04289     r1 = t;
04290     NEXTARG(r1)
04291     if ( r1 == tstop ) {
04292         /*
04293             One argument
04294         */
04295         if ( *t == ARGHEAD ) {
04296             if ( type == THETA ) return(1);
04297             else return(0); /* THETA2 */
04298         }
04299         if ( *t < 0 ) {
04300             if ( *t == -SNUMBER ) {
04301                 if ( t[1] < 0 ) return(0);
04302                 else {
04303                     if ( type == THETA2 && t[1] == 0 ) return(0);
04304                     else return(1);
04305                 }
04306             }
04307             return(-1);
04308         }
04309         k = t[*t-1];
04310         if ( *t == ABS(k)+1+ARGHEAD ) {
04311             if ( k > 0 ) return(1);
04312             else return(0);
04313         }
04314         return(-1);
04315     }
04316     /*
04317         At least two arguments
04318     */
04319     r2 = r1;
04320     NEXTARG(r2)
04321     if ( r2 < tstop ) return(-1); /* More than 2 arguments */
04322     /*
04323         Note now that zero has to be treated specially
04324         We take the criteria from the symmetrize routine
04325     */
04326     if ( *t == -SNUMBER && *r1 == -SNUMBER ) {
04327         if ( t[1] > r1[1] ) return(0);
04328         else if ( t[1] < r1[1] ) {
04329             return(1);
04330         }
04331         else if ( type == THETA ) return(1);
04332         else return(0); /* THETA2 */
04333     }
04334     else if ( t[1] == 0 && *t == -SNUMBER ) {
04335         if ( *r1 > 0 ) { }
04336         else if ( *t < *r1 ) return(1);
04337         else if ( *t > *r1 ) return(0);

```

```

04338     }
04339     else if ( r1[1] == 0 && *r1 == -SNUMBER ) {
04340         if ( *t > 0 ) { }
04341         else if ( *t < *r1 ) return(1);
04342         else if ( *t > *r1 ) return(0);
04343     }
04344     r2 = AT.WorkPointer;
04345     if ( *t < 0 ) {
04346         ta = r2;
04347         ToGeneral(t,ta,0);
04348         r2 += *r2;
04349     }
04350     else ta = t;
04351     if ( *r1 < 0 ) {
04352         tb = r2;
04353         ToGeneral(r1,tb,0);
04354     }
04355     else tb = r1;
04356     stopa = ta + *ta;
04357     stopb = tb + *tb;
04358     ta += ARGHEAD; tb += ARGHEAD;
04359     while ( ta < stopa ) {
04360         if ( tb >= stopb ) return(0);
04361         if ( ( ia = CompareTerms(BHEAD ta,tb,(WORD)1) ) < 0 ) return(0);
04362         if ( ia > 0 ) return(1);
04363         ta += *ta;
04364         tb += *tb;
04365     }
04366     if ( type == THETA ) return(1);
04367     else return(0); /* THETA2 */
04368 }
04369
04370 /*
04371     #] DoTheta :
04372     #[ DoDelta :
04373 */
04374
04375 int DoDelta(WORD *t)
04376 {
04377     WORD k, *r1, *r2, *tstop, isnum, isnum2, type = *t;
04378     if ( AC.BracketNormalize ) return(-1);
04379     k = t[1];
04380     if ( k <= FUNHEAD ) goto argzero;
04381     if ( k == FUNHEAD+ARGHEAD && t[FUNHEAD] == ARGHEAD ) goto argzero;
04382     tstop = t + k;
04383     t += FUNHEAD;
04384     r1 = t;
04385     NEXTARG(r1)
04386     if ( *t < 0 ) {
04387         k = 1;
04388         if ( *t == -SNUMBER ) { isnum = 1; k = t[1]; }
04389         else isnum = 0;
04390     }
04391     else {
04392         k = t[*t-1];
04393         k = ABS(k);
04394         if ( k == *t-ARGHEAD-1 ) isnum = 1;
04395         else isnum = 0;
04396         k = 1;
04397     }
04398     if ( r1 >= tstop ) { /* Single argument */
04399         if ( !isnum ) return(-1);
04400         if ( k == 0 ) goto argzero;
04401         goto argnonzero;
04402     }
04403     r2 = r1;
04404     NEXTARG(r2)
04405     if ( r2 < tstop ) return(-1);
04406     if ( *r1 < 0 ) {
04407         if ( *r1 == -SNUMBER ) { isnum2 = 1; }
04408         else isnum2 = 0;
04409     }
04410     else {
04411         k = r1[*r1-1];
04412         k = ABS(k);
04413         if ( k == *r1-ARGHEAD-1 ) isnum2 = 1;
04414         else isnum2 = 0;
04415     }
04416     if ( isnum != isnum2 ) return(-1);
04417     tstop = r1;
04418     while ( t < tstop && r1 < r2 ) {
04419         if ( *t != *r1 ) {
04420             if ( !isnum ) return(-1);
04421             goto argnonzero;
04422         }
04423         t++; r1++;
04424     }

```



```

04425     if ( t != tstop || r1 != r2 ) {
04426         if ( !isnum ) return(-1);
04427         goto argnonzero;
04428     }
04429 argzero:
04430     if ( type == DELTA2 ) return(1);
04431     else return(0);
04432 argnonzero:
04433     if ( type == DELTA2 ) return(0);
04434     else return(1);
04435 }
04436
04437 /*
04438     #] DoDelta :
04439     #[ DoRevert :
04440 */
04441 void DoRevert(WORD *fun, WORD *tmp)
04442 {
04443     WORD *t, *r, *m, *to, *tt, *mm, i, j;
04444     to = fun + fun[1];
04445     r = fun + FUNHEAD;
04446     while ( r < to ) {
04447         if ( *r <= 0 ) {
04448             if ( *r == -REVERSEFUNCTION ) {
04449                 m = r; mm = m+1;
04450                 while ( mm < to ) *m++ = *mm++;
04451                 to--;
04452                 (fun[1])--;
04453                 fun[2] |= DIRTYSYMFLAG;
04454             }
04455             else if ( *r <= -FUNCTION ) r++;
04456             else {
04457                 if ( *r == -INDEX && r[1] < MINSPEC ) *r = -VECTOR;
04458                 r += 2;
04459             }
04460         }
04461     }
04462     else {
04463         if ( ( *r > ARGHEAD )
04464             && ( r[ARGHEAD+1] == REVERSEFUNCTION )
04465             && ( *r == (r[ARGHEAD]+ARGHEAD) )
04466             && ( r[ARGHEAD] == (r[ARGHEAD+2]+4) )
04467             && ( *(r+r-3) == 1 )
04468             && ( *(r+r-2) == 1 )
04469             && ( *(r+r-1) == 3 ) ) {
04470             mm = r;
04471             r += ARGHEAD + 1;
04472             tt = r + r[1];
04473             r += FUNHEAD;
04474             m = tmp;
04475             t = r;
04476             j = 0;
04477             while ( t < tt ) {
04478                 NEXTARG(t)
04479                 j++;
04480             }
04481             while ( --j >= 0 ) {
04482                 i = j;
04483                 t = r;
04484                 while ( --i >= 0 ) {
04485                     NEXTARG(t)
04486                 }
04487                 if ( *t > 0 ) {
04488                     i = *t;
04489                     NCOPY(m,t,i);
04490                 }
04491                 else if ( *t <= -FUNCTION ) *m++ = *t++;
04492                 else { *m++ = *t++; *m++ = *t++; }
04493             }
04494             i = WORDDIF(m,tmp);
04495             m = tmp;
04496             t = mm;
04497             r = t + *t;
04498             NCOPY(t,m,i);
04499             m = r;
04500             r = t;
04501             i = WORDDIF(to,m);
04502             NCOPY(t,m,i);
04503             fun[1] = WORDDIF(t,fun);
04504             to = t;
04505             fun[2] |= DIRTYSYMFLAG;
04506         }
04507         else r += *r;
04508     }
04509 }
04510 }
04511

```

```

04512 /*
04513     #] DoRevert :
04514     #] Normalize :
04515     #[ DetCommu :
04516
04517     Determines the number of terms in an expression that contain
04518     noncommuting objects. This can be used to see whether products of
04519     this expression can be evaluated with binomial coefficients.
04520
04521     We don't try to be fancy. If a term contains noncommuting objects
04522     we are not looking whether they can commute with complete other
04523     terms.
04524
04525     If the number gets too large we cut it off.
04526 */
04527
04528 #define MAXNUMBEROFNONCOMTERMS 2
04529
04530 WORD DetCommu(WORD *terms)
04531 {
04532     WORD *t, *tnext, *tstop;
04533     WORD num = 0;
04534     if ( *terms == 0 ) return(0);
04535     if ( terms[*terms] == 0 ) return(0);
04536     t = terms;
04537     while ( *t ) {
04538         tnext = t + *t;
04539         tstop = tnext - ABS(tnext[-1]);
04540         t++;
04541         while ( t < tstop ) {
04542             if ( *t >= FUNCTION ) {
04543                 if ( functions[*t-FUNCTION].commute ) {
04544                     num++;
04545                     if ( num >= MAXNUMBEROFNONCOMTERMS ) return(num);
04546                     break;
04547                 }
04548             }
04549             else if ( *t == SUBEXPRESSION ) {
04550                 if ( cbuf[t[4]].CanCommu[t[2]] ) {
04551                     num++;
04552                     if ( num >= MAXNUMBEROFNONCOMTERMS ) return(num);
04553                     break;
04554                 }
04555             }
04556             else if ( *t == EXPRESSION ) {
04557                 num++;
04558                 if ( num >= MAXNUMBEROFNONCOMTERMS ) return(num);
04559                 break;
04560             }
04561             else if ( *t == DOLLAREXPRESSION ) {
04562 /*
04563         Technically this is not correct. We have to test first
04564         whether this is MODLOCAL (in TFORM) and if so, use the
04565         local version. Anyway, this should be rare to never
04566         occurring because dollars should be replaced.
04567 */
04568                 if ( cbuf[AM.dbufnum].CanCommu[t[2]] ) {
04569                     num++;
04570                     if ( num >= MAXNUMBEROFNONCOMTERMS ) return(num);
04571                     break;
04572                 }
04573             }
04574             t += t[1];
04575         }
04576         t = tnext;
04577     }
04578     return(num);
04579 }
04580
04581 /*
04582     #] DetCommu :
04583     #[ DoesCommu :
04584
04585     Determines the number of noncommuting objects in a term.
04586     If the number gets too large we cut it off.
04587 */
04588
04589 WORD DoesCommu(WORD *term)
04590 {
04591     WORD *tstop;
04592     WORD num = 0;
04593     if ( *term == 0 ) return(0);
04594     tstop = term + *term;
04595     tstop = tstop - ABS(tstop[-1]);
04596     term++;
04597     while ( term < tstop ) {
04598         if ( ( *term >= FUNCTION ) && ( functions[*term-FUNCTION].commute ) ) {

```

```

04599         num++;
04600         if ( num >= MAXNUMBEROFNONCOMTERMS ) return(num);
04601     }
04602     term += term[1];
04603 }
04604 return(num);
04605 }
04606
04607 /*
04608 #] DoesCommu :
04609 #[ PolyNormPoly :
04610
04611     Normalizes a polynomial
04612 */
04613
04614 #ifdef EVALUATEGCD
04615 WORD *PolyNormPoly (PHEAD WORD *Poly) {
04616
04617     GETBIDENTITY;
04618     WORD *buffer = AT.WorkPointer;
04619     WORD *p;
04620     if ( NewSort(BHEAD0) ) { Terminate(-1); }
04621     AR.CompareRoutine = (COMPAREDUMMY)(&CompareSymbols);
04622     while ( *Poly ) {
04623         p = Poly + *Poly;
04624         if ( SymbolNormalize(Poly) < 0 ) return(0);
04625         if ( StoreTerm(BHEAD Poly) ) {
04626             AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
04627             LowerSortLevel();
04628             Terminate(-1);
04629         }
04630         Poly = p;
04631     }
04632     if ( EndSort(BHEAD buffer,1) < 0 ) {
04633         AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
04634         Terminate(-1);
04635     }
04636     p = buffer;
04637     while ( *p ) p += *p;
04638     AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
04639     AT.WorkPointer = p + 1;
04640     return(buffer);
04641 }
04642 #endif
04643
04644 /*
04645 #] PolyNormPoly :
04646 #[ EvaluateGcd :
04647
04648     Try to evaluate the GCDFUNCTION gcd_.
04649     This function can have a number of arguments which can be numbers
04650     and/or polynomials. If there are objects that aren't SYMBOLS or numbers
04651     it cannot work currently.
04652
04653     To make this work properly we have to intervene in proces.c
04654     proces.c:         if ( Normalize(BHEAD m) ) {
04655 1060 proces.c:         if ( Normalize(BHEAD r) ) {
04656 1126?proces.c:         if ( Normalize(BHEAD term) ) {
04657     proces.c:         if ( Normalize(BHEAD AT.WorkPointer) ) goto PasErr;
04658 2308!proces.c:     if ( ( retnorm = Normalize(BHEAD term) ) != 0 ) {
04659     proces.c:         ReNumber(BHEAD term); Normalize(BHEAD term);
04660     proces.c:         if ( Normalize(BHEAD v) ) Terminate(-1);
04661     proces.c:         if ( Normalize(BHEAD w) ) { LowerSortLevel(); goto PolyCall; }
04662     proces.c:         if ( Normalize(BHEAD term) ) goto PolyCall;
04663 */
04664 #ifdef EVALUATEGCD
04665
04666 WORD *EvaluateGcd(PHEAD WORD *subterm)
04667 {
04668     GETBIDENTITY
04669     WORD *oldworkpointer = AT.WorkPointer, *work1, *work2, *work3;
04670     WORD *t, *tt, *ttt, *t1, *t2, *t3, *t4, *tstop;
04671     WORD ct, nnum;
04672     UWORD gcdnum, stor;
04673     WORD *lnum=n_llnum+1;
04674     WORD *num1, *num2, *num3, *den1, *den2, *den3;
04675     WORD sizenum1, sizenum2, sizenum3, sizeden1, sizeden2, sizeden3;
04676     int i, isnumeric = 0, numarg = 0 /*, sizearg */;
04677     LONG size;
04678 /*
04679     Step 1: Look for -SNUMBER or -SYMBOL arguments.
04680             If encountered, treat everybody with it.
04681 */
04682     tt = subterm + subterm[1]; t = subterm + FUNHEAD;
04683
04684     while ( t < tt ) {
04685         numarg++;

```

```

04686         if ( *t == -SNUMBER ) {
04687             if ( t[1] == 0 ) {
04688 gcdzero;;
04689                 MLOCK(ErrorMessageLock);
04690                 MesPrint("Trying to take the GCD involving a zero term.");
04691                 MUNLOCK(ErrorMessageLock);
04692                 return(0);
04693             }
04694             gcdnum = ABS(t[1]);
04695             t1 = subterm + FUNHEAD;
04696             while ( gcdnum > 1 && t1 < tt ) {
04697                 if ( *t1 == -SNUMBER ) {
04698                     stor = ABS(t1[1]);
04699                     if ( stor == 0 ) goto gcdzero;
04700                     if ( GcdLong(BHEAD (UWORD *)&stor,1,(UWORD *)&gcdnum,1,
04701                         (UWORD *)lnum,&nnum) ) goto FromGCD;
04702                     gcdnum = lnum[0];
04703                     t1 += 2;
04704                     continue;
04705                 }
04706                 else if ( *t1 == -SYMBOL ) goto gcdisone;
04707                 else if ( *t1 < 0 ) goto gcdillegal;
04708 /*
04709             Now we have to go through all the terms in the argument.
04710             This includes long numbers.
04711 */
04712             ttt = t1 + *t1;
04713             ct = *ttt; *ttt = 0;
04714             if ( t1[1] != 0 ) { /* First normalize the argument */
04715                 t1 = PolyNormPoly(BHEAD t1+ARGHEAD);
04716             }
04717             else t1 += ARGHEAD;
04718             while ( *t1 ) {
04719                 t1 += *t1;
04720                 i = ABS(t1[-1]);
04721                 t2 = t1 - i;
04722                 i = (i-1)/2;
04723                 t3 = t2+i-1;
04724                 while ( t3 > t2 && *t3 == 0 ) { t3--; i--; }
04725                 if ( GcdLong(BHEAD (UWORD *)t2,(WORD)i,(UWORD *)&gcdnum,1,
04726                     (UWORD *)lnum,&nnum) ) {
04727                     *ttt = ct;
04728                     goto FromGCD;
04729                 }
04730                 gcdnum = lnum[0];
04731                 if ( gcdnum == 1 ) {
04732                     *ttt = ct;
04733                     goto gcdisone;
04734                 }
04735             }
04736             *ttt = ct;
04737             t1 = ttt;
04738             AT.WorkPointer = oldworkpointer;
04739         }
04740         if ( gcdnum == 1 ) goto gcdisone;
04741         oldworkpointer[0] = 4;
04742         oldworkpointer[1] = gcdnum;
04743         oldworkpointer[2] = 1;
04744         oldworkpointer[3] = 3;
04745         oldworkpointer[4] = 0;
04746         AT.WorkPointer = oldworkpointer + 5;
04747         return(oldworkpointer);
04748     }
04749     else if ( *t == -SYMBOL ) {
04750         t1 = subterm + FUNHEAD;
04751         i = t[1];
04752         while ( t1 < tt ) {
04753             if ( *t1 == -SNUMBER ) goto gcdisone;
04754             if ( *t1 == -SYMBOL ) {
04755                 if ( t1[1] != i ) goto gcdisone;
04756                 t1 += 2; continue;
04757             }
04758             if ( *t1 < 0 ) goto gcdillegal;
04759             ttt = t1 + *t1;
04760             ct = *ttt; *ttt = 0;
04761             if ( t1[1] != 0 ) { /* First normalize the argument */
04762                 t2 = PolyNormPoly(BHEAD t1+ARGHEAD);
04763             }
04764             else t2 = t1 + ARGHEAD;
04765             while ( *t2 ) {
04766                 t3 = t2+1;
04767                 t2 = t2 + *t2;
04768                 tstop = t2 - ABS(t2[-1]);
04769                 while ( t3 < tstop ) {
04770                     if ( *t3 != SYMBOL ) {
04771                         *ttt = ct;
04772                         goto gcdillegal;

```

```

04773         }
04774         t4 = t3 + 2;
04775         t3 += t3[1];
04776         while ( t4 < t3 ) {
04777             if ( *t4 == i && t4[1] > 0 ) goto nextterminarg;
04778             t4 += 2;
04779         }
04780     }
04781     *ttt = ct;
04782     goto gcdisone;
04783 nextterminarg:;
04784     }
04785     *ttt = ct;
04786     t1 = ttt;
04787     AT.WorkPointer = oldworkpointer;
04788     }
04789     oldworkpointer[0] = 8;
04790     oldworkpointer[1] = SYMBOL;
04791     oldworkpointer[2] = 4;
04792     oldworkpointer[3] = t[1];
04793     oldworkpointer[4] = 1;
04794     oldworkpointer[5] = 1;
04795     oldworkpointer[6] = 1;
04796     oldworkpointer[7] = 3;
04797     oldworkpointer[8] = 0;
04798     AT.WorkPointer = oldworkpointer+9;
04799     return(oldworkpointer);
04800 }
04801 else if ( *t < 0 ) {
04802 gcdillegal:;
04803     MLOCK(ErrorMessageLock);
04804     MesPrint("Illegal object in gcd_ function. Object not a number or a symbol.");
04805     MUNLOCK(ErrorMessageLock);
04806     goto FromGCD;
04807 }
04808 else if ( ABS(t[*t-1]) == *t-ARGHEAD-1 ) isnumeric = numarg;
04809 else if ( t[1] != 0 ) {
04810     ttt = t + *t; ct = *ttt; *ttt = 0;
04811     t = PolyNormPoly(BHEAD t+ARGHEAD);
04812     *ttt = ct;
04813     if ( t[*t] == 0 && ABS(t[*t-1]) == *t-ARGHEAD-1 ) isnumeric = numarg;
04814     AT.WorkPointer = oldworkpointer;
04815     t = ttt;
04816 }
04817 t += *t;
04818 }
04819 /*
04820 At this point there are only generic arguments.
04821 There are however still two cases:
04822 1: There is an argument that is purely numerical
04823    In that case we have to take the gcd of all coefficients
04824 2: All arguments are nontrivial polynomials.
04825    Here we don't worry so much about the factor. (???)
04826 We know whether case 1 occurs when isnumeric > 0.
04827 We can look up numarg to get a good starting value.
04828 */
04829 AT.WorkPointer = oldworkpointer;
04830 if ( isnumeric ) {
04831     t = subterm + FUNHEAD;
04832     for ( i = 1; i < isnumeric; i++ ) {
04833         NEXTARG(t);
04834     }
04835     if ( t[1] != 0 ) { /* First normalize the argument */
04836         ttt = t + *t; ct = *ttt; *ttt = 0;
04837         t = PolyNormPoly(BHEAD t+ARGHEAD);
04838         *ttt = ct;
04839     }
04840     t += *t;
04841     i = (ABS(t[-1])-1)/2;
04842     den1 = t - 1 - i;
04843     num1 = den1 - i;
04844     sizenum1 = sizeden1 = i;
04845     while ( sizenum1 > 1 && num1[sizenum1-1] == 0 ) sizenum1--;
04846     while ( sizeden1 > 1 && den1[sizeden1-1] == 0 ) sizeden1--;
04847     work1 = AT.WorkPointer+1; work2 = work1+sizenum1;
04848     for ( i = 0; i < sizenum1; i++ ) work1[i] = num1[i];
04849     for ( i = 0; i < sizeden1; i++ ) work2[i] = den1[i];
04850     num1 = work1; den1 = work2;
04851     AT.WorkPointer = work2 = work2 + sizeden1;
04852     t = subterm + FUNHEAD;
04853     while ( t < tt ) {
04854         ttt = t + *t; ct = *ttt; *ttt = 0;
04855         if ( t[1] != 0 ) {
04856             t = PolyNormPoly(BHEAD t+ARGHEAD);
04857         }
04858         else t += ARGHEAD;
04859         while ( *t ) {

```

```

04860         t += *t;
04861         i = (ABS(t[-1])-1)/2;
04862         den2 = t - 1 - i;
04863         num2 = den2 - i;
04864         sizenum2 = sizeden2 = i;
04865         while ( sizenum2 > 1 && num2[sizenum2-1] == 0 ) sizenum2--;
04866         while ( sizeden2 > 1 && den2[sizeden2-1] == 0 ) sizeden2--;
04867         num3 = AT.WorkPointer;
04868         if ( GcdLong(BHEAD *(UWORD *)num2,sizenum2,(UWORD *)num1,sizenum1,
04869                     (UWORD *)num3,&sizenum3) ) goto FromGCD;
04870         sizenum1 = sizenum3;
04871         for ( i = 0; i < sizenum1; i++ ) num1[i] = num3[i];
04872         den3 = AT.WorkPointer;
04873         if ( GcdLong(BHEAD *(UWORD *)den2,sizeden2,(UWORD *)den1,sizeden1,
04874                     (UWORD *)den3,&sizeden3) ) goto FromGCD;
04875         sizeden1 = sizeden3;
04876         for ( i = 0; i < sizeden1; i++ ) den1[i] = den3[i];
04877         if ( sizenum1 == 1 && num1[0] == 1 && sizeden1 == 1 && den1[1] == 1 )
04878             goto gcdisone;
04879     }
04880     *ttt = ct;
04881     t = ttt;
04882     AT.WorkPointer = work2;
04883 }
04884 AT.WorkPointer = oldworkpointer;
04885 /*
04886     Now copy the GCD to the 'output'
04887 */
04888     if ( sizenum1 > sizeden1 ) {
04889         while ( sizenum1 > sizeden1 ) den1[sizeden1++] = 0;
04890     }
04891     else if ( sizenum1 < sizeden1 ) {
04892         while ( sizenum1 < sizeden1 ) num1[sizenum1++] = 0;
04893     }
04894     t = oldworkpointer;
04895     i = 2*sizenum1+1;
04896     *t++ = i+1;
04897     if ( num1 != t ) { NCOPY(t,num1,sizenum1); }
04898     else t += sizenum1;
04899     if ( den1 != t ) { NCOPY(t,den1,sizeden1); }
04900     else t += sizeden1;
04901     *t++ = i;
04902     *t++ = 0;
04903     AT.WorkPointer = t;
04904     return(oldworkpointer);
04905 }
04906 /*
04907     Now the real stuff with only polynomials.
04908     Pick up the shortest term to start.
04909     We are a bit brutish about this.
04910 */
04911     t = subterm + FUNHEAD;
04912     AT.WorkPointer += AM.MaxTer/sizeof(WORD);
04913     work2 = AT.WorkPointer;
04914 /*
04915     sizearg = subterm[1];
04916 */
04917     i = 0; work3 = 0;
04918     while ( t < tt ) {
04919         i++;
04920         work1 = AT.WorkPointer;
04921         ttt = t + *t; ct = *ttt; *ttt = 0;
04922         t = PolyNormPoly(BHEAD t+ARGHEAD);
04923         if ( *work1 < AT.WorkPointer-work1 ) {
04924 /*
04925             sizearg = AT.WorkPointer-work1;
04926 */
04927             numarg = i;
04928             work3 = work1;
04929         }
04930         *ttt = ct; t = ttt;
04931     }
04932     *AT.WorkPointer++ = 0;
04933 /*
04934     We have properly normalized arguments and the shortest is indicated in work3
04935 */
04936     work1 = work3;
04937     while ( *work2 ) {
04938         if ( work2 != work3 ) {
04939             work1 = PolyGCD2(BHEAD work1,work2);
04940         }
04941         while ( *work2 ) work2 += *work2;
04942         work2++;
04943     }
04944     work2 = work1;
04945     while ( *work2 ) work2 += *work2;
04946     size = work2 - work1 + 1;

```

```

04947     t = oldworkpointer;
04948     NCOPY(t,work1,size);
04949     AT.WorkPointer = t;
04950     return(oldworkpointer);
04951
04952 gcdisone;;
04953     oldworkpointer[0] = 4;
04954     oldworkpointer[1] = 1;
04955     oldworkpointer[2] = 1;
04956     oldworkpointer[3] = 3;
04957     oldworkpointer[4] = 0;
04958     AT.WorkPointer = oldworkpointer+5;
04959     return(oldworkpointer);
04960 FromGCD:
04961     MLOCK(ErrorMessageLock);
04962     MesCall("EvaluateGcd");
04963     MUNLOCK(ErrorMessageLock);
04964     return(0);
04965 }
04966
04967 #endif
04968
04969 /*
04970  #] EvaluateGcd :
04971  #[ TreatPolyRatFun :
04972
04973      if ( AR.PolyFunExp == 1 ) we have to trim the contents of the polyratfun
04974      down to its most divergent term and give it coefficient +1. This is done
04975      by taking the terms with the least power in the variable in the numerator
04976      and in the denominator and then combine them.
04977      Answer is either PolyRatFun(ep^n,1) or PolyRatFun(1,1) or PolyRatFun(1,ep^n)
04978  */
04979
04980 int TreatPolyRatFun(PHEAD WORD *prf)
04981 {
04982     WORD *t, *tstop, *r, *rstop, *m, *mstop;
04983     WORD exp1 = MAXPOWER, exp2 = MAXPOWER;
04984     t = prf+FUNHEAD;
04985     if ( *t < 0 ) {
04986         if ( *t == -SYMBOL && t[1] == AR.PolyFunVar ) {
04987             if ( exp1 > 1 ) exp1 = 1;
04988             t += 2;
04989         }
04990         else {
04991             if ( exp1 > 0 ) exp1 = 0;
04992             NEXTARG(t)
04993         }
04994     }
04995     else {
04996         tstop = t + *t;
04997         t += ARGHEAD;
04998         while ( t < tstop ) {
04999             /*
05000              Now look for the minimum power of AR.PolyFunVar
05001             */
05002             r = t+1;
05003             t += *t;
05004             rstop = t - ABS(t[-1]);
05005             while ( r < rstop ) {
05006                 if ( *r != SYMBOL ) { r += r[1]; continue; }
05007                 m = r;
05008                 mstop = m + m[1];
05009                 m += 2;
05010                 while ( m < mstop ) {
05011                     if ( *m == AR.PolyFunVar ) {
05012                         if ( m[1] < exp1 ) exp1 = m[1];
05013                         break;
05014                     }
05015                     m += 2;
05016                 }
05017                 if ( m == mstop ) {
05018                     if ( exp1 > 0 ) exp1 = 0;
05019                 }
05020                 break;
05021             }
05022             if ( r == rstop ) {
05023                 if ( exp1 > 0 ) exp1 = 0;
05024             }
05025         }
05026         t = tstop;
05027     }
05028     if ( *t < 0 ) {
05029         if ( *t == -SYMBOL && t[1] == AR.PolyFunVar ) {
05030             if ( exp2 > 1 ) exp2 = 1;
05031         }
05032         else {
05033             if ( exp2 > 0 ) exp2 = 0;

```

```

05034     }
05035 }
05036 else {
05037     tstop = t + *t;
05038     t += ARGHEAD;
05039     while ( t < tstop ) {
05040 /*
05041         Now look for the minimum power of AR.PolyFunVar
05042 */
05043         r = t+1;
05044         t += *t;
05045         rstop = t - ABS(t[-1]);
05046         while ( r < rstop ) {
05047             if ( *r != SYMBOL ) { r += r[1]; continue; }
05048             m = r;
05049             mstop = m + m[1];
05050             m += 2;
05051             while ( m < mstop ) {
05052                 if ( *m == AR.PolyFunVar ) {
05053                     if ( m[1] < exp2 ) exp2 = m[1];
05054                     break;
05055                 }
05056                 m += 2;
05057             }
05058             if ( m == mstop ) {
05059                 if ( exp2 > 0 ) exp2 = 0;
05060             }
05061             break;
05062         }
05063         if ( r == rstop ) {
05064             if ( exp2 > 0 ) exp2 = 0;
05065         }
05066     }
05067 }
05068 /*
05069     Now we can compose the output.
05070     Notice that the output can never be longer than the input provided
05071     we never can have arguments that consist of just a function.
05072 */
05073 exp1 = exp1-exp2;
05074 /* if ( exp1 > 0 ) exp1 = 0; */
05075 t = prf+FUNHEAD;
05076 if ( exp1 == 0 ) {
05077     *t++ = -SNUMBER; *t++ = 1;
05078     *t++ = -SNUMBER; *t++ = 1;
05079 }
05080 else if ( exp1 > 0 ) {
05081     if ( exp1 == 1 ) {
05082         *t++ = -SYMBOL; *t++ = AR.PolyFunVar;
05083     }
05084     else {
05085         *t++ = 8+ARGHEAD;
05086         *t++ = 0;
05087         FILLARG(t);
05088         *t++ = 8; *t++ = SYMBOL; *t++ = 4; *t++ = AR.PolyFunVar;
05089         *t++ = exp1; *t++ = 1; *t++ = 1; *t++ = 3;
05090     }
05091     *t++ = -SNUMBER; *t++ = 1;
05092 }
05093 else {
05094     *t++ = -SNUMBER; *t++ = 1;
05095     if ( exp1 == -1 ) {
05096         *t++ = -SYMBOL; *t++ = AR.PolyFunVar;
05097     }
05098     else {
05099         *t++ = 8+ARGHEAD;
05100         *t++ = 0;
05101         FILLARG(t);
05102         *t++ = 8; *t++ = SYMBOL; *t++ = 4; *t++ = AR.PolyFunVar;
05103         *t++ = -exp1; *t++ = 1; *t++ = 1; *t++ = 3;
05104     }
05105 }
05106 prf[2] = 0; /* Clean */
05107 prf[1] = t - prf;
05108 return(0);
05109 }
05110
05111 /*
05112     #] TreatPolyRatFun :
05113     #[ DropCoefficient :
05114 */
05115
05116 void DropCoefficient(PHEAD WORD *term)
05117 {
05118     GETBIDENTITY
05119     WORD *t = term + *term;
05120     WORD n, na;

```



```

05121     n = t[-1]; na = ABS(n);
05122     t -= na;
05123     if ( n == 3 && t[0] == 1 && t[1] == 1 ) return;
05124     *AN.RepPoint = 1;
05125     t[0] = 1; t[1] = 1; t[2] = 3;
05126     *term -= (na-3);
05127 }
05128
05129 /*
05130 #] DropCoefficient :
05131 #[ DropSymbols :
05132 */
05133
05134 void DropSymbols(PHEAD WORD *term)
05135 {
05136     GETBIDENTITY
05137     WORD *tend = term + *term, *t1, *t2, *tstop;
05138     tstop = tend - ABS(tend[-1]);
05139     t1 = term+1;
05140     while ( t1 < tstop ) {
05141         if ( *t1 == SYMBOL ) {
05142             *AN.RepPoint = 1;
05143             t2 = t1+t1[1];
05144             while ( t2 < tend ) *t1++ = *t2++;
05145             *term = t1 - term;
05146             break;
05147         }
05148         t1 += t1[1];
05149     }
05150 }
05151
05152 /*
05153 #] DropSymbols :
05154 #[ SymbolNormalize :
05155 */
05156 int SymbolNormalize(WORD *term)
05157 {
05158     GETBIDENTITY
05159     WORD buffer[7*NORMSIZE], *t, *b, *bb, *tt, *m, *tstop;
05160     int i;
05161     b = buffer;
05162     *b++ = SYMBOL; *b++ = 2;
05163     t = term + *term;
05164     tstop = t - ABS(t[-1]);
05165     t = term + 1;
05166     while ( t < tstop ) { /* Step 1: collect symbols */
05167         if ( *t == SYMBOL && t < tstop ) {
05168             for ( i = 2; i < t[1]; i += 2 ) {
05169                 bb = buffer+2;
05170                 while ( bb < b ) {
05171                     if ( bb[0] == t[i] ) { /* add powers */
05172                         bb[1] += t[i+1];
05173                         if ( bb[1] > MAXPOWER || bb[1] < -MAXPOWER ) {
05174                             MLOCK(ErrorMessageLock);
05175                             MesPrint("Power in SymbolNormalize out of range");
05176                             MUNLOCK(ErrorMessageLock);
05177                             return(-1);
05178                         }
05179                     }
05180                     if ( bb[1] == 0 ) {
05181                         b -= 2;
05182                         while ( bb < b ) {
05183                             bb[0] = bb[2]; bb[1] = bb[3]; bb += 2;
05184                         }
05185                     }
05186                     goto Nexti;
05187                 }
05188                 else if ( bb[0] > t[i] ) { /* insert it */
05189                     m = b;
05190                     while ( m > bb ) { m[1] = m[-1]; m[0] = m[-2]; m -= 2; }
05191                     b += 2;
05192                     bb[0] = t[i];
05193                     bb[1] = t[i+1];
05194                     goto Nexti;
05195                 }
05196             }
05197             bb += 2;
05198         }
05199         if ( bb >= b ) { /* add it to the end */
05200             *b++ = t[i]; *b++ = t[i+1];
05201         }
05202     }
05203     Nexti:;
05204 }
05205
05206 else {
05207     MLOCK(ErrorMessageLock);
05208     MesPrint("Illegal term in SymbolNormalize");
05209     MUNLOCK(ErrorMessageLock);
05210     return(-1);
05211 }

```

```

05216     }
05217     t += t[1];
05218 }
05219 buffer[1] = b - buffer;
05220 /*
05221 Veto negative powers
05222 */
05223 if ( AT.LeaveNegative == 0 ) {
05224     b = buffer; bb = b + b[1]; b += 3;
05225     while ( b < bb ) {
05226         if ( *b < 0 ) {
05227             MLOCK(ErrorMessageLock);
05228             MesPrint("Negative power in SymbolNormalize");
05229             MUNLOCK(ErrorMessageLock);
05230             return(-1);
05231         }
05232         b += 2;
05233     }
05234 }
05235 /*
05236 Now we use the fact that the new term will not be longer than the old one
05237 Actually it should be shorter when there is more than one subterm!
05238 Copy back.
05239 */
05240 i = buffer[1];
05241 b = buffer; tt = term + 1;
05242 if ( i > 2 ) { NCOPY(tt,b,i) }
05243 if ( tt < tstop ) {
05244     i = term[*term-1];
05245     if ( i < 0 ) i = -i;
05246     *term -= (tstop-tt);
05247     NCOPY(tt,tstop,i)
05248 }
05249 return(0);
05250 }
05251
05252 /*
05253 #] SymbolNormalize :
05254 #[ TestFunFlag :
05255
05256 Tests whether a function still has unsubstituted subexpressions
05257 This function has its dirtyflag on!
05258 */
05259 int TestFunFlag(PHEAD WORD *tfun)
05260 {
05261     WORD *t, *tstop, *r, *rstop, *m, *mstop;
05262     if ( functions[*tfun-FUNCTION].spec <= 0 ) return(0);
05263     tstop = tfun + tfun[1];
05264     t = tfun + FUNHEAD;
05265     while ( t < tstop ) {
05266         if ( *t < 0 ) { NEXTARG(t); continue; }
05267         rstop = t + *t;
05268         if ( t[1] == 0 ) { t = rstop; continue; }
05269         r = t + ARGHEAD;
05270         while ( r < rstop ) { /* Here we loop over terms */
05271             m = r+1; mstop = r+r; mstop -= ABS(mstop[-1]);
05272             while ( m < mstop ) { /* Loop over the subterms */
05273                 if ( *m == SUBEXPRESSION || *m == EXPRESSION || *m == DOLLAREXPRESSION ) return(1);
05274                 if ( ( *m >= FUNCTION ) && ( m[2] & DIRTYFLAG ) == DIRTYFLAG )
05275                     && ( *m != REPLACEMENT ) && TestFunFlag(BHEAD m) ) return(1);
05276                 m += m[1];
05277             }
05278             r += *r;
05279         }
05280         t += *t;
05281     }
05282     return(0);
05283 }
05284 }
05285
05286 /*
05287 #] TestFunFlag :
05288 #[ BracketNormalize :
05289 */
05290 int BracketNormalize(PHEAD WORD *term)
05291 {
05292     WORD *stop = term+*term-3, *t, *tt, *tstart, *r;
05293     WORD *oldwork = AT.WorkPointer;
05294     WORD *termout;
05295     WORD i, ii, j;
05296     termout = AT.WorkPointer = term+*term;
05297     /*
05298 First collect all functions and sort them
05299 */
05300 tt = termout+1; t = term+1;
05301 while ( t < stop ) {

```

```

05303 if ( *t >= FUNCTION ) { i = t[1]; NCOPY(tt,t,i); }
05304 else t += t[1];
05305 }
05306 if ( tt > termout+1 && tt-termout-1 > termout[2] ) { /* sorting */
05307 r = termout+1; ii = tt-r;
05308 for ( i = 0; i < ii-FUNHEAD; i += FUNHEAD ) { /* Bubble sort */
05309 for ( j = i+FUNHEAD; j > 0; j -= FUNHEAD ) {
05310 if ( functions[r[j-FUNHEAD]-FUNCTION].commute
05311 && functions[r[j]-FUNCTION].commute == 0 ) break;
05312 if ( r[j-FUNHEAD] > r[j] ) EXCH(r[j-FUNHEAD],r[j])
05313 else break;
05314 }
05315 }
05316 }
05317
05318 tstart = tt; t = term + 1; *tt++ = DELTA; *tt++ = 2;
05319 while ( t < stop ) {
05320 if ( *t == DELTA ) { i = t[1]-2; t += 2; tstart[1] += i; NCOPY(tt,t,i); }
05321 else t += t[1];
05322 }
05323 if ( tstart[1] > 2 ) {
05324 for ( r = tstart+2; r < tstart+tstart[1]; r += 2 ) {
05325 if ( r[0] > r[1] ) EXCH(r[0],r[1])
05326 }
05327 }
05328 if ( tstart[1] > 4 ) { /* sorting */
05329 r = tstart+2; ii = tstart[1]-2;
05330 for ( i = 0; i < ii-2; i += 2 ) { /* Bubble sort */
05331 for ( j = i+2; j > 0; j -= 2 ) {
05332 if ( r[j-2] > r[j] ) {
05333 EXCH(r[j-2],r[j])
05334 EXCH(r[j-1],r[j+1])
05335 }
05336 else if ( r[j-2] < r[j] ) break;
05337 else {
05338 if ( r[j-1] > r[j+1] ) EXCH(r[j-1],r[j+1])
05339 else break;
05340 }
05341 }
05342 }
05343 tt = tstart+tstart[1];
05344 }
05345 else if ( tstart[1] == 2 ) { tt = tstart; }
05346 else tt = tstart+4;
05347
05348 tstart = tt; t = term + 1; *tt++ = INDEX; *tt++ = 2;
05349 while ( t < stop ) {
05350 if ( *t == INDEX ) { i = t[1]-2; t += 2; tstart[1] += i; NCOPY(tt,t,i); }
05351 else t += t[1];
05352 }
05353 if ( tstart[1] >= 4 ) { /* sorting */
05354 r = tstart+2; ii = tstart[1]-2;
05355 for ( i = 0; i < ii-1; i += 1 ) { /* Bubble sort */
05356 for ( j = i+1; j > 0; j -= 1 ) {
05357 if ( r[j-1] > r[j] ) EXCH(r[j-1],r[j])
05358 else break;
05359 }
05360 }
05361 tt = tstart+tstart[1];
05362 }
05363 else if ( tstart[1] == 2 ) { tt = tstart; }
05364 else tt = tstart+3;
05365
05366 tstart = tt; t = term + 1; *tt++ = DOTPRODUCT; *tt++ = 2;
05367 while ( t < stop ) {
05368 if ( *t == DOTPRODUCT ) { i = t[1]-2; t += 2; tstart[1] += i; NCOPY(tt,t,i); }
05369 else t += t[1];
05370 }
05371 if ( tstart[1] > 5 ) { /* sorting */
05372 r = tstart+2; ii = tstart[1]-2;
05373 for ( i = 0; i < ii; i += 3 ) {
05374 if ( r[i] > r[i+1] ) EXCH(r[i],r[i+1])
05375 }
05376 for ( i = 0; i < ii-3; i += 3 ) { /* Bubble sort */
05377 for ( j = i+3; j > 0; j -= 3 ) {
05378 if ( r[j-3] < r[j] ) break;
05379 if ( r[j-3] > r[j] ) {
05380 EXCH(r[j-3],r[j])
05381 EXCH(r[j-2],r[j+1])
05382 }
05383 else {
05384 if ( r[j-2] > r[j+1] ) EXCH(r[j-2],r[j+1])
05385 else break;
05386 }
05387 }
05388 }
05389 tt = tstart+tstart[1];

```

```

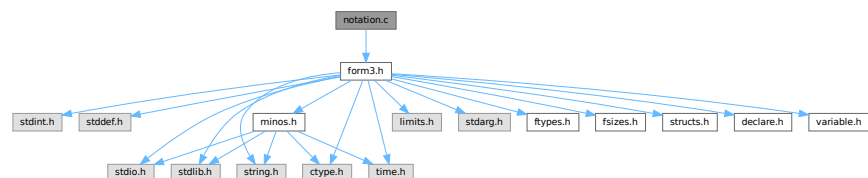
05390     }
05391     else if ( tstart[1] == 2 ) { tt = tstart; }
05392     else {
05393         if ( tstart[2] > tstart[3] ) EXCH(tstart[2],tstart[3])
05394         tt = tstart+5;
05395     }
05396
05397     tstart = tt; t = term + 1; *tt++ = SYMBOL; *tt++ = 2;
05398     while ( t < stop ) {
05399         if ( *t == SYMBOL ) { i = t[1]-2; t += 2; tstart[1] += i; NCOPY(tt,t,i); }
05400         else t += t[1];
05401     }
05402     if ( tstart[1] > 4 ) { /* sorting */
05403         r = tstart+2; ii = tstart[1]-2;
05404         for ( i = 0; i < ii-2; i += 2 ) { /* Bubble sort */
05405             for ( j = i+2; j > 0; j -= 2 ) {
05406                 if ( r[j-2] > r[j] ) EXCH(r[j-2],r[j])
05407                 else break;
05408             }
05409         }
05410         tt = tstart+tstart[1];
05411     }
05412     else if ( tstart[1] == 2 ) { tt = tstart; }
05413     else tt = tstart+4;
05414
05415     tstart = tt; t = term + 1; *tt++ = SETSET; *tt++ = 2;
05416     while ( t < stop ) {
05417         if ( *t == SETSET ) { i = t[1]-2; t += 2; tstart[1] += i; NCOPY(tt,t,i); }
05418         else t += t[1];
05419     }
05420     if ( tstart[1] > 4 ) { /* sorting */
05421         r = tstart+2; ii = tstart[1]-2;
05422         for ( i = 0; i < ii-2; i += 2 ) { /* Bubble sort */
05423             for ( j = i+2; j > 0; j -= 2 ) {
05424                 if ( r[j-2] > r[j] ) {
05425                     EXCH(r[j-2],r[j])
05426                     EXCH(r[j-1],r[j+1])
05427                 }
05428                 else break;
05429             }
05430         }
05431         tt = tstart+tstart[1];
05432     }
05433     else if ( tstart[1] == 2 ) { tt = tstart; }
05434     else tt = tstart+4;
05435     *tt++ = 1; *tt++ = 1; *tt++ = 3;
05436     t = term; i = *termout = tt - termout; tt = termout;
05437     NCOPY(t,tt,i);
05438     AT.WorkPointer = oldwork;
05439     return(0);
05440 }
05441
05442 /*
05443 #] BracketNormalize :
05444 */

```

4.89 notation.c File Reference

```
#include "form3.h"
```

Include dependency graph for notation.c:



Functions

- int [NormPolyTerm](#) (PHEAD WORD *term)

- int [ConvertToPoly](#) (PHEAD WORD *term, WORD *outterm, WORD *comlist, WORD par)
- int [LocalConvertToPoly](#) (PHEAD WORD *term, WORD *outterm, WORD startebuf, WORD par)
- int [ConvertFromPoly](#) (PHEAD WORD *term, WORD *outterm, WORD from, WORD to, WORD offset, WORD par)
- int [FindSubterm](#) (WORD *subterm)
- int [FindLocalSubterm](#) (PHEAD WORD *subterm, WORD startebuf)
- void [PrintSubtermList](#) (int from, int to)
- void [PrintExtraSymbol](#) (int num, WORD *terms, int par)
- int [FindSubexpression](#) (WORD *subexpr)
- int [ExtraSymFun](#) (PHEAD WORD *term, WORD level)
- int [PruneExtraSymbols](#) (WORD downto)

4.89.1 Detailed Description

Contains the functions that deal with the rewriting and manipulation of expressions/terms in polynomial representation.

Definition in file [notation.c](#).

4.89.2 Function Documentation

4.89.2.1 NormPolyTerm()

```
int NormPolyTerm (
    PHEAD WORD * term )
```

Definition at line 53 of file [notation.c](#).

4.89.2.2 ConvertToPoly()

```
int ConvertToPoly (
    PHEAD WORD * term,
    WORD * outterm,
    WORD * comlist,
    WORD par )
```

Definition at line 307 of file [notation.c](#).

4.89.2.3 LocalConvertToPoly()

```
int LocalConvertToPoly (
    PHEAD WORD * term,
    WORD * outterm,
    WORD startebuf,
    WORD par )
```

Converts a generic term to polynomial notation in which there are only symbols and brackets. During conversion there will be only symbols. Brackets are stripped. Objects that need 'translation' are put inside a special compiler buffer and represented by a symbol. The numbering of the extra symbols is down from the maximum. In principle there can be a problem when running into the already assigned ones. This uses the FindTree for searching in the global tree and then looks further in the AT.ebufnum. This allows fully parallel processing. Hence we need no locks. Cannot be used in the same module as ConvertToPoly.

Definition at line 510 of file [notation.c](#).

Referenced by [poly_factorize_expression\(\)](#).

4.89.2.4 ConvertFromPoly()

```
int ConvertFromPoly (
    PHEAD WORD * term,
    WORD * outterm,
    WORD from,
    WORD to,
    WORD offset,
    WORD par )
```

Definition at line 660 of file [notation.c](#).

4.89.2.5 FindSubterm()

```
int FindSubterm (
    WORD * subterm )
```

Definition at line 773 of file [notation.c](#).

4.89.2.6 FindLocalSubterm()

```
int FindLocalSubterm (
    PHEAD WORD * subterm,
    WORD startebuf )
```

Definition at line 843 of file [notation.c](#).

4.89.2.7 PrintSubtermList()

```
void PrintSubtermList (
    int from,
    int to )
```

Definition at line 897 of file [notation.c](#).

4.89.2.8 PrintExtraSymbol()

```
void PrintExtraSymbol (
    int num,
    WORD * terms,
    int par )
```

Definition at line 1009 of file [notation.c](#).

4.89.2.9 FindSubexpression()

```
int FindSubexpression (
    WORD * subexpr )
```

Definition at line 1096 of file [notation.c](#).

4.89.2.10 ExtraSymFun()

```
int ExtraSymFun (
    PHEAD WORD * term,
    WORD level )
```

Definition at line 1149 of file [notation.c](#).

4.89.2.11 PruneExtraSymbols()

```
int PruneExtraSymbols (
    WORD downto )
```

Definition at line 1217 of file [notation.c](#).

4.90 notation.c

[Go to the documentation of this file.](#)

```
00001
00006 /* # [ License : */
00007 /*
00008 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00009 * When using this file you are requested to refer to the publication
00010 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011 * This is considered a matter of courtesy as the development was paid
00012 * for by FOM the Dutch physics granting agency and we would like to
00013 * be able to track its scientific use to convince FOM of its value
00014 * for the community.
00015 *
00016 * This file is part of FORM.
00017 *
00018 * FORM is free software: you can redistribute it and/or modify it under the
00019 * terms of the GNU General Public License as published by the Free Software
00020 * Foundation, either version 3 of the License, or (at your option) any later
00021 * version.
00022 *
00023 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00026 * details.
00027 *
00028 * You should have received a copy of the GNU General Public License along
00029 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00030 */
00031 /* # [ License : */
00032 /*
00033 * # [ Includes :
00034 */
00035
00036 #include "form3.h"
00037
00038 /*
00039 * # ] Includes :
00040 * # [ NormPolyTerm :
00041
00042 * Brings a term to normal form.
00043
00044 * This routine knows objects of the following types:
00045 * SYMBOL
00046 * HAAKJE
00047 * SNUMBER
00048 * LNUMBER
00049 * The SNUMBER and LNUMBER are worked into the coefficient.
00050 * One of the essences here is that everything can be done in place.
00051 */
00052
00053 int NormPolyTerm(PHEAD WORD *term)
00054 {
00055     WORD *tcoef, ncoef, *tstop, *tfill, *t, *tt;
00056     int equal, i;
00057     WORD *r1, *r2, *r3, *r4, *r5, *rfirst, rv;
```

```

00058      WORD *lnum, nnum;          /* Scratch, originally for factorials */
00059  /*
00060      One: find the coefficient
00061  */
00062      tcoef = term+*term;
00063      ncoef = tcoef[-1];
00064      tstop = tcoef - ABS(tcoef[-1]);
00065      tfill = t = term + 1;
00066      rfirst = 0;
00067      if ( t >= tstop ) { return(*term); }
00068      while ( t < tstop ) {
00069          switch ( *t ) {
00070              case SYMBOL:
00071                  if ( rfirst == 0 ) {
00072  /*
00073                      Here we only need to sort
00074                      1: assume no equals. Bubble.
00075  */
00076                      rfirst = t;
00077                      r2 = rfirst+4; tt = r3 = t + t[1]; equal = 0;
00078                      while ( r2 < r3 ) {
00079                          r1 = r2 - 2;
00080                          if ( *r2 > *r1 ) { r2 += 2; continue; }
00081                          if ( *r2 == *r1 ) { r2 += 2; equal = 1; continue; }
00082                          rv = *r1; *r1 = *r2; *r2 = rv;
00083                          r1 -= 2; r2 -= 2; r4 = r2 + 2;
00084                          while ( r1 > t ) {
00085                              if ( *r2 >= *r1 ) { r2 = r4; break; }
00086                              rv = *r1; *r1 = *r2; *r2 = rv;
00087                              r1 -= 2; r2 -= 2;
00088                          }
00089                      }
00090  /*
00091                      2: hunt down the equal objects
00092                      postpone eliminating zero powers.
00093  */
00094                      if ( equal ) {
00095                          r1 = t+2; r2 = r1+2;
00096                          while ( r2 < r3 ) {
00097                              if ( *r1 == *r2 ) {
00098                                  r1[1] += r2[1];
00099                                  r4 = r2+2;
00100                                  while ( r4 < r3 ) *r2++ = *r4++;
00101                                  t[1] -= 2;
00102                                  r2 = r1 + 2; r3 -= 2;
00103                              }
00104                          }
00105                      }
00106                  }
00107                  else {
00108  /*
00109                      Here we only need to insert
00110  */
00111                      r1 = t + 2; tt = r3 = t + t[1];
00112                      while ( r1 < r3 ) {
00113                          r2 = rfirst+2; r4 = rfirst + rfirst[1];
00114                          while ( r2 < r4 ) {
00115                              if ( *r1 == *r2 ) {
00116                                  r2[1] += r1[1];
00117                                  break;
00118                              }
00119                              else if ( *r2 > *r1 ) {
00120                                  r5 = r4;
00121                                  while ( r5 > r2 ) { r5[1] = r5[-1]; r5[0] = r5[-2]; r5 -= 2; }
00122                                  rfirst[1] += 2;
00123                                  *r2 = *r1; r2[1] = r1[1];
00124                                  break;
00125                              }
00126                              r2 += 2;
00127                          }
00128                          if ( r2 == r4 ) {
00129                              rfirst[1] += 2;
00130                              *r2++ = *r1++; *r2++ = *r1++;
00131                          }
00132                          else r1 += 2;
00133                      }
00134                  }
00135                  t = tt;
00136                  break;
00137              case HAAKJE:      /* Here we skip brackets */
00138                  t += t[1];
00139                  break;
00140              case SNUMBER:
00141                  if ( t[2] < 0 ) {
00142                      t[2] = -t[2];
00143                      if ( t[3] & 1 ) ncoef = -ncoef;
00144                  }

```



```

00145         else if ( t[2] == 0 ) {
00146             if ( t[3] < 0 ) goto NormInf;
00147             goto NormZero;
00148         }
00149         lnum = TermMalloc("lnum");
00150         lnum[0] = t[2];
00151         nnum = 1;
00152         if ( t[3] && RaisPow(BHEAD (UWORD *)lnum,&nnum, (UWORD) (ABS(t[3]))) ) goto FromNorm;
00153         ncoef = REDLENG(ncoef);
00154         if ( t[3] < 0 ) {
00155             if ( Divvy(BHEAD (UWORD *)tstop,&ncoef, (UWORD *)lnum,nnum) )
00156                 goto FromNorm;
00157         }
00158         else if ( t[3] > 0 ) {
00159             if ( Mully(BHEAD (UWORD *)tstop,&ncoef, (UWORD *)lnum,nnum) )
00160                 goto FromNorm;
00161         }
00162         ncoef = INCLENG(ncoef);
00163         t += t[1];
00164         TermFree(lnum,"lnum");
00165         break;
00166     case LNUMBER:
00167         ncoef = REDLENG(ncoef);
00168         if ( Mully(BHEAD (UWORD *)tstop,&ncoef, (UWORD *) (t+3),t[2]) ) goto FromNorm;
00169         ncoef = INCLENG(ncoef);
00170         t += t[1];
00171         break;
00172     default:
00173         MLOCK(ErrorMessageLock);
00174         MesPrint("Illegal code in NormPolyTerm");
00175         MUNLOCK(ErrorMessageLock);
00176         Terminate(-1);
00177         break;
00178     }
00179 }
00180 /*
00181  Now we try to eliminate objects to the power zero.
00182 */
00183 if ( rfirst ) {
00184     r2 = rfirst+2;
00185     r3 = rfirst + rfirst[1];
00186     while ( r2 < r3 ) {
00187         if ( r2[1] == 0 ) {
00188             r1 = r2 + 2;
00189             while ( r1 < r3 ) { r1[-2] = r1[0]; r1[-1] = r1[1]; r1 += 2; }
00190             r3 -= 2;
00191             rfirst[1] -= 2;
00192         }
00193         else { r2 += 2; }
00194     }
00195     if ( rfirst[1] < 4 ) rfirst = 0;
00196 }
00197 /*
00198  Finally we put the term together
00199 */
00200 if ( rfirst ) {
00201     i = rfirst[1];
00202     NCOPY(tfill,rfirst,i)
00203 }
00204 i = ABS(ncoef)-1;
00205 NCOPY(tfill,tstop,i)
00206 *tfill++ = ncoef;
00207 *term = tfill - term;
00208 return(*term);
00209 NormZero:
00210 *term = 0;
00211 return(0);
00212 NormInf:
00213 MLOCK(ErrorMessageLock);
00214 MesPrint("0^0 in NormPolyTerm");
00215 MUNLOCK(ErrorMessageLock);
00216 Terminate(-1);
00217 return(-1);
00218 FromNorm:
00219 MLOCK(ErrorMessageLock);
00220 MesCall("NormPolyTerm");
00221 MUNLOCK(ErrorMessageLock);
00222 Terminate(-1);
00223 return(-1);
00224 }
00225
00226 /*
00227     #] NormPolyTerm :
00228     #[ ComparePoly :
00229 */
00251 #ifdef WITHCOMPAREPOLY
00252

```

```

00253 WORD ComparePoly(WORD *term1, WORD *term2, WORD level)
00254 {
00255     WORD *t1, *t2, *t3, *t4, *tstop1, *tstop2;
00256     tstop1 = term1 + *term1;
00257     tstop1 -= ABS(tstop1[-1]);
00258     tstop2 = term2 + *term2;
00259     tstop2 -= ABS(tstop2[-1]);
00260     t1 = term1+1;
00261     t2 = term2+1;
00262     while ( t1 < tstop1 && t2 < tstop2 ) {
00263         if ( *t1 == *t2 ) {
00264             if ( *t1 == HAAKJE ) {
00265                 if ( t1[2] != t2[2] ) return(t2[2]-t1[2]);
00266                 t1 += t1[1]; t2 += t2[1];
00267             }
00268             else { /* must be type SYMBOL */
00269                 t3 = t1 + t1[1]; t4 = t2 + t2[1];
00270                 t1 += 2; t2 += 2;
00271                 while ( t1 < t3 && t2 < t4 ) {
00272                     if ( *t1 != *t2 ) return(*t2-*t1);
00273                     if ( t1[1] != t2[1] ) return(t2[1]-t1[1]);
00274                     t1 += 2; t2 += 2;
00275                 }
00276                 if ( t1 < t3 ) return(-1);
00277                 if ( t2 < t4 ) return(1);
00278             }
00279         }
00280         else return(*t2-*t1);
00281     }
00282     if ( t1 < tstop1 ) return(-1);
00283     if ( t2 < tstop2 ) return(1);
00284     return(0);
00285 }
00286
00287 #endif
00288
00289 /*
00290     #] ComparePoly :
00291     #[ ConvertToPoly :
00292 */
00305 static int FirstWarnConvertToPoly = 1;
00306
00307 int ConvertToPoly(PHEAD WORD *term, WORD *outterm, WORD *comlist, WORD par)
00308 {
00309     WORD *tout, *tstop, ncoef, *t, *r, *tt, *ttwo = 0;
00310     int i, action = 0;
00311     tt = term + *term;
00312     ncoef = ABS(tt[-1]);
00313     tstop = tt - ncoef;
00314     tout = outterm+1;
00315     t = term + 1;
00316     if ( comlist[2] == DOALL ) {
00317         while ( t < tstop ) {
00318             if ( *t == SYMBOL ) {
00319                 r = t+2;
00320                 t += t[1];
00321                 while ( r < t ) {
00322                     if ( r[1] > 0 ) {
00323                         *tout++ = SYMBOL;
00324                         *tout++ = 4;
00325                         *tout++ = r[0];
00326                         *tout++ = r[1];
00327                     }
00328                     else {
00329                         tout[1] = SYMBOL;
00330                         tout[2] = 4;
00331                         tout[3] = r[0];
00332                         tout[4] = -1;
00333                         i = FindSubterm(tout+1);
00334                         *tout++ = SYMBOL;
00335                         *tout++ = 4;
00336                         *tout++ = MAXVARIABLES-i;
00337                         *tout++ = -r[1];
00338                         action = 1;
00339                     }
00340                     r += 2;
00341                 }
00342             }
00343             else if ( *t == DOTPRODUCT ) {
00344                 r = t + 2;
00345                 t += t[1];
00346                 while ( r < t ) {
00347                     tout[1] = DOTPRODUCT;
00348                     tout[2] = 5;
00349                     tout[3] = r[0];
00350                     tout[4] = r[1];
00351                     if ( r[2] < 0 ) {

```

```

00352         tout[5] = -1;
00353     }
00354     else {
00355         tout[5] = 1;
00356     }
00357     i = FindSubterm(tout+1);
00358     *tout++ = SYMBOL;
00359     *tout++ = 4;
00360     *tout++ = MAXVARIABLES-i;
00361     *tout++ = ABS(r[2]);
00362     r += 3;
00363     action = 1;
00364 }
00365 }
00366 else if ( *t == VECTOR ) {
00367     r = t + 2;
00368     t += t[1];
00369     while ( r < t ) {
00370         tout[1] = VECTOR;
00371         tout[2] = 4;
00372         tout[3] = r[0];
00373         tout[4] = r[1];
00374         i = FindSubterm(tout+1);
00375         *tout++ = SYMBOL;
00376         *tout++ = 4;
00377         *tout++ = MAXVARIABLES-i;
00378         *tout++ = 1;
00379         r += 2;
00380         action = 1;
00381     }
00382 }
00383 else if ( *t == INDEX ) {
00384     r = t + 2;
00385     t += t[1];
00386     while ( r < t ) {
00387         tout[1] = INDEX;
00388         tout[2] = 3;
00389         tout[3] = r[0];
00390         i = FindSubterm(tout+1);
00391         *tout++ = SYMBOL;
00392         *tout++ = 4;
00393         *tout++ = MAXVARIABLES-i;
00394         *tout++ = 1;
00395         r++;
00396         action = 1;
00397     }
00398 }
00399 else if ( *t == HAAKJE ) {
00400     if ( par ) {
00401         tout[0] = 1; tout[1] = 1; tout[2] = 3;
00402         *outterm = (tout+3)-outterm;
00403         if ( NormPolyTerm(BHEAD outterm) < 0 ) return(-1);
00404         tout = outterm + *outterm;
00405         tout -= 3;
00406         i = t[1]; NCOPY(tout,t,i);
00407         ttwo = tout-1;
00408     }
00409     else { t += t[1]; }
00410 }
00411 else if ( *t >= FUNCTION ) {
00412     i = FindSubterm(t);
00413     t += t[1];
00414     *tout++ = SYMBOL;
00415     *tout++ = 4;
00416     *tout++ = MAXVARIABLES-i;
00417     *tout++ = 1;
00418     action = 1;
00419 }
00420 else {
00421     if ( FirstWarnConvertToPoly ) {
00422         MLOCK(ErrorMessageLock);
00423         MesPrint("Illegal object in conversion to polynomial notation");
00424         MUNLOCK(ErrorMessageLock);
00425         FirstWarnConvertToPoly = 0;
00426     }
00427     return(-1);
00428 }
00429 }
00430 NCOPY(tout,tstop,nccoef)
00431 if ( ttwo ) {
00432     WORD hh = *ttwo;
00433     *ttwo = tout-ttwo;
00434     if ( ( i = NormPolyTerm(BHEAD ttwo) ) >= 0 ) i = action;
00435     tout = ttwo + *ttwo;
00436     *ttwo = hh;
00437     *outterm = tout - outterm;
00438 }

```

```

00439     else {
00440         *outterm = tout-outterm;
00441         if ( ( i = NormPolyTerm(BHEAD outterm) ) >= 0 ) i = action;
00442     }
00443 }
00444 else if ( comlist[2] == ONLYFUNCTIONS ) {
00445     while ( t < tstop ) {
00446         if ( *t >= FUNCTION ) {
00447             if ( comlist[1] == 3 ) {
00448                 i = FindSubterm(t);
00449                 t += t[1];
00450                 *tout++ = SYMBOL;
00451                 *tout++ = 4;
00452                 *tout++ = MAXVARIABLES-i;
00453                 *tout++ = 1;
00454                 action = 1;
00455             }
00456             else {
00457                 for ( i = 3; i < comlist[1]; i++ ) {
00458                     if ( *t == comlist[i] ) break;
00459                 }
00460                 if ( i < comlist[1] ) {
00461                     i = FindSubterm(t);
00462                     t += t[1];
00463                     *tout++ = SYMBOL;
00464                     *tout++ = 4;
00465                     *tout++ = MAXVARIABLES-i;
00466                     *tout++ = 1;
00467                     action = 1;
00468                 }
00469                 else {
00470                     i = t[1]; NCOPY(tout,t,i);
00471                 }
00472             }
00473         }
00474         else {
00475             i = t[1]; NCOPY(tout,t,i);
00476         }
00477     }
00478     NCOPY(tout,tstop,ncoef)
00479     *outterm = tout-outterm;
00480     Normalize(BHEAD outterm);
00481     i = action;
00482 }
00483 else {
00484     MLOCK(ErrorMessageLock);
00485     MesPrint("Illegal internal code in conversion to polynomial notation");
00486     MUNLOCK(ErrorMessageLock);
00487     i = -1;
00488 }
00489 return(i);
00490 }
00491
00492 /*
00493     #] ConvertToPoly :
00494     #[ LocalConvertToPoly :
00495 */
00510 int LocalConvertToPoly(PHEAD WORD *term, WORD *outterm, WORD startebuf, WORD par)
00511 {
00512     WORD *tout, *tstop, ncoef, *t, *r, *tt, *ttwo = 0;
00513     int i, action = 0;
00514     tt = term + *term;
00515     ncoef = ABS(tt[-1]);
00516     tstop = tt - ncoef;
00517     tout = outterm+1;
00518     t = term + 1;
00519     while ( t < tstop ) {
00520         if ( *t == SYMBOL ) {
00521             r = t+2;
00522             t += t[1];
00523             while ( r < t ) {
00524                 if ( r[1] > 0 ) {
00525                     *tout++ = SYMBOL;
00526                     *tout++ = 4;
00527                     *tout++ = r[0];
00528                     *tout++ = r[1];
00529                 }
00530                 else {
00531                     tout[1] = SYMBOL;
00532                     tout[2] = 4;
00533                     tout[3] = r[0];
00534                     tout[4] = -1;
00535                     i = FindLocalSubterm(BHEAD tout+1,startebuf);
00536                     *tout++ = SYMBOL;
00537                     *tout++ = 4;
00538                     *tout++ = MAXVARIABLES-i;
00539                     *tout++ = -r[1];

```

```

00540         action = 1;
00541     }
00542     r += 2;
00543 }
00544 }
00545 else if ( *t == DOTPRODUCT ) {
00546     r = t + 2;
00547     t += t[1];
00548     while ( r < t ) {
00549         tout[1] = DOTPRODUCT;
00550         tout[2] = 5;
00551         tout[3] = r[0];
00552         tout[4] = r[1];
00553         if ( r[2] < 0 ) {
00554             tout[5] = -1;
00555         }
00556         else {
00557             tout[5] = 1;
00558         }
00559         i = FindLocalSubterm(BHEAD tout+1,startebuf);
00560         *tout++ = SYMBOL;
00561         *tout++ = 4;
00562         *tout++ = MAXVARIABLES-i;
00563         *tout++ = ABS(r[2]);
00564         r += 3;
00565         action = 1;
00566     }
00567 }
00568 else if ( *t == VECTOR ) {
00569     r = t + 2;
00570     t += t[1];
00571     while ( r < t ) {
00572         tout[1] = VECTOR;
00573         tout[2] = 4;
00574         tout[3] = r[0];
00575         tout[4] = r[1];
00576         i = FindLocalSubterm(BHEAD tout+1,startebuf);
00577         *tout++ = SYMBOL;
00578         *tout++ = 4;
00579         *tout++ = MAXVARIABLES-i;
00580         *tout++ = 1;
00581         r += 2;
00582         action = 1;
00583     }
00584 }
00585 else if ( *t == INDEX ) {
00586     r = t + 2;
00587     t += t[1];
00588     while ( r < t ) {
00589         tout[1] = INDEX;
00590         tout[2] = 3;
00591         tout[3] = r[0];
00592         i = FindLocalSubterm(BHEAD tout+1,startebuf);
00593         *tout++ = SYMBOL;
00594         *tout++ = 4;
00595         *tout++ = MAXVARIABLES-i;
00596         *tout++ = 1;
00597         r++;
00598         action = 1;
00599     }
00600 }
00601 else if ( *t == HAAKJE ) {
00602     if ( par ) {
00603         tout[0] = 1; tout[1] = 1; tout[2] = 3;
00604         outterm = (tout+3)-outterm;
00605         if ( NormPolyTerm(BHEAD outterm) < 0 ) return(-1);
00606         tout = outterm + *outterm;
00607         tout -= 3;
00608         i = t[1]; NCOPY(tout,t,i);
00609         ttwo = tout-1;
00610     }
00611     else { t += t[1]; }
00612 }
00613 else if ( *t >= FUNCTION ) {
00614     i = FindLocalSubterm(BHEAD t,startebuf);
00615     t += t[1];
00616     *tout++ = SYMBOL;
00617     *tout++ = 4;
00618     *tout++ = MAXVARIABLES-i;
00619     *tout++ = 1;
00620     action = 1;
00621 }
00622 else {
00623     if ( FirstWarnConvertToPoly ) {
00624         MLOCK(ErrorMessageLock);
00625         MesPrint("Illegal object in conversion to polynomial notation");
00626         MUNLOCK(ErrorMessageLock);

```

```

00627         FirstWarnConvertToPoly = 0;
00628     }
00629     return(-1);
00630 }
00631 }
00632 NCOPY(tout,tstop,ncoef)
00633 if ( ttwo ) {
00634     WORD hh = *ttwo;
00635     *ttwo = tout-ttwo;
00636     if ( ( i = NormPolyTerm(BHEAD ttwo) ) >= 0 ) i = action;
00637     tout = ttwo + *ttwo;
00638     *ttwo = hh;
00639     *outterm = tout - outterm;
00640 }
00641 else {
00642     *outterm = tout-outterm;
00643     if ( ( i = NormPolyTerm(BHEAD outterm) ) >= 0 ) i = action;
00644 }
00645 return(i);
00646 }
00647
00648 /*
00649     #] LocalConvertToPoly :
00650     #[ ConvertFromPoly :
00651
00652     Converts a generic term from polynomial notation to the original
00653     in which the extra symbols have been replaced by their values.
00654     The output is in outterm.
00655     We only deal with the extra symbols in the range from < i <= to
00656     The output has to be sent to TestSub because it may contain
00657     subexpressions when extra symbols have been replaced.
00658 */
00659
00660 int ConvertFromPoly(PHEAD WORD *term, WORD *outterm, WORD from, WORD to, WORD offset, WORD par)
00661 {
00662     WORD *tout, *tstop, *tstop1, ncoef, *t, *r, *tt;
00663     int i;
00664     /* first = 1; */
00665     tt = term + *term;
00666     tout = outterm+1;
00667     ncoef = ABS(tt[-1]);
00668     tstop = tt - ncoef;
00669     /*
00670     r = t = term + 1;
00671     while ( t < tstop ) {
00672         if ( *t == SYMBOL ) {
00673             tstop1 = t + t[1];
00674             tt = t + 2;
00675             while ( tt < tstop1 ) {
00676                 if ( ( *tt < MAXVARIABLES - to )
00677                     || ( *tt >= MAXVARIABLES - from ) ) {
00678                     tt += 2;
00679                 }
00680                 else break;
00681             }
00682             if ( tt >= tstop1 ) { t = tstop1; continue; }
00683             while ( r < t ) *tout++ = *r++;
00684             t += 2;
00685             first = 0;
00686             while ( t < tstop1 ) {
00687                 if ( ( *t < MAXVARIABLES - to )
00688                     || ( *t >= MAXVARIABLES - from ) ) {
00689                     *tout++ = SYMBOL;
00690                     *tout++ = 4;
00691                     *tout++ = *t++;
00692                     *tout++ = *t++;
00693                 }
00694                 else {
00695                     *tout++ = SUBEXPRESSION;
00696                     *tout++ = SUBEXPSIZE;
00697                     *tout++ = MAXVARIABLES - *t++ + offset;
00698                     *tout++ = *t++;
00699                     if ( par ) *tout++ = AT.ebufnum;
00700                     else *tout++ = AM.sbufnum;
00701                     FILLSUB(tout)
00702                 }
00703             }
00704             r = t;
00705         }
00706         else {
00707             t += t[1];
00708         }
00709     }
00710     if ( first ) {
00711         i = *term; t = term;
00712         NCOPY(outterm,t,i);
00713         return(*term);

```

```

00714     }
00715     while ( r < t ) *tout++ = *r++;
00716     NCOPY(tout,tstop,ncoef)
00717     *outterm = tout-outterm;
00718 /*
00719     t = term + 1;
00720     while ( t < tstop ) {
00721         if ( *t == SYMBOL ) {
00722             tstop1 = t + t[1];
00723             tt = t + 2;
00724             while ( tt < tstop1 ) {
00725                 if ( ( *tt < MAXVARIABLES - to )
00726                     || ( *tt >= MAXVARIABLES - from ) ) {
00727                     tt += 2;
00728                 }
00729                 else {
00730                     *tout++ = SUBEXPRESSION;
00731                     *tout++ = SUBEXPSIZE;
00732                     *tout++ = MAXVARIABLES - *tt++ + offset;
00733                     *tout++ = *tt++;
00734                     if ( par ) *tout++ = AT.ebufnum;
00735                     else      *tout++ = AM.sbufnum;
00736                     FILLSUB(tout)
00737                 }
00738             }
00739             r = tout; t += 2;
00740             *tout++ = SYMBOL; *tout++ = 0;
00741             while ( t < tstop1 ) {
00742                 if ( ( *t < MAXVARIABLES - to )
00743                     || ( *t >= MAXVARIABLES - from ) ) {
00744                     *tout++ = *t++;
00745                     *tout++ = *t++;
00746                 }
00747                 else { t += 2; }
00748             }
00749             r[1] = tout - r;
00750             if ( r[1] <= 2 ) tout = r;
00751         }
00752         else {
00753             i = t[1]; NCOPY(tout,t,i)
00754         }
00755     }
00756     NCOPY(tout,tstop,ncoef)
00757     *outterm = tout-outterm;
00758     return(*outterm);
00759 }
00760
00761 /*
00762     #] ConvertFromPoly :
00763     #[ FindSubterm :
00764
00765     In this routine we look up a variable.
00766     If we don't find it we will enter it in the subterm compiler buffer
00767     Searching is by tree structure.
00768     Adding changes the tree.
00769
00770     Notice that in TFORM we should be in sequential mode.
00771 */
00772
00773 int FindSubterm(WORD *subterm)
00774 {
00775     WORD old[5], *ss, *term;
00776     int number;
00777     CBUF *C = cbuf + AM.sbufnum;
00778     LONG oldCpointer;
00779     term = subterm-1;
00780     ss = subterm+subterm[1];
00781 /*
00782     Convert to proper term
00783 */
00784     old[0] = *term; old[1] = ss[0]; old[2] = ss[1]; old[3] = ss[2]; old[4] = ss[3];
00785     ss[0] = 1; ss[1] = 1; ss[2] = 3; ss[3] = 0; *term = subterm[1]+4;
00786 /*
00787     We may have to add the term to the compiler
00788     buffer and then to the tree. This cannot be done in parallel and
00789     hence we have to set a lock.
00790 */
00791     LOCK(AM.sbuflock);
00792
00793     oldCpointer = C->Pointer-C->Buffer; /* Offset of course !!!!!*/
00794     AddRHS(AM.sbufnum,1);
00795     AddNtoC(AM.sbufnum,*term,term,8);
00796     AddToCB(C,0)
00797 /*
00798     See whether we have this one already. If not, insert it in the tree.
00799 */
00800     number = InsTree(AM.sbufnum,C->numrhs);

```

```

00801 /*
00802     Restore old values and return what is needed.
00803 */
00804 if ( number < (C->numrhs) ) { /* It existed already */
00805     C->Pointer = oldCpointer + C->Buffer;
00806     C->numrhs--;
00807 }
00808 else {
00809     GETIDENTITY
00810     WORD dim = DimensionSubterm(subterm);
00811
00812     if ( dim == -MAXPOSITIVE ) { /* Give error message but continue */
00813         WORD *old = AN.currentTerm;
00814         AN.currentTerm = term;
00815         MLOCK(ErrorMessageLock);
00816         MesPrint("Dimension out of range in %t");
00817         MUNLOCK(ErrorMessageLock);
00818         AN.currentTerm = old;
00819     }
00820 /*
00821     Store the dimension
00822 */
00823     C->dimension[number] = dim;
00824 }
00825 UNLOCK(AM.sbuflock);
00826
00827 *term = old[0]; ss[0] = old[1]; ss[1] = old[2]; ss[2] = old[3]; ss[3] = old[4];
00828 return(number);
00829 }
00830
00831 /*
00832     #] FindSubterm :
00833     #[ FindLocalSubterm :
00834
00835     In this routine we look up a variable.
00836     If we don't find it we will enter it in the subterm compiler buffer
00837     Searching is by tree structure.
00838     Adding changes the tree.
00839
00840     Notice that in TFORM we should be in sequential mode.
00841 */
00842
00843 int FindLocalSubterm(PHEAD WORD *subterm, WORD startebuf)
00844 {
00845     WORD old[5], *ss, *term, i, j, *t1, *t2;
00846     int number;
00847     CBUF *C = cbuf + AT.ebufnum;
00848     term = subterm-1;
00849     ss = subterm+subterm[1];
00850 /*
00851     Convert to proper term
00852 */
00853     old[0] = *term; old[1] = ss[0]; old[2] = ss[1]; old[3] = ss[2]; old[4] = ss[3];
00854     ss[0] = 1; ss[1] = 1; ss[2] = 3; ss[3] = 0; *term = subterm[1]+4;
00855 /*
00856     First see whether we have this one already in the global buffer.
00857 */
00858     number = FindTree(AM.sbufnum,term);
00859     if ( number > 0 ) goto wearehappy;
00860 /*
00861     Now look whether it is in the ebufnum between startebuf and numrhs
00862     Note however that we need an offset of (numxsymbol-startebuf)
00863 */
00864     for ( i = startebuf+1; i <= C->numrhs; i++ ) {
00865         t1 = C->rhs[i]; t2 = term;
00866         if ( *t1 == *t2 ) {
00867             j = *t1;
00868             while ( *t1 == *t2 && j > 0 ) { t1++; t2++; j--; }
00869             if ( j <= 0 ) {
00870                 number = i-startebuf+numxsymbol;
00871                 goto wearehappy;
00872             }
00873         }
00874     }
00875 /*
00876     Now we have to add it to cbuf[AT.ebufnum]
00877 */
00878     AddRHS(AT.ebufnum,1);
00879     AddNtoC(AT.ebufnum,*term,term,9);
00880     AddToCB(C,0)
00881     number = C->numrhs-startebuf+numxsymbol;
00882 wearehappy:
00883     *term = old[0]; ss[0] = old[1]; ss[1] = old[2]; ss[2] = old[3]; ss[3] = old[4];
00884     return(number);
00885 }
00886
00887 /*

```



```

00888      #] FindLocalSubterm :
00889      #[ PrintSubtermList :
00890
00891      Prints all the expressions in the subterm compiler buffer.
00892      The format is such that they give definitions of the temporary
00893      variables of which the contents are stored in this buffer.
00894      These variables have the names Z_123 etc.
00895  */
00896
00897 void PrintSubtermList(int from,int to)
00898 {
00899     UBYTE buffer[80], *out, outbuffer[300];
00900     int first, i, ii, inc = 1;
00901     WORD *term;
00902     CBUF *C = cbuf + AM.sbufnum;
00903  /*
00904     if ( to < from ) inc = -1;
00905     if ( to == from ) inc = 0;
00906  */
00907     if ( from <= to ) {
00908         inc = 1; to += inc;
00909     }
00910     else {
00911         inc = -1; to += inc;
00912     }
00913     AO.OutFill = AO.OutputLine = outbuffer;
00914     AO.OutStop = AO.OutputLine+AC.LineLength;
00915     AO.IsBracket = 0;
00916     AO.OutSkip = 3;
00917
00918     if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {
00919         TokenToLine((UBYTE *)"      ");
00920         AO.OutSkip = 7;
00921     }
00922     else if ( ( AO.Optimize.debugflags & 1 ) == 1 ) {}
00923     else if ( AO.OutSkip > 0 ) {
00924         for ( i = 0; i < AO.OutSkip; i++ ) TokenToLine((UBYTE *)" ");
00925     }
00926     i = from;
00927     do {
00928         if ( ( AO.Optimize.debugflags & 1 ) == 1 ) {
00929             TokenToLine((UBYTE *)"id ");
00930             for ( ii = 3; ii < AO.OutSkip; ii++ ) TokenToLine((UBYTE *)" ");
00931         }
00932  /*
00933         if ( AC.OutputMode == NORMALFORMAT ) {
00934             TokenToLine((UBYTE *)"id ");
00935         }
00936  */
00937         else if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {}
00938         else { TokenToLine((UBYTE *)" "); }
00939
00940         out = StrCopy((UBYTE *)AC.extrasym,buffer);
00941         if ( AC.extrasymbols == 0 ) {
00942             out = NumCopy(i,out);
00943             out = StrCopy((UBYTE *)"_",out);
00944         }
00945         else if ( AC.extrasymbols == 1 ) {
00946             out = AddArrayIndex(i,out);
00947         }
00948         out = StrCopy((UBYTE *)"=",out);
00949         TokenToLine(buffer);
00950         term = C->rhs[i];
00951         first = 1;
00952         if ( *term == 0 ) {
00953             out = StrCopy((UBYTE *)"0",buffer);
00954             if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE ) {
00955                 out = StrCopy((UBYTE *)";",out);
00956             }
00957             TokenToLine(buffer);
00958         }
00959         else {
00960             while ( *term ) {
00961                 if ( WriteInnerTerm(term,first) ) Terminate(-1);
00962                 term += *term;
00963                 first = 0;
00964             }
00965             if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE ) {
00966                 out = StrCopy((UBYTE *)";",buffer);
00967                 TokenToLine(buffer);
00968             }
00969         }
00970  /*
00971         There is a problem with FiniLine because it prepares for a
00972         continuation line in fortran mode.
00973         But the next statement should start on a blank line.
00974  */

```

```

00975  /*
00976      FiniLine();
00977      if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {
00978          AO.OutFill = AO.OutputLine;
00979          TokenToLine((UBYTE *)"      ");
00980          AO.OutSkip = 7;
00981      }
00982  */
00983      if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {
00984          AO.OutSkip = 6;
00985          FiniLine();
00986          AO.OutSkip = 7;
00987      }
00988      else {
00989          FiniLine();
00990      }
00991      i += inc;
00992  } while ( i != to );
00993 }
00994
00995  /*
00996      #] PrintSubtermList :
00997      #[ PrintExtraSymbol :
00998
00999      Prints the definition of extra symbol num as the contents
01000      of the expression in terms.
01001      The parameter par has three options:
01002          EXTRASYMBOL      num is interpreted as the number of an extra symbol
01003          REGULARSYMBOL    num is interpreted as the number of a symbol.
01004                          It could still be an extra symbol.
01005          EXPRESSIONNUMBER num is the number of an expression.
01006      terms contains the rhs expression.
01007  */
01008
01009 void PrintExtraSymbol(int num, WORD *terms,int par)
01010 {
01011     UBYTE buffer[80], *out, outbuffer[300];
01012     int first, i;
01013     WORD *term;
01014
01015     AO.OutFill = AO.OutputLine = outbuffer;
01016     AO.OutStop = AO.OutputLine+AC.LineLength;
01017     AO.IsBracket = 0;
01018
01019     if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {
01020         TokenToLine((UBYTE *)"      ");
01021         AO.OutSkip = 7;
01022     }
01023     else if ( ( AO.Optimize.debugflags & 1 ) == 1 ) {
01024         TokenToLine((UBYTE *)"id ");
01025         for ( i = 3; i < AO.OutSkip; i++ ) TokenToLine((UBYTE *)" ");
01026     }
01027     else if ( AO.OutSkip > 0 ) {
01028         for ( i = 0; i < AO.OutSkip; i++ ) TokenToLine((UBYTE *)" ");
01029     }
01030     out = buffer;
01031     switch ( par ) {
01032     case REGULARSYMBOL:
01033         if ( num >= MAXVARIABLES-cbuf[AM.sbufnum].numrhs ) {
01034             num = MAXVARIABLES-num;
01035         }
01036         else {
01037             out = StrCopy(FindSymbol(num),out);
01038             /* out = StrCopy(VARNAME(symbols,num),out); */
01039             break;
01040         }
01041         /* fall through */
01042     case EXTRASYMBOL:
01043         out = StrCopy(FindExtraSymbol(num),out);
01044         /*
01045             out = StrCopy((UBYTE *)AC.extrasym,out);
01046             if ( AC.extrasymbols == 0 ) {
01047                 out = NumCopy(num,out);
01048                 out = StrCopy((UBYTE *)"_",out);
01049             }
01050             else if ( AC.extrasymbols == 1 ) {
01051                 out = AddArrayIndex(num,out);
01052             }
01053         */
01054         break;
01055     case EXPRESSIONNUMBER:
01056         out = StrCopy(EXPRNAME(num),out);
01057         break;
01058     default:
01059         MesPrint("Illegal option in PrintExtraSymbol");
01060         Terminate(-1);
01061     }
}

```

```

01062     out = StrCopy((UBYTE *) "=", out);
01063     TokenToLine(buffer);
01064     term = terms;
01065     first = 1;
01066     if ( *term == 0 ) {
01067         out = StrCopy((UBYTE *) "0", buffer);
01068         TokenToLine(buffer);
01069     }
01070     else {
01071         while ( *term ) {
01072             if ( WriteInnerTerm(term, first) ) Terminate(-1);
01073             term += *term;
01074             first = 0;
01075         }
01076     }
01077     if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE ) {
01078         out = StrCopy((UBYTE *) ";", buffer);
01079         TokenToLine(buffer);
01080     }
01081     FiniLine();
01082 }
01083
01084 /*
01085     #] PrintExtraSymbol :
01086     #[ FindSubexpression :
01087
01088     In this routine we look up a subexpression.
01089     If we don't find it we will enter it in the subterm compiler buffer
01090     Searching is by tree structure.
01091     Adding changes the tree.
01092
01093     Notice that in TFORM we should be in sequential mode.
01094 */
01095
01096 int FindSubexpression(WORD *subexpr)
01097 {
01098     WORD *term;
01099     int number;
01100     CBUF *C = cbuf + AM.sbufnum;
01101     LONG oldCpointer;
01102
01103     term = subexpr;
01104     while ( *term ) term += *term;
01105     number = term - subexpr;
01106     /*
01107         We may have to add the subexpression to the tree.
01108         This requires a lock.
01109     */
01110     LOCK(AM.sbuflock);
01111
01112     oldCpointer = C->Pointer-C->Buffer; /* Offset of course !!!!!*/
01113     AddRHS(AM.sbufnum, 1);
01114     /*
01115         Add the terms to the compiler buffer. Paste on a zero.
01116     */
01117     AddNtoC(AM.sbufnum, number, subexpr, 10);
01118     AddToCB(C, 0)
01119     /*
01120         See whether we have this one already. If not, insert it in the tree.
01121     */
01122     number = InsTree(AM.sbufnum, C->numrhs);
01123     /*
01124         Restore old values and return what is needed.
01125     */
01126     if ( number < (C->numrhs) ) { /* It existed already */
01127         C->Pointer = oldCpointer + C->Buffer;
01128         C->numrhs--;
01129     }
01130     else {
01131         GETIDENTITY
01132         WORD dim = DimensionExpression(BHEAD subexpr);
01133     /*
01134         Store the dimension
01135     */
01136         C->dimension[number] = dim;
01137     }
01138
01139     UNLOCK(AM.sbuflock);
01140
01141     return(number);
01142 }
01143
01144 /*
01145     #] FindSubexpression :
01146     #[ ExtraSymFun :
01147 */
01148

```

```

01149 int ExtraSymFun(PHEAD WORD *term,WORD level)
01150 {
01151     WORD *oldworkpointer = AT.WorkPointer;
01152     WORD *termout, *t1, *t2, *t3, *tstop, *tend, i;
01153     int retval = 0;
01154     tend = termout = term + *term;
01155     tstop = tend - ABS(tend[-1]);
01156     t3 = t1 = term+1; t2 = termout+1;
01157 /*
01158     First refind the function(s). There is at least one.
01159 */
01160     while ( t1 < tstop ) {
01161         if ( *t1 == EXTRASYMFUN && t1[1] == FUNHEAD+2 ) {
01162             if ( t1[FUNHEAD] == -SNUMBER && t1[FUNHEAD+1] <= numxsymbol
01163                 && t1[FUNHEAD+1] > 0 ) {
01164                 i = t1[FUNHEAD+1];
01165             }
01166             else if ( t1[FUNHEAD] == -SYMBOL && t1[FUNHEAD+1] < MAXVARIABLES
01167                 && t1[FUNHEAD+1] >= MAXVARIABLES-numxsymbol ) {
01168                 i = MAXVARIABLES - t1[FUNHEAD+1];
01169             }
01170             else goto nocase;
01171             while ( t3 < t1 ) *t2++ = *t3++;
01172 /*
01173             Now inset the rhs pointer
01174 */
01175             *t2++ = SUBEXPRESSION;
01176             *t2++ = SUBEXPSIZE;
01177             *t2++ = i;
01178             *t2++ = 1;
01179             *t2++ = AM.sbufnum;
01180             FILLSUB(t2)
01181             t3 = t1 = t1 + t1[1];
01182         }
01183         else if ( *t1 == EXTRASYMFUN && t1[1] == FUNHEAD ) {
01184             while ( t3 < t1 ) *t2++ = *t3++;
01185             t3 = t1 = t1 + t1[1];
01186         }
01187         else {
01188 nocase:;
01189             t1 = t1 + t1[1];
01190         }
01191     }
01192     while ( t3 < tend ) *t2++ = *t3++;
01193     *termout = t2 - termout;
01194     AT.WorkPointer = t2;
01195     if ( AT.WorkPointer >= AT.WorkTop ) {
01196         MLOCK(ErrorMessageLock);
01197         MesWork();
01198         MUNLOCK(ErrorMessageLock);
01199         AT.WorkPointer = oldworkpointer;
01200         return(-1);
01201     }
01202     retval = Generator(BHEAD termout,level);
01203     AT.WorkPointer = oldworkpointer;
01204     if ( retval < 0 ) {
01205         MLOCK(ErrorMessageLock);
01206         MesCall("ExtraSymFun");
01207         MUNLOCK(ErrorMessageLock);
01208     }
01209     return(retval);
01210 }
01211 /*
01212 #] ExtraSymFun :
01213 #[ PruneExtraSymbols :
01214 */
01215 */
01216
01217 int PruneExtraSymbols(WORD downto)
01218 {
01219     CBUF *C = cbuf + AM.sbufnum;
01220     if ( downto < C->numrhs && downto >= 0 ) { /* !!!!! */
01221         ClearTree(AM.sbufnum);
01222         C->numrhs = downto;
01223         if ( downto == 0 ) {
01224             C->Pointer = C->Buffer;
01225         }
01226         else {
01227             WORD *w = C->rhs[downto], i;
01228             while ( *w ) w += *w;
01229             C->Pointer = w+1;
01230             for ( i = 1; i <= downto; i++ ) {
01231                 InsTree(AM.sbufnum,i);
01232             }
01233         }
01234     }
01235     return(0);

```

```

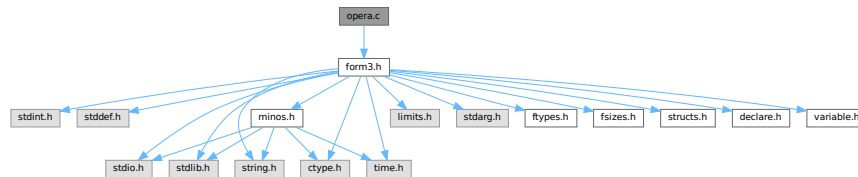
01236 }
01237
01238 /*
01239      #] PruneExtraSymbols :
01240 */

```

4.91 opera.c File Reference

```
#include "form3.h"
```

Include dependency graph for opera.c:



Functions

- int [EpFind](#) (PHEAD WORD *term, WORD *params)
- int [EpCon](#) (PHEAD WORD *term, WORD *params, WORD num, WORD level)
- WORD [EpGen](#) (WORD number, WORD *inlist, WORD *kron, WORD *perm, WORD sgn)
- WORD [Trick](#) (WORD *in, [TRACES](#) *t)
- int [Trace4no](#) (WORD number, WORD *kron, [TRACES](#) *t)
- int [Trace4](#) (PHEAD WORD *term, WORD *params, WORD num, WORD level)
- int [Trace4Gen](#) (PHEAD [TRACES](#) *t, WORD number)
- WORD [TraceNno](#) (WORD number, WORD *kron, [TRACES](#) *t)
- int [TraceN](#) (PHEAD WORD *term, WORD *params, WORD num, WORD level)
- int [TraceNgen](#) (PHEAD [TRACES](#) *t, WORD number)
- int [Traces](#) (PHEAD WORD *term, WORD *params, WORD num, WORD level)
- int [TraceFind](#) (PHEAD WORD *term, WORD *params)
- int [Chisholm](#) (PHEAD WORD *term, WORD level)
- int [TenVecFind](#) (PHEAD WORD *term, WORD *params)
- int [TenVec](#) (PHEAD WORD *term, WORD *params, WORD num, WORD level)

4.91.1 Detailed Description

Contains the 'operations' These are the trace routines, the contractions of the Levi-Civita tensors and the tensor to vector/vector to tensor routines. The trace and contraction routines are done in a special way (see commentary with the FIXEDGLOBALS struct)

Definition in file [opera.c](#).

4.91.2 Function Documentation

4.91.2.1 EpFind()

```

int EpFind (
    PHEAD WORD * term,
    WORD * params )

```

Definition at line 58 of file [opera.c](#).

4.91.2.2 EpfCon()

```
int EpfCon (
    PHEAD WORD * term,
    WORD * params,
    WORD num,
    WORD level )
```

Definition at line [212](#) of file [opera.c](#).

4.91.2.3 EpfGen()

```
WORD EpfGen (
    WORD number,
    WORD * inlist,
    WORD * kron,
    WORD * perm,
    WORD sgn )
```

Definition at line [282](#) of file [opera.c](#).

4.91.2.4 Trick()

```
WORD Trick (
    WORD * in,
    TRACES * t )
```

Definition at line [331](#) of file [opera.c](#).

4.91.2.5 Trace4no()

```
int Trace4no (
    WORD number,
    WORD * kron,
    TRACES * t )
```

Definition at line [418](#) of file [opera.c](#).

4.91.2.6 Trace4()

```
int Trace4 (
    PHEAD WORD * term,
    WORD * params,
    WORD num,
    WORD level )
```

Definition at line [695](#) of file [opera.c](#).

4.91.2.7 Trace4Gen()

```
int Trace4Gen (
    PHEAD TRACES * t,
    WORD number )
```

Definition at line [858](#) of file [opera.c](#).

4.91.2.8 TraceNno()

```
WORD TraceNno (
    WORD number,
    WORD * kron,
    TRACES * t )
```

Definition at line [1343](#) of file [opera.c](#).

4.91.2.9 TraceN()

```
int TraceN (
    PHEAD WORD * term,
    WORD * params,
    WORD num,
    WORD level )
```

Definition at line [1384](#) of file [opera.c](#).

4.91.2.10 TraceNgen()

```
int TraceNgen (
    PHEAD TRACES * t,
    WORD number )
```

Definition at line [1449](#) of file [opera.c](#).

4.91.2.11 Traces()

```
int Traces (
    PHEAD WORD * term,
    WORD * params,
    WORD num,
    WORD level )
```

Definition at line [1834](#) of file [opera.c](#).

4.91.2.12 TraceFind()

```
int TraceFind (
    PHEAD WORD * term,
    WORD * params )
```

Definition at line [1856](#) of file [opera.c](#).

4.91.2.13 Chisholm()

```
int Chisholm (
    PHEAD WORD * term,
    WORD level )
```

Definition at line 1966 of file [opera.c](#).

4.91.2.14 TenVecFind()

```
int TenVecFind (
    PHEAD WORD * term,
    WORD * params )
```

Definition at line 2181 of file [opera.c](#).

4.91.2.15 TenVec()

```
int TenVec (
    PHEAD WORD * term,
    WORD * params,
    WORD num,
    WORD level )
```

Definition at line 2315 of file [opera.c](#).

4.92 opera.c

[Go to the documentation of this file.](#)

```
00001
00009 /* #[ License : */
00010 /*
00011 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00012 * When using this file you are requested to refer to the publication
00013 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014 * This is considered a matter of courtesy as the development was paid
00015 * for by FOM the Dutch physics granting agency and we would like to
00016 * be able to track its scientific use to convince FOM of its value
00017 * for the community.
00018 *
00019 * This file is part of FORM.
00020 *
00021 * FORM is free software: you can redistribute it and/or modify it under the
00022 * terms of the GNU General Public License as published by the Free Software
00023 * Foundation, either version 3 of the License, or (at your option) any later
00024 * version.
00025 *
00026 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00029 * details.
00030 *
00031 * You should have received a copy of the GNU General Public License along
00032 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00033 */
00034 /* #[ License : */
00035 /*
00036 * #[ Includes : opera.c
00037 */
00038
00039 #include "form3.h"
00040 /*
00041 int hulp;
```



```

00042 */
00043
00044 /*
00045  #] Includes :
00046  #[ Operations :
00047      #[ EpfFind :          WORD EpfFind(term,params)
00048
00049      Searches for a pair of Levi-Civita tensors that should be
00050      contracted.
00051      If a match is found its settings are recorded in AT.TMout.
00052      type indicates the number of indices that is searched for,
00053      unless all are searched for (type = 0).
00054      number is the number of tensors that should survive contraction.
00055
00056 */
00057
00058 int EpfFind(PHEAD WORD *term, WORD *params)
00059 {
00060     GETBIDENTITY
00061     WORD *t, *m, *r, n1 = 0, n2, min = -1, count, fac;
00062     WORD *c1 = 0, *c2 = 0, sgn = 1;
00063     WORD *tstop, *mstop;
00064     UWORD *facto = (UWORD *)AT.WorkPointer;
00065     WORD ncoef, nfac;
00066     WORD number, type;
00067     if ( ( AT.WorkPointer = (WORD *) (facto + AM.MaxTal) ) > AT.WorkTop ) {
00068         MLOCK(ErrorMessageLock);
00069         MesWork();
00070         MUNLOCK(ErrorMessageLock);
00071         return(-1);
00072     }
00073     number = params[3];
00074     type = params[4];
00075     t = term;
00076     GETSTOP(t,tstop);
00077     t++;
00078     if ( !type ) {
00079         while ( *t != LEVICIVITA && t < tstop ) t += t[1];
00080         if ( t >= tstop ) return(0);
00081         m = t;
00082         while ( *m == LEVICIVITA && m < tstop ) { n1++; m += m[1]; }
00083     AllLev:
00084         if ( n1 <= (number+1) || n1 <= 1 ) return(0);
00085         mstop = m;
00086         m = t + t[1];
00087         do {
00088             while ( m[1] == t[1] ) {
00089                 m += FUNHEAD;
00090                 r = t+FUNHEAD;
00091                 count = fac = n1 = n2 = t[1] - FUNHEAD;
00092                 while ( n1 && n2 ) {
00093                     if ( *m > *r ) {
00094                         r++; n2--;
00095                     }
00096                     else if ( *m < *r ) {
00097                         m++; n1--;
00098                     }
00099                     else {
00100                         if ( *m >= AM.OffsetIndex &&
00101                             ( ( *m >= AM.IndDum && AC.lDefDim == fac ) ||
00102                               ( *m < AM.IndDum &&
00103                                 indices[*m-AM.OffsetIndex].dimension == fac ) ) ) {
00104                             count--;
00105                         }
00106                         r++; m++; n1--; n2--;
00107                     }
00108                 }
00109                 m += n1;
00110                 if ( min < 0 || count < min ) {
00111                     c1 = t;
00112                     c2 = m - fac - FUNHEAD;
00113                     min = count;
00114                 }
00115                 if ( m >= mstop ) break;
00116             }
00117             t += t[1];
00118         } while ( ( m = t + t[1] ) < mstop );
00119     }
00120     else {
00121         fac = type + FUNHEAD;
00122         while ( *t != LEVICIVITA && t < tstop ) t += t[1];
00123         while ( *t == LEVICIVITA && t < tstop && t[1] != fac ) t += t[1];
00124         if ( t >= tstop ) return(0);
00125         m = t;
00126         while ( *m == LEVICIVITA && m < tstop && m[1] == fac ) { n1++; m += m[1]; }
00127         goto AllLev;
00128     }
}

```

```

00129 /*
00130     We have now the two tensors that give the minimum contraction
00131     in c1 and c2.
00132     Prepare the AT.TMout array;
00133 */
00134     if ( min < 0 ) return(0); /* No matching pair! */
00135     t = c1;
00136     mstop = c2;
00137     fac = t[1] - FUNHEAD;
00138     m = AT.TMout;
00139     *m++ = 3 + (min*2); /* The full length */
00140     *m++ = CONTRACT;
00141     if ( number < 0 ) *m++ = 1;
00142     else *m++ = 0;
00143     n1 = n2 = t[1] - FUNHEAD;
00144     r = c1 + FUNHEAD;
00145     c1 = m;
00146     m = c2 + FUNHEAD;
00147     c2 = c1 + min;
00148     while ( n1 && n2 ) {
00149         if ( *m > *r ) { *c1++ = *r++; n2--; }
00150         else if ( *m < *r ) { *c2++ = *m++; n1--; }
00151         else {
00152             if ( *m < AM.OffsetIndex || ( *m < AM.IndDum &&
00153                 ( indices[*m-AM.OffsetIndex].dimension != fac ) ) ||
00154                 ( *m >= AM.IndDum && AC.lDefDim != fac ) ) {
00155                 *c1++ = *r++; *c2++ = *m++;
00156             }
00157             else { if ( ( n1 ^ n2 ) & 1 ) sgn = -sgn; r++; m++; }
00158             n1--; n2--;
00159         }
00160     }
00161     if ( n1 ) { NCOPY(c2,m,n1); }
00162     else if ( n2 ) { NCOPY(c1,r,n2); }
00163     fac -= min;
00164     m = t + t[1];
00165     while ( m < mstop ) *t++ = *m++;
00166     m += m[1];
00167     while ( m < tstop ) *t++ = *m++;
00168     *t++ = SUBEXPRESSION;
00169     *t++ = SUBEXPSIZE;
00170     *t++ = -1;
00171     *t++ = 1;
00172     *t++ = DUMMYBUFFER;
00173     FILLSUB(t)
00174     r = term;
00175     r += *r - 1;
00176     mstop = r;
00177     ncoef = REDLENG(*r);
00178     tstop = t;
00179     while ( m < mstop ) *t++ = *m++;
00180     if ( Factorial(BHEAD fac,facto,&nfac) || Mully(BHEAD (UWORD *)tstop,&ncoef,facto,nfac) ) {
00181         MLOCK(ErrorMessageLock);
00182         MesCall("EpFFind");
00183         MUNLOCK(ErrorMessageLock);
00184         SETERROR(-1)
00185     }
00186     tstop += (ABS(ncoef))*2;
00187     if ( sgn < 0 ) ncoef = -ncoef;
00188     ncoef *= 2;
00189     *tstop++ = (ncoef<0)?(ncoef-1):(ncoef+1);
00190     *term = WORDDIF(tstop,term);
00191     return(1);
00192 }
00193
00194 /*
00195     #] EpFFind :
00196     #[ EpFCon :          WORD EpFCon(term,params,num,level)
00197
00198     Contraction of two strings of indices/vectors. They come
00199     from LeviCivita tensors that are being contracted.
00200     term is the term with the subterm to be replaced.
00201     params is the full indicator:
00202         Length, number, raise, parameters.
00203     Length is the length of the parameter field.
00204     number is the number of the operation.
00205     raise tells whether level should be raised afterwards.
00206     the parameters are the two strings.
00207     level is the id level.
00208     The factorial has been multiplied in already.
00209
00210 */
00211
00212 int EpFCon(PHEAD WORD *term, WORD *params, WORD num, WORD level)
00213 {
00214     GETBIDENTITY
00215     WORD *kron, *perm, *termout, *tstop, size2;

```

```

00216     WORD *m, *t, sizes, sgn = 0, i;
00217     sizes = *params - 3;
00218     kron = AT.WorkPointer;
00219     perm = (AT.WorkPointer += sizes);
00220     termout = (AT.WorkPointer += sizes);
00221     AT.WorkPointer = (WORD *)(((UBYTE *) (AT.WorkPointer)) + AM.MaxTer);
00222     if ( AT.WorkPointer > AT.WorkTop ) {
00223         MLOCK(ErrorMessageLock);
00224         MesWork();
00225         MUNLOCK(ErrorMessageLock);
00226         return(-1);
00227     }
00228     params += 2;
00229     if ( !(*params++) ) level--;
00230     size2 = sizes>>1;
00231     if ( !size2 ) goto DoOnce;
00232     while ( ( sgn = EpfGen(size2,params,kron,perm,sgn) ) != 0 ) {
00233 DoOnce:
00234         t = term;
00235         GETSTOP(t,tstop);
00236         m = termout;
00237         tstop -= SUBEXPSIZE;
00238         while ( t < tstop ) *m++ = *t++;
00239         if ( t[2] != num || *t != SUBEXPRESSION ) {
00240             MLOCK(ErrorMessageLock);
00241             MesPrint("Serious error in EpfCon");
00242             MUNLOCK(ErrorMessageLock);
00243             return(-1);
00244         }
00245         tstop += SUBEXPSIZE;
00246         if ( sizes ) {
00247             *m++ = DELTA;
00248             *m++ = sizes + 2;
00249             t = kron;
00250             i = sizes;
00251             if ( i ) { NCOPY(m,t,i); }
00252         }
00253         t = tstop;
00254         tstop = term + *term;
00255         while ( t < tstop ) *m++ = *t++;
00256         *termout = WORDDIF(m,termout);
00257         m--;
00258         if ( sgn < 0 ) *m = - *m;
00259         if ( *termout ) {
00260             *AN.RepPoint = 1;
00261             AR.expchanged = 1;
00262             AT.WorkPointer = termout + *termout;
00263             if ( Generator(BHEAD termout,level) < 0 ) goto EpfCall;
00264         }
00265     }
00266     AT.WorkPointer = kron;
00267     return(0);
00268 EpfCall:
00269     if ( AM.tracebackflag ) {
00270         MLOCK(ErrorMessageLock);
00271         MesCall("EpfCon");
00272         MUNLOCK(ErrorMessageLock);
00273     }
00274     return(-1);
00275 }
00276
00277 /*
00278     #] EpfCon :
00279     #[ EpfGen :          WORD EpfGen(number,inlist,kron,perm,sgn)
00280 */
00281
00282 WORD EpfGen(WORD number, WORD *inlist, WORD *kron, WORD *perm, WORD sgn)
00283 {
00284     WORD i, *in2, k, a;
00285     if ( !sgn ) {
00286         in2 = inlist + number;
00287         number *= 2;
00288         for ( i = 1; i < number; i += 2 ) {
00289             *perm++ = i;
00290             *perm++ = i;
00291             *kron++ = *inlist++;
00292             *kron++ = *in2++;
00293         }
00294         if ( number <= 0 ) return(0);
00295         else return(1);
00296     }
00297     number *= 2;
00298     i = number - 1;
00299     while ( ( i -= 2 ) >= 0 ) {
00300         if ( ( k = perm[i] ) != i ) {
00301             sgn = -sgn;
00302             a = kron[i];

```

```

00303         kron[i] = kron[k];
00304         kron[k] = a;
00305     }
00306     if ( ( k = ( perm[i] += 2 ) ) < number ) {
00307         a = kron[i];
00308         kron[i] = kron[k];
00309         kron[k] = a;
00310         sgn = - sgn;
00311         for ( k = i + 2; k < number; k += 2 ) perm[k] = k;
00312         return(sgn);
00313     }
00314 }
00315 return(0);
00316 }
00317
00318 /*
00319     #] EpfGen :
00320     #[ Trick :          WORD Trick(in,t)
00321
00322     This routine implements the identity:
00323     g_(j,mu)*g_(j,nu)*g_(j,ro)=e_(mu,nu,ro,si)*g5_(j)*g_(j,si)
00324     +d_(mu,nu)*g_(j,ro)-d_(mu,ro)*g_(j,nu)+d_(nu,ro)*g_(j,mu)
00325     which is for 4 dimensions only!
00326
00327     Note that z->gamm = 1 if there is no g5 present.
00328
00329 */
00330
00331 WORD Trick(WORD *in, TRACES *t)
00332 {
00333     WORD n, n1;
00334     n = t->stap;
00335     n1 = t->step1;
00336     switch ( t->eers[n] ) {
00337     case 0: {
00338         WORD *p;
00339         p = t->pepf + t->mepf[n];
00340         *p++ = *in++;
00341         *p++ = *in++;
00342         *p++ = *in;
00343         *p = ++(t->mdum);
00344         (t->mepf[n1]) += 4;
00345         *in = t->mdum;
00346         t->gamm = - t->gamm;
00347         t->eers[n] = 5;
00348         break;
00349     }
00350     case 4: {
00351         WORD *p;
00352         p = t->pdel + t->mdel[n];
00353         (t->mepf[n1]) -= 4;
00354         (t->mdum)--;
00355         *p++ = *in++;
00356         *p = *in++;
00357         *in = *(t->pepf + t->mepf[n] + 2);
00358         (t->mdel[n1]) += 2;
00359         t->gamm = - t->gamm;
00360         break;
00361     }
00362     case 3: {
00363         t->sign1 = - t->sign1;
00364         *(t->pdel + t->mdel[n] + 1) = in[2];
00365         in[2] = in[1];
00366         break;
00367     }
00368     case 2: {
00369         t->sign1 = - t->sign1;
00370         in[2] = in[0];
00371         *(t->pdel + t->mdel[n]) = in[1];
00372         break;
00373     }
00374     case 1: {
00375         in[2] = *(t->pdel + t->mdel[n] + 1);
00376         (t->mdel[n1]) -= 2;
00377         break;
00378     }
00379     default: {
00380         return(0);
00381     }
00382 }
00383 return(--(t->eers[n]));
00384 }
00385
00386 /*
00387     #] Trick :
00388     #[ Trace4no :          WORD Trace4no(number,kron,t)
00389

```

```

00390      Takes the trace of a string of gamma matrices in 4 dimensions.
00391      There is no test for indices or vectors that are the same.
00392      The four dimensions refer to the contraction in the algebra:
00393      g_(i,a,b,c) = e_(a,b,c,d)*g_(i,5_,d) + g_(i,a)*d_(b,c)
00394                  - g_(i,b)*d_(a,c) + g_(i,c)*d_(a,b)
00395      This simplifies life very much and leads to shorter expressions
00396      than in the n dimensional case.
00397
00398      Parameters:
00399      number: the number of vectors/indices in inlist.
00400      inlist: the indices/vectors in the string.
00401      kron:   the output delta's.
00402      gamma5: the potential gamma5 in front.
00403      t:      the struct for scratch manipulations.
00404      stack:  the space to put all scratch arrays in.
00405
00406      The return value is zero if there are no more terms, 1 if a
00407      term was generated with a positive sign and -1 if a term was
00408      generated with a negative sign.
00409      The value of one is increased to two if the first 4 values
00410      in kron should be interpreted as a Levi-Civita tensor.
00411
00412      Note that kron should have more places reserved than the number
00413      of indices in inlist, because it will contain dummy indices
00414      temporarily. In principle there can be 'number*1/4' extra dummies.
00415
00416  */
00417
00418  int Trace4no(WORD number, WORD *kron, TRACES *t)
00419  {
00420      WORD i;
00421      WORD *p, *m;
00422      WORD *stop, oldsign;
00423      int retval;
00424      if ( !t->sgn ) {          /* Startup */
00425          if ( ( number < 0 ) || ( number & 1 ) ) return(0);
00426          if ( number <= 2 ) {
00427              if ( t->gamma5 == GAMMA5 ) return(0);
00428              if ( number == 2 ) {
00429                  *kron++ = *t->inlist;
00430                  *kron++ = t->inlist[1];
00431              }
00432              return(1);
00433          }
00434          t->sgn = 1;
00435          {
00436              WORD nhalf = number >> 1;
00437              WORD ndouble = number * 2;
00438              p = t->eers;
00439              t->eers = p;      p += nhalf;
00440              t->mepf = p;      p += nhalf;
00441              t->mdel = p;     p += nhalf;
00442              t->pdel = p;     p += number + nhalf;
00443              t->pepf = p;     p += ndouble;
00444              t->e4 = p;      p += number;
00445              t->e3 = p;      p += ndouble;
00446              t->nt3 = p;     p += nhalf;
00447              t->nt4 = p;     p += nhalf;
00448              t->j3 = p;      p += ndouble;
00449              t->j4 = p;
00450          }
00451          t->mepf[0] = 0;
00452          t->mdel[0] = 0;
00453          t->mdum = AM.mTraceDum;
00454          t->kstep = -2;
00455          t->step1 = 0;
00456          t->sign1 = 1;
00457          t->lc3 = -1;
00458          t->lc4 = -1;
00459          t->gamm = 1;
00460
00461          do {
00462              t->stap = (t->step1)++;
00463              t->kstep += 2;
00464              t->eers[t->stap] = 0;
00465              t->mepf[t->step1] = t->mepf[t->stap];
00466              t->mdel[t->step1] = t->mdel[t->stap];
00467          CallTrick: while ( !Trick(t->inlist+t->kstep,t) ) {
00468              t->kstep -= 2;
00469              t->step1 = (t->stap)--;
00470              if ( t->stap < 0 ) {
00471                  return(0);
00472              }
00473          }
00474          } while ( t->kstep < (number-4) );
00475  /*
00476      Take now the trace of the leftover matrices.

```

```

00477      If gamma5 causes the term to vanish there will be a
00478      renewed call to Trick for its next term.
00479  */
00480      t->sign2 = t->sign1;
00481      if ( ( t->gamma5 == GAMMA7 ) && ( t->gamm == -1 ) ) {
00482          t->sign2 = - t->sign2;
00483      }
00484      else if ( ( t->gamma5 == GAMMA5 ) && ( t->gamm == 1 ) ) {
00485          goto CallTrick;
00486      }
00487      else if ( ( t->gamma5 == GAMMA1 ) && ( t->gamm == -1 ) ) {
00488          goto CallTrick;
00489      }
00490      p = t->pdel + t->mdel[t->step1];
00491      *p++ = t->inlist[t->kstep+2];
00492      *p++ = t->inlist[t->kstep+3];
00493  /*
00494      Now the trace has been expressed in terms of Levi-Civita tensors
00495      and Kronecker delta's.
00496      The Levi-Civita tensors are in t->pepf
00497      and there are t->mepf[step1] elements in this array.
00498      The Kronecker delta's are in t->pdel
00499      and there are t->mdel[step1] elements in this array.
00500
00501      Next we rake the Levi-Civita tensors together such that there
00502      is an optimal use of the contractions.
00503  */
00504      {
00505          WORD ae;
00506          ae = t->mepf[t->step1];
00507          t->ad = t->mdel[t->step1]+2;
00508          t->a4 = 0;
00509          t->a3 = 0;
00510          while ( ( ae -= 4 ) >= 0 ) {
00511              if ( t->pepf[ae] > AM.mTraceDum && t->pepf[ae] <= t->mdum ) {
00512                  p = t->e3 + t->a3;
00513                  m = t->pepf + ae;
00514                  for ( i = 0; i < 3; i++ ) {
00515                      p[3] = m[3-i];
00516                      *p++ = m[i-4];
00517                  }
00518                  t->a3 += 6;
00519                  ae -= 4;
00520              }
00521              else {
00522                  p = t->e4 + t->a4;
00523                  m = t->pepf + ae;
00524                  for ( i = 0; i < 4; i++ ) *p++ = *m++;
00525                  t->a4 += 4;
00526              }
00527          }
00528      }
00529  /*
00530      Now e3 contains pairs of LeviCivita tensors that have
00531      three indices each and a3 is the total number of indices.
00532      e4 contains individual tensors with 4 indices.
00533      Some indices may be contracted with Kronecker delta's.
00534
00535      Contract the e3 tensors first.
00536  */
00537
00538      while ( t->a3 > 0 ) {
00539          t->nt3[++(t->lc3)] = 0;
00540          while ( ( t->nt3[t->lc3] = EpfGen(3,t->e3+t->a3-6,
00541              t->pdel+t->ad,t->j3+6*t->lc3,oldsign = t->nt3[t->lc3]) ) == 0 ) {
00542              if ( oldsign < 0 ) t->sign2 = - t->sign2;
00543              (t->lc3)--;
00544          NextE3:
00545              if ( t->lc3 < 0 ) goto CallTrick;
00546              t->ad -= 6;
00547              t->a3 += 6;
00548          }
00549          if ( oldsign ) {
00550              if ( oldsign != t->nt3[t->lc3] ) t->sign2 = - t->sign2;
00551          }
00552          else if ( t->nt3[t->lc3] < 0 ) t->sign2 = - t->sign2;
00553          t->a3 -= 6;
00554          t->ad += 6;
00555      }
00556  /*
00557      Contract the e4 tensors.
00558  */
00559      while ( t->a4 > 4 ) {
00560          t->nt4[++(t->lc4)] = 0;
00561          while ( ( t->nt4[t->lc4] = EpfGen(4,t->e4+t->a4-8,
00562              t->pdel+t->ad,t->j4+8*t->lc4,oldsign = t->nt4[t->lc4]) ) == 0 ) {
00563              if ( oldsign < 0 ) t->sign2 = - t->sign2;
              (t->lc4)--;

```

```

00564 NextE4:      if ( t->lc4 < 0 ) goto NextE3;
00565                t->ad -= 8;
00566                t->a4 += 8;
00567            }
00568            if ( oldsign ) {
00569                if ( oldsign != t->nt4[t->lc4] ) t->sign2 = - t->sign2;
00570            }
00571            else if ( t->nt4[t->lc4] < 0 ) t->sign2 = - t->sign2;
00572            t->a4 -= 8;
00573            t->ad += 8;
00574        }
00575    /*
00576        Finally the extra dummy indices can be eliminated.
00577        Note that there are currently t->ad words in t->pdel forming
00578        t->ad / 2 Kronecker delta's. We are however not allowed to
00579        alter anything in these arrays, so the results should be
00580        copied to kron.
00581    */
00582    m = kron;
00583    if ( t->a4 == 4 ) {
00584        p = t->e4;
00585        *m++ = *p++; *m++ = *p++; *m++ = *p++; *m++ = *p++;
00586        retval = 2;
00587    }
00588    else retval = 1;
00589    if ( t->sign2 < 0 ) retval = - retval;
00590    p = t->pdel;
00591    for ( i = 0; i < t->ad; i++ ) *m++ = *p++;
00592    p = kron;
00593    if ( t->a4 == 4 ) {
00594    /*
00595        Test for dummies in the last position of the e_.
00596    */
00597        stop = p + t->ad + 4;
00598        p += 3;
00599        while ( *p >= AM.mTraceDum && *p <= t->mdum ) {
00600            m = p + 1;
00601            do {
00602                if ( *m == *p ) {
00603                    *p = m[1];
00604                    *m = *--stop;
00605                    m[1] = *--stop;
00606                    break;
00607                }
00608                else if ( m[1] == *p ) {
00609                    *p = *m;
00610                    *m = *--stop;
00611                    m[1] = *--stop;
00612                    break;
00613                }
00614                else m += 2;
00615            } while ( m < stop );
00616        }
00617        p++;
00618    }
00619    else stop = p + t->ad;
00620    while ( p < (stop-2) ) {
00621        while ( *p >= AM.mTraceDum && *p <= t->mdum ) {
00622            m = p + 2;
00623            do {
00624                if ( *m == *p ) {
00625                    *p = m[1];
00626                    *m = *--stop;
00627                    m[1] = *--stop;
00628                    break;
00629                }
00630                else if ( m[1] == *p ) {
00631                    *p = *m;
00632                    *m = *--stop;
00633                    m[1] = *--stop;
00634                    break;
00635                }
00636                else m += 2;
00637            } while ( m < stop );
00638        }
00639        p++;
00640        while ( *p >= AM.mTraceDum && *p <= t->mdum ) {
00641            m = p + 1;
00642            do {
00643                if ( *m == *p ) {
00644                    *p = m[1];
00645                    *m = *--stop;
00646                    m[1] = *--stop;
00647                    break;
00648                }
00649                else if ( m[1] == *p ) {
00650                    *p = *m;

```

```

00651             *m = *--stop;
00652             m[1] = *--stop;
00653             break;
00654         }
00655         else m += 2;
00656     } while ( m < stop );
00657 }
00658     p++;
00659 }
00660     return(retval);
00661 }
00662 if ( number <= 2 ) return(0);
00663 else { goto NextE4; }
00664 }
00665
00666 /*
00667     #] Trace4no :
00668     #[ Trace4 :          WORD Trace4(term,params,num,level)
00669
00670     Generates traces of the string of gamma matrices in 'instring'.
00671
00672     The difference with the routine tracen ( for n dimensions )
00673     lies in the treatment of gamma 5 and the specific form
00674     of the Chisholm identities. The identities used here are
00675     g(mu)*g(al)*...*g(an)*g(mu)=
00676     n=odd: -2*g(an)*...*g(al)      ( reversed order )
00677     n=even: 2*g(an)*g(al)*...*g(a(n-1))
00678             +2*g(a(n-1))*...*g(al)*g(an)
00679     There is a special case for n=2 : 4*d(al,a2)*gi
00680
00681     The main difference with the old fortran version lies in
00682     the recursion that is used here. That cleans up the variables
00683     very much.
00684
00685     The contents of the AT.TMout array are:
00686     length,type,gamma5,gamma's
00687
00688     The space for the vectors in t is at most 14 * number.
00689
00690     The condition params[5] == 0 corresponds to finding gamma6*gamma7
00691     during the pick up of the matrices.
00692 */
00693
00694 int Trace4(PHEAD WORD *term, WORD *params, WORD num, WORD level)
00695 {
00696     GETBIDENTITY
00697     TRACES *t;
00698     WORD *p, *m, number, i;
00699     WORD *OldW;
00700     WORD j, minimum, minimum2, *min, *stopper;
00701     int ret;
00702     OldW = AT.WorkPointer;
00703     if ( AN.numtracesctack >= AN.intracestack ) {
00704         number = AN.intracestack + 2;
00705         t = (TRACES *)Malloca1(number*sizeof(TRACES),"TRACES-struct");
00706         if ( AN.tracestack ) {
00707             for ( i = 0; i < AN.intracestack; i++ ) { t[i] = AN.tracestack[i]; }
00708             M_free(AN.tracestack,"TRACES-struct");
00709         }
00710         AN.tracestack = t;
00711         AN.intracestack = number;
00712     }
00713
00714     number = *params - 6;
00715     if ( number < 0 || ( number & 1 ) || !params[5] ) return(0);
00716
00717     t = AN.tracestack + AN.numtracesctack;
00718     AN.numtracesctack++;
00719
00720     t->finalstep = ( params[2] & 16 ) ? 1 : 0;
00721     t->gamma5 = params[3];
00722     if ( t->finalstep && t->gamma5 != GAMMA1 ) {
00723         MLOCK(ErrorMessageLock);
00724         MesPrint("Gamma5 not allowed in this option of the trace command");
00725         MUNLOCK(ErrorMessageLock);
00726         AN.numtracesctack--;
00727         SETERROR(-1)
00728     }
00729     t->inlist = AT.WorkPointer;
00730     t->accup = t->accu = t->inlist + number;
00731     t->perm = t->accu + (number*2);
00732     t->eers = t->perm + number;
00733     if ( ( AT.WorkPointer += 19 * number ) >= AT.WorkTop ) {
00734         MLOCK(ErrorMessageLock);
00735         MesWork();
00736         MUNLOCK(ErrorMessageLock);
00737     }

```



```

00738         return(-1);
00739     }
00740     t->num = num;
00741     t->level = level;
00742     p = t->inlist;
00743     m = params+6;
00744     for ( i = 0; i < number; i++ ) *p++ = *m++;
00745     t->termp = term;
00746     t->factor = params[4];
00747     t->allsign = params[5];
00748     if ( number >= 10 || ( t->gamma5 != GAMMA1 && number > 4 ) ) {
00749 /*
00750         The next code should `normal order' the string
00751         We need the lexicographic smallest string, taking also the
00752         reverse strings into account.
00753 */
00754         minimum = 0; min = t->inlist;
00755         stopper = min + number;
00756         for ( i = 1; i < number; i++ ) {
00757             p = min;
00758             m = t->inlist + i;
00759             for ( j = 0; j < number; j++ ) {
00760                 if ( *p < *m ) break;
00761                 if ( *p > *m ) {
00762                     min = t->inlist+i;
00763                     minimum = i;
00764                     break;
00765                 }
00766                 p++; m++;
00767                 if ( m >= stopper ) m = t->inlist;
00768                 if ( p >= stopper ) p = t->inlist;
00769             }
00770         }
00771         p = min;
00772         min = m = AT.WorkPointer;
00773         i = number;
00774         while ( --i >= 0 ) {
00775             *m++ = *p++;
00776             if ( p >= stopper ) p = t->inlist;
00777         }
00778         p = t->inlist;
00779         m = min;
00780         i = number;
00781         while ( --i >= 0 ) *p++ = *m++;
00782         p = t->inlist;
00783         m = stopper;
00784         while ( p < m ) { /* reverse string */
00785             i = *p; *p++ = *--m; *m = i;
00786         }
00787         minimum2 = 0;
00788         for ( i = 0; i < number; i++ ) {
00789             p = min;
00790             m = t->inlist + i;
00791             for ( j = 0; j < number; j++ ) {
00792                 if ( *p < *m ) break;
00793                 if ( *p > *m ) {
00794                     m = t->inlist + i;
00795                     p = min;
00796                     j = number;
00797                     while ( --j >= 0 ) {
00798                         *p++ = *m++;
00799                         if ( m >= stopper ) m = t->inlist;
00800                     }
00801                     minimum2 = i;
00802                     break;
00803                 }
00804                 p++; m++;
00805                 if ( m >= stopper ) m = t->inlist;
00806             }
00807         }
00808         minimum ^= minimum2;
00809         if ( ( minimum & 1 ) != 0 ) {
00810             if ( t->gamma5 == GAMMA5 ) t->allsign = - t->allsign;
00811             else if ( t->gamma5 != GAMMA1 )
00812                 t->gamma5 = GAMMA6 + GAMMA7 - t->gamma5;
00813         }
00814         p = min; m = t->inlist; i = number;
00815         while ( --i >= 0 ) *m++ = *p++;
00816 /*
00817         Now the trace is in normal order
00818 */
00819     }
00820     ret = Trace4Gen(BHEAD t,number);
00821     AT.WorkPointer = OldW;
00822     AN.numtracesctack--;
00823     return(ret);
00824 }

```

```

00825
00826 /*
00827     #] Trace4 :
00828     #[ Trace4Gen :          WORD Trace4Gen(t,number)
00829
00830     The recursive breakdown of a trace in 4 dimensions.
00831     We test first whether the trace has zero or two gamma's left.
00832     This case can be done quickly.
00833     Next we test whether we can eliminate adjacent objects that are
00834     the same.
00835     Then we test for Chisholm identities (I). First for identities
00836     with an odd number of gamma matrices (II), then for those with an
00837     even number of matrices (III). The special thing here is the demand
00838     that the contraction be between indices with 4 dimensions only.
00839     Then there is a scan for objects that are the same, not regarding
00840     their type (IV). This is exactly the same as in n dimensions.
00841     Finally we have a string left in which all objects are different (V).
00842     This case is treated by the routine Trace4no (no stands for no
00843     objects are the same).
00844
00845     In case I we have one d_ of which the result of the contraction
00846     has not yet been fixed.
00847     Case II gives just a reordering of the matrices and a factor -2.
00848     Case III gives two terms: one for the anti commutation, such that
00849     the number of intermediate matrices becomes odd and the other
00850     from the Chisholm rule for an odd number of matrices. Both have
00851     a factor 2.
00852     Case IV gives m+1 terms when m is the number of matrices inbetween.
00853     We take the shortest path. The sign alternates and all terms have
00854     a factor two, except for the last one.
00855
00856 */
00857
00858 int Trace4Gen(PHEAD TRACES *t, WORD number)
00859 {
00860     GETBIDENTITY
00861     WORD *termout, *stop;
00862     WORD *p, *m, oldval;
00863     WORD *pold, *mold, diff, *oldstring, cp;
00864 /*
00865     #] Special cases :
00866 */
00867     if ( number <= 2 ) { /* Special cases */
00868         if ( t->gamma5 == GAMMA5 ) return(0);
00869         termout = AT.WorkPointer;
00870         p = t->termp;
00871         stop = p + *p;
00872         m = termout;
00873         p++;
00874         if ( p < stop ) do {
00875             if ( *p == SUBEXPRESSION && p[2] == t->num ) {
00876                 oldstring = p;
00877                 p = t->termp;
00878                 do { *m++ = *p++; } while ( p < oldstring );
00879                 p += p[1];
00880                 *m++ = AC.lUniTrace[0];
00881                 *m++ = AC.lUniTrace[1];
00882                 *m++ = AC.lUniTrace[2];
00883                 *m++ = AC.lUniTrace[3];
00884                 if ( number == 2 || t->accup > t->accu ) {
00885                     oldstring = m;
00886                     *m++ = DELTA;
00887                     *m++ = 4;
00888                     if ( number == 2 ) {
00889                         *m++ = t->inlist[0];
00890                         *m++ = t->inlist[1];
00891                     }
00892                     if ( t->accup > t->accu ) {
00893                         pold = p;
00894                         p = t->accu;
00895                         while ( p < t->accup ) *m++ = *p++;
00896                         oldstring[1] = WORDDIF(m,oldstring);
00897                         p = pold;
00898                     }
00899                 }
00900                 if ( t->factor ) {
00901                     *m++ = SNUMBER;
00902                     *m++ = 4;
00903                     *m++ = 2;
00904                     *m++ = t->factor;
00905                 }
00906                 do { *m++ = *p++; } while ( p < stop );
00907                 *termout = WORDDIF(m,termout);
00908                 if ( t->allsign < 0 ) m[-1] = -m[-1];
00909                 if ( ( AT.WorkPointer = m ) > AT.WorkTop ) {
00910                     MLOCK(ErrorMessageLock);
00911                     MesWork();

```

```

00912             MUNLOCK(ErrorMessageLock);
00913             return(-1);
00914         }
00915         *AN.RepPoint = 1;
00916         AR.expchanged = 1;
00917         if ( *termout ) {
00918             *AN.RepPoint = 1;
00919             AR.expchanged = 1;
00920             if ( Generator(BHEAD termout,t->level) ) goto TracCall;
00921         }
00922         AT.WorkPointer= termout;
00923         return(0);
00924     }
00925     p += p[1];
00926 } while ( p < stop );
00927 return(0);
00928 }
00929 /*
00930     #] Special cases :
00931     #[ Adjacent objects :
00932 */
00933 p = t->inlist;
00934 stop = p + number - 1;
00935 if ( *p == *stop ) {          /* First and last of string */
00936     oldval = *p;
00937     *(t->accup)++ = *p;
00938     *(t->accup)++ = *p;
00939     m = p+1;
00940     while ( m < stop ) *p++ = *m++;
00941     if ( t->gamma5 != GAMMA1 ) {
00942         if ( t->gamma5 == GAMMA5 ) t->allsign = - t->allsign;
00943         else if ( t->gamma5 == GAMMA6 ) t->gamma5 = GAMMA7;
00944         else if ( t->gamma5 == GAMMA7 ) t->gamma5 = GAMMA6;
00945     }
00946     if ( Trace4Gen(BHEAD t,number-2) ) goto TracCall;
00947     t = AN.tracestack + AN.numtracesctack - 1;
00948     if ( t->gamma5 != GAMMA1 ) {
00949         if ( t->gamma5 == GAMMA5 ) t->allsign = - t->allsign;
00950         else if ( t->gamma5 == GAMMA6 ) t->gamma5 = GAMMA7;
00951         else if ( t->gamma5 == GAMMA7 ) t->gamma5 = GAMMA6;
00952     }
00953     while ( p > t->inlist ) *--m = *--p;
00954     *p = *stop = oldval;
00955     t->accup -= 2;
00956     return(0);
00957 }
00958 do {
00959     if ( *p == p[1] ) {          /* Adjacent in string */
00960         oldval = *p;
00961         pold = p;
00962         m = p+2;
00963         *(t->accup)++ = *p;
00964         *(t->accup)++ = *p;
00965         while ( m <= stop ) *p++ = *m++;
00966         if ( Trace4Gen(BHEAD t,number-2) ) goto TracCall;
00967         t = AN.tracestack + AN.numtracesctack - 1;
00968         while ( p > pold ) *--m = *--p;
00969         *p++ = oldval;
00970         *p++ = oldval;
00971         t->accup -= 2;
00972         return(0);
00973     }
00974     p++;
00975 } while ( p < stop );
00976 /*
00977     #] Adjacent objects :
00978     #[ Odd Contraction :
00979 */
00980 p = t->inlist;
00981 stop = p + number;
00982 do {
00983     if ( *p >= AM.OffsetIndex && (
00984         ( *p < WILDOFFSET + AM.OffsetIndex &&
00985         indices[*p-AM.OffsetIndex].dimension == 4 )
00986         || ( *p >= WILDOFFSET + AM.OffsetIndex && AC.lDefDim == 4 ) ) ) {
00987         m = p+2;
00988         while ( m < stop ) {
00989             if ( *p == *m ) {
00990                 pold = p;
00991                 mold = m;
00992                 oldval = *p;
00993                 (t->factor)++;
00994                 t->allsign = - t->allsign;
00995                 *p++ = *--m;
00996                 m--;
00997                 while ( m > p ) { diff = *p; *p++ = *m; *m-- = diff; }
00998                 p = mold - 1;

```

```

00999          m = mold + 1;
01000          while ( m < stop ) *p++ = *m++;
01001          if ( Trace4Gen(BHEAD t,number-2) ) goto TracCall;
01002          t = AN.tracestack + AN.numtracesetack - 1;
01003          m--;
01004          while ( m > mold ) *m-- = *--p;
01005          p = pold;
01006          *m-- = oldval;
01007          *m-- = *p;
01008          *p++ = oldval;
01009          while ( m > p ) { diff = *p; *p++ = *m; *m-- = diff; }
01010          t->allsign = - t->allsign;
01011          (t->factor)--;
01012          return(0);
01013      }
01014      m += 2;
01015  }
01016  }
01017  p++;
01018  } while ( p < stop );
01019  /*
01020      #] Odd Contraction :
01021      #[ Even Contraction :
01022      First the case with two matrices inbetween.
01023  */
01024  p = t->inlist;
01025  stop = p + number;
01026  do {
01027      if ( *p >= AM.OffsetIndex && (
01028          ( *p < WILDOFFSET + AM.OffsetIndex &&
01029              indices[*p-AM.OffsetIndex].dimension == 4 )
01030          || ( *p >= WILDOFFSET + AM.OffsetIndex && AC.lDefDim == 4 ) ) ) {
01031          m = p+3;
01032          if ( m >= stop ) m -= number;
01033          if ( *p == *m ) {
01034              WORD oldfactor, old5;
01035              oldstring = AT.WorkPointer;
01036              AT.WorkPointer += number;
01037              oldfactor = t->allsign;
01038              old5 = t->gamma5;
01039              if ( m < p ) cp = (WORDDIF(m,t->inlist) + 1) & 1;
01040              else cp = 0;
01041              if ( cp && ( t->gamma5 != GAMMA1 ) ) {
01042                  if ( t->gamma5 == GAMMA5 ) t->allsign = -t->allsign;
01043                  else if ( t->gamma5 == GAMMA6 ) t->gamma5 = GAMMA7;
01044                  else if ( t->gamma5 == GAMMA7 ) t->gamma5 = GAMMA6;
01045              }
01046              mold = m;
01047              p = oldstring;
01048              m = t->inlist;
01049              while ( m < stop ) *p++ = *m++;
01050  /*
01051      Rotate the string
01052  */
01053          m = mold + 1;
01054          p = t->inlist;
01055          while ( m < stop ) *p++ = *m++;
01056          m = oldstring;
01057          if ( !cp && ((WORDDIF(stop,p))&1) != 0 && ( t->gamma5 != GAMMA1 ) ) {
01058              if ( t->gamma5 == GAMMA5 ) t->allsign = -t->allsign;
01059              else if ( t->gamma5 == GAMMA6 ) t->gamma5 = GAMMA7;
01060              else if ( t->gamma5 == GAMMA7 ) t->gamma5 = GAMMA6;
01061          }
01062          while ( p < stop ) *p++ = *m++;
01063          t->factor += 2;
01064          m = p - 3;
01065          p = t->inlist;
01066          oldval = number - 4;
01067          while ( oldval > 0 ) {
01068              if ( *p >= AM.OffsetIndex && (
01069                  ( *p < WILDOFFSET + AM.OffsetIndex &&
01070                      indices[*p-AM.OffsetIndex].dimension )
01071                  || ( *p >= WILDOFFSET + AM.OffsetIndex && AC.lDefDim ) ) ) {
01072                  if ( *p == *m ) {
01073                      *p = m[1];
01074                      break;
01075                  }
01076                  else if ( *p == m[1] ) {
01077                      *p = *m;
01078                      break;
01079                  }
01080              }
01081              p++; oldval--;
01082          }
01083          if ( oldval <= 0 ) {
01084              *(t->accup)++ = *m++;
01085              *(t->accup)++ = *m++;

```

```

01086         }
01087         if ( Trace4Gen(BHEAD t,number-4) ) goto TracCall;
01088         t = AN.tracestack + AN.numtracesctack - 1;
01089         t->factor -= 2;
01090         if ( oldval <= 0 ) t->accup -= 2;
01091         t->gamma5 = old5;
01092         t->allsign = oldfactor;
01093         AT.WorkPointer = p = oldstring;
01094         m = t->inlist;
01095         while ( m < stop ) *m++ = *p++;
01096         return(0);
01097     }
01098 }
01099 p++;
01100 } while ( p < stop );
01101 /*
01102     The case with at least 4 matrices inbetween.
01103 */
01104 p = t->inlist;
01105 stop = p + number;
01106 do {
01107     if ( *p >= AM.OffsetIndex && (
01108         ( *p < WILDOFFSET + AM.OffsetIndex &&
01109         indices[*p-AM.OffsetIndex].dimension == 4 )
01110         || ( *p >= WILDOFFSET + AM.OffsetIndex && AC.lDefDim == 4 ) ) ) {
01111         m = p+5;
01112         while ( m < stop ) {
01113             if ( *p == *m ) {
01114                 WORD *pex, *mex;
01115                 pold = p;
01116                 mold = m;
01117                 oldval = *p;
01118             /*
01119                 g_(1,mu)*g_(1,a1)*...*g_(1,aj)*g_(1,an)*g_(1,mu) ->
01120                 first:
01121                 2*g_(1,an)*g_(1,a1)*...*g_(1,aj)
01122             */
01123                 (t->factor)++;
01124             /*
01125                 The variable hulp seems unnecessary
01126                 *p = hulp = m[-1];
01127             */
01128                 *p = m[-1];
01129                 p = m - 1;
01130                 m++;
01131                 while ( m < stop ) *p++ = *m++;
01132                 if ( Trace4Gen(BHEAD t,number-2) ) goto TracCall;
01133                 t = AN.tracestack + AN.numtracesctack - 1;
01134                 pex = p; mex = m;
01135                 p = pold;
01136                 m = mold - 2;
01137                 while ( m > p ) { diff = *p; *p++ = *m; *m-- = diff; }
01138             /*
01139                 and then:
01140                 2*g_(1,aj)*...*g_(1,a1)*g_(1,an)
01141             */
01142                 if ( Trace4Gen(BHEAD t,number-2) ) goto TracCall;
01143                 t = AN.tracestack + AN.numtracesctack - 1;
01144                 p = pold;
01145                 m = mold - 2;
01146                 while ( m > p ) { diff = *p; *p++ = *m; *m-- = diff; }
01147                 m = mex;
01148                 p = pex;
01149                 m--;
01150                 while ( m > mold ) *m-- = *--p;
01151                 m = mold;
01152                 *m-- = oldval;
01153                 p = pold;
01154                 *m = *p;
01155                 *p = oldval;
01156                 (t->factor)--;
01157                 return(0);
01158             }
01159             m += 2;
01160         }
01161     }
01162     p++;
01163 } while ( p < stop );
01164 /*
01165     #] Even Contraction :
01166     #[ Same Objects :
01167 */
01168 p = t->inlist;
01169 stop = p + number - 1;
01170 diff = 2;
01171 do {
01172     p = t->inlist;

```

```

01173     while ( p <= stop ) {
01174         m = p + diff;
01175         if ( m > stop ) m -= number;
01176         if ( *p == *m ) {
01177             WORD oldfactor, c, old5;
01178             oldfactor = t->allsign;
01179             old5 = t->gamma5;
01180             cp = (WORDDIFF(m,t->inlist)) & 1;
01181             if ( !cp && ( t->gamma5 != GAMMA1 ) ) {
01182                 if ( t->gamma5 == GAMMA5 ) t->allsign = -t->allsign;
01183                 else if ( t->gamma5 == GAMMA6 ) t->gamma5 = GAMMA7;
01184                 else if ( t->gamma5 == GAMMA7 ) t->gamma5 = GAMMA6;
01185             }
01186             oldstring = AT.WorkPointer;
01187             AT.WorkPointer += number;
01188             mold = m;
01189             oldval = *p;
01190             p = oldstring;
01191             m = t->inlist;
01192             while ( m <= stop ) *p++ = *m++;
01193         /*
01194         Rotate the string
01195         */
01196         m = mold + 1;
01197         p = t->inlist;
01198         while ( m <= stop ) *p++ = *m++;
01199         m = oldstring;
01200         while ( p <= stop ) *p++ = *m++;
01201         (t->factor)++;
01202         p -= diff + 1;
01203         m = stop;
01204         *(t->accup) = oldval;
01205         t->accup += 2;
01206         m--;
01207         while ( m > p ) {
01208             c = t->accup[-1];
01209             t->accup[-1] = *m;
01210             *m = c;
01211             if ( Trace4Gen(BHEAD t,number-2) ) goto Trac4Call;
01212             t = AN.tracestack + AN.numtracesctack - 1;
01213             m--;
01214             t->allsign = - t->allsign;
01215         }
01216         c = t->accup[-1];
01217         t->accup[-1] = *m;
01218         *m = c;
01219         (t->factor)--;
01220         if ( Trace4Gen(BHEAD t,number-2) ) goto Trac4Call;
01221         t = AN.tracestack + AN.numtracesctack - 1;
01222         t->accup -= 2;
01223         t->allsign = oldfactor;
01224         AT.WorkPointer = p = oldstring;
01225         m = t->inlist;
01226         while ( m <= stop ) *m++ = *p++;
01227         t->gamma5 = old5;
01228         return(0);
01229     }
01230     p++;
01231 }
01232 } while ( ++diff <= (number>>1) );
01233 /*
01234     #] Same Objects :
01235     #[ All Different :
01236
01237     Here we have a string with all different objects.
01238 */
01239 t->sgn = 0;
01240 termout = AT.WorkPointer;
01241 for(;;) {
01242     if ( t->finalstep == 0 ) diff = Trace4no(number,t->accup,t);
01243     else diff = TraceNno(number,t->accup,t);
01244     /* while ( ( diff = Trace4no(number,t->accup,t) ) != 0 ) */
01245     if ( diff == 0 ) break;
01246     p = t->term;
01247     stop = p + *p;
01248     m = termout;
01249     p++;
01250     if ( p < stop ) do {
01251         if ( *p == SUBEXPRESSION && p[2] == t->num ) {
01252             oldstring = p;
01253             p = t->term;
01254             do { *m++ = *p++; } while ( p < oldstring );
01255             p += p[1];
01256             pold = p;
01257             *m++ = AC.lUniTrace[0];
01258             *m++ = AC.lUniTrace[1];

```

```

01260         *m++ = AC.lUniTrace[2];
01261         *m++ = AC.lUniTrace[3];
01262         *m++ = SNUMBER;
01263         *m++ = 4;
01264         *m++ = 2;
01265         *m++ = t->factor;
01266         p = t->accup;
01267         oldval = number;
01268         if ( diff == 2 || diff == -2 ) {
01269             *m++ = LEVICIVITA;
01270             *m++ = 4+FUNHEAD;
01271             *m++ = DIRTYFLAG;
01272             FILLFUN3(m)
01273             *m++ = *p++; *m++ = *p++; *m++ = *p++; *m++ = *p++;
01274             oldval -= 4;
01275         }
01276         if ( oldval > 0 || t->accup > t->accu ) {
01277             oldstring = m;
01278             *m++ = DELTA;
01279             *m++ = oldval + 2;
01280             if ( oldval > 0 ) NCOPY(m,p,oldval);
01281             if ( t->accup > t->accu ) {
01282                 p = t->accu;
01283                 while ( p < t->accup ) *m++ = *p++;
01284                 oldstring[1] = WORDDIF(m,oldstring);
01285             }
01286         }
01287         p = pold;
01288         do { *m++ = *p++; } while ( p < stop );
01289         *termout = WORDDIF(m,termout);
01290         if ( ( diff ^ t->allsign ) < 0 ) m[-1] = - m[-1];
01291         if ( ( AT.WorkPointer = m ) > AT.WorkTop ) {
01292             MLOCK(ErrorMessageLock);
01293             MesWork();
01294             MUNLOCK(ErrorMessageLock);
01295             return(-1);
01296         }
01297         if ( *termout ) {
01298             *AN.RepPoint = 1;
01299             AR.expchanged = 1;
01300             if ( Generator(BHEAD termout,t->level) ) {
01301                 AT.WorkPointer = termout;
01302                 goto TracCall;
01303             }
01304             t = AN.tracestack + AN.numtracesctack - 1;
01305         }
01306         break;
01307     }
01308     p += p[1];
01309 } while ( p < stop );
01310 }
01311 AT.WorkPointer = termout;
01312 return(0);
01313
01314 /*
01315     #] All Different :
01316 */
01317 Trac4Call:
01318     AT.WorkPointer = oldstring;
01319 TracCall:
01320     if ( AM.tracebackflag ) {
01321         MLOCK(ErrorMessageLock);
01322         MesCall("Trace4Gen");
01323         MUNLOCK(ErrorMessageLock);
01324     }
01325     return(-1);
01326 }
01327
01328 /*
01329     #] Trace4Gen :
01330     #[ TraceNno :           WORD TraceNno(number,kron,t)
01331
01332     Routine takes the trace in N dimensions of a string
01333     of gamma matrices. It is assumed that there are no
01334     contractions and no vectors that are the same. For
01335     the treatment of those cases there are special routines,
01336     that call this routine as a final stage.
01337     The calling routine must reserve 'number' WORDs for perm
01338     and kron each.
01339     kron and perm may not be altered during the generation!
01340
01341 */
01342
01343 WORD TraceNno(WORD number, WORD *kron, TRACES *t)
01344 {
01345     WORD i, j, a, *p;
01346     if ( !t->sgn ) {

```

```

01347         if ( !number || ( number & 1 ) ) return(0);
01348         p = t->inlist;
01349         for ( i = 0; i < number; i++ ) {
01350             t->perm[i] = i;
01351             *kron++ = *p++;
01352         }
01353         t->sgn = 1;
01354         return(1);
01355     }
01356     else {
01357         i = number - 3;
01358         while ( i > 0 ) {
01359             a = kron[i];
01360             p = t->perm + i;
01361             for ( j = i + 1; j <= *p; j++ ) kron[j-1] = kron[j];
01362             kron[*p++] = a;
01363             if ( *p < number ) {
01364                 a = kron[*p];
01365                 j = *p;
01366                 while ( j >= (i+1) ) { kron[j] = kron[j-1]; j--; }
01367                 kron[i] = a;
01368                 number -= 2;
01369                 for ( j = i+2; j < number; j += 2 ) t->perm[j] = j;
01370                 t->sgn = - t->sgn;
01371                 return(t->sgn);
01372             }
01373             i -= 2;
01374         }
01375     }
01376     return(0);
01377 }
01378
01379 /*
01380     #] TraceNno :
01381     #[ TraceN :          WORD TraceN(term,params,num,level)
01382 */
01383
01384 int TraceN(PHEAD WORD *term, WORD *params, WORD num, WORD level)
01385 {
01386     GETBIDENTITY
01387     TRACES *t;
01388     WORD *p, *m, number, i;
01389     WORD *OldW;
01390     int ret;
01391     if ( params[3] != GAMMA1 ) {
01392         MLOCK(ErrorMessageLock);
01393         MesPrint("Gamma5 not allowed in n-trace");
01394         MUNLOCK(ErrorMessageLock);
01395         SETERROR(-1)
01396     }
01397     OldW = AT.WorkPointer;
01398     if ( AN.numtracesctack >= AN.intracesctack ) {
01399         number = AN.intracesctack + 2;
01400         t = (TRACES *)Malloc1(number*sizeof(TRACES),"TRACES-struct");
01401         if ( AN.tracesctack ) {
01402             for ( i = 0; i < AN.intracesctack; i++ ) { t[i] = AN.tracesctack[i]; }
01403             M_free(AN.tracesctack,"TRACES-struct");
01404         }
01405         AN.tracesctack = t;
01406         AN.intracesctack = number;
01407     }
01408     number = *params - 6;
01409     if ( number < 0 || ( number & 1 ) || !params[5] ) return(0);
01410
01411     t = AN.tracesctack + AN.numtracesctack;
01412     AN.numtracesctack++;
01413
01414     t->inlist = AT.WorkPointer;
01415     t->accup = t->accu = t->inlist + number;
01416     t->perm = t->accu + number;
01417     if ( ( AT.WorkPointer += 3 * number ) >= AT.WorkTop ) {
01418         AN.numtracesctack--;
01419         MLOCK(ErrorMessageLock);
01420         MesWork();
01421         MUNLOCK(ErrorMessageLock);
01422         return(-1);
01423     }
01424     t->num = num;
01425     t->level = level;
01426     p = t->inlist;
01427     m = params+6;
01428     for ( i = 0; i < number; i++ ) *p++ = *m++;
01429     t->termp = term;
01430     t->factor = params[4];
01431     t->allsign = params[5];
01432     ret = TraceNgen(BHEAD t,number);
01433     AT.WorkPointer = OldW;

```



```

01434     AN.numtracesctack--;
01435     return(ret);
01436 }
01437
01438 /*
01439     #] TraceN :
01440     #[ TraceNgen :          WORD TraceNgen(t,number)
01441
01442     This routine is a simplified version of Trace4Gen. We know here
01443     only three cases: Adjacent objects, same objects and all different.
01444     The other difference lies of course in the struct which is now
01445     not of type TRACES, but of type TRACES.
01446
01447 */
01448
01449 int TraceNgen(PHEAD TRACES *t, WORD number)
01450 {
01451     GETBIDENTITY
01452     WORD *termout, *stop;
01453     WORD *p, *m, oldval;
01454     WORD *pold, *mold, diff, *oldstring;
01455 /*
01456     #[ Special cases :
01457 */
01458     if ( number <= 2 ) { /* Special cases */
01459         termout = AT.WorkPointer;
01460         p = t->term;
01461         stop = p + *p;
01462         m = termout;
01463         p++;
01464         if ( p < stop ) do {
01465             if ( *p == SUBEXPRESSION && p[2] == t->num ) {
01466                 oldstring = p;
01467                 p = t->term;
01468                 do { *m++ = *p++; } while ( p < oldstring );
01469                 p += p[1];
01470                 *m++ = AC.lUniTrace[0];
01471                 *m++ = AC.lUniTrace[1];
01472                 *m++ = AC.lUniTrace[2];
01473                 *m++ = AC.lUniTrace[3];
01474                 if ( number == 2 || t->accup > t->accu ) {
01475                     oldstring = m;
01476                     *m++ = DELTA;
01477                     *m++ = 4;
01478                     if ( number == 2 ) {
01479                         *m++ = t->inlist[0];
01480                         *m++ = t->inlist[1];
01481                     }
01482                     if ( t->accup > t->accu ) {
01483                         pold = p;
01484                         p = t->accu;
01485                         while ( p < t->accup ) *m++ = *p++;
01486                         oldstring[1] = WORDDIFF(m,oldstring);
01487                         p = pold;
01488                     }
01489                 }
01490                 if ( t->factor ) {
01491                     *m++ = SNUMBER;
01492                     *m++ = 4;
01493                     *m++ = 2;
01494                     *m++ = t->factor;
01495                 }
01496                 do { *m++ = *p++; } while ( p < stop );
01497                 *termout = WORDDIFF(m,termout);
01498                 if ( t->allsign < 0 ) m[-1] = -m[-1];
01499                 if ( ( AT.WorkPointer = m ) > AT.WorkTop ) {
01500                     MLOCK(ErrorMessageLock);
01501                     MesWork();
01502                     MUNLOCK(ErrorMessageLock);
01503                     return(-1);
01504                 }
01505                 if ( *termout ) {
01506                     *AN.RepPoint = 1;
01507                     AR.expchanged = 1;
01508                     if ( Generator(BHEAD termout,t->level) ) goto TracCall;
01509                 }
01510                 AT.WorkPointer= termout;
01511                 return(0);
01512             }
01513             p += p[1];
01514         } while ( p < stop );
01515         return(0);
01516     }
01517 /*
01518     #[ Special cases :
01519     #[ Adjacent objects :
01520 */

```

```

01521     p = t->inlist;
01522     stop = p + number - 1;
01523     if ( *p == *stop ) {          /* First and last of string */
01524         oldval = *p;
01525         *(t->accup)++ = *p;
01526         *(t->accup)++ = *p;
01527         m = p+1;
01528         while ( m < stop ) *p++ = *m++;
01529         if ( TraceNgen(BHEAD t,number-2) ) goto TracCall;
01530         t = AN.tracestack + AN.numtracesctack - 1;
01531         while ( p > t->inlist ) *--m = *--p;
01532         *p = *stop = oldval;
01533         t->accup -= 2;
01534         return(0);
01535     }
01536     do {
01537         if ( *p == p[1] ) {        /* Adjacent in string */
01538             oldval = *p;
01539             pold = p;
01540             m = p+2;
01541             *(t->accup)++ = *p;
01542             *(t->accup)++ = *p;
01543             while ( m <= stop ) *p++ = *m++;
01544             if ( TraceNgen(BHEAD t,number-2) ) goto TracCall;
01545             t = AN.tracestack + AN.numtracesctack - 1;
01546             while ( p > pold ) *--m = *--p;
01547             *p++ = oldval;
01548             *p++ = oldval;
01549             t->accup -= 2;
01550             return(0);
01551         }
01552         p++;
01553     } while ( p < stop );
01554 /*
01555     #] Adjacent objects :
01556     #[ Same Objects :
01557 */
01558     p = t->inlist;
01559     stop = p + number - 1;
01560     diff = 2;
01561     do {
01562         p = t->inlist;
01563         while ( p <= stop ) {
01564             m = p + diff;
01565             if ( m > stop ) m -= number;
01566             if ( *p == *m ) {
01567                 WORD oldfactor, c;
01568                 oldstring = AT.WorkPointer;
01569                 AT.WorkPointer += number;
01570                 mold = m;
01571                 oldval = *p;
01572                 p = oldstring;
01573                 m = t->inlist;
01574                 while ( m <= stop ) *p++ = *m++;
01575 /*
01576                 Rotate the string
01577 */
01578                 {
01579                     m = mold + 1;
01580                     p = t->inlist;
01581                     while ( m <= stop ) *p++ = *m++;
01582                     m = oldstring;
01583                     while ( p <= stop ) *p++ = *m++;
01584                 }
01585                 oldfactor = t->allsign;
01586                 (t->factor)++;
01587                 p -= diff + 1;
01588                 m = stop;
01589                 if ( oldval >= ( AM.OffsetIndex + WILDOFFSET ) ||
01590                     ( oldval >= AM.OffsetIndex
01591                       && indices[oldval-AM.OffsetIndex].dimension ) ) {
01592                     m--;
01593 /*
01594                     We distinguish 4 cases:
01595                     m-p=1   Use g_(1,mu,a,mu) = (2-d_(mu,mu))*g_(1,a)
01596                     m-p=2   Use g_(1,mu,a,b,mu) = 4*d_(a,b)+(d_(mu,mu)-4)*g_(1,a,b)
01597                     m-p=3   Use g_(1,mu,a,b,c,mu) = -2*g_(1,c,b,a)
01598                             -(d_(mu,mu)-4)*g_(1,a,b,c)
01599                     m-p>3   Reduce down to m-p=3 with the old technique
01600 */
01601                     while ( m > (p+3) ) {
01602                         c = *p;
01603                         *p = *m;
01604                         *m = c;
01605                         if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01606                         t = AN.tracestack + AN.numtracesctack - 1;
01607                         m--;

```

```

01608         t->allsign = - t->allsign;
01609     }
01610     switch ( WORDDIF(m,p) ) {
01611     case 1:
01612         c = *p;
01613         *p = *m;
01614         *m = c;
01615         if ( oldval < ( AM.OffsetIndex + WILDOFFSET )
01616             && indices[oldval-AM.OffsetIndex].nmin4
01617             != -NMIN4SHIFT ) {
01618             t->allsign = - t->allsign;
01619             if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01620             t = AN.tracestack + AN.numtracesctack - 1;
01621             (t->factor)--;
01622             * (t->accup)++ = SUMMEDIND;
01623             * (t->accup)++ =
01624                 indices[oldval-AM.OffsetIndex].nmin4;
01625         }
01626     else
01627     {
01628         if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01629         t = AN.tracestack + AN.numtracesctack - 1;
01630         t->allsign = - t->allsign;
01631         (t->factor)--;
01632         * (t->accup)++ = oldval;
01633         * (t->accup)++ = oldval;
01634     }
01635     if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01636     t = AN.tracestack + AN.numtracesctack - 1;
01637     t->accup -= 2;
01638     break;
01639 case 2:
01640     { WORD one, two;
01641     one = *p = p[1];
01642     two = p[1] = *m;
01643     (t->factor)++;
01644     * (t->accup)++ = *p;      /* 4 */
01645     * (t->accup)++ = *m;      /* d_(a,b) */
01646     if ( TraceNgen(BHEAD t,number-4) ) goto TracnCall;
01647     t = AN.tracestack + AN.numtracesctack - 1;
01648     *p = one; p[1] = two;
01649     t->accup -= 2;
01650     if ( oldval < ( AM.OffsetIndex + WILDOFFSET )
01651         && indices[oldval-AM.OffsetIndex].nmin4
01652         != -NMIN4SHIFT ) {
01653         t->factor -= 2;
01654         * (t->accup)++ = SUMMEDIND;
01655         * (t->accup)++ =
01656             indices[oldval-AM.OffsetIndex].nmin4;
01657     }
01658     else {
01659         t->allsign = - t->allsign;
01660         if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01661         t = AN.tracestack + AN.numtracesctack - 1;
01662         t->allsign = - t->allsign;
01663         t->factor -= 2;
01664         * (t->accup)++ = oldval;
01665         * (t->accup)++ = oldval;
01666     }
01667     if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01668     t = AN.tracestack + AN.numtracesctack - 1;
01669     t->accup -= 2;
01670     }
01671     break;
01672 default:
01673     c = *p;
01674     *p = *m;
01675     *m = c;
01676     c = m[-1]; m[-1] = m[-2]; m[-2] = c;
01677     t->allsign = - t->allsign;
01678     if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01679     t = AN.tracestack + AN.numtracesctack - 1;
01680     m--;
01681     c = *p;
01682     *p = *m;
01683     *m = c;
01684     (t->factor)--;
01685     if ( oldval < ( AM.OffsetIndex + WILDOFFSET )
01686         && indices[oldval-AM.OffsetIndex].nmin4
01687         != -NMIN4SHIFT ) {
01688         * (t->accup)++ = SUMMEDIND;
01689         * (t->accup)++ =
01690             indices[oldval-AM.OffsetIndex].nmin4;
01691         if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01692         t = AN.tracestack + AN.numtracesctack - 1;
01693         t->accup -= 2;
01694         t->allsign = - t->allsign;

```

```

01695         }
01696         else
01697         {
01698             *(t->accup)++ = oldval;
01699             *(t->accup)++ = oldval;
01700             if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01701             t = AN.tracestack + AN.numtracesctack - 1;
01702             t->accup -= 2;
01703             t->allsign = - t->allsign;
01704             t->factor += 2;
01705             if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01706             t = AN.tracestack + AN.numtracesctack - 1;
01707             t->factor -= 2;
01708         }
01709         break;
01710     }
01711 }
01712 else {
01713     *(t->accup) = oldval;
01714     t->accup += 2;
01715     m--;
01716     while ( m > p ) {
01717         c = t->accup[-1];
01718         t->accup[-1] = *m;
01719         *m = c;
01720         if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01721         t = AN.tracestack + AN.numtracesctack - 1;
01722         m--;
01723         t->allsign = - t->allsign;
01724     }
01725     c = t->accup[-1];
01726     t->accup[-1] = *m;
01727     *m = c;
01728     (t->factor)--;
01729     if ( TraceNgen(BHEAD t,number-2) ) goto TracnCall;
01730     t = AN.tracestack + AN.numtracesctack - 1;
01731     t->accup -= 2;
01732 }
01733 t->allsign = oldfactor;
01734 p = oldstring;
01735 m = t->inlist;
01736 while ( m <= stop ) *m++ = *p++;
01737 AT.WorkPointer = oldstring;
01738 return(0);
01739 }
01740 p++;
01741 }
01742 diff++;
01743 } while ( diff <= (number>>1) );
01744 /*
01745     #] Same Objects :
01746     #[ All Different :
01747
01748     Here we have a string with all different objects.
01749 */
01750 /*
01751     t->sgn = 0;
01752     termout = AT.WorkPointer;
01753     while ( ( diff = TraceNno(number,t->accup,t) ) != 0 ) {
01754         p = t->termp;
01755         stop = p + *p;
01756         m = termout;
01757         p++;
01758         if ( p < stop ) do {
01759             if ( *p == SUBEXPRESSION && p[2] == t->num ) {
01760                 oldstring = p;
01761                 p = t->termp;
01762                 do { *m++ = *p++; } while ( p < oldstring );
01763                 p += p[1];
01764                 pold = p;
01765                 *m++ = AC.lUniTrace[0];
01766                 *m++ = AC.lUniTrace[1];
01767                 *m++ = AC.lUniTrace[2];
01768                 *m++ = AC.lUniTrace[3];
01769                 *m++ = SNUMBER;
01770                 *m++ = 4;
01771                 *m++ = 2;
01772                 *m++ = t->factor;
01773                 p = t->accup;
01774                 oldval = number;
01775                 oldstring = m;
01776                 *m++ = DELTA;
01777                 *m++ = oldval + 2;
01778                 NCOPY(m,p,oldval);
01779                 if ( t->accup > t->accu ) {
01780                     p = t->accu;
01781                     while ( p < t->accup ) *m++ = *p++;

```

```

01782         oldstring[1] = WORDDIFF(m,oldstring);
01783     }
01784     p = pold;
01785     do { *m++ = *p++; } while ( p < stop );
01786     *termout = WORDDIFF(m,termout);
01787     if ( ( diff ^ t->allsign ) < 0 ) m[-1] = - m[-1];
01788     if ( ( AT.WorkPointer = m ) > AT.WorkTop ) {
01789         MLOCK(ErrorMessageLock);
01790         MesWork();
01791         MUNLOCK(ErrorMessageLock);
01792         return(-1);
01793     }
01794     if ( *termout ) {
01795         *AN.RepPoint = 1;
01796         AR.expchanged = 1;
01797         if ( Generator(BHEAD termout,t->level) ) {
01798             AT.WorkPointer = termout;
01799             goto TracCall;
01800         }
01801         t = AN.tracestack + AN.numtracesctack - 1;
01802     }
01803     break;
01804 }
01805 p += p[1];
01806 } while ( p < stop );
01807 }
01808 AT.WorkPointer = termout;
01809 return(0);
01810
01811 /*
01812     #] All Different :
01813 */
01814 TracnCall:
01815     AT.WorkPointer = oldstring;
01816 TracCall:
01817     if ( AM.tracebackflag ) {
01818         MLOCK(ErrorMessageLock);
01819         MesCall("TraceNgen");
01820         MUNLOCK(ErrorMessageLock);
01821     }
01822     return(-1);
01823 }
01824
01825 /*
01826     #] TraceNgen :
01827     #[ Traces :           WORD Traces(term,params,num,level)
01828
01829     The contents of the AT.TMout array are:
01830     length,type,subtype,gamma5,factor,sign,gamma's
01831 */
01832
01833
01834 int Traces(PHEAD WORD *term, WORD *params, WORD num, WORD level)
01835 {
01836     GETBIDENTITY
01837     switch ( AT.TMout[2] ) { /* Subtype gives dimension */
01838     case 0:
01839         return(TraceN(BHEAD term,params,num,level));
01840     case 4:
01841         return(Trace4(BHEAD term,params,num,level));
01842     case 12:
01843         return(Trace4(BHEAD term,params,num,level));
01844     case 20:
01845         return(Trace4(BHEAD term,params,num,level));
01846     default:
01847         return(0);
01848     }
01849 }
01850
01851 /*
01852     #] Traces :
01853     #[ TraceFind :       WORD TraceFind(term,params)
01854 */
01855
01856 int TraceFind(PHEAD WORD *term, WORD *params)
01857 {
01858     GETBIDENTITY
01859     WORD *p, *m, *to;
01860     WORD *termout, *stop, *stop2, number = 0;
01861     WORD first = 1;
01862     WORD type, spinline, sp;
01863     type = params[3];
01864     spinline = params[4];
01865     if ( spinline < 0 ) { /* $ variable. Evaluate */
01866         sp = DolToIndex(BHEAD -spinline);
01867         if ( AN.ErrorInDollar || sp < 0 ) {
01868             MLOCK(ErrorMessageLock);

```

```

01869         MesPrint("$%s does not have an index value in trace statement in module %1",
01870                 DOLLARNAME(Dollars,-spline),AC.CModule);
01871         MUNLOCK(ErrorMessageLock);
01872         return(0);
01873     }
01874     spline = sp;
01875 }
01876 to = AT.TMout;
01877 to++;
01878 *to++ = TAKETRACE;
01879 *to++ = type;
01880 *to++ = GAMMA1;
01881 *to++ = 0;           /* Powers of two */
01882 *to++ = 1;          /* sign */
01883 p = term;
01884 m = p + *p - 1;
01885 stop = m - ABS(*m);
01886 termout = m = AT.WorkPointer;
01887 m++;
01888 p++;
01889 while ( p < stop ) {
01890     stop2 = p + p[1];
01891     if ( *p == GAMMA && p[FUNHEAD] == spline ) {
01892         if ( first ) {
01893             *m++ = SUBEXPRESSION;
01894             *m++ = SUBEXPSIZE;
01895             *m++ = -1;
01896             *m++ = 1;
01897             *m++ = DUMMYBUFFER;
01898             FILLSUB(m)
01899             first = 0;
01900         }
01901         p += FUNHEAD+1;
01902         while ( p < stop2 ) {
01903             if ( *p == GAMMA5 ) {
01904                 if ( AT.TMout[3] == GAMMA5 ) AT.TMout[3] = GAMMA1;
01905                 else if ( AT.TMout[3] == GAMMA1 ) AT.TMout[3] = GAMMA5;
01906                 else if ( AT.TMout[3] == GAMMA7 ) AT.TMout[5] = -AT.TMout[5];
01907                 if ( number & 1 ) AT.TMout[5] = - AT.TMout[5];
01908                 p++;
01909             }
01910             else if ( *p == GAMMA6 ) {
01911                 if ( number & 1 ) goto F7;
01912                 if ( AT.TMout[3] == GAMMA6 ) (AT.TMout[4])++;
01913                 else if ( AT.TMout[3] == GAMMA1 ) AT.TMout[3] = GAMMA6;
01914                 else if ( AT.TMout[3] == GAMMA5 ) AT.TMout[3] = GAMMA6;
01915                 else if ( AT.TMout[3] == GAMMA7 ) AT.TMout[5] = 0;
01916                 p++;
01917             }
01918             else if ( *p == GAMMA7 ) {
01919                 if ( number & 1 ) goto F6;
01920                 if ( AT.TMout[3] == GAMMA7 ) (AT.TMout[4])++;
01921                 else if ( AT.TMout[3] == GAMMA1 ) AT.TMout[3] = GAMMA7;
01922                 else if ( AT.TMout[3] == GAMMA5 ) {
01923                     AT.TMout[3] = GAMMA7;
01924                     AT.TMout[5] = -AT.TMout[5];
01925                 }
01926                 else if ( AT.TMout[3] == GAMMA6 ) AT.TMout[5] = 0;
01927                 p++;
01928             }
01929             else {
01930                 *to++ = *p++;
01931                 number++;
01932             }
01933         }
01934     }
01935     else {
01936         while ( p < stop2 ) *m++ = *p++;
01937     }
01938 }
01939 if ( first ) return(0);
01940 AT.TMout[0] = WORDDIF(to,AT.TMout);
01941 to = term;
01942 to += *to;
01943 while ( p < to ) *m++ = *p++;
01944 *termout = WORDDIF(m,termout);
01945 to = term;
01946 p = termout;
01947 do { *to++ = *p++; } while ( p < m );
01948 AT.WorkPointer = term + *term;
01949 return(1);
01950 }
01951
01952 /*
01953     #] TraceFind :
01954     #[ Chisholm :          WORD Chisholm(term,level,num)
01955
```

```

01956      Routines for reorganizing traces.
01957      The command
01958          Chisholm,1;
01959      will collect the gamma matrices in spinline 1 and see whether
01960      they have an index in common with another gamma matrix. If this
01961      is the case the identity
01962           $g_{(2,\mu)} \text{Tr}[g_{(1,\mu)} S(2)] = S(2) + SR(2)$ 
01963      is applied (SR is the reversed string).
01964  */
01965
01966  int Chisholm(PHEAD WORD *term, WORD level)
01967  {
01968      GETBIDENTITY
01969      WORD *t, *r, *m, *s, *tt, *rr;
01970      WORD *mat, *matpoint, *termout, *rdo;
01971      CBUF *C = cbuf+AM.rbufnum;
01972      WORD i, j, num = C->lhs[level][2], gam5;
01973      WORD norm = 0, k, *matp;
01974  /*
01975      #[ Find : Find possible matrices
01976  */
01977      mat = matpoint = AT.WorkPointer;
01978      t = term;
01979      r = t + *t - 1; r -= ABS(*r);
01980      t++;
01981      i = 0;
01982      gam5 = GAMMA1;
01983      while ( t < r ) {
01984          if ( *t == GAMMA && t[FUNHEAD] == num ) {
01985              m = t + t[1];
01986              t += FUNHEAD+1;
01987              while ( t < m ) {
01988                  if ( *t >= 0 || *t < MINSPEC ) i++;
01989                  else {
01990                      if ( gam5 == GAMMA1 ) gam5 = *t;
01991                      else if ( gam5 == GAMMA5 ) {
01992                          if ( *t == GAMMA5 ) gam5 = GAMMA1;
01993                          else if ( *t != GAMMA1 ) gam5 = *t;
01994                      }
01995                  }
01996                  *matpoint++ = *t++;
01997              }
01998          }
01999          else t += t[1];
02000      }
02001      if ( ( i & 1 ) != 0 ) return(0); /* odd trace */
02002  /*
02003      #] Find :
02004      #[ Test : Test for contracted index
02005
02006      This code should be modified.
02007
02008      We have to check for all possible matches if C->lhs[level][3] == 1
02009      and the trace contains a gamma5, gamma6 or gamma7.
02010      Then we normalize by the number of possible contractions (norm) and
02011      do all of them. This way the Levi-Civita tensors have a maximum
02012      chance of cancelling each other. This option is activated with
02013      'contract' and 'symmetrize'. Defaults are that they are on, but
02014      they can be switched off with nocontract and nosymmetrize.
02015  */
02016      s = mat;
02017      while ( s < matpoint ) {
02018  /*
02019          if ( *s < AM.OffsetIndex || ( *s < ( AM.OffsetIndex + WILDOFFSET ) &&
02020              indices[*s-AM.OffsetIndex].dimension == 0 ) ) {
02021  /*
02022              if ( *s < AM.OffsetIndex || ( *s < ( AM.OffsetIndex + WILDOFFSET ) &&
02023                  indices[*s-AM.OffsetIndex].dimension != 4 )
02024              || ( ( AC.lDefDim != 4 ) && ( *s >= ( AM.OffsetIndex + WILDOFFSET ) ) ) ) {
02025                  s++; continue;
02026              }
02027              t = term+1;
02028              while ( t < r ) {
02029                  if ( *t == GAMMA && t[FUNHEAD] != num ) {
02030                      m = t + t[1];
02031                      t += FUNHEAD+1;
02032                      while ( t < m ) {
02033                          if ( *t == *s ) {
02034                              norm++;
02035                          }
02036                          t++;
02037                      }
02038                  }
02039                  else t += t[1];
02040              }
02041              s++;
02042          }

```

```

02043     if ( norm == 0 ) return(Generator(BHEAD term,level)); /* No Action */
02044 /*
02045     #] Test :
02046     #[ Do : Process the string
02047
02048     tt: The subterm
02049     t: The matrix
02050     s: The matrix in the relevant string
02051
02052     Cycle the string in mat so that s is at the end.
02053     Copy the part till the critical GAMMA.
02054     Copy inside the critical string, copy S, copy tail inside string.
02055     Important to remember where S is so that we can reverse it later.
02056     Add term UnitTrace/2/norm.
02057     Copy rest of term.
02058     Continue execution with S.
02059     Reverse S.
02060     Continue execution with SR.
02061 */
02062
02063     if ( C->lhs[level][3] == 0 /* || gam5 == GAMMA1 */ ) norm = 1;
02064
02065     matp = matpoint;
02066     for ( k = 0; k < norm; k++ ) {
02067         matpoint = matp;
02068         s = mat;
02069         while ( s < matpoint ) {
02070 /*
02071             if ( *s < AM.OffsetIndex || ( *s < ( AM.OffsetIndex + WILDOFFSET ) &&
02072             indices[*s-AM.OffsetIndex].dimension == 0 ) ) {
02073 */
02074             if ( *s < AM.OffsetIndex || ( *s < ( AM.OffsetIndex + WILDOFFSET ) &&
02075             indices[*s-AM.OffsetIndex].dimension != 4 ) ) {
02076                 s++; continue;
02077             }
02078             t = term+1;
02079             while ( t < r ) {
02080                 if ( *t == GAMMA && t[FUNHEAD] != num ) {
02081                     tt = t;
02082                     m = t + t[1];
02083                     t += FUNHEAD+1;
02084                     while ( t < m ) {
02085                         if ( *t == *s ) {
02086                             i = WORDDIF(t,tt);
02087                             m = mat;
02088                             while ( m <= s ) *matpoint++ = *m++;
02089                             t = mat;
02090                             while ( m < matpoint ) *t++ = *m++;
02091                             termout = t;
02092                             m = termout + 1;
02093                             t = term + 1;
02094                             while ( t < tt ) {
02095                                 if ( *t != GAMMA || t[FUNHEAD] != num ) {
02096                                     j = t[1];
02097                                     NCOPY(m,t,j);
02098                                 }
02099                                 else t += t[1];
02100                             }
02101                         }
02102                     tt += tt[1];
02103                     rdo = m;
02104                     j = i;
02105                     while ( --j >= 0 ) *m++ = *t++;
02106                     matpoint = m;
02107                     s = mat;
02108                     while ( s < termout ) *m++ = *s++;
02109                     m--;
02110                     t++;
02111                     while ( t < tt ) *m++ = *t++;
02112                     rdo[1] = WORDDIF(m,rdo);
02113
02114                     *m++ = AC.lUniTrace[0];
02115                     *m++ = AC.lUniTrace[1];
02116                     *m++ = AC.lUniTrace[2];
02117                     *m++ = AC.lUniTrace[3];
02118                     *m++ = SNUMBER;
02119                     *m++ = 4;
02120                     *m++ = 2*norm;
02121                     *m++ = -1;
02122
02123                     while ( t < r ) {
02124                         if ( *t != GAMMA || t[FUNHEAD] != num ) {
02125                             j = t[1];
02126                             NCOPY(m,t,j);
02127                         }
02128                         else t += t[1];
02129                     }

```



```

02130             rr = term + *term;
02131             while ( t < rr ) *m++ = *t++;
02132
02133             *termout = WORDDIFF(m,termout);
02134             rr = m;
02135             t = termout;
02136             j = *termout;
02137             NCOPY(m,t,j);
02138             AT.WorkPointer = m;
02139             if ( Generator(BHEAD t,level) ) goto ChisCall;
02140
02141             j = WORDDIFF(termout,mat)-1;
02142             t = matpoint;
02143             m = t + j;
02144             AT.WorkPointer = rr;
02145             while ( m > t ) {
02146                 i = *--m; *m = *t; *t++ = i;
02147             }
02148
02149             if ( Generator(BHEAD termout,level) ) goto ChisCall;
02150             AT.WorkPointer = mat;
02151
02152             goto NextK;
02153         }
02154         t++;
02155     }
02156 }
02157 else t += t[1];
02158 }
02159 s++;
02160 }
02161 NextK;
02162 }
02163 return(0);
02164 /*
02165 #] Do :
02166 */
02167 ChisCall:
02168     if ( AM.tracebackflag ) {
02169         MLOCK(ErrorMessageLock);
02170         MesCall("Chisholm");
02171         MUNLOCK(ErrorMessageLock);
02172     }
02173     return(-1);
02174 }
02175
02176 /*
02177 #] Chisholm :
02178 #[ TenVecFind :      WORD TenVecFind(term,params)
02179 */
02180
02181 int TenVecFind(PHEAD WORD *term, WORD *params)
02182 {
02183     GETBIDENTITY
02184     WORD *t, *w, *m, *tstop;
02185     WORD i, mode, thevector, thetensor, spectator;
02186     thetensor = params[3];
02187     thevector = params[4];
02188     mode = params[5];
02189     if ( thetensor < 0 ) { /* $-expression */
02190         thetensor = DolToTensor(BHEAD -thetensor);
02191         if ( thetensor < FUNCTION ) {
02192             if ( thevector > 0 ) {
02193                 thetensor = DolToTensor(BHEAD thevector);
02194                 if ( thetensor < FUNCTION ) {
02195                     MLOCK(ErrorMessageLock);
02196                     MesPrint("%s should have been a tensor in module %1"
02197                             ,DOLLARNAME(Dollars,params[4]),AC.CModule);
02198                     MUNLOCK(ErrorMessageLock);
02199                     return(-1);
02200                 }
02201                 thevector = DolToVector(BHEAD -params[3]);
02202                 if ( thevector >= 0 ) {
02203                     MLOCK(ErrorMessageLock);
02204                     MesPrint("%s should have been a vector in module %1"
02205                             ,DOLLARNAME(Dollars,-params[3]),AC.CModule);
02206                     MUNLOCK(ErrorMessageLock);
02207                     return(-1);
02208                 }
02209             }
02210         }
02211         else {
02212             MLOCK(ErrorMessageLock);
02213             MesPrint("%s should have been a tensor in module %1"
02214                     ,DOLLARNAME(Dollars,-params[3]),AC.CModule);
02215             MUNLOCK(ErrorMessageLock);
02216             return(-1);
02217         }
02218     }
02219 }

```

```

02217     }
02218 }
02219 if ( thevector > 0 ) { /* $-expression */
02220     thevector = DolToVector(BHEAD thevector);
02221     if ( thevector >= 0 ) {
02222         MLOCK(ErrorMessageLock);
02223         MesPrint("$%s should have been a vector in module %1"
02224             , DOLLARNAME(Dollars,params[4]),AC.CModule);
02225         MUNLOCK(ErrorMessageLock);
02226         return(-1);
02227     }
02228 }
02229 if ( ( mode & 1 ) != 0 ) { /* Vector to tensor */
02230     GETSTOP(term,tstop);
02231     t = term + 1;
02232     while ( t < tstop ) {
02233         if ( *t == DOTPRODUCT ) {
02234             i = t[1] - 2; t += 2;
02235             while ( i > 0 ) {
02236                 spectator = 0;
02237                 if ( t[2] < 0 ) {}
02238                 else if ( *t == thevector && t[1] == thevector ) {
02239                     if ( ( mode & 2 ) == 0 ) spectator = thevector;
02240                 }
02241                 else if ( *t == thevector ) spectator = t[1];
02242                 else if ( t[1] == thevector ) spectator = *t;
02243                 if ( spectator ) {
02244                     if ( ( mode & 8 ) == 0 ) goto match;
02245                     w = SetElements + Sets[params[6]].first;
02246                     m = SetElements + Sets[params[6]].last;
02247                     while ( w < m ) {
02248                         if ( *w == spectator ) break;
02249                         w++;
02250                     }
02251                     if ( w >= m ) goto match;
02252                 }
02253                 t += 3;
02254                 i -= 3;
02255             }
02256         }
02257         else if ( *t == VECTOR ) {
02258             i = t[1] - 2; t += 2;
02259             while ( i > 0 ) {
02260                 if ( *t == thevector ) goto match;
02261                 t += 2;
02262                 i -= 2;
02263             }
02264         }
02265         else if ( *t == thetensor ) t += t[1];
02266         else if ( *t >= FUNCTION ) {
02267             if ( functions[*t-FUNCTION].spec > 0 ) {
02268                 w = t + t[1];
02269                 t += FUNHEAD;
02270                 while ( t < w ) {
02271                     if ( *t == thevector ) goto match;
02272                     t++;
02273                 }
02274             }
02275             else if ( ( mode & 4 ) != 0 ) {
02276                 w = t + t[1];
02277                 t += FUNHEAD;
02278                 while ( t < w ) {
02279                     if ( *t == -VECTOR && t[1] == thevector ) goto match;
02280                     else if ( *t > 0 ) t += *t;
02281                     else if ( *t <= -FUNCTION ) t++;
02282                     else t += 2;
02283                 }
02284             }
02285             else t += t[1];
02286         }
02287         else t += t[1];
02288     }
02289 }
02290 else { /* Tensor to Vector */
02291     GETSTOP(term,tstop);
02292     t = term+1;
02293     while ( t < tstop ) {
02294         if ( *t == thetensor ) goto match;
02295         t += t[1];
02296     }
02297 }
02298 return(0);
02299 match:
02300     AT.TMout[0] = 5;
02301     AT.TMout[1] = TENVEC;
02302     AT.TMout[2] = thetensor;
02303     AT.TMout[3] = thevector;

```

```

02304     AT.TMout[4] = mode;
02305     if ( ( mode & 8 ) != 0 ) { AT.TMout[0] = 6; AT.TMout[5] = params[6]; }
02306     return(1);
02307 }
02308 }
02309
02310 /*
02311     #] TenVecFind :
02312     #[ TenVec :          WORD TenVec(term,params,num,level)
02313 */
02314
02315 int TenVec(PHEAD WORD *term, WORD *params, WORD num, WORD level)
02316 {
02317     GETBIDENTITY
02318     WORD *t, *m, *w, *termout, *tstop, *outlist, *ou, *ww, *mm;
02319     WORD i, j, k, x, mode, thevector, thetensor, DumNow, spectator;
02320     DUMMYUSE(num);
02321     thetensor = params[2];
02322     thevector = params[3];
02323     mode = params[4];
02324     termout = AT.WorkPointer;
02325     DumNow = AR.CurDum = DetCurDum(BHEAD term);
02326     if ( ( mode & 1 ) != 0 ) { /* Vector to tensor */
02327         AT.WorkPointer += *term;
02328         ou = outlist = AT.WorkPointer;
02329         GETSTOP(term,tstop);
02330         t = term + 1;
02331         m = termout + 1;
02332         while ( t < tstop ) {
02333             if ( *t == DOTPRODUCT ) {
02334                 i = t[1] - 2;
02335                 w = m;
02336                 *m++ = *t++; *m++ = *t++;
02337                 while ( i > 0 ) {
02338                     spectator = 0;
02339                     if ( t[2] < 0 ) {
02340                         *m++ = *t++; *m++ = *t++; *m++ = *t++;
02341                     }
02342                     else if ( *t == thevector && t[1] == thevector ) {
02343                         if ( ( mode & 2 ) == 0 ) spectator = thevector;
02344                         else {
02345                             *m++ = *t++; *m++ = *t++; *m++ = *t++;
02346                         }
02347                     }
02348                     else if ( *t == thevector ) spectator = t[1];
02349                     else if ( t[1] == thevector ) spectator = *t;
02350                     else {
02351                         *m++ = *t++; *m++ = *t++; *m++ = *t++;
02352                     }
02353                     if ( spectator ) {
02354                         if ( ( mode & 8 ) == 0 ) goto noveto;
02355                         ww = SetElements + Sets[params[5]].first;
02356                         mm = SetElements + Sets[params[5]].last;
02357                         while ( ww < mm ) {
02358                             if ( *ww == spectator ) break;
02359                             ww++;
02360                         }
02361                         if ( ww < mm ) {
02362                             *m++ = *t++; *m++ = *t++; *m++ = *t++;
02363                         }
02364                         else {
02365                             if ( spectator == thevector ) {
02366                                 for ( j = 0; j < t[2]; j++ ) {
02367                                     *ou++ = ++AR.CurDum;
02368                                     *ou++ = AR.CurDum;
02369                                 }
02370                                 t += 3;
02371                             }
02372                             else {
02373                                 for ( j = 0; j < t[2]; j++ ) *ou++ = spectator;
02374                                 t += 3;
02375                             }
02376                         }
02377                         i -= 3;
02378                     }
02379                     w[1] = WORDDIF(m,w);
02380                     if ( w[1] == 2 ) m = w;
02381                 }
02382             else if ( *t == VECTOR ) {
02383                 i = t[1] - 2; w = m;
02384                 *m++ = *t++; *m++ = *t++;
02385                 while ( i > 0 ) {
02386                     if ( *t == thevector ) {
02387                         *ou++ = t[1];
02388                         t += 2;
02389                     }
02390                     else { *m++ = *t++; *m++ = *t++; }

```

```

02391         i -= 2;
02392     }
02393     w[1] = WORDDIF(m,w);
02394     if ( w[1] == 2 ) m = w;
02395 }
02396 else if ( *t == thetensor ) {
02397     i = t[1] - FUNHEAD;
02398     t += FUNHEAD;
02399     NCOPY(ou,t,i);
02400 }
02401 else if ( *t >= FUNCTION ) {
02402     if ( functions[*t-FUNCTION].spec > 0 ) {
02403         w = t + t[1];
02404         i = FUNHEAD;
02405         NCOPY(m,t,i);
02406         while ( t < w ) {
02407             if ( *t == thevector ) {
02408                 *m++ = ++AR.CurDum;
02409                 *ou++ = AR.CurDum;
02410                 t++;
02411             }
02412             else *m++ = *t++;
02413         }
02414     }
02415     else if ( ( mode & 4 ) != 0 ) {
02416         w = t + t[1];
02417         i = FUNHEAD;
02418         NCOPY(m,t,i);
02419         while ( t < w ) {
02420             if ( *t == -VECTOR && t[1] == thevector ) {
02421                 *m++ = -INDEX;
02422                 *m++ = ++AR.CurDum;
02423                 *ou++ = AR.CurDum;
02424                 t += 2;
02425             }
02426             else if ( *t > 0 ) {
02427                 i = *t;
02428                 NCOPY(m,t,i);
02429             }
02430             else if ( *t <= -FUNCTION ) *m++ = *t++;
02431             else { *m++ = *t++; *m++ = *t++; }
02432         }
02433     }
02434     else goto docopy;
02435 }
02436 else {
02437 docopy:
02438     i = t[1];
02439     NCOPY(m,t,i);
02440 }
02441 }
02442 i = WORDDIF(ou,outlist);
02443 if ( i > 0 ) {
02444     for ( j = 1; j < i; j++ ) {
02445         if ( outlist[j-1] > outlist[j] ) {
02446             x = outlist[j-1]; outlist[j-1] = outlist[j]; outlist[j] = x;
02447             for ( k = j-1; k > 0; k-- ) {
02448                 if ( outlist[k-1] <= outlist[k] ) break;
02449                 x = outlist[k-1]; outlist[k-1] = outlist[k]; outlist[k] = x;
02450             }
02451         }
02452     }
02453 }
02454 *m++ = thetensor;
02455 *m++ = FUNHEAD + i;
02456 *m++ = DIRTYSYMFLAG;
02457 FILLFUN3(m)
02458 ou = outlist;
02459 NCOPY(m,ou,i);
02460 }
02461 w = term + *term;
02462 while ( t < w ) *m++ = *t++;
02463 }
02464 else { /* Tensor to Vector */
02465     GETSTOP(term,tstop);
02466     t = term+1;
02467     m = termout+1;
02468     while ( t < tstop ) {
02469         if ( *t != thetensor ) {
02470             i = t[1];
02471             NCOPY(m,t,i);
02472         }
02473         else {
02474             i = t[1] - FUNHEAD;
02475             t += FUNHEAD;
02476             if ( i > 0 ) {
02477                 w = m; m += 2;

```

```

02478             while ( --i >= 0 ) {
02479                 *m++ = thevector;
02480                 *m++ = *t++;
02481             }
02482             *w = DELTA;
02483             w[1] = WORDDIF(m,w);
02484         }
02485     }
02486 }
02487 w = term + *term;
02488 while ( t < w ) *m++ = *t++;
02489 }
02490 *termout = WORDDIF(m,termout);
02491 AT.WorkPointer = m;
02492 *AT.TMout = 0;
02493 if ( Generator(BHEAD termout,level) ) goto fromTenVec;
02494 AR.CurDum = DumNow;
02495 AT.WorkPointer = termout;
02496 return(0);
02497 fromTenVec:
02498 if ( AM.tracebackflag ) {
02499     MLOCK(ErrorMessageLock);
02500     MesCall("TenVec");
02501     MUNLOCK(ErrorMessageLock);
02502 }
02503 return(-1);
02504 }
02505
02506 /*
02507     #] TenVec :
02508     #] Operations :
02509 */

```

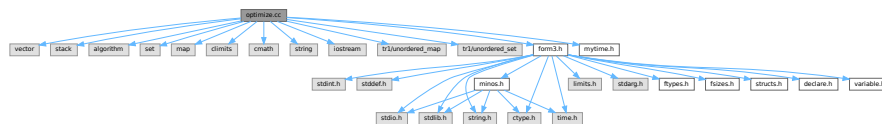
4.93 optimize.cc File Reference

```

#include <vector>
#include <stack>
#include <algorithm>
#include <set>
#include <map>
#include <climits>
#include <cmath>
#include <string>
#include <iostream>
#include <tr1/unordered_map>
#include <tr1/unordered_set>
#include "form3.h"
#include "mytime.h"

```

Include dependency graph for optimize.cc:



Data Structures

- class [tree_node](#)
- struct [CSEHash](#)
- struct [CSEEq](#)
- struct [node](#)
- struct [NodeHash](#)
- struct [NodeEq](#)
- class [optimization](#)

Typedefs

- typedef struct [node](#) **NODE**

Functions

- void [print_instr](#) (const vector< WORD > &instr, WORD num)
- template<class RandomAccessIterator >
void [my_random_shuffle](#) (PHEAD RandomAccessIterator fr, RandomAccessIterator to)
- LONG [get_expression](#) (int exprnr)
- vector< vector< WORD > > [get_brackets](#) ()
- int [count_operators](#) (const WORD *expr, bool print=false)
- int [count_operators](#) (const vector< WORD > &instr, bool print=false)
- vector< WORD > [occurrence_order](#) (const WORD *expr, bool rev)
- WORD [getpower](#) (const WORD *term, int var, int pos)
- void [fixarg](#) (UWORD *t, WORD &n)
- void [GcdLong_fix_args](#) (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- void [DivLong_fix_args](#) (UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc, UWORD *d, WORD *nd)
- void [build_Horner_tree](#) (const WORD **terms, int numterms, int var, int maxvar, int pos, vector< WORD > *res)
- bool [term_compare](#) (const WORD *a, const WORD *b)
- vector< WORD > [Horner_tree](#) (const WORD *expr, const vector< WORD > &order)
- void [print_tree](#) (const vector< WORD > &tree)
- template<typename T >
size_t [hash_range](#) (T *array, int size)
- vector< WORD > [generate_instructions](#) (const vector< WORD > &tree, bool do_CSE)
- int [count_operators_cse](#) (const vector< WORD > &tree)
- **NODE** * [buildTree](#) (vector< WORD > &tree)
- int [count_operators_cse_topdown](#) (vector< WORD > &tree)
- vector< WORD > [simulated_annealing](#) ()
- void [find_Horner_MCTS_expand_tree](#) ()
- void [find_Horner_MCTS](#) ()
- vector< WORD > [merge_operators](#) (const vector< WORD > &all_instr, bool move_coeff)
- vector< [optimization](#) > [find_optimizations](#) (const vector< WORD > &instr)
- bool [do_optimization](#) (const [optimization](#) optim, vector< WORD > &instr, int newid)
- int [partial_factorize](#) (vector< WORD > &instr, int n, int improve)
- vector< WORD > [optimize_greedy](#) (vector< WORD > instr, LONG time_limit)
- vector< WORD > [recycle_variables](#) (const vector< WORD > &all_instr)
- void [optimize_expression_given_Horner](#) ()
- void [generate_output](#) (const vector< WORD > &instr, int exprnr, int extraoffset, const vector< vector< WORD > > &brackets)
- WORD [generate_expression](#) (WORD exprnr)
- void [optimize_print_code](#) (int print_expr)
- int [Optimize](#) (WORD exprnr, int do_print)
- int [ClearOptimize](#) ()

Variables

- const WORD [OPER_ADD](#) = -1
- const WORD [OPER_MUL](#) = -2
- const WORD [OPER_COMMA](#) = -3
- WORD * [optimize_expr](#)
- vector< vector< WORD > > [optimize_best_Horner_schemes](#)
- int [optimize_num_vars](#)
- int [optimize_best_num_oper](#)
- vector< WORD > [optimize_best_instr](#)
- vector< WORD > [optimize_best_vars](#)
- bool [mcts_factorized](#)
- bool [mcts_separated](#)
- vector< WORD > [mcts_vars](#)
- [tree_node](#) [mcts_root](#)
- int [mcts_expr_score](#)
- set< pair< int, vector< WORD > > > [mcts_best_schemes](#)

4.93.1 Detailed Description

experimental routines for the optimization of FORTRAN or C output.

Definition in file [optimize.cc](#).

4.93.2 Function Documentation

4.93.2.1 [print_instr\(\)](#)

```
void print_instr (
    const vector< WORD > & instr,
    WORD num )
```

Definition at line [179](#) of file [optimize.cc](#).

4.93.2.2 [my_random_shuffle\(\)](#)

```
template<class RandomAccessIterator >
void my_random_shuffle (
    PHEAD RandomAccessIterator fr,
    RandomAccessIterator to )
```

Random shuffle

4.93.3 Description

Randomly permutes elements in the range [fr,to). Functionality is the same as C++'s "random_shuffle", but it uses Form's "wranf".

Definition at line [201](#) of file [optimize.cc](#).

Referenced by [find_Horner_MCTS\(\)](#).

4.93.3.1 `get_expression()`

```
LONG get_expression (
    int exprnr )
```

Get expression

4.93.4 Description

Reads an expression from the input file into a buffer (called `optimize_expr`). This buffer is used during the optimization process. Non-symbols are removed by `ConvertToPoly` and are put in temporary symbols.

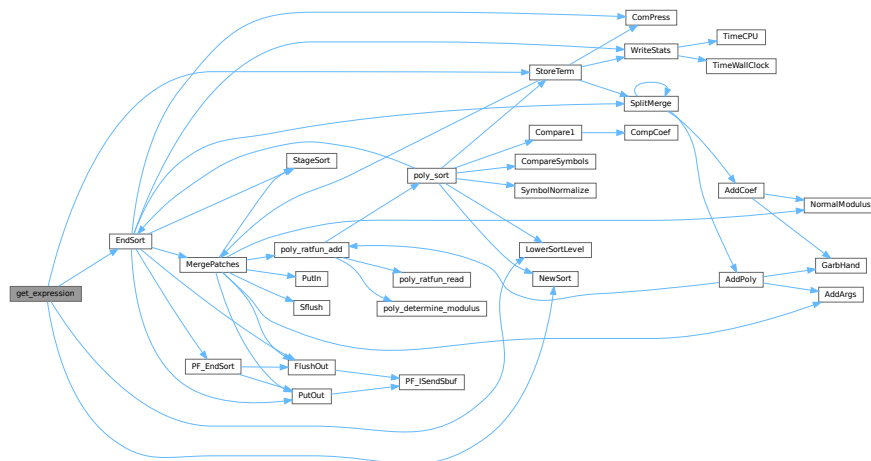
The return value is the length of the expression in WORDs, or a negative number if it failed.

Definition at line 225 of file [optimize.cc](#).

References [EndSort\(\)](#), [LowerSortLevel\(\)](#), [NewSort\(\)](#), and [StoreTerm\(\)](#).

Referenced by [Optimize\(\)](#).

Here is the call graph for this function:



4.93.4.1 `get_brackets()`

```
vector< vector< WORD > > get_brackets ( )
```

Get brackets

4.93.5 Description

Checks whether the input expression (stored in `optimize_expr`) contains brackets. If so, this method replaces terms outside brackets by powers of `SEPERATESYMBOL` (equal brackets have equal powers) and the brackets are returned. If not, the result is empty.

Brackets are used for simultaneous optimization. The symbol `SEPERATESYMBOL` is always the first one used in a Horner scheme.

Definition at line 302 of file [optimize.cc](#).

Referenced by [Optimize\(\)](#).

4.93.5.1 `count_operators()` [1/2]

```
int count_operators (
    const WORD * expr,
    bool print = false )
```

Count operators

4.93.6 Description

Counts the number of operators in a Form-style expression.

Definition at line 422 of file [optimize.cc](#).

Referenced by [find_Horner_MCTS\(\)](#), [generate_instructions\(\)](#), [Optimize\(\)](#), [optimize_expression_given_Horner\(\)](#), and [optimize_greedy\(\)](#).

4.93.6.1 `count_operators()` [2/2]

```
int count_operators (
    const vector< WORD > & instr,
    bool print = false )
```

Count operators

4.93.7 Description

Counts the number of operators in a vector of instructions

Definition at line 476 of file [optimize.cc](#).

4.93.7.1 `occurrence_order()`

```
vector< WORD > occurrence_order (
    const WORD * expr,
    bool rev )
```

Occurrence order

4.93.8 Description

Extracts all variables from an expression and sorts them with most occurring first (or last, with `rev=true`)

Definition at line 519 of file [optimize.cc](#).

Referenced by [Optimize\(\)](#).

4.93.8.1 `getpower()`

```
WORD getpower (
    const WORD * term,
    int var,
    int pos )
```

Horner tree building

4.93.9 Description

Given a Form-style expression (in a buffer in memory), this builds an expression tree. The tree is determined by a multivariate Horner scheme, i.e., something of the form:

$$1+y+x*(2+y*(1+y)+x*(3-y*(...)))$$

The order of the variables is given to the method "Horner_tree", which rennumbers ad reorders the terms of the expression. Next, the recursive method "build_Horner_tree" does the actual tree construction.

The tree is represented in postfix notation. Tokens are of the following forms:

- SNUMBER tokenlength num den coefflength
- SYMBOL tokenlength variable power
- OPER_ADD or OPER_MUL

4.93.10 Note

Sets AN.poly_num_vars and allocates AN.poly_vars. The latter should be freed later. Get power of variable (helper function for build_Horner_tree)

4.93.11 Description

Returns the power of the variable "var", which is at position "pos" in this term, if it is present.

Definition at line 600 of file [optimize.cc](#).

Referenced by [build_Horner_tree\(\)](#).

4.93.11.1 `fixarg()`

```
void fixarg (
    UWORD * t,
    WORD & n )
```

Call GcdLong/DivLong with leading zeroes

4.93.12 Description

These method remove leading zeroes, so that GcdLong and DivLong can safely be called.

Definition at line 614 of file [optimize.cc](#).

4.93.12.1 GcdLong_fix_args()

```
void GcdLong_fix_args (
    PHEAD UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc )
```

Definition at line 620 of file [optimize.cc](#).

4.93.12.2 DivLong_fix_args()

```
void DivLong_fix_args (
    UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc,
    UWORD * d,
    WORD * nd )
```

Definition at line 626 of file [optimize.cc](#).

4.93.12.3 build_Horner_tree()

```
void build_Horner_tree (
    const WORD ** terms,
    int numterms,
    int var,
    int maxvar,
    int pos,
    vector< WORD > * res )
```

Build the Horner tree

4.93.13 Description

Constructs the Horner tree. The method processes one variable and continues recursively until the Horner scheme is completed.

"terms" is a pointer to the starts of the terms. "numterms" is the number of terms to be processed. "var" is the next variable to be processed (index between 0 and #maxvar) and "maxvar" is the last variable to be processed, so that partial Horner trees can also be constructed. "pos" is the position that the power of "var" should be in (one level further in the recursion, "pos" has increased by 0 or 1 depending on whether the previous power was 0 or not). The result is written at the pointer "res".

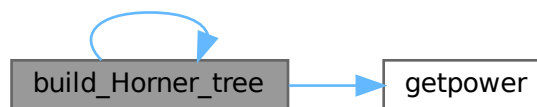
This method also factors out gcds of the coefficients. The result should end with "gcd OPER_MUL" at all times, so that one level higher gcds can be extracted again.

Definition at line 652 of file [optimize.cc](#).

References [build_Horner_tree\(\)](#), and [getpower\(\)](#).

Referenced by [build_Horner_tree\(\)](#), and [Horner_tree\(\)](#).

Here is the call graph for this function:



4.93.13.1 term_compare()

```

bool term_compare (
    const WORD * a,
    const WORD * b )
  
```

Term compare (helper function for Horner_tree)

4.93.14 Description

Compares two terms of the form "L SYM 4 x n coeff" or "L coeff". Lower powers of lower-indexed symbols come first. This is used to sort the terms in correct order.

Definition at line 852 of file [optimize.cc](#).

Referenced by [Horner_tree\(\)](#).

4.93.14.1 Horner_tree()

```
vector< WORD > Horner_tree (
    const WORD * expr,
    const vector< WORD > & order )
```

Prepare Horner tree building

4.93.15 Description

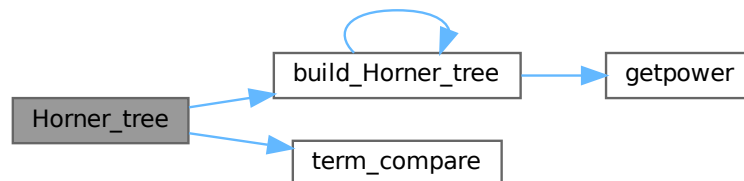
This method rennumbers the variables to 0...#vars-1 according to the specified order. Next, it stored pointer to individual terms and sorts the terms with higher power first. Then the sorted lists of power is used for the construction of the Horner tree.

Definition at line 873 of file [optimize.cc](#).

References [build_Horner_tree\(\)](#), and [term_compare\(\)](#).

Referenced by [optimize_expression_given_Horner\(\)](#).

Here is the call graph for this function:



4.93.15.1 print_tree()

```
void print_tree (
    const vector< WORD > & tree )
```

Definition at line 1001 of file [optimize.cc](#).

4.93.15.2 hash_range()

```
template<typename T >
size_t hash_range (
    T * array,
    int size )
```

Definition at line 1070 of file [optimize.cc](#).

4.93.15.3 generate_instructions()

```
vector< WORD > generate_instructions (
    const vector< WORD > & tree,
    bool do_CSE )
```

Generate instructions

4.93.16 Description

Converts the expression tree to a list of instructions that directly translate to code. Instructions are of the form:

expr.nr operator length [operands]+ trailing.zero

The operands are of the form:

length [(EXTRA)SYMBOL length variable power] coeff

This method only generates binary operators. Merging is done later. The method also checks for common subexpressions and eliminates them and the flag "do_CSE" is set.

4.93.17 Implementation details

The map "ID" keeps track of which subexpressions already exist. The key is formatted as one of the following:

SYMBOL x n SNUMBER coeff OPERATOR LHS RHS

with LHS/RHS formatted as one of the following:

SNUMBER idx 0 (EXTRA)SYMBOL x n

ID[symbol] or ID[operator] equals a subexpression number. ID[coeff] equals the position of the number in the input.

The stack s is used to process the postfix expression tree. Three-word tokens of the form:

SNUMBER idx.of.coeff 0 SYMBOL x n EXTRASYMBOL x 1

are pushed onto it. Operators pop two operands and push the resulting expression.

(Extra)symbols are 1-indexed, because -X is also needed to represent -1 times this term.

There is currently a bug. The notation cannot tell if there is a single bracket and then ignores the bracket.

TODO: check if this method performs properly if do_CSE=false

Definition at line 1132 of file [optimize.cc](#).

References [count_operators\(\)](#).

Referenced by [optimize_expression_given_Horner\(\)](#).

Here is the call graph for this function:



4.93.17.1 count_operators_cse()

```
int count_operators_cse (
    const vector< WORD > & tree )
```

Count number of operators in a binary tree, while removing CSEs on the fly. The instruction set is not created, which makes this method slightly faster.

A hash is created on the fly and is passed through the stack. TODO: find better hash functions

Definition at line 1341 of file [optimize.cc](#).

4.93.17.2 buildTree()

```
NODE * buildTree (
    vector< WORD > & tree )
```

Definition at line 1563 of file [optimize.cc](#).

4.93.17.3 count_operators_cse_topdown()

```
int count_operators_cse_topdown (
    vector< WORD > & tree )
```

Definition at line 1625 of file [optimize.cc](#).

4.93.17.4 simulated_annealing()

```
vector< WORD > simulated_annealing ( )
```

Definition at line 1680 of file [optimize.cc](#).

4.93.17.5 find_Horner_MCTS_expand_tree()

```
void find_Horner_MCTS_expand_tree ( )
```

Definition at line 2053 of file [optimize.cc](#).

4.93.17.6 find_Horner_MCTS()

```
void find_Horner_MCTS ( )
```

Find best Horner schemes using MCTS

4.93.18 Description

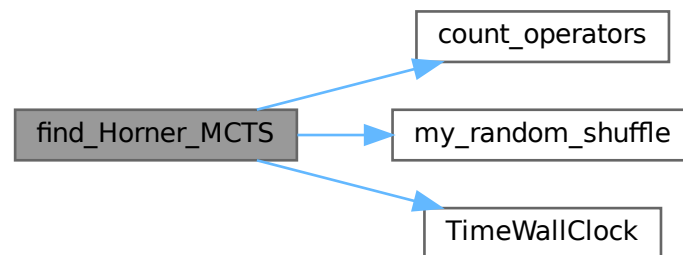
The method governs the MCTS for the best Horner schemes. It does some pre-processing, calls "find_Horner_MCTS_expand_tree" a number of times and does some post-processing.

Definition at line 2254 of file [optimize.cc](#).

References [count_operators\(\)](#), [my_random_shuffle\(\)](#), and [TimeWallClock\(\)](#).

Referenced by [Optimize\(\)](#).

Here is the call graph for this function:



4.93.18.1 merge_operators()

```
vector< WORD > merge_operators (
    const vector< WORD > & all_instr,
    bool move_coeff )
```

Merge operators

4.93.19 Description

The input instructions form a binary DAG. This method merges expressions like

$Z1 = a+b; Z2 = Z1+c;$

into

$Z2 = a+b+c;$

An instruction is merged iff it only has one parent and the operator equals its parent's operator.

This still leaves some freedom: where should the coefficients end up in cases as:

$Z1 = Z2 + x \Leftrightarrow Z1 = 2*Z2 + x \quad Z2 = 2*x*y \quad Z2 = x*y$

Both are relevant, e.g. for CSE of the form "2*x" and "2*Z2". The flag "move_coeff" moves coefficients from LHS-like expressions to RHS-like expressions.

Furthermore, this method removes empty equation ($Z1=0$), that are introduced by some "optimize_greedy" substitutions.

4.93.20 Implementation details

Expressions are mostly traversed via a stack, so that parents are evaluated before their children.

With "move_coeff" set coefficients are moved, but this leads to some tricky cases, e.g.

$$Z1 = Z2 + x \quad Z2 = 2*y$$

Here Z2 reduces to the trivial equation $Z2=y$, which should be eliminated. Here the array skip[i] comes in.

Furthermore in the case

$$Z1 = Z2 + x \quad Z2 = 2*Z3 \quad Z3 = x*Z4 \quad Z4 = y*z$$

after substituting $Z1 = 2*Z3 + x$, the parent expression for Z4 becomes Z3 instead of Z2. This is where renum_par[i] comes in.

Finally, once a coefficient has been moved, skip_coeff[i] is set and this coefficient is copied into the new expression anymore.

Definition at line 2402 of file [optimize.cc](#).

Referenced by [optimize_expression_given_Horner\(\)](#).

4.93.20.1 find_optimizations()

```
vector< optimization > find_optimizations (
    const vector< WORD > & instr )
```

Find optimizations

4.93.21 Description

This method find all optimization of the form described in "class Optimization". It process every equation, looking for possible optimizations and stores them in a fast-access data structure to count the total improvement of an optimization.

Definition at line 2697 of file [optimize.cc](#).

Referenced by [optimize_greedy\(\)](#).

4.93.21.1 do_optimization()

```
bool do_optimization (
    const optimization optim,
    vector< WORD > & instr,
    int newid )
```

Do optimization

4.93.22 Description

This method performs an optimization. It scans through the equations of "optim.eqnidxs" and looks in which this optimization can still be performed (due to other performed optimizations this isn't always the case). If possible, it substitutes the common subexpression by a new extra symbol numbered "newid". Finally, the new extrasymbol is defined accordingly.

Substitutions may lead to trivial equations of the form " $Z_i=Z_j$ ", but these are removed in the end of the method. The method returns whether the substitution has been done once or more (or not).

Definition at line 2930 of file [optimize.cc](#).

Referenced by [optimize_greedy\(\)](#).

4.93.22.1 partial_factorize()

```
int partial_factorize (
    vector< WORD > & instr,
    int n,
    int improve )
```

Partial factorization of instructions

4.93.23 Description

This method performs partial factorization of instructions. In particular the following instructions

$Z_1 = x*a*b$ $Z_2 = x*c*d*e$ $Z_3 = 2*x + Z_1 + Z_2 + \text{more}$

are replaced by

$Z_1 = a*b$ $Z_2 = c*d*e$ $Z_3 = Z_j + \text{more}$ $Z_i = 2 + Z_1 + Z_2$ $Z_j = x*Z_i$

Here it is necessary that no other equations refer to Z_1 and Z_2 . The generation of trivial instructions ($Z_i=Z_j$ or $Z_i=x$) is prevented.

Definition at line 3479 of file [optimize.cc](#).

Referenced by [optimize_greedy\(\)](#).

4.93.23.1 optimize_greedy()

```
vector< WORD > optimize_greedy (
    vector< WORD > instr,
    LONG time_limit )
```

Optimize instructions greedily

4.93.24 Description

This method optimizes an expression greedily. It calls "find_optimizations" to obtain candidates and performs the best one(s) by calling "do_optimization".

How many different optimization are done, before "find_optimization" is called again, is determined by the settings "greedyminnum" and "greedymaxperc".

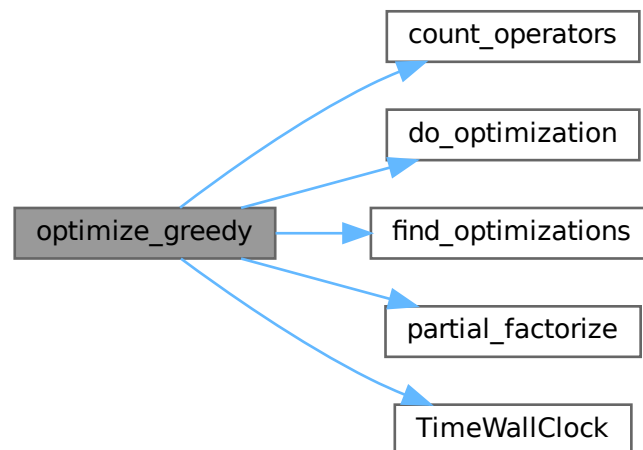
During the optimization process, sequences of zeroes are introduced in the instructions, since moving all instructions when one gets optimized, is very costly. Therefore, in the end, the instructions are "compressed" again to remove these extra zeroes.

Definition at line 3773 of file [optimize.cc](#).

References [count_operators\(\)](#), [do_optimization\(\)](#), [find_optimizations\(\)](#), [partial_factorize\(\)](#), and [TimeWallClock\(\)](#).

Referenced by [optimize_expression_given_Horner\(\)](#).

Here is the call graph for this function:



4.93.24.1 recycle_variables()

```
vector< WORD > recycle_variables (
    const vector< WORD > & all_instr )
```

Recycle variables

4.93.25 Description

The current input uses many temporary variables. Many of them become obsolete at some point during the evaluation of the code, so can be recycled. This method rennumbers the temporary variables, so that they are recycled. Furthermore, the input is order in depth-first order, so that the instructions can be performed consecutively.

4.93.26 Implementation details

First, for each subDAG, an estimate for the number of variables needed is made. This is done by the following recursive formula:

$$\#vars(x) = \max(\#vars(ch\_i(x)) + i),$$

with $ch\_i(x)$ the i -th child of x , where the childs are ordered w.r.t. $\#vars(ch\_i)$. This formula is exact if the input forms a tree, and otherwise gives a reasonable estimate.

Then, the instructions are reordered in a depth-first order with childs ordered w.r.t. $\#vars$. Next, the times that variables become obsolete are found. Each LHS of an instruction is renumbered to the lowest-numbered temporary variable that is available at that time.

Definition at line 3913 of file [optimize.cc](#).

Referenced by [optimize_expression_given_Horner\(\)](#).

4.93.26.1 optimize_expression_given_Horner()

```
void optimize_expression_given_Horner ( )
```

Optimize expression given a Horner scheme

4.93.27 Description

This method picks one Horner scheme from the list of best Horner schemes, applies this scheme to the expression and then, depending on `optimize.settings`, does a common subexpression elimination (CSE) or performs greedy optimizations.

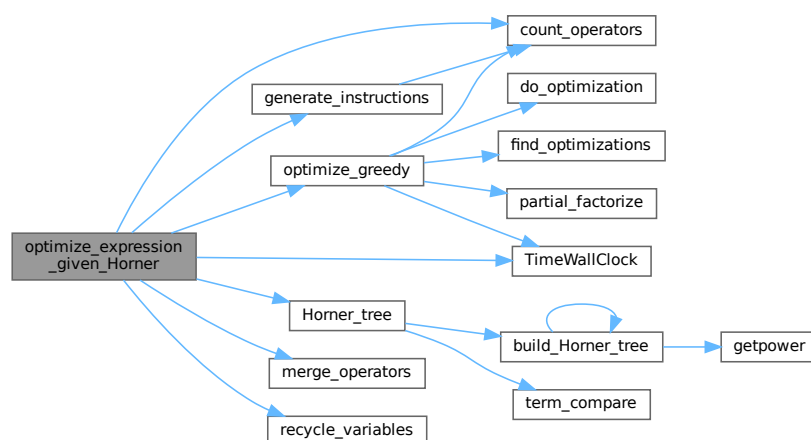
CSE is fast, while greedy might be slow. CSE followed by greedy is faster than greedy alone, but typically results in slightly worse code (not proven; just observed). eventually do greedy optimizations

Definition at line 4060 of file [optimize.cc](#).

References [count_operators\(\)](#), [generate_instructions\(\)](#), [Horner_tree\(\)](#), [merge_operators\(\)](#), [optimize_greedy\(\)](#), [recycle_variables\(\)](#), and [TimeWallClock\(\)](#).

Referenced by [Optimize\(\)](#).

Here is the call graph for this function:



4.93.27.1 generate_output()

```
void generate_output (
    const vector< WORD > & instr,
    int exprnr,
    int extraoffset,
    const vector< vector< WORD > > & brackets )
```

Generate output

4.93.28 Description

This method prepares the instructions for printing. They are stored in Form format, so that they can be printed by "PrintExtraSymbol". The results are stored in the buffer AO.OptimizeResult.

Definition at line [4291](#) of file [optimize.cc](#).

Referenced by [Optimize\(\)](#).

4.93.28.1 generate_expression()

```
WORD generate_expression (
    WORD exprnr )
```

Generate expression

4.93.29 Description

This method modifies the original optimized expression by an expression with extra symbols. This is used for "#Optimize".

Definition at line [4440](#) of file [optimize.cc](#).

References [EndSort\(\)](#), [Generator\(\)](#), [LowerSortLevel\(\)](#), [NewSort\(\)](#), and [PutOut\(\)](#).

Referenced by [Optimize\(\)](#).

4.93.31 Description

This method takes an input expression and generates optimized code to calculate its value. The following methods are called to do so:

- (1) `get_expression` : to read to expression
- (2) `get_brackets` : find brackets for simultaneous optimization
- (3) `occurrence_order` or `find_Horner_MCTS` : to determine (the) Horner scheme(s) to use; this depends on `AO.optimize.horner`
- (4) `optimize_expression_given_Horner` : to do the optimizations for each Horner scheme; this method does either CSE or greedy optimizations dependings on `AO.optimize.method`
- (5) `generate_output` : to format the output in Form notation and store it in a buffer
- (6a) `optimize_print_code` : to print the expression (for "Print") or (6b) `generate_expression` : to modify the expression (for "#Optimize")

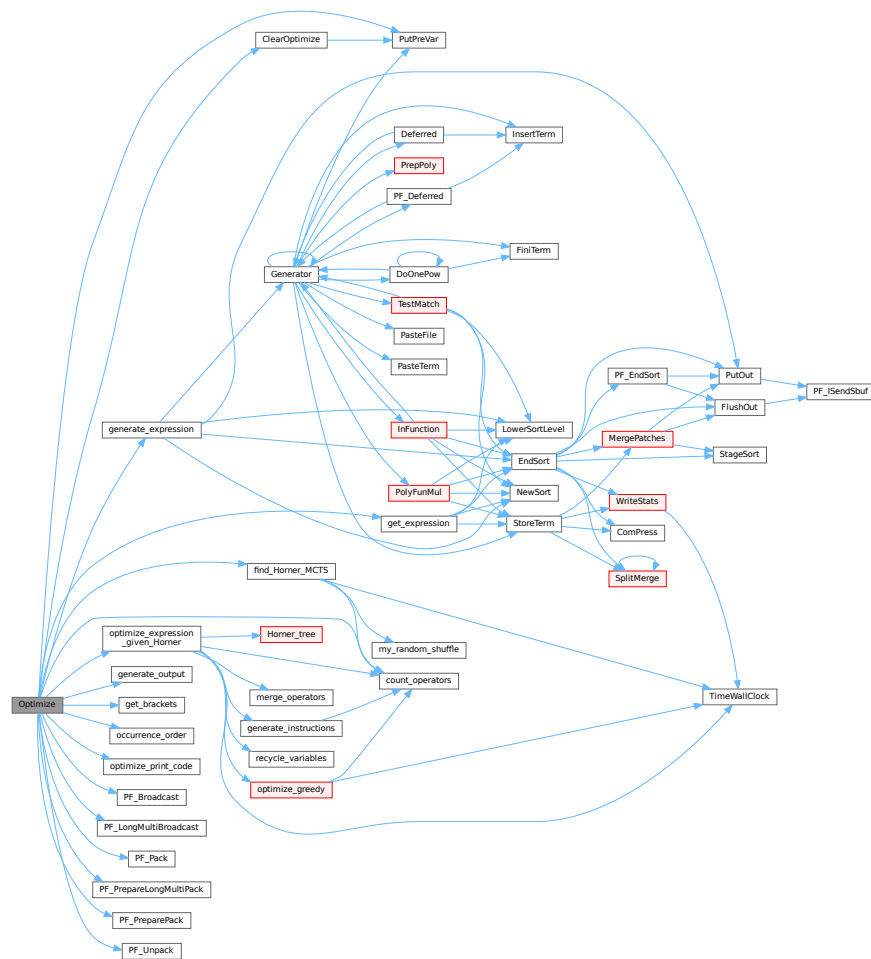
On ParFORM, all the processes must call this function at the same time. Then

- (1) Because only the master can access to the expression to be optimized, the master broadcast the expression to all the slaves after reading the expression (`PF_get_expression`).
- (2) `get_brackets` reads `optimize_expr` as the input and it works also on the slaves. We leave it although the bracket information is not needed on the slaves (used in (5) on the master).
- (3) and (4) `find_Horner_MCTS` and `optimize_expression_given_Horner` are parallelized.
- (5), (6a) and (6b) are needed only on the master.

Definition at line 4637 of file [optimize.cc](#).

References [ClearOptimize\(\)](#), [count_operators\(\)](#), [find_Horner_MCTS\(\)](#), [generate_expression\(\)](#), [generate_output\(\)](#), [get_brackets\(\)](#), [get_expression\(\)](#), [occurrence_order\(\)](#), [optimize_expression_given_Horner\(\)](#), [optimize_print_code\(\)](#), [PF_Broadcast\(\)](#), [PF_LongMultiBroadcast\(\)](#), [PF_Pack\(\)](#), [PF_PrepareLongMultiPack\(\)](#), [PF_PreparePack\(\)](#), [PF_Unpack\(\)](#), and [PutPreVar\(\)](#).

Here is the call graph for this function:



4.93.31.1 ClearOptimize()

```
int ClearOptimize (
    void )
```

Optimization of expression

4.93.32 Description

Clears the buffers that were used for optimization output. Clears the expression from the buffers (marks it to be dropped). Note: we need to use the expression by its name, because numbers may change if we drop other expressions between when we do the optimizations and clear the results (in [execute.c](#)). Also this is not 100% safe, because we could overwrite the optimized expression. But that can be done only in a Local or Global statement and hence we only have to test there that we might have to call ClearOptimize first. (in file [comexpr.c](#))

Definition at line 4974 of file [optimize.cc](#).

References [PutPreVar\(\)](#).

Referenced by [Optimize\(\)](#).

Here is the call graph for this function:



4.93.33 Variable Documentation

4.93.33.1 OPER_ADD

```
const WORD OPER_ADD = -1
```

Definition at line 120 of file [optimize.cc](#).

4.93.33.2 OPER_MUL

```
const WORD OPER_MUL = -2
```

Definition at line 121 of file [optimize.cc](#).

4.93.33.3 OPER_COMMA

```
const WORD OPER_COMMA = -3
```

Definition at line 122 of file [optimize.cc](#).

4.93.33.4 optimize_expr

```
WORD* optimize_expr
```

Definition at line 156 of file [optimize.cc](#).

4.93.33.5 optimize_best_Horner_schemes

```
vector<vector<WORD>> > optimize_best_Horner_schemes
```

Definition at line 157 of file [optimize.cc](#).

4.93.33.6 `optimize_num_vars`

```
int optimize_num_vars
```

Definition at line 158 of file [optimize.cc](#).

4.93.33.7 `optimize_best_num_oper`

```
int optimize_best_num_oper
```

Definition at line 159 of file [optimize.cc](#).

4.93.33.8 `optimize_best_instr`

```
vector<WORD> optimize_best_instr
```

Definition at line 160 of file [optimize.cc](#).

4.93.33.9 `optimize_best_vars`

```
vector<WORD> optimize_best_vars
```

Definition at line 161 of file [optimize.cc](#).

4.93.33.10 `mcts_factorized`

```
bool mcts_factorized
```

Definition at line 164 of file [optimize.cc](#).

4.93.33.11 `mcts_separated`

```
bool mcts_separated
```

Definition at line 164 of file [optimize.cc](#).

4.93.33.12 `mcts_vars`

```
vector<WORD> mcts_vars
```

Definition at line 165 of file [optimize.cc](#).

4.93.33.13 `mcts_root`

```
tree\_node mcts_root
```

Definition at line 166 of file [optimize.cc](#).

4.93.33.14 mcts_expr_score

```
int mcts_expr_score
```

Definition at line 167 of file [optimize.cc](#).

4.93.33.15 mcts_best_schemes

```
set<pair<int,vector<WORD>> > > mcts_best_schemes
```

Definition at line 168 of file [optimize.cc](#).

4.94 optimize.cc

[Go to the documentation of this file.](#)

```
00001
00006 /*  #[ License : */
00007 /*
00008  *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  *   When using this file you are requested to refer to the publication
00010  *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  *   This is considered a matter of courtesy as the development was paid
00012  *   for by FOM the Dutch physics granting agency and we would like to
00013  *   be able to track its scientific use to convince FOM of its value
00014  *   for the community.
00015  *
00016  *   This file is part of FORM.
00017  *
00018  *   FORM is free software: you can redistribute it and/or modify it under the
00019  *   terms of the GNU General Public License as published by the Free Software
00020  *   Foundation, either version 3 of the License, or (at your option) any later
00021  *   version.
00022  *
00023  *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00026  *   details.
00027  *
00028  *   You should have received a copy of the GNU General Public License along
00029  *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00030 */
00031 /*
00032  *   #[ License :
00033  *   #[ includes :
00034 */
00035
00036 // #define DEBUG
00037 // #define DEBUG_MORE
00038 // #define DEBUG_MCTS
00039 // #define DEBUG_GREEDY
00040
00041 #ifdef HAVE_CONFIG_H
00042 #ifndef CONFIG_H_INCLUDED
00043 #define CONFIG_H_INCLUDED
00044 #include <config.h>
00045 #endif
00046 #endif
00047
00048 #include <vector>
00049 #include <stack>
00050 #include <algorithm>
00051 #include <set>
00052 #include <map>
00053 #include <limits>
00054 #include <cmath>
00055 #include <string>
00056 #include <algorithm>
00057 #include <iostream>
00058
00059 #ifdef APPLE64
00060 #ifndef HAVE_UNORDERED_MAP
00061 #define HAVE_UNORDERED_MAP
00062 #endif
```

```

00063 #ifndef HAVE_UNORDERED_SET
00064 #define HAVE_UNORDERED_SET
00065 #endif
00066 #endif
00067
00068 #ifdef HAVE_UNORDERED_MAP
00069     #include <unordered_map>
00070     using std::unordered_map;
00071 #elif !defined(HAVE_TR1_UNORDERED_MAP) && defined(HAVE_BOOST_UNORDERED_MAP_HPP)
00072     #include <boost/unordered_map.hpp>
00073     using boost::unordered_map;
00074 #else
00075     #include <tr1/unordered_map>
00076     using std::tr1::unordered_map;
00077 #endif
00078 #ifdef HAVE_UNORDERED_SET
00079     #include <unordered_set>
00080     using std::unordered_set;
00081 #elif !defined(HAVE_TR1_UNORDERED_SET) && defined(HAVE_BOOST_UNORDERED_SET_HPP)
00082     #include <boost/unordered_set.hpp>
00083     using boost::unordered_set;
00084 #else
00085     #include <tr1/unordered_set>
00086     using std::tr1::unordered_set;
00087 #endif
00088
00089 #if defined(HAVE_BUILTIN_POPCOUNT)
00090     static inline int popcount(unsigned int x) {
00091         return __builtin_popcount(x);
00092     }
00093 #elif defined(HAVE_POPCNT)
00094     #include <intrin.h>
00095     static inline int popcount(unsigned int x) {
00096         return __popcnt(x);
00097     }
00098 #else
00099     static inline int popcount(unsigned int x) {
00100         int count = 0;
00101         while (x > 0) {
00102             if ((x & 1) == 1) count++;
00103             x >>= 1;
00104         }
00105         return count;
00106     }
00107 #endif
00108
00109 extern "C" {
00110     #include "form3.h"
00111 }
00112
00113 // #ifdef DEBUG
00114 #include "mytime.h"
00115 // #endif
00116
00117 using namespace std;
00118
00119 // operators
00120 const WORD OPER_ADD = -1;
00121 const WORD OPER_MUL = -2;
00122 const WORD OPER_COMMA = -3;
00123
00124 // class for a node of the MCTS tree
00125 class tree_node {
00126 public:
00127     vector<tree_node> childs;
00128     double sum_results;
00129     int num_visits;
00130     WORD var;
00131     bool finished;
00132
00133     tree_node(int _var=0):
00134         sum_results(0), num_visits(0), var(_var), finished(false) {}
00135
00136     tree_node(const tree_node& other):
00137         childs(other.childs),
00138         sum_results(other.sum_results),
00139         num_visits(other.num_visits),
00140         var(other.var),
00141         finished(other.finished) {}
00142
00143     tree_node& operator=(const tree_node& other) {
00144         if (this != &other) {
00145             childs = other.childs;
00146             sum_results = other.sum_results;
00147             num_visits = other.num_visits;
00148             var = other.var;
00149             finished = other.finished;

```

```

00150     }
00151     return *this;
00152 }
00153 };
00154
00155 // global variables for multithreading
00156 WORD *optimize_expr;
00157 vector<vector<WORD> > optimize_best_Horner_schemes;
00158 int optimize_num_vars;
00159 int optimize_best_num_oper;
00160 vector<WORD> optimize_best_instr;
00161 vector<WORD> optimize_best_vars;
00162
00163 // global variables for MCTS
00164 bool mcts_factorized, mcts_separated;
00165 vector<WORD> mcts_vars;
00166 tree_node mcts_root;
00167 int mcts_expr_score;
00168 set<pair<int,vector<WORD> > > mcts_best_schemes;
00169
00170 #ifdef WITHPTHREADS
00171 pthread_mutex_t optimize_lock;
00172 #endif
00173
00174 /*
00175  #] includes :
00176  #[ print_instr :
00177  */
00178
00179 void print_instr (const vector<WORD> &instr, WORD num)
00180 {
00181     const WORD *tbegin = &instr.begin();
00182     const WORD *tend = tbegin+instr.size();
00183     for (const WORD *t=tbegin; t!=tend; t+=*(t+2)) {
00184         MesPrint("<%d> %a",num,t[2],t);
00185     }
00186 }
00187
00188 /*
00189  #] print_instr :
00190  #[ my_random_shuffle :
00191  */
00192
00193 template <class RandomAccessIterator>
00194 void my_random_shuffle (PHEAD RandomAccessIterator fr, RandomAccessIterator to) {
00195     for (int i=to-fr-1; i>0; --i)
00196         swap (fr[i],fr[wrarf(BHEAD0) % (i+1)]);
00197 }
00198
00199 /*
00200  #] my_random_shuffle :
00201  #[ get_expression :
00202  */
00203
00204 static WORD comlist[3] = {TYPETOPOLYNOMIAL,3,DOALL};
00205
00206 LONG get_expression (int exprnr) {
00207     GETIDENTITY;
00208
00209     AR.NoCompress = 1;
00210
00211     NewSort(BHEAD0);
00212     EXPRESSIONS e = Expressions+exprnr;
00213     SetScratch(AR.infile,&(e->onfile));
00214
00215     // get header term
00216     WORD *term = AT.WorkPointer;
00217     GetTerm(BHEAD term);
00218
00219     NewSort(BHEAD0);
00220
00221     // get terms
00222     while (GetTerm(BHEAD term) > 0) {
00223         AT.WorkPointer = term + *term;
00224         WORD *t1 = term;
00225         WORD *t2 = term + *term;
00226         if (ConvertToPoly(BHEAD t1,t2,comlist,1) < 0) return -1;
00227         int n = *t2;
00228         NCOPY(t1,t2,n);
00229         AT.WorkPointer = term + *term;
00230         if (StoreTerm(BHEAD term)) return -1;
00231     }
00232
00233     // sort and store in buffer
00234     LONG len = EndSort(BHEAD (WORD *) ((void *) (&optimize_expr)),2);
00235     LowerSortLevel();

```

```

00256     AT.WorkPointer = term;
00257
00258     return len;
00259 }
00260
00261 /*
00262     #] get_expression :
00263     #[ PF_get_expression :
00264 */
00265 #ifdef WITHMPI
00266
00267 // get_expression for ParFORM
00268 LONG PF_get_expression (int exprnr) {
00269     LONG len;
00270     if (PF.me == MASTER) {
00271         len = get_expression(exprnr);
00272     }
00273     if (PF.numtasks > 1) {
00274         PF_BroadcastBuffer(&optimize_expr, &len);
00275     }
00276     return len;
00277 }
00278
00279 // replace get_expression called in Optimize
00280 #define get_expression PF_get_expression
00281
00282 #endif
00283 /*
00284     #] PF_get_expression :
00285     #[ get_brackets :
00286 */
00287
00302 vector<vector<WORD> > get_brackets () {
00303
00304     // check for brackets in expression
00305     bool has_brackets = false;
00306     for (WORD *t=optimize_expr; *t!=0; t+=*t) {
00307         WORD *tend=t+*t; tend-=ABS(*tend-1));
00308         for (WORD *t1=t+1; t1<tend; t1+=*(t1+1))
00309             if (*t1 == HAAKJE)
00310                 has_brackets=true;
00311     }
00312
00313     // replace brackets by SEPARATESYMBOL
00314     vector<vector<WORD> > brackets;
00315
00316     if (has_brackets) {
00317         int exprlen=10; // we need potential space for an empty bracket
00318         for (WORD *t=optimize_expr; *t!=0; t+=*t)
00319             exprlen += *t;
00320         WORD *newexpr = (WORD *)Malloc1(exprlen*sizeof(WORD), "optimize newexpr");
00321
00322         int i=0;
00323         int sep_power = 0;
00324
00325         for (WORD *t=optimize_expr; *t!=0; t+=*t) {
00326             WORD *t1 = t+1;
00327
00328             // scan for bracket
00329             vector<WORD> bracket;
00330             for (; *t1!=HAAKJE; t1+=*(t1+1))
00331                 bracket.insert(bracket.end(), t1, t1+*(t1+1));
00332
00333             if (brackets.size()==0 || bracket!=brackets.back()) {
00334                 sep_power++;
00335                 brackets.push_back(bracket);
00336             }
00337             t1+=*(t1+1);
00338
00339             WORD left = t + *t - t1;
00340             bool more_symbols = (left != ABS(*t+*t-1));
00341
00342             // add power of SEPARATESYMBOL
00343             newexpr[i++] = 1 + left + (more_symbols ? 2 : 4);
00344             newexpr[i++] = SYMBOL;
00345             newexpr[i++] = (more_symbols ? *(t1+1) + 2 : 4);
00346             newexpr[i++] = SEPARATESYMBOL;
00347             newexpr[i++] = sep_power;
00348
00349             // add remaining terms
00350             if (more_symbols) {
00351                 t1+=2;
00352                 left-=2;
00353             }
00354             while (left-->0)
00355                 newexpr[i++] = *(t1++);
00356         }

```

```

00357 /*
00358     We insert here an empty bracket that is zero.
00359     It is used for the case that there is only a single bracket which is
00360     outside the notation for trees at a later stage.
00361
00362     There may be a problem now in that in the case of sep_power==1
00363     newexpr is bigger than optimize_expr. We have to check that.
00364 */
00365     if ( sep_power == 1 )
00366     {
00367         WORD *t;
00368         vector<WORD> bracket;
00369         bracket.push_back(0);
00370         bracket.push_back(0);
00371         bracket.push_back(0);
00372         bracket.push_back(0);
00373         sep_power++;
00374         brackets.push_back(bracket);
00375         newexpr[i++] = 8;
00376         newexpr[i++] = SYMBOL;
00377         newexpr[i++] = 4;
00378         newexpr[i++] = SEPARATESYMBOL;
00379         newexpr[i++] = sep_power;
00380         newexpr[i++] = 1;
00381         newexpr[i++] = 1;
00382         newexpr[i++] = 3;
00383         newexpr[i++] = 0;
00384         for (t=optimize_expr; *t!=0; t+=*t) {}
00385         if ( t-optimize_expr+1 < i ) { // We have to redo this
00386             M_free(optimize_expr, "$-sort space");
00387             optimize_expr = (WORD *)Malloc1(i*sizeof(WORD), "$-sort space");
00388         }
00389     }
00390     else {
00391         newexpr[i++] = 0;
00392     }
00393     memcpy(optimize_expr, newexpr, i*sizeof(WORD));
00394     M_free(newexpr, "optimize newexpr");
00395
00396     // if factorized, replace SEP by FAC and remove brackets
00397     if (brackets[0].size()>0 && brackets[0][2]==FACTORSYMBOL) {
00398         for (WORD *t=optimize_expr; *t!=0; t+=*t) {
00399             if (*t == ABS(*(t+*t-1))+1) continue;
00400             if (t[1]==SYMBOL)
00401                 for (int i=3; i<t[2]; i+=2)
00402                     if (t[i]==SEPARATESYMBOL) t[i]=FACTORSYMBOL;
00403         }
00404         return vector<vector<WORD> >();
00405     }
00406 }
00407
00408 return brackets;
00409 }
00410
00411 /*
00412     #] get_brackets :
00413     #[ count_operators :
00414 */
00415
00422 int count_operators (const WORD *expr, bool print=false) {
00423
00424     int n=0;
00425     while (*(expr+n)!=0) n+=*(expr+n);
00426
00427     int cntpow=0, cntmul=0, cntadd=0, sumpow=0;
00428     WORD maxpowfac=1, maxpowsep=1;
00429
00430     for (const WORD *t=expr; *t!=0; t+=*t) {
00431         if (t!=expr) cntadd++; // new term
00432         if (*t==ABS(*(t+*t-1))+1) continue; // only coefficient
00433
00434         int cntsym=0;
00435
00436         if (t[1]==SYMBOL)
00437             for (int i=3; i<t[2]; i+=2) {
00438                 if (t[i]==FACTORSYMBOL) {
00439                     maxpowfac = max(maxpowfac, t[i+1]);
00440                     continue;
00441                 }
00442                 if (t[i]==SEPARATESYMBOL) {
00443                     maxpowsep = max(maxpowsep, t[i+1]);
00444                     continue;
00445                 }
00446                 if (t[i+1]>2) { // (extra)symbol power>2
00447                     cntpow++;
00448                     sumpow += (int)floor(log(t[i+1])/log(2.0)) + popcount(t[i+1]) - 1;
00449                 }
00450             }
00451     }

```

```

00450             if (t[i+1]==2) cntmul++; // (extra)symbol squared
00451             cntsym++;
00452         }
00453
00454         if (ABS(*(t+*t-1))!=3 || *(t+*t-2)!=1 || *(t+*t-3)!=1) cntsym++; // non +/-1 coefficient
00455
00456         if (cntsym > 0) cntmul+=cntsym-1;
00457     }
00458
00459     cntadd -= maxpowfac-1;
00460     cntmul += maxpowfac-1;
00461
00462     cntadd -= maxpowsep-1;
00463
00464     if (print)
00465         MesPrint ("*** STATS: original %1P %1M %1A : %1", cntpow,cntmul,cntadd,sumpow+cntmul+cntadd);
00466
00467     return sumpow+cntmul+cntadd;
00468 }
00469
00476 int count_operators (const vector<WORD> &instr, bool print=false) {
00477     int cntpow=0, cntmul=0, cntadd=0, sumpow=0;
00478
00479     const WORD *ebegin = &instr.begin();
00480     const WORD *eend = ebegin+instr.size();
00481
00482     for (const WORD *e=ebegin; e!=eend; e+=*(e+2)) {
00483         for (const WORD *t=e+3; *t!=0; t+=*t) {
00484             if (t!=e+3) {
00485                 if (*(e+1)==OPER_ADD) cntadd++; // new term
00486                 if (*(e+1)==OPER_MUL) cntmul++; // new term
00487             }
00488             if (*t == ABS(*(t+*t-1))+1) continue; // only coefficient
00489             if (*(t+1)==SYMBOL || *(t+1)==EXTRASymbol) {
00490                 if (*(t+4)==2) cntmul++; // (extra)symbol squared
00491                 if (*(t+4)>2) { // (extra)symbol power>2
00492                     cntpow++;
00493                     sumpow += (int)floor(log(*(t+4))/log(2.0)) + popcount(*(t+4)) - 1;
00494                 }
00495             }
00496             if (ABS(*(t+*t-1))!=3 || *(t+*t-2)!=1 || *(t+*t-3)!=1) cntmul++; // non +/-1 coefficient
00497         }
00498     }
00499 }
00500
00501 if (print)
00502     MesPrint ("*** STATS: optimized %1P %1M %1A : %1", cntpow,cntmul,cntadd,sumpow+cntmul+cntadd);
00503
00504 return sumpow+cntmul+cntadd;
00505 }
00506
00507 /*
00508 #] count_operators :
00509 #[ occurrence_order :
00510 */
00511
00519 vector<WORD> occurrence_order (const WORD *expr, bool rev) {
00520     // count the number of occurrences of variables
00521     map<WORD,int> cnt;
00522     for (const WORD *t=expr; *t!=0; t+=*t) {
00523         if (*t == ABS(*(t+*t-1))+1) continue; // skip constant term
00524         if (t[1] == SYMBOL)
00525             for (int i=3; i<t[2]; i+=2)
00526                 cnt[t[i]]++;
00527     }
00528
00529     bool is_fac=false, is_sep=false;
00530     if (cnt.count(FACTORSYMBOL)) {
00531         is_fac=true;
00532         cnt.erase(FACTORSYMBOL);
00533     }
00534     if (cnt.count(SEPARATESYMBOL)) {
00535         is_sep=true;
00536         cnt.erase(SEPARATESYMBOL);
00537     }
00538
00539     // determine the order of the variables
00540     vector<pair<int,WORD> > cnt_order;
00541     for (map<WORD,int>::iterator i=cnt.begin(); i!=cnt.end(); i++)
00542         cnt_order.push_back(make_pair(i->second, i->first));
00543     sort(cnt_order.rbegin(), cnt_order.rend());
00544
00545     // create resulting order
00546     vector<WORD> order;
00547     for (int i=0; i<(int)cnt_order.size(); i++)
00548         order.push_back(cnt_order[i].second);
00549 }

```



```

00550
00551     if (rev) reverse(order.begin(),order.end());
00552
00553     // add FACTORSYMBOL/SEPARATESYMBOL
00554     if (is_fac) order.insert(order.begin(), FACTORSYMBOL);
00555     if (is_sep) order.insert(order.begin(), SEPARATESYMBOL);
00556
00557     return order;
00558 }
00559
00560 /*
00561     #] occurrence_order :
00562     #[ Horner_tree :
00563 */
00564
00600 WORD getpower (const WORD *term, int var, int pos) {
00601     if (*term == ABS(*(term+*term-1))+1) return 0; // constant term
00602     if (2*pos+2 >= term[2]) return 0;           // too few symbols
00603     if (term[2*pos+3] != var) return 0;         // incorrect symbol
00604     return term[2*pos+4];                       // return power
00605 }
00606
00614 void fixarg (UWORD *t, WORD &n) {
00615     int an=ABS(n), sn=SGN(n);
00616     while (*(t+an-1)==0) an--;
00617     n=an*sn;
00618 }
00619
00620 void GcdLong_fix_args (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc) {
00621     fixarg(a,na);
00622     fixarg(b,nb);
00623     GcdLong(BHEAD a,na,b,nb,c,nc);
00624 }
00625
00626 void DivLong_fix_args(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc, UWORD *d, WORD *nd)
00627 {
00628     fixarg(a,na);
00629     fixarg(b,nb);
00629     DivLong(a,na,b,nb,c,nc,d,nd);
00630 }
00631
00652 void build_Horner_tree (const WORD **terms, int numterms, int var, int maxvar, int pos, vector<WORD>
*res) {
00653
00654     GETIDENTITY;
00655
00656     if (var == maxvar) {
00657         // Horner tree is finished, so add remaining terms unfactorized
00658         // (note: since only complete Horner schemes seem to be useful, numterms=1 here
00659
00660         for (int fr=0; fr<numterms; fr++) {
00661
00662             bool empty = true;
00663             const WORD *t = terms[fr];
00664
00665             // add symbols
00666             if (*t != ABS(*(t+*t-1))+1)
00667                 for (int i=3+2*pos; i<t[2]; i+=2) {
00668                     res->push_back(SYMBOL);
00669                     res->push_back(4);
00670                     res->push_back(t[i]);
00671                     res->push_back(t[i+1]);
00672                     if (!empty) res->push_back(OPER_MUL);
00673                     empty = false;
00674                 }
00675
00676             // if empty, add a 1, since the result should look like "... coeff *"
00677             if (empty) {
00678                 res->push_back(SNUMBER);
00679                 res->push_back(5);
00680                 res->push_back(1);
00681                 res->push_back(1);
00682                 res->push_back(3);
00683             }
00684
00685             // add coefficient
00686             res->push_back(SNUMBER);
00687             WORD n = ABS(*(t+*t-1));
00688             res->push_back(n+2);
00689             for (int i=0; i<n; i++)
00690                 res->push_back(*(t+*t-n+i));
00691             res->push_back(OPER_MUL);
00692
00693             if (fr>0) res->push_back(OPER_ADD);
00694         }
00695
00696         // result should end with gcd of the terms; right now this never

```

```

00697         // triggers, but if one would allow for incomplete Horner schemes,
00698         // one should extract the gcd here
00699         if (numterms > 1) {
00700             res->push_back(SNUMBER);
00701             res->push_back(5);
00702             res->push_back(1);
00703             res->push_back(1);
00704             res->push_back(3);
00705             res->push_back(OPER_MUL);
00706         }
00707     }
00708     else {
00709         // extract variable "var" and the gcd and proceed recursively
00710
00711         WORD nnum = 0, nden = 0, ntmp = 0, ndum = 0;
00712         UWORD *num = NumberMalloc("build_Horner_tree");
00713         UWORD *den = NumberMalloc("build_Horner_tree");
00714         UWORD *tmp = NumberMalloc("build_Horner_tree");
00715         UWORD *dum = NumberMalloc("build_Horner_tree");
00716
00717         // previous coefficient for gcd extraction or coefficient multiplication
00718         int prev_coeff_idx = -1;
00719
00720         for (int fr=0; fr<numterms;) {
00721
00722             // find power of current term
00723             WORD pow = getpower(terms[fr], var, pos);
00724
00725             // find all terms with that power
00726             int to=fr+1;
00727             while (to<numterms && getpower(terms[to], var, pos) == pow) to++;
00728
00729             // recursively build Horner tree of all terms proportional to var^pow
00730             build_Horner_tree (terms+fr, to-fr, var+1, maxvar, pow==0?pos:pos+1, res);
00731
00732             if (AN.poly_vars[var] != FACTORSYMBOL && AN.poly_vars[var] != SEPARATESYMBOL) {
00733                 // if normal symbol, find gcd(numerators) and gcd(denominators)
00734                 WORD n1 = res->at(res->size()-2) / 2;
00735                 WORD *t1 = &res->at(res->size()-2-2*ABS(n1));
00736
00737                 WORD *t2 = fr==0 ? t1 : &res->at(prev_coeff_idx);
00738                 WORD n2 = fr==0 ? n1 : *(t2+(t2+1)-1) / 2;
00739                 if (fr>0) t2+=2;
00740
00741                 GcdLong_fix_args(BHEAD (UWORD *)t1,n1, (UWORD *)t2,n2,num,&nnum);
00742                 GcdLong_fix_args(BHEAD (UWORD *)t1+ABS(n1),ABS(n1), (UWORD
00743 *t2+ABS(n2),ABS(n2),den,&nden);
00744
00745             // divide out gcds; note: leading zeroes can be added here
00746             for (int i=0; i<2; i++) {
00747                 if (i==1 && fr==0) break;
00748
00749                 WORD *t = i==0 ? t1 : t2;
00750                 WORD n = i==0 ? n1 : n2;
00751
00752                 DivLong_fix_args((UWORD *)t, n, num, nnum, tmp, &ntmp, dum, &ndum);
00753                 for (int j=0; j<ABS(ntmp); j++) *t++ = tmp[j];
00754                 for (int j=0; j<ABS(n)-ABS(ntmp); j++) *t++ = 0;
00755
00756                 if (SGN(ntmp) != SGN(n)) n=-n;
00757
00758                 DivLong_fix_args((UWORD *)t, n, den, nden, tmp, &ntmp, dum, &ndum);
00759                 for (int j=0; j<ABS(ntmp); j++) *t++ = tmp[j];
00760                 for (int j=0; j<ABS(n)-ABS(ntmp); j++) *t++ = 0;
00761
00762                 *t++ = SGN(n) * (2*ABS(n)+1);
00763             }
00764
00765             // add the addition operator
00766             if (fr>0) res->push_back(OPER_ADD);
00767
00768             // add the power of "var"
00769             WORD nextpow = (to==numterms ? 0 : getpower(terms[to], var, pos));
00770
00771             if (pow-nextpow > 0) {
00772                 res->push_back(SYMBOL);
00773                 res->push_back(4);
00774                 res->push_back(var);
00775                 res->push_back(pow-nextpow);
00776                 res->push_back(OPER_MUL);
00777             }
00778
00779             // add the extracted gcd
00780             res->push_back(SNUMBER);
00781             WORD n = MaX(ABS(nnum),nden);
00782             res->push_back(n*2+3);
00783             for (int i=0; i<ABS(nnum); i++) res->push_back(num[i]);

```

```

00783         for (int i=0; i<n-ABS(nnum); i++) res->push_back(0);
00784         for (int i=0; i<nden; i++) res->push_back(den[i]);
00785         for (int i=0; i<n-ABS(nden); i++) res->push_back(0);
00786         res->push_back(SGN(nnum)*(2*n+1));
00787         res->push_back(OPER_MUL);
00788
00789         prev_coeff_idx = res->size() - ABS(res->at(res->size()-2)) - 3;
00790     }
00791     else if (AN.poly_vars[var]==FACTORSYMBOL) {
00792
00793         // if factorsymbol, multiply overall integer contents
00794
00795         if (fr>0) {
00796             WORD n1 = res->at(res->size()-2) / 2;
00797             WORD *t1 = &res->at(res->size()-2-2*ABS(n1));
00798             WORD *t2 = &res->at(prev_coeff_idx);
00799             WORD n2 = *(t2+(t2+1)-1) / 2;
00800             t2+=2;
00801
00802             MulRat(BHEAD (UWORD *)t1,n1, (UWORD *)t2,n2,tmp,&ntmp);
00803
00804             // replace previous coefficient with 1
00805             n2=ABS(n2);
00806             for (int i=0; i<ABS(n2); i++)
00807                 t2[i] = t2[n2+i] = i==0 ? 1 : 0;
00808             t2[2*n2] = 2*n2+1;
00809
00810             // remove this coefficient
00811             for (int i=0; i<2*ABS(n1)+2; i++)
00812                 res->pop_back();
00813
00814             // add product
00815             res->back() = 2*ABS(ntmp)+3; // adjust size of term
00816             res->insert(res->end(), tmp, tmp+2*ABS(ntmp)); // num/den coefficient
00817             res->push_back(SGN(ntmp)*(2*ABS(ntmp)+1)); // size of coefficient
00818             res->push_back(OPER_MUL); // operator
00819         }
00820
00821         prev_coeff_idx = res->size() - ABS(res->at(res->size()-2)) - 3;
00822
00823         // multiply previous factors with this factor
00824         if (fr>0)
00825             res->push_back(OPER_MUL);
00826     }
00827     else { // AN.poly_vars[var]==SEPARATESYMBOL
00828         if (fr>0)
00829             res->push_back(OPER_COMMA);
00830         prev_coeff_idx = -1;
00831     }
00832
00833     fr=to;
00834 }
00835
00836 // cleanup
00837 NumberFree(dum,"build_Horner_tree");
00838 NumberFree(tmp,"build_Horner_tree");
00839 NumberFree(den,"build_Horner_tree");
00840 NumberFree(num,"build_Horner_tree");
00841 }
00842 }
00843
00852 bool term_compare(const WORD *a, const WORD *b) {
00853     if (*a == ABS(*(a+a-1))+1) return true; // coefficient comes first
00854     if (*b == ABS(*(b+b-1))+1) return false;
00855     if (a[1]!=SYMBOL) return true;
00856     if (b[1]!=SYMBOL) return false;
00857     for (int i=3; i<a[2] && i<b[2]; i+=2) {
00858         if (a[i] !=b[i] ) return a[i] >b[i];
00859         if (a[i+1]!=b[i+1]) return a[i+1]<b[i+1];
00860     }
00861     return a[2]<b[2];
00862 }
00863
00873 vector<WORD> Horner_tree(const WORD *expr, const vector<WORD> &order) {
00874     #ifdef DEBUG
00875         MesPrint ("*** [%s, w=%w] CALL: Horner_tree(%a)", thetime_str().c_str(), order.size(), &order[0]);
00876     #endif
00877
00878     GETIDENTITY;
00879
00880     // find the renumbering scheme (new numbers are 0,1,...,#vars-1)
00881     map<WORD,WORD> renum;
00882     AN.poly_num_vars = order.size();
00883     AN.poly_vars = (WORD *)Malloca(AN.poly_num_vars*sizeof(WORD), "AN.poly_vars");
00884     for (int i=0; i<AN.poly_num_vars; i++) {
00885         AN.poly_vars[i] = order[i];
00886         renum[order[i]] = i;
00887     }

```

```

00887     }
00888
00889     // sort variables in individual terms using bubble sort
00890     WORD* sorted;
00891     WORD* dynamicAlloc = 0;
00892     LONG sumsize = 0;
00893
00894     for (const WORD *t=expr; *t!=0; t+=*t) {
00895         sumsize += *t;
00896     }
00897
00898     // We actually need sumsize + 1 WORDS available, due to the "*sorted = 0;"
00899     // at the end of the sort loop.
00900     if ( AT.WorkPointer + sumsize + 1 > AT.WorkTop ) {
00901         dynamicAlloc = (WORD*)Malloc1(sizeof(*dynamicAlloc)*(sumsize+1), "Horner_tree alloc");
00902         sorted = dynamicAlloc;
00903     }
00904     else {
00905         sorted = AT.WorkPointer;
00906     }
00907
00908     for (const WORD *t=expr; *t!=0; t+=*t) {
00909         memcpy(sorted, t, *t*sizeof(WORD));
00910
00911         if (*t != ABS(*t+*t-1))+1) {
00912             // non-constant term
00913
00914             // renumber variables, adding new variables if necessary
00915             for (int i=3; i<sorted[2]; i+=2) {
00916                 if (!renum.count(sorted[i])) {
00917                     renum[sorted[i]] = AN.poly_num_vars;
00918
00919                     WORD *new_poly_vars = (WORD *)Malloc1((AN.poly_num_vars+1)*sizeof(WORD),
00920 "AN.poly_vars");
00921                     memcpy(new_poly_vars, AN.poly_vars, AN.poly_num_vars*sizeof(WORD));
00922                     M_free(AN.poly_vars, "poly_vars");
00923                     AN.poly_vars = new_poly_vars;
00924                     AN.poly_vars[AN.poly_num_vars] = sorted[i];
00925                     AN.poly_num_vars++;
00926                 }
00927                 sorted[i] = renum[sorted[i]];
00928             }
00929             // order variables
00930             for (int i=0; i<sorted[2]/2; i++)
00931                 for (int j=3; j+2<sorted[2]; j+=2)
00932                     if (sorted[j] > sorted[j+2]) {
00933                         swap(sorted[j], sorted[j+2]);
00934                         swap(sorted[j+1], sorted[j+3]);
00935                     }
00936
00937             sorted += *sorted;
00938         }
00939
00940         *sorted = 0;
00941         if ( dynamicAlloc ) {
00942             sorted = dynamicAlloc;
00943         }
00944         else {
00945             sorted = AT.WorkPointer;
00946         }
00947
00948         // find pointers to all terms and sort them efficiently
00949         vector<const WORD *> terms;
00950         for (const WORD *t=sorted; *t!=0; t+=*t)
00951             terms.push_back(t);
00952         sort(terms.rbegin(), terms.rend(), term_compare);
00953
00954         // construct the Horner tree
00955         vector<WORD> res;
00956         build_Horner_tree(&terms[0], terms.size(), 0, AN.poly_num_vars, 0, &res);
00957
00958         // remove leading zeroes in coefficients
00959         int j=0;
00960         for (int i=0; i<(int)res.size(); i++) {
00961             if (res[i]==OPER_ADD || res[i]==OPER_MUL || res[i]==OPER_COMMA)
00962                 res[j++] = res[i++];
00963             else if (res[i]==SYMBOL) {
00964                 memmove(&res[j], &res[i], res[i+1]*sizeof(WORD));
00965                 i+=res[j+1];
00966                 j+=res[j+1];
00967             }
00968             else if (res[i]==SNUMBER) {
00969                 int n = (res[i+1]-2)/2;
00970                 int dn = 0;
00971                 while (res[i+1+n-dn]==0 && res[i+1+2*n-dn]==0) dn++;
00972                 res[j++] = SNUMBER;

```

```

00973         res[j++] = 2*(n-dn) + 3;
00974         memmove(&res[j], &res[i+2], (n-dn)*sizeof(WORD));
00975         j+=n-dn;
00976         memmove(&res[j], &res[i+n+2], (n-dn)*sizeof(WORD));
00977         j+=n-dn;
00978         res[j++] = SGN(res[i+2*n+2])*(2*(n-dn)+1);
00979         i+=2*n+3;
00980     }
00981 }
00982 res.resize(j);
00983
00984 if ( dynamicAlloc ) {
00985     M_free(dynamicAlloc, "Horner_tree alloc");
00986 }
00987
00988 #ifdef DEBUG
00989     MesPrint ("*** [%s, w=%w] DONE: Horner_tree(%a)", thetime_str().c_str(), order.size(), &order[0]);
00990 #endif
00991
00992     return res;
00993 }
00994
00995 /*
00996  #] Horner_tree :
00997  #[ print_tree :
00998  */
00999
01000 // print Horner tree (for debugging)
01001 void print_tree (const vector<WORD> &tree) {
01002
01003     GETIDENTITY;
01004
01005     for (int i=0; i<(int)tree.size();) {
01006         if (tree[i]==OPER_ADD) {
01007             MesPrint("+");
01008             i++;
01009         }
01010         else if (tree[i]==OPER_MUL) {
01011             MesPrint("*");
01012             i++;
01013         }
01014         else if (tree[i]==OPER_COMMA) {
01015             MesPrint(",");
01016             i++;
01017         }
01018         else if (tree[i]==SNUMBER) {
01019             UBYTE buf[100];
01020             int n = tree[i+tree[i+1]-1]/2;
01021             PrtLong((UWORD *)&tree[i+2], n, buf);
01022             int l = strlen((char *)buf);
01023             buf[l]='/';
01024             n=ABS(n);
01025             PrtLong((UWORD *)&tree[i+2+n], n, buf+l+1);
01026             MesPrint("%s",buf);
01027             i+=tree[i+1];
01028         }
01029         else if (tree[i]==SYMBOL) {
01030             if (AN.poly_vars[tree[i+2]] < 10000)
01031                 MesPrint("%s^%d", VARNAME(symbols,AN.poly_vars[tree[i+2]]), tree[i+3]);
01032             else
01033                 MesPrint("Z%d^%d", MAXVARIABLES-AN.poly_vars[tree[i+2]], tree[i+3]);
01034             i+=tree[i+1];
01035         }
01036         else {
01037             MesPrint("error");
01038             exit(1);
01039         }
01040
01041         MesPrint(" ");
01042     }
01043
01044     MesPrint("");
01045 }
01046
01047 /*
01048  #] print_tree :
01049  #[ generate_instructions :
01050  */
01051
01052
01053 struct CSEHash {
01054     size_t operator()(const vector<WORD>& n) const {
01055         return n[0];
01056     }
01057 };
01058
01059 struct CSEEq {

```

```

01060     bool operator() (const vector<WORD>& lhs, const vector<WORD>& rhs) const {
01061         if (lhs.size() != rhs.size()) return false;
01062         for (unsigned int i = 1; i < lhs.size(); i++) {
01063             if (lhs[i] != rhs[i]) return false;
01064         }
01065         return true;
01066     }
01067 };
01068
01069
01070 template<typename T> size_t hash_range(T* array, int size) {
01071     size_t hash = 0;
01072
01073     for (int i = 0; i < size; i++) {
01074         hash ^= array[i] + 0x9e3779b9 + (hash « 6) + (hash » 2);
01075     }
01076
01077     return hash;
01078 }
01079
01132 vector<WORD> generate_instructions (const vector<WORD> &tree, bool do_CSE) {
01133 #ifdef DEBUG
01134     MesPrint ("*** [%s, w=%w] CALL: generate_instructions(cse=%d)",
01135             thetime_str().c_str(), do_CSE?1:0);
01136 #endif
01137
01138     typedef unordered_map<vector<WORD>, int, CSEHash, CSEEq> csemap;
01139     csemap ID;
01140
01141     // reserve lots of space, to prevent later rehashes
01142     // TODO: what if this is too large? make a parameter?
01143     if (do_CSE) {
01144         ID.rehash(mcts_expr_score * 2);
01145     }
01146
01147     // s is a stack of operands to process when you encounter operators
01148     // in the postfix expression tree. Operands consist of three WORDs,
01149     // formatted as the LHS/RHS of the keys in ID.
01150     stack<int> s;
01151     vector<WORD> instr;
01152     WORD numinstr = 0;
01153     vector<WORD> x;
01154
01155     // process the expression tree
01156     for (int i=0; i<(int)tree.size(); i) {
01157         x.clear();
01158
01159         if (tree[i]==SNUMBER) {
01160             WORD hash = hash_range(&tree[i], tree[i + 1]);
01161             x.push_back(hash);
01162             x.push_back(SNUMBER);
01163             x.insert(x.end(),&tree[i],&tree[i+1]);
01164             int sign = SGN(x.back());
01165             x.back() *= sign;
01166
01167             std::pair<csemap::iterator, bool> suc = ID.insert(csemap::value_type(x, i + 1));
01168             s.push(0);
01169             s.push(sign * suc.first->second);
01170             s.push(SNUMBER);
01171             s.push(hash);
01172             i+=tree[i+1];
01173         }
01174         else if (tree[i]==SYMBOL) {
01175             WORD hash = hash_range(&tree[i], tree[i + 1]);
01176             if (tree[i+3]>1) {
01177                 x.push_back(hash);
01178                 x.push_back(SYMBOL);
01179                 x.push_back(tree[i+2]+1); // variable (1-indexed)
01180                 x.push_back(tree[i+3]); // power
01181
01182                 csemap::iterator it = ID.find(x);
01183                 if (do_CSE && it != ID.end()) {
01184                     // already-seen power of a symbol
01185                     s.push(1);
01186                     s.push(it->second);
01187                     s.push(EXTRASYMBOL);
01188                 } else {
01189                     //MesPrint ("strange");
01190                     if (numinstr == MAXPOSITIVE) {
01191                         MesPrint ((char *) "ERROR: too many temporary variables needed in
optimization");
01192                         Terminate(-1);
01193                     }
01194
01195                     // new power greater than 1 of a symbol
01196                     instr.push_back(numinstr); // expr.nr
01197                     instr.push_back(OPER_MUL); // operator

```

```

01198         instr.push_back(9+OPTHEAD); // length total
01199         instr.push_back(8);          // length operand
01200         instr.push_back(SYMBOL);     // SYMBOL
01201         instr.push_back(4);          // length symbol
01202         instr.push_back(tree[i+2]);  // variable
01203         instr.push_back(tree[i+3]);  // power
01204         instr.push_back(1);          // numerator
01205         instr.push_back(1);          // denominator
01206         instr.push_back(3);          // length coeff
01207         instr.push_back(0);          // trailing 0
01208
01209         ID[x] = ++numinstr;
01210         s.push(1);
01211         s.push(numinstr);
01212         s.push(EXTRASymbol);
01213     }
01214 }
01215 else {
01216     // power of 1
01217     s.push(tree[i+3]); // power
01218     s.push(tree[i+2]+1); // variable (1-indexed)
01219     s.push(SYMBOL);
01220 }
01221
01222 s.push(hash); // push hash
01223 i+=tree[i+1];
01224 }
01225 else { // tree[i]==OPERATOR
01226     int oper = tree[i];
01227     i++;
01228
01229     x.push_back(0); // placeholder for hash
01230     x.push_back(oper);
01231     vector<WORD> hash;
01232     hash.push_back(oper);
01233
01234     // get two operands from the stack
01235     for (int operand=0; operand<2; operand++) {
01236         hash.push_back(s.top()); s.pop();
01237         x.push_back(s.top()); s.pop();
01238         x.push_back(s.top()); s.pop();
01239         x.push_back(s.top()); s.pop();
01240     }
01241
01242     x[0] = hash_range(&hash[0], 3);
01243
01244     // get rid of multiplications by +/-1
01245     if (oper==OPER_MUL) {
01246         bool do_continue = false;
01247
01248         for (int operand=0; operand<2; operand++) {
01249             int idx_oper1 = operand==0 ? 2 : 5;
01250             int idx_oper2 = operand==0 ? 5 : 2;
01251
01252             // check whether operand 1 equals +/-1
01253             if (x[idx_oper1]==SNUMBER) {
01254
01255                 int idx = ABS(x[idx_oper1+1])-1;
01256                 if (tree[idx+2]==1 && tree[idx+3]==1 && ABS(tree[idx+4])==3) {
01257                     // push +/- other operand and continue
01258                     s.push(x[idx_oper2+2]);
01259                     s.push(x[idx_oper2+1]*SGN(x[idx_oper1+1]));
01260                     s.push(x[idx_oper2]);
01261                     s.push(hash[1 + (operand + 1) % 2]);
01262                     do_continue = true;
01263                     break;
01264                 }
01265             }
01266         }
01267
01268         if (do_continue) continue;
01269     }
01270
01271     // check whether this subexpression has been seen before
01272     // if not, generate instruction to define it
01273     csemap::iterator it = ID.find(x);
01274     if (!do_CSE || it == ID.end()) {
01275         if (numinstr == MAXPOSITIVE) {
01276             MesPrint((char *) "ERROR: too many temporary variables needed in optimization");
01277             Terminate(-1);
01278         }
01279
01280         instr.push_back(numinstr); // expr.nr.
01281         instr.push_back(x[1]);     // operator
01282         instr.push_back(3);        // length
01283
01284         ID[x] = ++numinstr;

```

```

01285
01286         int lenidx = instr.size()-1;
01287
01288         for (int j=0; j<2; j++)
01289             if (x[3*j+2]==SYMBOL || x[3*j+2]==EXTRASYMBOL) {
01290                 instr.push_back(8); // length total
01291                 instr.push_back(x[3*j+2]); // (extra)symbol
01292                 instr.push_back(4); // length (extra)symbol
01293                 instr.push_back(ABS(x[3*j+3])-1); // variable (0-indexed)
01294                 instr.push_back(x[3*j+4]); // power
01295                 instr.push_back(1); // numerator
01296                 instr.push_back(1); // denominator
01297                 instr.push_back(3*SGN(x[3*j+3])); // length coeff
01298                 instr[lenidx] += 8;
01299             }
01300             else { // x[3*j+1]==SNUMBER
01301                 int t = ABS(x[3*j+3])-1;
01302                 instr.push_back(tree[t+1]-1); // length number
01303                 instr.insert(instr.end(), &tree[t+2], &tree[t]+tree[t+1]); // digits
01304                 instr.back() *= SGN(instr.back()) * SGN(x[3*j+3]);
01305                 instr[lenidx] += tree[t+1]-1;
01306             }
01307
01308         instr.push_back(0); // trailing 0
01309         instr[lenidx]++;
01310     }
01311
01312     // push new expression on the stack
01313     s.push(1);
01314     s.push(ID[x]);
01315     s.push(EXTRASYMBOL);
01316     s.push(x[0]); // push hash
01317 }
01318 }
01319
01320 #ifdef DEBUG
01321     MesPrint ("*** [%s, w=%w] DONE: generate_instructions(cse=%d) : numoper=%d",
01322             thetime_str().c_str(), do_CSE?1:0, count_operators(instr));
01323 #endif
01324
01325     return instr;
01326 }
01327
01328 /*
01329 #] generate_instructions :
01330 #[ count_operators_cse :
01331 */
01332
01333
01334 int count_operators_cse (const vector<WORD> &tree) {
01342     //MesPrint ("*** [%s] Starting CSEE", thetime_str().c_str());
01343
01344     typedef unordered_map<vector<WORD>, int, CSEHash, CSEEq> csemap;
01345     csemap ID;
01346
01347     // reserve lots of space, to prevent later rehashes
01348     // TODO: what if this is too large? make a parameter?
01349     ID.rehash(mcts_expr_score * 2);
01350
01351     // s is a stack of operands to process when you encounter operators
01352     // in the postfix expression tree. Operands consist of three WORDs,
01353     // formatted as the LHS/RHS of the keys in ID.
01354     stack<int> s;
01355     int numinstr = 0, numcommas = 0;
01356     vector<WORD> x;
01357
01358     // process the expression tree
01359     for (int i=0; i<(int)tree.size();) {
01360         x.clear();
01361
01362         if (tree[i]==SNUMBER) {
01363             WORD hash = hash_range(&tree[i], tree[i + 1]);
01364             x.push_back(hash);
01365             x.push_back(SNUMBER);
01366             x.insert(x.end(), &tree[i], &tree[i]+tree[i+1]);
01367             int sign = SGN(x.back());
01368             x.back() *= sign;
01369
01370             std::pair<csemap::iterator, bool> suc = ID.insert(csemap::value_type(x, i + 1));
01371             s.push(0);
01372             s.push(sign * suc.first->second);
01373             s.push(SNUMBER);
01374             s.push(hash);
01375             i+=tree[i+1];
01376         }
01377         else if (tree[i]==SYMBOL) {
01378             WORD hash = hash_range(&tree[i], tree[i + 1]);

```



```

01379         if (tree[i+3]>1) {
01380             x.push_back(hash);
01381             x.push_back(SYMBOL);
01382             x.push_back(tree[i+2]+1); // variable (1-indexed)
01383             x.push_back(tree[i+3]); // power
01384
01385             csemap::iterator it = ID.find(x);
01386             if (it != ID.end()) {
01387                 // already-seen power of a symbol
01388                 s.push(1);
01389                 s.push(it->second);
01390                 s.push(EXTRASymbol);
01391             } else {
01392                 if (tree[i + 3] == 2)
01393                     numinstr++;
01394                 else
01395                     numinstr += (int)floor(log(tree[i+3])/log(2.0)) + popcount(tree[i+3]) - 1;
01396
01397                 ID[x] = numinstr;
01398                 s.push(1);
01399                 s.push(numinstr);
01400                 s.push(EXTRASymbol);
01401             }
01402         }
01403     else {
01404         // power of 1
01405         s.push(tree[i+3]); // power
01406         s.push(tree[i+2]+1); // variable (1-indexed)
01407         s.push(SYMBOL);
01408     }
01409
01410     s.push(hash); // push hash
01411
01412     i+=tree[i+1];
01413 }
01414 else { // tree[i]==OPERATOR
01415     int oper = tree[i];
01416     i++;
01417
01418     x.push_back(0); // placeholder for hash
01419     x.push_back(oper);
01420     vector<WORD> hash;
01421     hash.push_back(oper);
01422
01423     // get two operands from the stack
01424     for (int operand=0; operand<2; operand++) {
01425         hash.push_back(s.top()); s.pop();
01426         x.push_back(s.top()); s.pop();
01427         x.push_back(s.top()); s.pop();
01428         x.push_back(s.top()); s.pop();
01429     }
01430
01431     x[0] = hash_range(&hash[0], 3);
01432
01433     // get rid of multiplications by +/-1
01434     if (oper==OPER_MUL) {
01435         bool do_continue = false;
01436
01437         for (int operand=0; operand<2; operand++) {
01438             int idx_oper1 = operand==0 ? 2 : 5;
01439             int idx_oper2 = operand==0 ? 5 : 2;
01440
01441             // check whether operand 1 equals +/-1
01442             if (x[idx_oper1]==SNUMBER) {
01443                 int idx = ABS(x[idx_oper1+1])-1;
01444                 if (tree[idx+2]==1 && tree[idx+3]==1 && ABS(tree[idx+4])==3) {
01445                     // push +/- other operand and continue
01446                     s.push(x[idx_oper2+2]);
01447                     s.push(x[idx_oper2+1]*SGN(x[idx_oper1+1]));
01448                     s.push(x[idx_oper2]);
01449                     s.push(hash[1 + (operand + 1) % 2]);
01450                     do_continue = true;
01451                     break;
01452                 }
01453             }
01454         }
01455     }
01456
01457     if (do_continue) continue;
01458 }
01459
01460 // check whether this subexpression has been seen before
01461 // if not, generate instruction to define it
01462 csemap::iterator it = ID.find(x);
01463 if (it == ID.end()) {
01464     if (numinstr == MAXPOSITIVE) {

```

```

01466         MesPrint((char *)"ERROR: too many temporary variables needed in optimization");
01467         Terminate(-1);
01468     }
01469
01470     if (oper == OPER_COMMA) numcommas++;
01471     ID[x] = ++numinstr;
01472     s.push(1);
01473     s.push(numinstr);
01474     s.push(EXTRASymbol);
01475 } else {
01476     // push new expression on the stack
01477     s.push(1);
01478     s.push(it->second);
01479     s.push(EXTRASymbol);
01480 }
01481
01482     s.push(x[0]); // push hash
01483 }
01484 }
01485
01486 //MesPrint ("*** [%s] Stopping CSEE", thetime_str().c_str());
01487 return numinstr - numcommas;
01488 }
01489
01490 /*
01491 #] count_operators_cse :
01492 #[ count_operators_cse_topdown :
01493 */
01494
01495 typedef struct node {
01496     const WORD* data;
01497     struct node* l;
01498     struct node* r; // TODO: add l,r to data?
01499     WORD sign; // TODO: use data for this?
01500     UWORD hash;
01501
01502     node() : l(NULL), r(NULL), sign(1), hash(0) {};
01503     node(const WORD* data) : data(data), l(NULL), r(NULL), sign(1), hash(0) {};
01504
01505     // a minus sign in the tree should only count as a different entry if
01506     // it is a compound expression: a = -a, but T+V != T+V
01507     int cmp(const struct node* rhs) const {
01508         if (this == rhs) return 0;
01509
01510         if (data[0] != rhs->data[0]) return data[0] < rhs->data[0] ? -1 : 1;
01511         int mod = data[0] == SNUMBER ? -1 : 0; // don't check sign, for numbers
01512         if (data[0] == SYMBOL || data[0] == SNUMBER) {
01513             for (int i = 0; i < data[1] + mod; i++) {
01514                 if (data[i] != rhs->data[i]) return data[i] < rhs->data[i] ? -1 : 1;
01515             }
01516         } else {
01517             int lv = l->cmp(rhs->l);
01518             if (lv != 0) return lv;
01519             int rv = r->cmp(rhs->r);
01520             if (rv != 0) return rv;
01521
01522             // TODO: only for ADD operation
01523             if (l->sign != rhs->l->sign) return l->sign < rhs->l->sign ? -1 : 1;
01524             if (r->sign != rhs->r->sign) return r->sign < rhs->r->sign ? -1 : 1;
01525         }
01526
01527         return 0;
01528     }
01529
01530     // less than operator
01531     bool operator() (const struct node* lhs, const struct node* rhs) const
01532     {
01533         return lhs->cmp(rhs) < 0;
01534     }
01535
01536     void calcHash() {
01537         int mod = data[0] == SNUMBER ? -1 : 0; // don't check sign, for numbers
01538         if (data[0] == SYMBOL || data[0] == SNUMBER) {
01539             hash = hash_range(data, data[1] + mod);
01540         } else {
01541             if (l->hash == 0) l->calcHash();
01542             if (r->hash == 0) r->calcHash();
01543
01544             // signs only matter for compound expressions
01545             size_t newr[] = {(size_t)data[0], l->hash, (size_t)l->sign, r->hash, (size_t)r->sign};
01546             hash = hash_range(newr, 5);
01547         }
01548     }
01549 } NODE;
01550
01551 struct NodeHash {
01552     size_t operator() (const NODE* n) const {

```

```

01553         return n->hash; // already computed
01554     }
01555 };
01556
01557 struct NodeEq {
01558     bool operator()(const NODE* lhs, const NODE* rhs) const {
01559         return lhs->cmp(rhs) == 0;
01560     }
01561 };
01562
01563 NODE* buildTree(vector<WORD> &tree) {
01564     //MesPrint ("*** [%s] Starting CSEE topdown", thetime_str().c_str());
01565
01566     // allocate spaces for the tree, cannot be more nodes than tree size
01567     NODE* ar = (NODE*)Malloca(tree.size() * sizeof(NODE), "CSE tree");
01568     NODE* c = 0;
01569     unsigned int curIndex = 0;
01570
01571     stack<NODE*> st;
01572     for (int i=0; i<(int)tree.size(); i++) {
01573         c = ar + curIndex;
01574         new (c) NODE(&tree[i]); // placement new
01575         curIndex++;
01576
01577         if (tree[i]==SYMBOL || tree[i] == SNUMBER) {
01578             // extract the sign to a new class member
01579             if (tree[i] == SNUMBER) {
01580                 c->sign = SGN(tree[i + tree[i + 1] -1]);
01581             }
01582
01583             c->calcHash();
01584             st.push(c);
01585             i+=tree[i+1];
01586         } else {
01587             c->r = st.top(); st.pop();
01588             c->l = st.top(); st.pop();
01589
01590             // filter *1 and *-1
01591             // TODO: also multiply if there are two numbers?
01592             if (c->data[0] == OPER_MUL) {
01593                 NODE* ch[] = {c->r, c->l};
01594                 for (int j = 0; j < 2; j++)
01595                     if (ch[j]->data[0] == SNUMBER && ch[j]->data[1] == 5 && ch[j]->data[2]==1 &&
01596                         ch[j]->data[3]==1) {
01597                         ch[(j+1)%2]->sign *= ch[j]->sign; // transfer sign
01598                         c = ch[(j+1)%2];
01599                         break;
01600                     }
01601
01602             c->calcHash();
01603             st.push(c);
01604             i++;
01605         }
01606     }
01607
01608     // TODO: reallocate to smaller size? Could save memory
01609     //MesPrint("Memory difference: %d vs %d", curIndex, tree.size());
01610
01611     // we want to make the root of the tree the first element
01612     // so that we can easily free the array.
01613     // we swap the first element with the root
01614     // we need to change the pointer in the operator node that has this element as a child
01615     // TODO: check performance
01616     for (unsigned int i = 0; i < curIndex; i++) {
01617         if (ar[i].l == ar) ar[i].l = st.top();
01618         if (ar[i].r == ar) ar[i].r = st.top();
01619     }
01620     swap(ar[0], *st.top());
01621     return ar;
01622 }
01623
01624 int count_operators_cse_topdown (vector<WORD> &tree) {
01625     typedef unordered_set<NODE*, NodeHash, NodeEq> nodeset;
01626     nodeset ID;
01627
01628     // reserve lots of space, to prevent later rehashes
01629     // TODO: what if this is too large? make a parameter?
01630     ID.rehash(mcts_expr_score * 2);
01631
01632     int numinstr = 0;
01633
01634     NODE* root = buildTree(tree);
01635
01636     stack<NODE*> stack;
01637     stack.push(root);
01638

```

```

01639     while (!stack.empty())
01640     {
01641         NODE* c = stack.top();
01642         stack.pop();
01643
01644         if (c->data[0] == SYMBOL) {
01645             if (c->data[3] > 1) {
01646                 std::pair<nodeset::iterator, bool> suc = ID.insert(c);
01647                 if (suc.second) { // new
01648                     if (c->data[3] == 2)
01649                         numinstr++;
01650                     else
01651                         numinstr += (int)floor(log(c->data[3])/log(2.0)) + popcount(c->data[3]) - 1;
01652                 }
01653             }
01654         } else {
01655             if (c->data[0] != SNUMBER) {
01656                 // operator
01657                 std::pair<nodeset::iterator, bool> suc = ID.insert(c);
01658                 if (suc.second) {
01659                     stack.push(c->r);
01660                     stack.push(c->l);
01661
01662                     // ignore OPER_COMMA
01663                     if (c->data[0] == OPER_MUL || c->data[0] == OPER_ADD)
01664                         numinstr++;
01665                 }
01666             }
01667         }
01668     }
01669
01670     //MesPrint ("*** [%s] Stopping CSEE", thetime_str().c_str());
01671     M_free(root, "CSE tree");
01672
01673     return numinstr;
01674 }
01675
01676 /*
01677 #] count_operators_cse_topdown :
01678 #[ simulated_annealing :
01679 */
01680 vector<WORD> simulated_annealing() {
01681     float minT = AO.Optimize.saMinT.fval;
01682     float maxT = AO.Optimize.saMaxT.fval;
01683     float T = maxT;
01684     float coolrate = pow(minT / maxT, 1 / (float)AO.Optimize.saIter);
01685
01686     GETIDENTITY;
01687
01688     // create a valid state where FACTORSYMBOL/SEPARATESYMBOL remains first
01689     vector<WORD> state = occurrence_order(optimize_expr, false);
01690     int startindex = 0;
01691     if ((state.size() > 0 && state[0] == SEPARATESYMBOL) || (state.size() > 1 && state[1] ==
FACTORSYMBOL)) startindex++;
01692     if (state.size() > 1 && state[1] == FACTORSYMBOL) startindex++;
01693
01694     my_random_shuffle(BHEAD state.begin() + startindex, state.end()); // start from random scheme
01695
01696     vector<WORD> tree = Horner_tree(optimize_expr, state);
01697     int curscore = count_operators_cse_topdown(tree);
01698
01699     // clean poly_vars, that are allocated by Horner_tree
01700     AN.poly_num_vars = 0;
01701     M_free(AN.poly_vars, "poly_vars");
01702
01703     std::vector<WORD> best = state; // best state
01704     int bestscore = curscore;
01705
01706     if (startindex == (int)state.size() || (int)state.size() - startindex < 2) {
01707         return best;
01708     }
01709
01710     for (int o = 0; o < AO.Optimize.saIter; o++) {
01711         int inda = iranf(BHEAD state.size() - startindex) + startindex;
01712         int indb = iranf(BHEAD state.size() - startindex) + startindex;
01713
01714         if (inda == indb) {
01715             continue;
01716         }
01717
01718         swap(state[inda], state[indb]); // swap works best for Horner
01719
01720         vector<WORD> tree = Horner_tree(optimize_expr, state);
01721         int newscore = count_operators_cse_topdown(tree);
01722
01723         // clean poly_vars, that are allocated by Horner_tree
01724         AN.poly_num_vars = 0;

```

```

01725         M_free(AN.poly_vars,"poly_vars");
01726
01727         if (newscore <= curscore || 2.0 * wranf(BHEAD0) / (float)(UWORD)(-1) < exp((curscore -
newscore) / T)) {
01728             curscore = newscore;
01729
01730             if (curscore < bestscore) {
01731                 bestscore = curscore;
01732                 best = state;
01733             }
01734         } else {
01735             swap(state[inda], state[indb]);
01736         }
01737
01738 #ifdef DEBUG_SA
01739     MesPrint("Score at step %d: %d", o, curscore);
01740 #endif
01741     T *= coolrate;
01742 }
01743
01744 #ifdef DEBUG_SA
01745     MesPrint("Simulated annealing score: %d", bestscore);
01746 #endif
01747
01748     return best;
01749 }
01750
01751 /*
01752  #] simulated_annealing :
01753  #[ printpstree :
01754  */
01755
01756 /*
01757 // print MCTS tree with LaTeX/pstricks (for analysis)
01758 void printpstree_rec (tree_node x, string pre="") {
01759
01760     if (x.num_visits==1) {
01761         MesPrint("%s\\TR{%d}",pre.c_str(),x.var);
01762     }
01763     else {
01764         MesPrint("%s\\pstree%s{\\TR{%d}}{",pre.c_str(),
01765                 pre==" "?[nodesep=0, levelsep=40]:"",
01766                 x.var);
01767         for (int i=0; i<(int)x.childs.size(); i++)
01768             if (x.childs[i].num_visits>0)
01769                 printpstree_rec(x.childs[i], pre+" ");
01770         MesPrint("%s",pre.c_str());
01771     }
01772 }
01773
01774 void printpstree () {
01775     // draw tree with pstricks
01776     MesPrint ("\\documentclass{article}");
01777     MesPrint ("\\usepackage{pstricks,pst-node,pst-tree,graphicx}");
01778     MesPrint ("\\begin{document}");
01779     MesPrint ("\\scalebox{0.02}{");
01780     printpstree_rec(mcts_root," ");
01781     MesPrint ("}");
01782     MesPrint ("\\end{document}");
01783 }
01784 */
01785
01786 /*
01787  #] printpstree :
01788  #[ find_Horner_MCTS_expand_tree :
01789  */
01790
01791 /*
01822  #[ next_MCTS_scheme :
01823  */
01824
01825
01826 // find a Horner scheme to be used for the next simulation
01827 inline static void next_MCTS_scheme (PHEAD vector<WORD> *porder, vector<WORD> *pscheme,
vector<tree_node *> *ppath) {
01828
01829     vector<WORD> &order = *porder;
01830     vector<WORD> &schemev = *pscheme;
01831     vector<tree_node *> &path = *ppath;
01832     int depth = 0, nchild0;
01833     float slide_down_factor = 1.0;
01834
01835     order.clear();
01836     path.clear();
01837
01838     // MCTS step I: select
01839     tree_node *select = &mcts_root;
01840     path.push_back(select);

```

```

01841     nchild0 = select->childs.size();
01842     while (select->childs.size() > 0) {
01843         // add virtual loss
01844         select->num_visits++;
01845         select->sum_results+=mcts_expr_score;
01846
01847         //-----
01848         switch ( AO.Optimize.mctsdecaymode ) {
01849             case 1: // Based on http://arxiv.org/abs/arXiv:1312.0841
01850                 slide_down_factor = 1.0-(1.0*AT.optentimes)/(1.0*AO.Optimize.mctsnumexpand);
01851                 break;
01852             case 2: // This gives a bit more cleanup time at the end.
01853                 if ( 2*AT.optentimes < AO.Optimize.mctsnumexpand ) {
01854                     slide_down_factor = 1.0*(AO.Optimize.mctsnumexpand-2*AT.optentimes);
01855                     slide_down_factor /= 1.0*AO.Optimize.mctsnumexpand;
01856                 }
01857                 else {
01858                     slide_down_factor = 0.0001;
01859                 }
01860                 break;
01861             case 3: // depth dependent factor combined with case 1
01862                 float dd = 1.0-(1.0*depth)/(1.0*nchild0);
01863                 slide_down_factor = 1.0-(1.0*AT.optentimes)/(1.0*AO.Optimize.mctsnumexpand);
01864                 if ( dd <= 0.000001 ) slide_down_factor = 1.0;
01865                 else slide_down_factor /= dd;
01866                 if ( slide_down_factor > 1.0 ) slide_down_factor = 1.0;
01867                 break;
01868         }
01869         //-----
01870
01871         #ifdef DEBUG_MCTS
01872             MesPrint("select %d",select->var);
01873         #endif
01874
01875         // find most-promising node
01876         double best=0;
01877         tree_node *next=NULL;
01878         for (vector<tree_node>::iterator p=select->childs.begin(); p<select->childs.end(); p++) {
01879             double score;
01880             if (p->num_visits >= 1) {
01881
01882                 // there are results calculated, so select with the UCT formula
01883                 score = mcts_expr_score / (p->sum_results/p->num_visits) +
01884                     //-----
01885                     slide_down_factor *
01886                     //-----
01887                     2 * AO.Optimize.mctsconstant.fval * sqrt(2*log(select->num_visits) /
01888                         p->num_visits);
01889
01890             #ifdef DEBUG_MCTS
01891                 printf("%d: %.2lf [x=%.2lf n=%d fin=%i]\n",p->var,score,mcts_expr_score /
01892                     (p->sum_results/p->num_visits),
01893                     p->num_visits,p->finished?1:0);
01894                 fflush(stdout);
01895             #endif
01896             }
01897             else {
01898                 // no results yet, so select this node by setting score=infinite
01899                 score = 1e100;
01900             #ifdef DEBUG_MCTS
01901                 printf("%d: inf\n",p->var); fflush(stdout);
01902             #endif
01903             }
01904
01905             // update best candidate
01906             if (!p->finished && score>best) {
01907                 best=score;
01908                 next=&*p;
01909             }
01910
01911             // if no node is found, this node is finished
01912             if (next==NULL) {
01913                 select->finished=true;
01914                 break;
01915             }
01916
01917             // traverse down the tree
01918             select = next;
01919             path.push_back(select);
01920             order.push_back(select->var);
01921             depth++;
01922         }
01923
01924         // MCTS step II: expand
01925

```

```

01926 #ifdef DEBUG_MCTS
01927     MesPrint("expand %d",select->var);
01928 #endif
01929
01930     // variables used so far
01931     set<WORD> var_used;
01932
01933     for (int i=0; i<(int)order.size(); i++)
01934         var_used.insert(ABS(order[i])-1);
01935
01936     // if this a new node, create node and add children
01937     if (!select->finished && select->childs.size()==0) {
01938         tree_node new_node(select->var);
01939         int sign = (order.size() == 0) ? 1 : SGN(order.back());
01940         for (int i=0; i<(int)mcts_vars.size(); i++)
01941             if (!var_used.count(mcts_vars[i])) {
01942                 new_node.chlds.push_back(tree_node(sign*(mcts_vars[i]+1)));
01943                 if (AO.Optimize.hornerdirection==O_FORWARDANDBACKWARD)
01944                     new_node.chlds.push_back(tree_node(-sign*(mcts_vars[i]+1)));
01945             }
01946         my_random_shuffle(BHEAD new_node.chlds.begin(), new_node.chlds.end());
01947
01948         // here locking is necessary, since operator=(tree_node) is a
01949         // non-atomic operation (using pointers makes this lock obsolete)
01950         LOCK(optimize_lock);
01951         *select = new_node;
01952         UNLOCK(optimize_lock);
01953     }
01954     // set finished if necessary
01955     if (select->childs.size()==0)
01956         select->finished = true;
01957
01958     // add virtual loss of number of operators in original expression
01959     select->num_visits++;
01960     select->sum_results+=mcts_expr_score;
01961
01962     // MCTS step III: simulation
01963
01964     // create complete Horner scheme
01965     deque<WORD> scheme;
01966
01967     for (int i=0; i<(int)mcts_vars.size(); i++)
01968         if (!var_used.count(mcts_vars[i]))
01969             scheme.push_back(mcts_vars[i]);
01970     my_random_shuffle(BHEAD scheme.begin(), scheme.end());
01971
01972     for (int i=(int)order.size()-1; i>=0; i--) {
01973         if (order[i] > 0)
01974             scheme.push_front(order[i]-1);
01975         else
01976             scheme.push_back(-order[i]-1);
01977     }
01978
01979     // add FACTORSYMBOL/SEPARATESYMBOL is necessary
01980     if (mcts_factorized)
01981         scheme.push_front(FACTORSYMBOL);
01982     if (mcts_separated)
01983         scheme.push_front(SEPARATESYMBOL);
01984
01985     // Horner scheme as a vector
01986     schemev = vector<WORD>(scheme.begin(), scheme.end());
01987 }
01988
01989 /*
01990 #] next_MCTS_scheme :
01991 #[ try_MCTS_scheme :
01992 */
01993
01994 // count the number of operators in the given Horner scheme
01995 inline static void try_MCTS_scheme (PHEAD const vector<WORD> &scheme, int *pnum_oper) {
01996
01997     // do Horner, CSE and count the number of operators
01998     vector<WORD> tree = Horner_tree(optimize_expr, scheme);
01999     //vector<WORD> instr = generate_instructions(tree, true);
02000     //int num_oper = count_operators(instr);
02001     //int num_oper2 = count_operators_cse(tree);
02002     //int num_oper2 = count_operators_cse_topdown(tree);
02003     //MesPrint("%d %d", num_oper, num_oper2);
02004
02005     int num_oper = count_operators_cse_topdown(tree);
02006
02007     // clean poly_vars, that is allocated by Horner_tree
02008     AN.poly_num_vars = 0;
02009     M_free(AN.poly_vars,"poly_vars");
02010
02011     *pnum_oper = num_oper;

```

```

02013
02014 }
02015
02016 /*
02017     #] try_MCTS_scheme :
02018     #[ update_MCTS_scheme :
02019 */
02020
02021 // update the best score and the search tree
02022 inline static void update_MCTS_scheme (int num_oper, const vector<WORD> &scheme, vector<tree_node *>
    *ppath) {
02023
02024     vector<tree_node *> &path = *ppath;
02025
02026     // update the (global) list of best Horner scheme
02027     if ((int)mcts_best_schemes.size() < AO.Optimize.mctsnumkeep ||
02028         (--mcts_best_schemes.end()->first > num_oper) {
02029         // here locking is necessary, for otherwise best_schemes may
02030         // become corrupted; lock can be prevented if each thread keeps
02031         // track of it's own list and those lists are merged in the end,
02032         // but this seems not useful to implement
02033         LOCK(optimize_lock);
02034         mcts_best_schemes.insert(make_pair(num_oper,scheme));
02035         if ((int)mcts_best_schemes.size() > AO.Optimize.mctsnumkeep)
02036             mcts_best_schemes.erase(--mcts_best_schemes.end());
02037         UNLOCK(optimize_lock);
02038     }
02039
02040     // MCTS step IV: backpropagate
02041
02042     // add number of operator and subtract mcts_expr_score, which
02043     // behaves as a "virtual loss"
02044     for (vector<tree_node *>::iterator p=path.begin(); p<path.end(); p++)
02045         (*p)->sum_results += num_oper - mcts_expr_score;
02046
02047 }
02048
02049 /*
02050     #] update_MCTS_scheme :
02051 */
02052
02053 void find_Horner_MCTS_expand_tree () {
02054
02055     #ifdef DEBUG
02056         MesPrint ("*** [%s, w=%w] CALL: find_Horner_MCTS_expand_tree", thetime_str().c_str());
02057     #endif
02058
02059     GETIDENTITY;
02060
02061     // the order for the Horner scheme up to the selected node, with signs
02062     // indicating forward or backward.
02063     vector<WORD> order;
02064
02065     // complete Horner scheme
02066     vector<WORD> scheme;
02067
02068     // path to the selected node
02069     vector<tree_node *> path;
02070
02071     // the number of operations obtained by the simulation
02072     int num_oper;
02073
02074     next_MCTS_scheme(BHEAD &order, &scheme, &path);
02075     try_MCTS_scheme(BHEAD scheme, &num_oper);
02076     #ifdef DEBUG_MCTS
02077         // Actually "order" is needed only for this debug output.
02078         MesPrint ("[%a] -> [%a] -> %d", order.size(), &order[0], scheme.size(), &scheme[0], num_oper);
02079     #endif
02080     update_MCTS_scheme(num_oper, scheme, &path);
02081
02082     #ifdef DEBUG
02083         MesPrint ("*** [%s, w=%w] DONE: find_Horner_MCTS_expand_tree(%a-> %d)",
02084             thetime_str().c_str(), scheme.size(), &scheme[0], num_oper);
02085     #endif
02086 }
02087
02088
02089 /*
02090     #] find_Horner_MCTS_expand_tree :
02091     #[ PF_find_Horner_MCTS_expand_tree :
02092 */
02093 #ifdef WITHMPI
02094
02095 // To remember which task is assigned to each slave. A task is represented by
02096 // a pair of a Horner scheme and the selected path for the scheme.
02097 // The index range is from 1 to PF.numtasks-1.
02098 vector<pair<vector<WORD>, vector<tree_node *> > > PF_opt_MCTS_tasks;

```



```

02099
02100 // Initialization.
02101 void PF_find_Horner_MCTS_expand_tree_master_init () {
02102
02103     PF_opt_MCTS_tasks.resize(PF.numtasks);
02104     for (int i = 1; i < PF.numtasks; i++) {
02105         pair<vector<WORD>, vector<tree_node *> > &p = PF_opt_MCTS_tasks[i];
02106         p.first.clear();
02107         p.second.clear();
02108     }
02109
02110 }
02111
02112 // Wait for an idle slave and return the process number.
02113 int PF_find_Horner_MCTS_expand_tree_master_next () {
02114
02115     // Find an idle slave.
02116     int next;
02117     PF_Receive(PF_ANY_SOURCE, PF_OPT_MCTS_MSGTAG, &next, NULL);
02118
02119     // Check if the slave had a task.
02120     pair<vector<WORD>, vector<tree_node *> > &p = PF_opt_MCTS_tasks[next];
02121     if (!p.first.empty()) {
02122         // If so, update the result.
02123         int num_oper;
02124         PF_Unpack(&num_oper, 1, PF_INT);
02125         update_MCTS_scheme(num_oper, p.first, &p.second);
02126
02127         // Clear the task.
02128         p.first.clear();
02129         p.second.clear();
02130     }
02131
02132     return next;
02133 }
02134 }
02135
02136 // The main function on the master.
02137 void PF_find_Horner_MCTS_expand_tree_master () {
02138
02139     #ifdef DEBUG
02140         MesPrint ("*** [%s, w=%w] CALL: PF_find_Horner_MCTS_expand_tree_master", thetime_str().c_str());
02141     #endif
02142
02143     vector<WORD> order;
02144     vector<WORD> scheme;
02145     vector<tree_node *> path;
02146
02147     next_MCTS_scheme(BHEAD &order, &scheme, &path);
02148
02149     // Find an idle slave.
02150     int next = PF_find_Horner_MCTS_expand_tree_master_next();
02151
02152     // Send a new task to the slave.
02153     PF_PrepareLongSinglePack();
02154     int len = scheme.size();
02155     PF_LongSinglePack(&len, 1, PF_INT);
02156     PF_LongSinglePack(&scheme[0], len, PF_WORD);
02157     PF_LongSingleSend(next, PF_OPT_MCTS_MSGTAG);
02158
02159     // Remember the task.
02160     pair<vector<WORD>, vector<tree_node *> > &p = PF_opt_MCTS_tasks[next];
02161     p.first = scheme;
02162     p.second = path;
02163
02164     #ifdef DEBUG
02165         MesPrint ("*** [%s, w=%w] DONE: PF_find_Horner_MCTS_expand_tree_master", thetime_str().c_str());
02166     #endif
02167 }
02168 }
02169
02170 // Wait for all the slaves to finish their tasks.
02171 void PF_find_Horner_MCTS_expand_tree_master_wait () {
02172
02173     #ifdef DEBUG
02174         MesPrint ("*** [%s, w=%w] CALL: PF_find_Horner_MCTS_expand_tree_master_wait",
02175             thetime_str().c_str());
02176     #endif
02177
02178     // Wait for all the slaves.
02179     for (int i = 1; i < PF.numtasks; i++) {
02180         int next = PF_find_Horner_MCTS_expand_tree_master_next();
02181         // Send a null task.
02182         PF_PrepareLongSinglePack();
02183         int len = 0;
02184         PF_LongSinglePack(&len, 1, PF_INT);
02185         PF_LongSingleSend(next, PF_OPT_MCTS_MSGTAG);

```

```

02185     }
02186
02187 #ifdef DEBUG
02188     MesPrint ("*** [%s, w=%w] DONE: PF_find_Horner_MCTS_expand_tree_master_wait",
02189             thetime_str().c_str());
02189 #endif
02190
02191 }
02192
02193 // The main function on the slaves.
02194 void PF_find_Horner_MCTS_expand_tree_slave () {
02195
02196 #ifdef DEBUG
02197     MesPrint ("*** [%s, w=%w] CALL: PF_find_Horner_MCTS_expand_tree_slave", thetime_str().c_str());
02198 #endif
02199
02200     vector<WORD> scheme;
02201
02202     {
02203         // Send the first message to the master, which indicates I am idle.
02204         PF_PreparePack();
02205         int dummy = 0;
02206         PF_Pack(&dummy, 1, PF_INT);
02207         PF_Send(MASTER, PF_OPT_MCTS_MSGTAG);
02208     }
02209
02210     for (;;) {
02211         // Get a task from the master.
02212         PF_LongSingleReceive(MASTER, PF_OPT_MCTS_MSGTAG, NULL, NULL);
02213
02214         // Length of the task.
02215         int len;
02216         PF_LongSingleUnpack(&len, 1, PF_INT);
02217
02218         // No task remains.
02219         if (len == 0) break;
02220
02221         // Perform the given task.
02222         scheme.resize(len);
02223         PF_LongSingleUnpack(&scheme[0], len, PF_WORD);
02224         int num_oper;
02225         try_MCTS_scheme(scheme, &num_oper);
02226
02227         // Send the result to the master.
02228         PF_PreparePack();
02229         PF_Pack(&num_oper, 1, PF_INT);
02230         PF_Send(MASTER, PF_OPT_MCTS_MSGTAG);
02231     }
02232
02233 #ifdef DEBUG
02234     MesPrint ("*** [%s, w=%w] DONE: PF_find_Horner_MCTS_expand_tree_slave", thetime_str().c_str());
02235 #endif
02236 }
02237
02238 #endif
02239
02240 /*
02241 #] PF_find_Horner_MCTS_expand_tree :
02242 #[ find_Horner_MCTS :
02243 */
02244
02253 //vector<vector<WORD> > find_Horner_MCTS () {
02254 void find_Horner_MCTS () {
02255
02256 #ifdef WITHMPI
02257     if (PF.me != MASTER) {
02258         if (PF.numtasks <= 1) return;
02259         PF_find_Horner_MCTS_expand_tree_slave();
02260         return;
02261     }
02262 #endif
02263
02264 #ifdef DEBUG
02265     MesPrint ("*** [%s, w=%w] CALL: find_Horner_MCTS", thetime_str().c_str());
02266 #endif
02267
02268     GETIDENTITY;
02269
02270     LONG start_time = TimeWallClock(1);
02271
02272     // initialize the used global variables
02273     mcts_expr_score = count_operators(optimize_expr);
02274     mcts_root = tree_node();
02275
02276     // extract all symbols from the expression
02277     set<WORD> var_set;
02278     for (WORD *t=optimize_expr; *t!=0; t+=*t)

```

```

02279         if (t[1] == SYMBOL)
02280             for (int i=3; i<t[2]; i+=2)
02281                 var_set.insert(t[i]);
02282
02283         // check for factorized/separated expression and make sure that
02284         // FACTORSYMBOL/SEPARATESYMBOL isn't included in the MCTS
02285         mcts_factorized = var_set.count(FACTORSYMBOL);
02286         if (mcts_factorized) var_set.erase(FACTORSYMBOL);
02287         mcts_separated = var_set.count(SEPARATESYMBOL);
02288         if (mcts_separated) var_set.erase(SEPARATESYMBOL);
02289
02290         mcts_vars = vector<WORD>(var_set.begin(), var_set.end());
02291         optimize_num_vars = (int)mcts_vars.size();
02292         // initialize MCTS tree root
02293         for (int i=0; i<(int)mcts_vars.size(); i++) {
02294             if (AO.Optimize.hornerdirection != O_BACKWARD)
02295                 mcts_root.chlds.push_back(tree_node(+ (mcts_vars[i]+1)));
02296             if (AO.Optimize.hornerdirection != O_FORWARD)
02297                 mcts_root.chlds.push_back(tree_node(- (mcts_vars[i]+1)));
02298         }
02299         my_random_shuffle(BHEAD mcts_root.chlds.begin(), mcts_root.chlds.end());
02300
02301         #if defined(WITHMPI)
02302         PF_find_Horner_MCTS_expand_tree_master_init();
02303         #endif
02304
02305         // initialize a potential variable mctsconstant scheme.
02306         AT.optentimes = 0;
02307
02308         // call expand_tree until it is called "mctsnunexpand" times, the
02309         // time limit is reached or the tree is fully finished
02310         for (int times=0; times<AO.Optimize.mctsnunexpand && !mcts_root.finished &&
02311             (AO.Optimize.mctsttimelimit==0 || (TimeWallClock(1)-start_time)/100 <
02312             AO.Optimize.mctsttimelimit);
02313             times++) {
02314             AT.optentimes = times;
02315             // call expand_tree routine depending on threading mode
02316             #if defined(WITHPTHREADS)
02317             if (AM.totalnumberofthreads > 1)
02318                 find_Horner_MCTS_expand_tree_threaded();
02319             #elif defined(WITHMPI)
02320             if (PF.numtasks > 1)
02321                 PF_find_Horner_MCTS_expand_tree_master();
02322             else
02323                 #endif
02324                 find_Horner_MCTS_expand_tree();
02325             }
02326
02327             // if TForm or ParForm, wait for everyone to finish
02328             #ifdef WITHPTHREADS
02329             MasterWaitAll();
02330             #endif
02331             #ifdef WITHMPI
02332             PF_find_Horner_MCTS_expand_tree_master_wait();
02333             #endif
02334             #ifdef DEBUG
02335             MesPrint ("*** [%s, w=%w] DONE: find_Horner_MCTS", thetime_str().c_str());
02336             #endif
02337         }
02338     }
02339
02340     /*
02341     #] find_Horner_MCTS :
02342     #[ merge_operators :
02343     */
02344
02402     vector<WORD> merge_operators (const vector<WORD> &all_instr, bool move_coeff) {
02403
02404         #ifdef DEBUG_MORE
02405         MesPrint ("*** [%s, w=%w] CALL: merge_operators", thetime_str().c_str());
02406         #endif
02407
02408         // get starting positions of instructions
02409         vector<const WORD *> instr;
02410         const WORD *tbegin = &all_instr.begin();
02411         const WORD *tend = tbegin+all_instr.size();
02412         // copy all instructions to temp space. There will be n of them in instr.
02413         for (const WORD *t=tbegin; t!=tend; t+=*(t+2)) {
02414             instr.push_back(t);
02415         }
02416         int n = instr.size();
02417
02418         // find parents and number of parents of instructions
02419         vector<int> par(n, numpar(n,0));
02420         for (int i=0; i<n; i++) par[i]=i;
02421

```

```

02422     for (int i=0; i<n; i++) {
02423         for (const WORD *t=instr[i]+OPTHEAD; *t!=0; t+=*t) {
02424             if ( (*t+1)==EXTRASymbol && *t!=1+ABS(*(t+*t-1)) ) {
02425                 // extra symbol t[3] is referred to in instr i.
02426                 par[*t+3]=i;
02427                 numpar[*t+3]++;
02428
02429                 // if coefficient isn't +/-1 or power > 1, increase numpar,
02430                 // so that this is not merged
02431                 if (*(t+*t-3)!=1 || *(t+*t-2)!=1 || ABS(*(t+*t-1))!=3) numpar[*t+3]++;
02432                 if (*(t+4)>1) numpar[*t+3]++;
02433             }
02434         }
02435     }
02436     // determine which instructions to merge
02437     stack<int> s;
02438     s.push(n-1);
02439     vector<bool> vis(n,false);
02440
02441     while (!s.empty()) {
02442
02443         int i=s.top(); s.pop();
02444         if (vis[i]) continue;
02445         vis[i]=true;
02446
02447         for (const WORD *t=instr[i]+OPTHEAD; *t!=0; t+=*t)
02448             if ( (*t+1)==EXTRASymbol && *t!=1+ABS(*(t+*t-1)) )
02449                 s.push(*t+3);
02450
02451         // condition: one parent and equal operator as parent
02452         if (numpar[i]==1 && *(instr[i+1])==*(instr[par[i]+1]))
02453             par[i] = par[par[i]]; // The expr into which we subst par[i] to get i
02454         else
02455             par[i] = i;
02456     }
02457
02458     // merge instructions into new instructions
02459     vector<WORD> newinstr;
02460
02461     // stack of new expressions, all 0-indexed
02462     stack<WORD> new_expr;
02463     new_expr.push(n-1);
02464     vis = vector<bool>(n,false);
02465
02466     // skip empty equations (might be introduced by greedy optimizations)
02467     vector<bool> skip(n,false), skipcoeff(n,false);
02468     for (int i=0; i<n; i++)
02469         if (*(instr[i]+OPTHEAD) == 0) skip[i]=skipcoeff[i]=true;
02470
02471     // for renumbering merged parents
02472     vector<int> renum_par(n);
02473     for (int i=0; i<n; i++)
02474         renum_par[i]=i;
02475
02476     while (!new_expr.empty()) {
02477         int x = new_expr.top(); new_expr.pop();
02478         if (vis[x]) continue;
02479         vis[x] = true;
02480
02481         // find all instructions with parent=x and copy their arguments
02482         // into a new expression
02483         bool first_copy=true;
02484         int lenidx = newinstr.size()+2;
02485
02486         // 1-indexed, since signs may occur
02487         stack<WORD> this_expr;
02488         this_expr.push(x+1);
02489
02490         while (!this_expr.empty()) {
02491             // pop from stack, determine expr.nr and sign
02492             int i = this_expr.top(); this_expr.pop();
02493             int sign = SGN(i);
02494             i = ABS(i)-1;
02495             for (const WORD *t=instr[i]+OPTHEAD; *t!=0; t+=*t) { // terms in i
02496                 // don't copy a term if:
02497                 // (1) skip=true, since then it's merged into the parent
02498                 // (2) extrasymbol with parent=x, because its children should be copied
02499                 // (3) coefficient with skipcoeff=true, since it's already copied
02500                 bool copy = !skip[i];
02501                 if (*t!=1+ABS(*(t+*t-1)) && *(t+1)==EXTRASymbol) {
02502                     if (par[*t+3] == x) {
02503                         // parent of term refers to x. we push it with its sign if no skip is true
02504                         // and the sign of the expr.
02505                         this_expr.push(sign * (skip[i]||skipcoeff[i] ? 1 : SGN(*(t+*t-1))) *
(1+*(t+3)));
02506                     }
02507                     if (*(instr[i+1]) == OPER_MUL) sign=1;
02508                     copy=false;

```

```

02508         }
02509     else {
02510         new_expr.push(*(t+3));
02511     }
02512 }
02513
02514 if (*t == 1+ABS(*(t+*t-1)) && skipcoeff[i]) {
02515     copy=false;
02516 }
02517
02518 if (copy) {
02519     // first term, so add header
02520     if (first_copy) {
02521         newinstr.push_back(renum_par[x]); // expr.nr.
02522         newinstr.push_back(*(instr[x]+1)); // operator
02523         newinstr.push_back(3); // length OPTHEAD?
02524         first_copy=false;
02525     }
02526
02527     // copy term and adjust sign
02528     int thislenidx = newinstr.size();
02529     newinstr.insert(newinstr.end(), t, t+*t); // Put the whole term in newinstr
02530     newinstr.back() *= sign;
02531     if (*(instr[i]+1) == OPER_MUL) sign=1;
02532     newinstr[lenidx] += *t;
02533
02534     // check for moving coefficients up
02535     // necessary condition: MUL-expression with 1 parent
02536     if (move_coeff && *t!=1+ABS(*(t+*t-1)) && *(instr[i]+1)!=OPER_COMMA &&
02537         *(t+1)==EXTRASYNBOL && numpar[*(t+3)]=1 && *(instr[*(t+3)+1]==OPER_MUL)
02538 {
02539
02539     // coefficient is always the first term (that's how Horner+generate works)
02540     const WORD *t1 = instr[*(t+3)]+OPTHEAD;
02541     const WORD *t2 = t1+*t1;
02542
02543     if (*t1 == 1+ABS(*(t1+*t1-1))) {
02544         // t1 pointer to a coefficient, so move it
02545
02546         // remove old coefficient of 1
02547         WORD *t3 = &newinstr.end(); //
02548         int sign2 = SGN(t3[-1]); //
02549         newinstr.erase(newinstr.end()-3, newinstr.end());
02550         // count number of arguments; iff it is 2 move the (extra)symbol too
02551         int numargs=0;
02552         for (const WORD *tt=t1; *tt!=0; tt+=*tt) {
02553             numargs++;
02554         }
02555         if (numargs==2 && *(t2+4)==1) {
02556             // replace (extra)symbol
02557             newinstr[newinstr.size()-4] = *(t2+1);
02558             newinstr[newinstr.size()-3] = *(t2+2);
02559             newinstr[newinstr.size()-2] = *(t2+3);
02560             newinstr[newinstr.size()-1] = *(t2+4);
02561             sign2 *= SGN(*(t2+*t2-1)); // was t2[7]
02562
02563             // ignore this expression from now on
02564             skip[*(t+3)]=true;
02565             if (*(t2+1)==EXTRASYNBOL)
02566                 renum_par[*(t+3)] = *(t2+3);
02567         }
02568         else {
02569             // otherwise, ignore coefficient from now on
02570             // we need to collect the signs of the terms
02571             // first and set them to one. This was forgotten
02572             // before. Gave occasional errors.
02573             if ( numargs > 2 || ( numargs == 2 && t2[4] > 1 ) ) {
02574                 for (WORD *tt=(WORD *)t2; *tt!=0; tt+=*tt) {
02575                     if ( tt[*tt-1] < 0 ) {
02576                         tt[*tt-1] = -tt[*tt-1];
02577                         sign2 = -sign2;
02578                     }
02579                 }
02580             }
02581             skipcoeff[*(t+3)]=true;
02582         }
02583
02584         // add new coefficient
02585         newinstr.insert(newinstr.end(), t1+1, t1+*t1);
02586         newinstr.back() *= sign2;
02587         newinstr[thislenidx] += ABS(newinstr.back()) - 3;
02588         newinstr[lenidx] += ABS(newinstr.back()) - 3;
02589     }
02590 }
02591 }
02592 }
02593 }

```

```

02594
02595     // if something has been copied, add trailing zero
02596     if (!first_copy) {
02597         newinstr.push_back(0);
02598         newinstr[lenidx]++;
02599     }
02600 }
02601
02602 // renumber the expressions to 0,1,2,...; only keep expressions with
02603 // skip=false which are their own parent after a renumbering in case
02604 // of moved coefficients
02605
02606 // find renumber scheme
02607 vector<int> renum(n,-1);
02608 int next=0;
02609 for (int i=0; i<n; i++)
02610     if (!skip[i] && renum_par[par[i]]==i) renum[renum_par[i]]=next++;
02611
02612 // find new instruction index
02613 tbegin = &*newinstr.begin();
02614 tend = tbegin+newinstr.size();
02615 for (const WORD *t=tbegin; t!=tend; t+=(t+2))
02616     instr[renum[*t]] = t;
02617
02618 // renumbering expressions and sort them lexicographically (in
02619 // new_instr they are in the preorder of the expression tree)
02620 vector<WORD> sortinstr;
02621 for (int i=0; i<next; i++) {
02622     int idx = sortinstr.size();
02623     sortinstr.insert(sortinstr.end(), instr[i], instr[i]+(instr[i]+2));
02624
02625     sortinstr[idx] = renum[sortinstr[idx]];
02626
02627     // renumber content of an expression
02628     for (WORD *t2=&sortinstr[idx]+3; *t2!=0; t2+=*t2)
02629         if ((*t2!=1+ABS(*t2+*t2-1)) && *(t2+1)==EXTRASYMBOL)
02630             *(t2+3) = renum[*t2+3];
02631 }
02632
02633 #ifdef DEBUG_MORE
02634     MesPrint ("*** [%s, w=%w] DONE: merge_operators", thetime_str().c_str());
02635 #endif
02636
02637     return sortinstr;
02638 }
02639
02640 /*
02641     #] merge_operators :
02642     #[ class Optimization :
02643 */
02644
02669 class optimization {
02670 public:
02671     int type, arg1, arg2, improve;
02672     vector<WORD> coeff;
02673     vector<int> eqnidxs;
02674
02675     bool operator< (const optimization &a) const {
02676         if (arg1 != a.arg1) return arg1 < a.arg1;
02677         if (arg2 != a.arg2) return arg2 < a.arg2;
02678         if (type != a.type) return type < a.type;
02679         return coeff < a.coeff;
02680     }
02681 };
02682
02683 /*
02684     #] class Optimization :
02685     #[ find_optimizations :
02686 */
02687
02697 vector<optimization> find_optimizations (const vector<WORD> &instr) {
02698
02699     #ifdef DEBUG_GREEDY
02700         MesPrint ("*** [%s, w=%w] CALL: find_optimizations", thetime_str().c_str());
02701     #endif
02702
02703     // #[ Startup :
02704     // the resulting vector of optimizations
02705     vector<optimization> res;
02706
02707     // a map to count the improvement of an optimization; the
02708     // improvement is stored as a vector<int> with equation numbers
02709     map<optimization, vector<int> > cnt;
02710
02711     // a map to identify coefficients
02712     map<vector<int>,int> idx_coeff;
02713

```

```

02714     const WORD *ebegin = &*instr.begin();
02715     const WORD *eend = ebegin+instr.size();
02716
02717     for (int optim_type=0; optim_type<=4; optim_type++) {
02718
02719         cnt.clear();
02720         idx_coeff.clear();
02721
02722         optimization optim;
02723         optim.type = optim_type;
02724         // #] Startup :
02725
02726         // #[ type 0 : find optimizations of the form z=x^n (optim.type==0)
02727         if (optim_type == 0) {
02728             for (const WORD *e=ebegin; e!=eend; e+=*(e+2)) {
02729                 if (*(e+1) != OPER_MUL) continue;
02730                 for (const WORD *t=e+3; *t!=0; t+=*t) {
02731                     if (*t == ABS(*(t+*t-1))+1) continue;
02732                     if (*(t+4) > 1) {
02733                         optim.arg1 = (*(t+1)==SYMBOL ? 1 : -1) * (*(t+3) + 1);
02734                         optim.arg2 = *(t+4);
02735                         cnt[optim].push_back(e-ebegin);
02736                     }
02737                 }
02738             }
02739         }
02740         // #] type 0 :
02741         // #[ type 1 : find optimizations of the form z=x*y (optim.type==1)
02742         if (optim_type == 1) {
02743             for (const WORD *e=ebegin; e!=eend; e+=*(e+2)) {
02744                 if (*(e+1) != OPER_MUL) continue;
02745
02746                 for (const WORD *t1=e+3; *t1!=0; t1+=*t1) {
02747                     if (*t1 == ABS(*(t1+*t1-1))+1) continue;
02748                     int x1 = (*(t1+1)==SYMBOL ? 1 : -1) * (*(t1+3) + 1);
02749
02750                     for (const WORD *t2=t1+*t1; *t2!=0; t2+=*t2) {
02751                         if (*t2 == ABS(*(t2+*t2-1))+1) continue;
02752                         int x2 = (*(t2+1)==SYMBOL ? 1 : -1) * (*(t2+3) + 1);
02753
02754                         if (*(t1+4) == *(t2+4)) {
02755                             optim.arg1 = x1;
02756                             optim.arg2 = x2;
02757                             if (optim.arg1 > optim.arg2)
02758                                 swap(optim.arg1, optim.arg2);
02759
02760                             if (*(t1+4) == 1)
02761                                 cnt[optim].push_back(e-ebegin);
02762                             else {
02763                                 // E=x^n*y^n -> z=x*y; E=z^n is double improvement
02764                                 cnt[optim].push_back(e-ebegin);
02765                                 cnt[optim].push_back(e-ebegin);
02766                             }
02767                         }
02768                     }
02769                 }
02770             }
02771         }
02772         // #] type 1 :
02773         // #[ type 2 : find optimizations of the form z=c*x (optim.type==2)
02774         if (optim_type == 2) {
02775             for (const WORD *e=ebegin; e!=eend; e+=*(e+2)) {
02776
02777                 if (*(e+1) == OPER_ADD) {
02778                     // in ADD-equation
02779
02780                     for (const WORD *t=e+3; *t!=0; t+=*t) {
02781                         if (*t == ABS(*(t+*t-1))+1) continue;
02782
02783                         if (*(t+4)==1) {
02784                             if (ABS(*(t+*t-1))==3 && *(t+*t-2)==1 && *(t+*t-3)==1) continue;
02785
02786                             optim.coeff = vector<WORD>(t+*t-ABS(*(t+*t-1)), t+*t);
02787                             optim.coeff.back() = ABS(optim.coeff.back());
02788
02789                             optim.arg1 = (*(t+1)==SYMBOL ? 1 : -1) * (*(t+3) + 1);
02790                             optim.arg2 = 0;
02791
02792                             cnt[optim].push_back(e-ebegin);
02793                         }
02794                     }
02795                 }
02796                 else if (*(e+1) == OPER_MUL) {
02797                     // in MUL-equation
02798                     optim.coeff.clear();
02799
02800                     for (const WORD *t=e+3; *t!=0; t+=*t)

```

```

02801         if (*t == ABS(*(t+*t-1))+1) {
02802             if (ABS(*(t+*t-1))==3 && *(t+*t-2)==1 && *(t+*t-3)==1) continue;
02803             optim.coeff = vector<WORD>(t+*t-ABS(*(t+*t-1)), t+*t);
02804             optim.coeff.back() = ABS(optim.coeff.back());
02805         }
02806
02807         if (!optim.coeff.empty())
02808             for (const WORD *t=e+3; *t!=0; t+=*t) {
02809                 if (*t == ABS(*(t+*t-1))+1) continue;
02810                 if (*(t+4) != 1) continue;
02811                 optim.arg1 = (*(t+1)==SYMBOL ? 1 : -1) * (*(t+3) + 1);
02812                 optim.arg2 = 0;
02813                 cnt[optim].push_back(e-ebegin);
02814             }
02815     }
02816 }
02817 }
02818 // #] type 2 :
02819 // #[ type 3 : find optimizations of the form z=x+c (optim.type==3)
02820 if (optim_type == 3) {
02821     for (const WORD *e=ebegin; e!=eend; e+=*(e+2)) {
02822
02823         if (*(e+1) != OPER_ADD) continue;
02824
02825         optim.coeff.clear();
02826
02827         for (const WORD *t=e+3; *t!=0; t+=*t)
02828             if (*t == ABS(*(t+*t-1))+1) {
02829                 optim.coeff = vector<WORD>(t+*t-ABS(*(t+*t-1)), t+*t);
02830             }
02831
02832         if (!optim.coeff.empty()) {
02833             for (const WORD *t=e+3; *t!=0; t+=*t) {
02834                 if (*t == ABS(*(t+*t-1))+1) continue;
02835                 if (ABS(*(t+*t-1))!=3 || *(t+*t-2)!=1 || *(t+*t-3)!=1) continue;
02836                 if (*(t+*t-1)==-3) optim.coeff.back() *= -1;
02837
02838                 optim.arg1 = (*(t+1)==SYMBOL ? 1 : -1) * (*(t+3) + 1);
02839                 optim.arg2 = 0;
02840                 cnt[optim].push_back(e-ebegin);
02841                 if (*(t+*t-1)==-3) optim.coeff.back() *= -1;
02842             }
02843         }
02844     }
02845 }
02846 // #] type 3 :
02847 // #[ type 4,5 : find optimizations of the form z=x+y or z=x-y (optim.type==4 or 5)
02848 if (optim_type == 4) {
02849     for (const WORD *e=ebegin; e!=eend; e+=*(e+2)) {
02850         if (*(e+1) != OPER_ADD) continue;
02851
02852         for (const WORD *t1=e+3; *t1!=0; t1+=*t1) {
02853             if (*t1 == ABS(*(t1+*t1-1))+1) continue;
02854             int x1 = (*(t1+1)==SYMBOL ? 1 : -1) * (*(t1+3) + 1);
02855
02856             for (const WORD *t2=t1+*t1; *t2!=0; t2+=*t2) {
02857                 if (*t2 == ABS(*(t2+*t2-1))+1) continue;
02858                 int x2 = (*(t2+1)==SYMBOL ? 1 : -1) * (*(t2+3) + 1);
02859
02860                 int sign1 = SGN(*(t1+*t1-1));
02861                 int sign2 = SGN(*(t2+*t2-1));
02862
02863                 if (BigLong((UWORD *)t1+5, ABS(*(t1+*t1-1))-1, (UWORD *)t2+5,
02864                     ABS(*(t2+*t2-1))-1) == 0) {
02865
02866                     optim.type = (sign1 * sign2 == 1 ? 4 : 5); // optimization type
02867                     optim.arg1 = x1;
02868                     optim.arg2 = x2;
02869                     if (optim.arg1 > optim.arg2) {
02870                         swap(optim.arg1, optim.arg2);
02871                     }
02872
02873                     if (ABS(*(t1+*t1-1))==3 && *(t1+*t1-2)==1 && *(t1+*t1-3)==1)
02874                         cnt[optim].push_back(e-ebegin);
02875                     else {
02876                         // E=2x+2y -> z=x+y; E=2z is improvement bby itself
02877                         cnt[optim].push_back(e-ebegin);
02878                         cnt[optim].push_back(e-ebegin);
02879                     }
02880                 }
02881             }
02882         }
02883     }
02884 // #] type 4,5 :
02885 // #[ add :
02886

```



```

02887         // add optimizations with positive improvement to the result
02888         for (map<optimization, vector<int> >::iterator i=cnt.begin(); i!=cnt.end(); i++) {
02889             int improve = i->second.size() - 1;
02890             if (improve > 0) {
02891                 res.push_back(i->first);
02892                 res.back().improve = improve;
02893                 res.back().eqnidxs = i->second;
02894             }
02895             // remove duplicates, that were add to get the correct improvement
02896             res.back().eqnidxs.erase(unique(res.back().eqnidxs.begin(), res.back().eqnidxs.end()),
res.back().eqnidxs.end());
02897         }
02898     }
02899 }
02900 // #] add :
02901
02902 #ifdef DEBUG_GREEDY
02903     MesPrint ("*** [%s, w=%w] DONE: find_optimizations", thetime_str().c_str());
02904 #endif
02905
02906     return res;
02907 }
02908
02909 /*
02910     #] find_optimizations :
02911     #[ do_optimization :
02912 */
02913
02930 bool do_optimization (const optimization optim, vector<WORD> &instr, int newid) {
02931
02932     // #[ Debug code :
02933     #ifdef DEBUG_GREEDY
02934         if (optim.type==0)
02935             MesPrint ("*** [%s, w=%w] CALL: do_optimization(improve=%d, %c%d^%d)",
thetime_str().c_str(), optim.improve,
optim.arg1>0?'x':'Z', ABS(optim.arg1)-1, optim.arg2);
02936         else if (optim.type==1 || optim.type==4)
02937             MesPrint ("*** [%s, w=%w] CALL: do_optimization(improve=%d, %c%d%c%c%d)",
thetime_str().c_str(), optim.improve,
optim.arg1>0?'x':'Z', ABS(optim.arg1)-1,
optim.type==1 ? '+' : optim.type==4 ? '+' : '-',
optim.arg2>0?'x':'Z', ABS(optim.arg2)-1);
02938         else {
02939             WORD n = optim.coeff.back()/2;
02940             UBYTE num[BITINWORD*ABS(n)], den[BITINWORD*ABS(n)];
02941             PrtLong((UWORD *)&optim.coeff[0], n, num);
02942             PrtLong((UWORD *)&optim.coeff[ABS(n)], ABS(n), den);
02943             MesPrint ("*** [%s, w=%w] CALL: do_optimization(improve=%d, %c%d%c%s/%s)",
thetime_str().c_str(), optim.improve,
optim.arg1>0?'x':'Z', ABS(optim.arg1)-1,
optim.type==2 ? '+' : '+' , num, den);
02944         }
02945     #endif
02946     // #] Debug code :
02947
02948     bool substituted = false;
02949     WORD *ebegin = &instr.begin();
02950
02951     // #[ type 0 : substitution of the form z=x^n (optim.type==0)
02952     if (optim.type == 0) {
02953
02954         int vartypex = optim.arg1>0 ? SYMBOL : EXTRASYMBOL;
02955         int varnumx = ABS(optim.arg1) - 1;
02956         int n = optim.arg2;
02957
02958         for (int i=0; i<(int)optim.eqnidxs.size(); i++) {
02959             WORD *e = ebegin + optim.eqnidxs[i];
02960             if (*(e+1) != OPER_MUL) continue;
02961
02962             // scan through equation
02963             for (WORD *t=e+3; *t!=0; t+=*t) {
02964                 if (*t == ABS(*(t+*t-1))+1) continue;
02965                 if (*(t+1)==vartypex &&
*(t+3)==varnumx &&
*(t+4) % n == 0) {
02966
02967                     // substitute
02968                     *(t+1) = EXTRASYMBOL;
02969                     *(t+3) = newid;
02970                     *(t+4) /= n;
02971
02972                     substituted = true;
02973                 }
02974             }
02975         }
02976     }
02977
02978     if (!substituted) {

```

```

02989 #ifdef DEBUG_GREEDY
02990     MesPrint ("*** [%s, w=%w] DONE: do_optimization : res=false", thetime_str().c_str(),
optim.improve);
02991 #endif
02992     return false;
02993 }
02994
02995 // add extra equation (Tnew = x^n)
02996 instr.push_back(newid); // eqn.nr
02997 instr.push_back(OPER_MUL); // operator
02998 instr.push_back(12); // total length
02999 instr.push_back(8); // term length
03000 instr.push_back(vartypex); // (extra)symbol
03001 instr.push_back(4); // symbol length
03002 instr.push_back(varnumx); // symbol id
03003 instr.push_back(n); // power
03004 instr.push_back(1);
03005 instr.push_back(1); // coefficient 1
03006 instr.push_back(3);
03007 instr.push_back(0); // trailing 0
03008 }
03009 // #] type 0 :
03010 // #[ type 1 : substitution of the form z=x*y (optim.type==1)
03011 if (optim.type == 1) {
03012
03013     int vartypex = optim.arg1>0 ? SYMBOL : EXTRASYMBOL;
03014     int varnumx = ABS(optim.arg1) - 1;
03015     int vartypey = optim.arg2>0 ? SYMBOL : EXTRASYMBOL;
03016     int varnumy = ABS(optim.arg2) - 1;
03017
03018     for (int i=0; i<(int)optim.eqnidxs.size(); i++) {
03019         WORD *e = ebegin + optim.eqnidxs[i];
03020         if (*(e+1) != OPER_MUL) continue;
03021
03022         // scan through equation
03023         int powx=0, powy=0;
03024         for (WORD *t=e+3; *t!=0; t+=*t) {
03025             if (*t == ABS(*(t+*t-1))+1) continue;
03026             if (*(t+1)==vartypex && *(t+3)==varnumx) powx = *(t+4);
03027             if (*(t+1)==vartypey && *(t+3)==varnumy) powy = *(t+4);
03028         }
03029
03030         // substitute if found
03031         if (powx>0 && powy>0 && powx==powy) {
03032
03033             WORD sign = 1;
03034             WORD *newt = e+3;
03035
03036             for (WORD *t=e+3; *t!=0;) {
03037                 int dt=*t;
03038
03039                 if (*t == ABS(*(t+*t-1))+1 ||
03040                     (!(*(t+1)==vartypex && *(t+3)==varnumx) &&
03041                      !(*(t+1)==vartypey && *(t+3)==varnumy))) {
03042                     memmove(newt, t, *t*sizeof(WORD));
03043                     newt += dt;
03044                 }
03045                 else {
03046                     sign *= SGN(*(t+*t-1));
03047                 }
03048
03049                 t+=dt;
03050             }
03051
03052             *newt++ = 8; // term length
03053             *newt++ = EXTRASYMBOL; // extrasymbol
03054             *newt++ = 4; // symbol length
03055             *newt++ = newid; // symbol id
03056             *newt++ = powx; // power
03057             *newt++ = 1;
03058             *newt++ = 1; // coefficient +/-1
03059             *newt++ = 3*sign;
03060             *newt++ = 0; // trailing 0
03061
03062             substituted = true;
03063         }
03064     }
03065
03066     if (!substituted) {
03067 #ifdef DEBUG_GREEDY
03068         MesPrint ("*** [%s, w=%w] DONE: do_optimization : res=false", thetime_str().c_str(),
optim.improve);
03069 #endif
03070         return false;
03071     }
03072
03073     // add extra equation (Tnew = x*y)

```

```

03074      instr.push_back(newid);      // eqn.nr
03075      instr.push_back(OPER_MUL);   // operator
03076      instr.push_back(20);         // total length
03077      instr.push_back(8);          // LHS length
03078      instr.push_back(vartypex);   // (extra)symbol
03079      instr.push_back(4);          // symbol length
03080      instr.push_back(varnumx);    // symbol id
03081      instr.push_back(1);          // power 1
03082      instr.push_back(1);
03083      instr.push_back(1);          // coefficient 1
03084      instr.push_back(3);
03085      instr.push_back(8);          // RHS length
03086      instr.push_back(vartypey);   // (extra)symbol
03087      instr.push_back(4);          // symbol length
03088      instr.push_back(varnumy);    // symbol id
03089      instr.push_back(1);          // power 1
03090      instr.push_back(1);
03091      instr.push_back(1);          // coefficient 1
03092      instr.push_back(3);
03093      instr.push_back(0);          // trailing 0
03094  }
03095  // #] type 1 :
03096  // #[ type 2 : substitution of the form z=c*x (optim.type==2)
03097
03098  if (optim.type == 2) {
03099      int vartype = optim.arg1>0 ? SYMBOL : EXTRASYMBOL;
03100      int varnum = ABS(optim.arg1) - 1;
03101
03102      WORD ncoeff = optim.coeff.back();
03103
03104      // scan through equations
03105      for (int i=0; i<(int)optim.eqnidxs.size(); i++) {
03106          WORD *e = ebegin + optim.eqnidxs[i];
03107
03108          if (*(e+1) == OPER_ADD) {
03109              // scan through ADD-equation
03110              for (WORD *t=e+3; *t!=0; t+=*t) {
03111
03112                  if (*t == ABS(*(t+*t-1))+1) continue;
03113
03114                  if (*(t+1)==vartype && ABS(*(t+3))==varnum && *(t+4)==1 &&
03115                      BigLong((UWORD *) &optim.coeff[0], ncoeff-1,
03116                              (UWORD *) t+*t-ABS(*(t+*t-1)), ABS(*(t+*t-1))-1) == 0) {
03117                      // substitute
03118
03119                      int sign = SGN(*(t+*t-1));
03120
03121                      WORD *tend = t;
03122                      while (*tend!=0) tend+=*tend;
03123                      WORD nmove = tend - t - *t;
03124                      memmove(t, t+*t, nmove*sizeof(WORD));
03125                      t += nmove;
03126
03127                      *t++ = 8;          // term length
03128                      *t++ = EXTRASYMBOL; // (extra)symbol
03129                      *t++ = 4;          // symbol length
03130                      *t++ = newid;      // symbol id
03131                      *t++ = 1;          // power of 1
03132                      *t++ = 1;
03133                      *t++ = 1;          // coefficient of +/-1
03134                      *t++ = 3 * sign;
03135                      *t++ = 0;          // trailing 0
03136
03137                      substituted = true;
03138                      break;
03139                  }
03140              }
03141          }
03142          else if (*(e+1) == OPER_MUL) {
03143
03144              bool coeff_match=false, var_match=false;
03145              int sign = 1;
03146
03147              // scan through MUL-equation
03148              for (WORD *t=e+3; *t!=0; t+=*t) {
03149                  if (*t == ABS(*(t+*t-1))+1 && BigLong((UWORD *) &optim.coeff[0], ncoeff-1,
03150                      *(t+*t-ABS(*(t+*t-1))), ABS(*(t+*t-1))-1) == 0) {
03151                      coeff_match = true;
03152                      sign *= SGN(*(t+*t-1));
03153                  }
03154                  else if (*(t+1)==vartype && ABS(*(t+3))==varnum && *(t+4)==1) {
03155                      var_match = true;
03156                      sign *= SGN(*(t+*t-1));
03157                  }
03158              }
03159          }

```

```

03160         // substitute if found
03161         if (coeff_match && var_match) {
03162
03163             WORD *newt = e+3;
03164
03165             for (WORD *t=e+3; *t!=0;) {
03166
03167                 int dt=*t;
03168
03169                 if (*t!=ABS(*(t+*t-1))+1 && !(*t+1)==vartype && ABS(*(t+3))==varnum &&
* (t+4)==1) ) {
03170                     memmove(newt, t, dt*sizeof(WORD));
03171                     newt += dt;
03172                 }
03173
03174                 t+=dt;
03175             }
03176
03177             *newt++ = 8;           // term length
03178             *newt++ = EXTRASYMBOL; // extrasymbol
03179             *newt++ = 4;           // symbol length
03180             *newt++ = newid;       // symbol id
03181             *newt++ = 1;           // power of 1
03182             *newt++ = 1;
03183             *newt++ = 1;           // coefficient of +/-1
03184             *newt++ = 3 * sign;
03185             *newt++ = 0;           // trailing 0
03186
03187             substituted = true;
03188         }
03189     }
03190 }
03191
03192     if (!substituted) {
03193 #ifdef DEBUG_GREEDY
03194         MesPrint ("*** [%s, w=%w] DONE: do_optimization : res=false", thetime_str().c_str(),
optim.improve);
03195 #endif
03196         return false;
03197     }
03198
03199     // add extra equation (Tnew = c*y)
03200     instr.push_back(newid); // eqn.nr
03201     instr.push_back(OPER_ADD); // operator
03202     instr.push_back(9+ABS(ncoeff)); // total length
03203     instr.push_back(5+ABS(ncoeff)); // term length
03204     instr.push_back(vartype); // (extra)symbol
03205     instr.push_back(4); // symbol length
03206     instr.push_back(varnum); // symbol id
03207     instr.push_back(1); // power of 1
03208     for (int i=0; i<ABS(ncoeff); i++) // coefficient
03209         instr.push_back(optim.coeff[i]);
03210     instr.push_back(0); // trailing 0
03211 }
03212 // #] type 2 :
03213 // #[ type 3 : substitution of the form z=x+c (optim.type==3)
03214 if (optim.type == 3) {
03215     int vartype = optim.arg1>0 ? SYMBOL : EXTRASYMBOL;
03216     int varnum = ABS(optim.arg1) - 1;
03217
03218     WORD ncoeff = optim.coeff.back();
03219
03220     // scan through equation
03221     for (int i=0; i<(int)optim.eqnidxs.size(); i++) {
03222         WORD *e = ebegin + optim.eqnidxs[i];
03223
03224         if (*(e+1) != OPER_ADD) continue;
03225
03226         int coeff_match=0, var_match=0;
03227
03228         for (WORD *t=e+3; *t!=0; t+=*t) {
03229             if (*t == ABS(*(t+*t-1))+1 && BigLong((UWORD *) &optim.coeff[0],ABS(ncoeff)-1,
03230 (UWORD
* )t+*t-ABS(*(t+*t-1)),ABS(*(t+*t-1))-1) == 0)
03231                 coeff_match = SGN(ncoeff) * SGN(*(t+*t-1));
03232             else if (*t+1==vartype && ABS(*(t+3))==varnum && *(t+4)==1)
03233                 var_match = SGN(*(t+7));
03234         }
03235
03236         // substitute if found (x+c and -x-c and matches)
03237         if (coeff_match * var_match == 1) {
03238
03239             WORD *newt = e+3;
03240
03241             for (WORD *t=e+3; *t!=0;) {
03242                 int dt=*t;
03243                 if (*t!=ABS(*(t+*t-1))+1 && !(*t+1)==vartype && ABS(*(t+3))==varnum &&

```

```

    *(t+4)==1)) {
03244             memmove(newt, t, dt*sizeof(WORD));
03245             newt += dt;
03246         }
03247         t+=dt;
03248     }
03249
03250     *newt++ = 8;           // term length
03251     *newt++ = EXTRASYMBOL; // extrasymbol
03252     *newt++ = 4;           // symbol length
03253     *newt++ = newid;       // symbol id
03254     *newt++ = 1;           // power of 1
03255     *newt++ = 1;
03256     *newt++ = 1;           // coefficient of +/-1
03257     *newt++ = 3*coeff_match;
03258     *newt++ = 0;           // trailing zero
03259
03260     substituted = true;
03261 }
03262 }
03263
03264     if (!substituted) {
03265 #ifdef DEBUG_GREEDY
03266         MesPrint ("*** [%s, w=%w] DONE: do_optimization : res=false", thetime_str().c_str(),
optim.improve);
03267 #endif
03268         return false;
03269     }
03270
03271     // add extra equation (Tnew = x+c)
03272     instr.push_back(newid);           // eqn.nr
03273     instr.push_back(OPER_ADD);        // operator
03274     instr.push_back(13+ABS(ncoeff)); // total length
03275     instr.push_back(8);               // x-term length
03276     instr.push_back(vartype);         // (extra)symbol
03277     instr.push_back(4);               // symbol length
03278     instr.push_back(varnum);          // symbol id
03279     instr.push_back(1);               // power of 1
03280     instr.push_back(1);
03281     instr.push_back(1);               // coefficient of 1
03282     instr.push_back(3);
03283     instr.push_back(ABS(ncoeff)+1);   // c-term length
03284     for (int i=0; i<ABS(ncoeff); i++) // coefficient
03285         instr.push_back(optim.coeff[i]);
03286     instr.push_back(0);               // trailing zero
03287 }
03288 // #] type 3 :
03289 // #[ type 4,5 : substitution of the form z=x+y or z=x-y (optim.type=4 or 5)
03290 if (optim.type >= 4) {
03291
03292     int vartypex = optim.arg1>0 ? SYMBOL : EXTRASYMBOL;
03293     int varnumx = ABS(optim.arg1) - 1;
03294     int vartypey = optim.arg2>0 ? SYMBOL : EXTRASYMBOL;
03295     int varnumy = ABS(optim.arg2) - 1;
03296
03297     // scan through equations
03298     for (int i=0; i<(int)optim.eqnidxs.size(); i++) {
03299         WORD *e = ebegin + optim.eqnidxs[i];
03300
03301         if (*(e+1) != OPER_ADD) continue;
03302
03303         const WORD *coeffx=NULL, *coeffy=NULL;
03304         WORD ncoeffx=0,ncoeffy=0;
03305
03306         // looks for terms
03307         for (WORD *t=e+3; *t!=0; t+=*t) {
03308             if (*t == ABS(*(t+*t-1))+1) continue; // constant
03309             if (*(t+1)==vartypex && *(t+3)==varnumx && *(t+4)==1) {
03310                 coeffx = t+5;
03311                 ncoeffx = *(t+*t-1);
03312             }
03313             if (*(t+1)==vartypey && *(t+3)==varnumy && *(t+4)==1) {
03314                 coeffy = t+5;
03315                 ncoeffy = *(t+*t-1);
03316             }
03317         }
03318
03319         // check signs (type=4: x+y and -x-y, type=5: x-y and -x+y) ??????
03320         // check signs (type=4: x+y, type=5: x-y) !!!!!!!!!!!
03321         if (SGN(ncoeffx) * SGN(ncoeffy) * (optim.type==4 ? 1 : -1) == 1) {
03322             // check absolute value of coefficients
03323             if (BigLong((UWORD *)coeffx, ABS(ncoeffx)-1, (UWORD *)coeffy, ABS(ncoeffy)-1) == 0) {
03324                 // substitute
03325                 vector<WORD> coeff(coeffx, coeffx+ABS(ncoeffx));
03326
03327                 WORD *newt = e+3;
03328 /*

```

```

03329 if ( optim.type == 5 ) {
03330     while ( *newt ) newt+=*newt;
03331     int i = (newt - e) - 3;
03332     MesPrint(" < %a",i,e+3);
03333     newt = e+3;
03334 }
03335 /*
03336         for (WORD *t=e+3; *t!=0;) {
03337             int dt=*t;
03338             if (*t == ABS(*t+*t-1))+1 ||
03339                 (!(*t+1)==vartypex && *(t+3)==varnumx) &&
03340                 (!(*t+1)==vartypey && *(t+3)==varnumy)) {
03341                 memmove(newt, t, dt*sizeof(WORD));
03342                 newt += dt;
03343             }
03344             t+=dt;
03345         }
03346
03347         *newt++ = 5 + ABS(ncoeffx);           // term length
03348         *newt++ = EXTRASYMBOL;               // extrasymbol
03349         *newt++ = 4;                         // symbol length
03350         *newt++ = newid;                     // symbol id
03351         *newt++ = 1;                         // power of 1
03352         for (int j=0; j<ABS(ncoeffx); j++) // coefficient
03353             *newt++ = coeff[j];
03354         *newt++ = 0;                         // trailing 0
03355         substituted = true;
03356 /*
03357 if ( optim.type == 5 ) {
03358     int i = (newt - e) - 4;
03359     MesPrint(" > %a",i,e+3);
03360 }
03361 */
03362     }
03363 }
03364 }
03365
03366     if (!substituted) {
03367 #ifdef DEBUG_GREEDY
03368         MesPrint ("*** [%s, w=%w] DONE: do_optimization : res=false", thetime_str().c_str(),
03369             optim.improve);
03369 #endif
03370         return false;
03371     }
03372 /*
03373 if ( optim.type == 5 )
03374 MesPrint ("improve=%d, %c%d%c%c%d", optim.improve,
03375     optim.arg1>0?'x':'Z', ABS(optim.arg1)-1,
03376     optim.type==1 ? '*' : optim.type==4 ? '+' : '-',
03377     optim.arg2>0?'x':'Z', ABS(optim.arg2)-1);
03378 */
03379     // add extra equation (Tnew = x+/-y)
03380     instr.push_back(newid); // eqn.nr
03381     instr.push_back(OPER_ADD); // operator
03382     instr.push_back(20); // total length
03383     instr.push_back(8); // term length
03384     instr.push_back(vartypex); // (extra)symbol
03385     instr.push_back(4); // symbol length
03386     instr.push_back(varnumx); // symbol id
03387     instr.push_back(1); // power of 1
03388     instr.push_back(1);
03389     instr.push_back(1); // coefficient of 1
03390     instr.push_back(3);
03391     instr.push_back(8); // term length
03392     instr.push_back(vartypey); // (extra)symbol
03393     instr.push_back(4); // symbol length
03394     instr.push_back(varnumy); // symbol id
03395     instr.push_back(1); // power of 1
03396     instr.push_back(1);
03397     instr.push_back(1); // coefficient of +/-1
03398     instr.push_back(3*(optim.type==4?1:-1));
03399     instr.push_back(0); // trailing 0
03400 }
03401 // #] type 4,5 :
03402 // #[ trivial : remove trivial equations of the form Zi = +/-Zj
03403 vector<int> renum(newid+1, 0);
03404 bool do_renum=false;
03405
03406 // vector may be moved when it is extended
03407 ebegin = &*instr.begin();
03408
03409 for (int i=0; i<(int)optim.eqnidxs.size(); i++) {
03410     WORD *e = ebegin + optim.eqnidxs[i];
03411     WORD *t = e+3;
03412     if (*t==0) continue; // empty (removed) equation
03413     if (*t+*t)!=0 continue; // more than 1 term
03414     if (*t!=8) continue; // term not of correct form

```

```

03415         if (*(t+4)!=1) continue; // power != 1
03416         if (*(t+5)!=1 || *(t+6)!=1) continue; // coeff != +/-1
03417
03418         // trivial term, so renumber this one
03419         renum[*e] = SGN(*(t+7)) * (*(t+3) + 1);
03420         do_renum = true;
03421
03422         // remove equation
03423         *t=0;
03424     }
03425     // #] trivial :
03426     // #[ renumbering :
03427
03428     // there are renumberings to be done, so loop through all equations
03429     if (do_renum) {
03430         WORD *eend = ebegin+instr.size();
03431
03432         for (WORD *e=ebegin; e!=eend; e+=*(e+2)) {
03433             for (WORD *t=e+3; *t!=0; t+=*t) {
03434                 if (*t == ABS(*(t+*t-1))+1) continue;
03435                 if (*(t+1)==EXTRASYNBOL && renum[* (t+3)]!=0) {
03436                     *(t+*t-1) *= SGN(renum[* (t+3)]);
03437                     *(t+3) = ABS(renum[* (t+3)]) - 1;
03438                 }
03439             }
03440         }
03441     }
03442     // #] renumbering :
03443
03444     #ifdef DEBUG_GREEDY
03445     MesPrint ("*** [%s, w=%w] DONE: do_optimization : res=true", thetime_str().c_str(),
03446             optim.improve);
03447     #endif
03448     return true;
03449 }
03450
03451 /*
03452 #] do_optimization :
03453 #[ partial_factorize :
03454 */
03455
03479 int partial_factorize (vector<WORD> &instr, int n, int improve) {
03480
03481     #ifdef DEBUG_GREEDY
03482     MesPrint ("*** [%s, w=%w] CALL: partial_factorize (n=%d)", thetime_str().c_str(), n);
03483     #endif
03484
03485     GETIDENTITY;
03486
03487     // get starting positions of instructions
03488     vector<int> instr_idx(n);
03489     WORD *ebegin = &instr.begin();
03490     WORD *eend = ebegin+instr.size();
03491     for (WORD *e=ebegin; e!=eend; e+=*(e+2)) {
03492         instr_idx[*e] = e - ebegin;
03493     }
03494
03495     // get reference counts
03496     /*
03497     * The next construction replaces the vector construction which is
03498     * rather costly for valgrind (and maybe also in normal running)
03499     */
03500     int nmax = 2*n;
03501     WORD *numpar = (WORD *)Malloc1(nmax*sizeof(WORD), "numpar");
03502     for ( int i = 0; i < nmax; i++ ) numpar[i] = 0;
03503     // vector<WORD> numpar(n);
03504     for (WORD *e=ebegin; e!=eend; e+=*(e+2))
03505         for (WORD *t=e+3; *t!=0; t+=*t) {
03506             if (*t == ABS(*(t+*t-1))+1) continue;
03507             if (*(t+1) == EXTRASYMBOL) numpar[* (t+3)]++;
03508         }
03509
03510     // find factorizable expressions
03511     for (int i=0; i<n; i++) {
03512         WORD *e = &instr.begin() + instr_idx[i];
03513         if (*(e+1) != OPER_ADD) continue;
03514
03515         // count symbol occurrences
03516         map<WORD,WORD> cnt; // 1-indexed, <0:EXTRASYNBOL, >0:SYMBOL
03517
03518         for (WORD *t=e+3; *t!=0; t+=*t) {
03519             if (*t==ABS(*(t+*t-1))+1) continue;
03520
03521             // count symbols in t
03522             if (*(t+4)==1)
03523                 cnt[(*(t+1)==SYMBOL ? 1 : -1) * (*(t+3)+1)]++;

```

```

03524
03525 // count symbols in extrasymbols of t
03526 if (*(t+1)==EXTRASymbol && *(t+4)==1 && numpar[* (t+3)]==1) {
03527     WORD *t2 = &*instr.begin() + instr_idx[* (t+3)];
03528     if (*(t2+1) != OPER_MUL) continue;
03529     for (t2+=3; *t2!=0; t2+=*t2) {
03530         if (*t2 == ABS(* (t2+*t2-1))+1) continue;
03531         if (*(t2+4)==1)
03532             cnt[(* (t2+1))==SYMBOL ? 1 : -1] * (* (t2+3)+1)++;
03533     }
03534 }
03535 }
03536
03537 // find most-occurring symbol
03538 WORD x=0, best=0;
03539 for (map<WORD,WORD>::iterator it=cnt.begin(); it!=cnt.end(); it++)
03540     if (it->second > best) { x=it->first; best=it->second; }
03541
03542 // occurrence>=2 and occurrence>improve, so factorize
03543
03544 if (best>=2 && best>improve) {
03545     // initialize new equation (Zi from example above)
03546     vector<WORD> new_eqn;
03547     new_eqn.push_back(n);
03548     new_eqn.push_back(OPER_ADD);
03549     new_eqn.push_back(0); // length
03550
03551     WORD dt;
03552     WORD *newt=e+3;
03553     for (WORD *t=e+3; *t!=0; t+=dt) {
03554         dt = *t;
03555         bool keep=true;
03556
03557         if (*t!=ABS(* (t+*t-1))+1) {
03558
03559             // factorized symbol is in t itself
03560             if (*(t+4)==1) {
03561                 WORD y = (* (t+1))==SYMBOL ? 1 : -1] * (* (t+3)+1);
03562                 if (y==x) {
03563                     new_eqn.push_back(*t-4);
03564                     new_eqn.insert(new_eqn.end(), t+5, t+dt);
03565                     keep=false;
03566                 }
03567             }
03568
03569             // look in extrasymbol of t with ref.count=1
03570             if (*(t+1)==EXTRASymbol && *(t+4)==1 && numpar[* (t+3)]==1) {
03571                 WORD *t2 = &*instr.begin() + instr_idx[* (t+3)];
03572                 if (*(t2+1) == OPER_MUL) {
03573                     bool has_x=false;
03574                     for (t2+=3; *t2!=0; t2+=*t2) {
03575                         if (*t2 == ABS(* (t2+*t2-1))+1) continue;
03576                         WORD y = (* (t2+1))==SYMBOL ? 1 : -1] * (* (t2+3)+1);
03577                         // extrasymbol has factorized symbol
03578                         if (y==x && *(t2+4)==1) {
03579                             has_x=true;
03580                             // copy remaining part
03581                             WORD *tend=t2+*t2;
03582                             WORD sign = SGN(* (tend-1));
03583                             while (*tend!=0) tend+=*tend;
03584                             int dt2 = tend - (t2+*t2);
03585                             memmove(t2, t2+*t2, (dt2+1)*sizeof(WORD));
03586                             t2 += dt2;
03587                             * (t2-1) *= sign;
03588                             break;
03589                         }
03590                     }
03591                     if (has_x) {
03592                         // extrasymbol has x, so add it to new equation
03593                         keep=false;
03594                         int thisidx=new_eqn.size();
03595                         new_eqn.insert(new_eqn.end(), t, t+dt);
03596                         t2 = &*instr.begin() + instr_idx[* (t+3)] + 3;
03597                         // if becomes trivial, substitute the term
03598                         if (* (t2+*t2)==0) {
03599                             // it's a number
03600                             if (*t2 == ABS(* (t2+*t2-1))+1) {
03601                                 if (ABS(new_eqn[new_eqn.size()-1])==3 &&
03602                                     new_eqn[new_eqn.size()-2]==1 && new_eqn[new_eqn.size()-3]==1) {
03603                                     // original equation has coefficient of +/-1, so replace
03604                                     it
03605                                     WORD sign = SGN(new_eqn.back());
03606                                     new_eqn.erase(new_eqn.begin()+thisidx, new_eqn.end());
03607                                     new_eqn.insert(new_eqn.end(), t2, t2+*t2);
03608                                     new_eqn.back() *= sign;
03609                                     *t2 = 0;
03610                                 }
03611                             }
03612                         }
03613                     }
03614                 }
03615             }
03616         }
03617     }
03618 }

```



```

03609                                     else {
03610                                         // two non-trivial coefficients, so multiply them
03611                                         // note: untested code (found no way to trigger it)
03612                                         UWORD *tmp = NumberMalloc("partial_factorize");
03613                                         WORD ntmp=0;
03614                                         MulRat (BHEAD (UWORD *)t2+*t2-ABS(* (t2+*t2-1)),
* (t2+*t2-1),
03615                                                         (UWORD
*)&*(new_eqn.end()-ABS(new_eqn.back())), new_eqn.back(),
03616                                                         tmp, &ntmp);
03617                                         new_eqn.erase(new_eqn.begin()+thisidx, new_eqn.end());
03618                                         new_eqn.push_back(ABS(ntmp)+1);
03619                                         new_eqn.insert(new_eqn.end(), tmp, tmp+ABS(ntmp));
03620                                         NumberFree(tmp,"partial_factorize");
03621                                         *t2 = 0;
03622                                     }
03623                                 }
03624                             else if (*(t2+4)==1) {
03625                                 // it's a variable
03626                                 new_eqn.back() *= SGN(* (t2+*t2-1));
03627                                 new_eqn[thisidx+1] = *(t2+1);
03628                                 new_eqn[thisidx+2] = *(t2+2);
03629                                 new_eqn[thisidx+3] = *(t2+3);
03630                                 new_eqn[thisidx+4] = *(t2+4);
03631                                 *t2 = 0;
03632                             }
03633                         }
03634                     }
03635                 }
03636             }
03637         }
03638     // no x, so copy it
03639     if (keep) {
03640         memmove(newt, t, dt*sizeof(WORD));
03641         newt += dt;
03642     }
03643 }
03644 }
03645
03646 // finalize new equation
03647 new_eqn.push_back(0);
03648 new_eqn[2] = new_eqn.size();
03649
03650 bool empty = newt == e+3;
03651 if ( n+1 >= nmax ) {
03652     int i, newnmax = nmax*2;
03653     WORD *newnumpar = (WORD *)Malloc1(newnmax*sizeof(WORD),"newnumpar");
03654     for ( i = 0; i < n; i++ ) newnumpar[i] = numpar[i];
03655     for ( i < newnmax; i++ ) newnumpar[i] = 0;
03656     M_free(numpar,"numpar");
03657     numpar = newnumpar;
03658     nmax = newnmax;
03659 }
03660 // numpar.push_back(0);
03661 n++;
03662
03663 // if original is not empty, add new equation (Zj) to it
03664 // otherwise replace it later
03665 if (!empty) {
03666     *newt++ = 8;
03667     *newt++ = EXTRASYMBOL;
03668     *newt++ = 4;
03669     *newt++ = n;
03670     *newt++ = 1;
03671     *newt++ = 1;
03672     *newt++ = 1;
03673     *newt++ = 3;
03674     *newt++ = 0;
03675 }
03676
03677 // add new equation to instructions
03678 instr_idx.push_back(instr.size());
03679 instr.insert(instr.end(), new_eqn.begin(), new_eqn.end());
03680
03681 // generate another new equation (Zj=x*Zi)
03682 new_eqn.clear();
03683 new_eqn.push_back(n);
03684 new_eqn.push_back(OPER_MUL);
03685 new_eqn.push_back(20);
03686 new_eqn.push_back(8);
03687
03688 // add factorized symbol
03689 if (x>0) {
03690     new_eqn.push_back(SYMBOL);
03691     new_eqn.push_back(4);
03692     new_eqn.push_back(x-1);
03693     new_eqn.push_back(1);

```

```

03694     }
03695     else {
03696         new_eqn.push_back (EXTRASymbol);
03697         new_eqn.push_back (4);
03698         new_eqn.push_back (-x-1);
03699         new_eqn.push_back (1);
03700     }
03701     new_eqn.push_back (1);
03702     new_eqn.push_back (1);
03703     new_eqn.push_back (3);
03704     new_eqn.push_back (8);
03705     // add new equation (Zi)
03706     new_eqn.push_back (EXTRASymbol);
03707     new_eqn.push_back (4);
03708     new_eqn.push_back (n-1);
03709     new_eqn.push_back (1);
03710     new_eqn.push_back (1);
03711     new_eqn.push_back (1);
03712     new_eqn.push_back (3);
03713     new_eqn.push_back (0);
03714
03715     if (!empty) {
03716         // add new equation (Zj) to instructions
03717         instr_idx.push_back (instr.size());
03718         instr.insert (instr.end(), new_eqn.begin(), new_eqn.end());
03719         if ( n+1 >= nmax ) {
03720             int i, newnmax = nmax*2;
03721             WORD *newnumpar = (WORD *) Malloc1 (newnmax*sizeof(WORD), "newnumpar");
03722             for ( i = 0; i < n; i++ ) newnumpar[i] = numpar[i];
03723             for ( ; i < newnmax; i++ ) newnumpar[i] = 0;
03724             M_free (numpar, "numpar");
03725             numpar = newnumpar;
03726             nmax = newnmax;
03727         }
03728         // numpar.push_back (0);
03729         n++;
03730     }
03731     else {
03732         // replace e with Zj
03733         e = &*instr.begin() + instr_idx[i];
03734         e[1] = OPER_MUL;
03735         memcpy (e+3, &new_eqn[3], (new_eqn.size()-3)*sizeof(WORD));
03736     }
03737
03738     // decrease i, so this expression is factorized again if possible
03739     i--;
03740 }
03741 }
03742
03743 #ifdef DEBUG_GREEDY
03744     MesPrint ("*** [%s, w=%w] DONE: partial_factorize (n=%d)", thetime_str().c_str(), n);
03745 #endif
03746     M_free (numpar, "numpar");
03747     return n;
03748 }
03749
03750 /*
03751 #] partial_factorize :
03752 #[ optimize_greedy :
03753 */
03754
03755 vector<WORD> optimize_greedy (vector<WORD> instr, LONG time_limit) {
03756
03757     #ifdef DEBUG
03758         int old_num_oper = count_operators (instr);
03759         MesPrint ("*** [%s, w=%w] CALL: optimize_greedy (numoper=%d)",
03760                 thetime_str().c_str(), old_num_oper);
03761     #endif
03762
03763     LONG start_time = TimeWallClock (1);
03764
03765     WORD *ebegin = &*instr.begin();
03766     WORD *eend = ebegin+instr.size();
03767
03768     // store final equation, since it must be the last equation later
03769     int final_eqn_idx = 0;
03770     int next_eqn = 0;
03771
03772     for (WORD *e=ebegin; e!=eend; e+=*(e+2)) {
03773         next_eqn = *e + 1;
03774         final_eqn_idx = e-ebegin;
03775     }
03776
03777     // optimize instructions
03778     while (TimeWallClock (1)-start_time < time_limit) {
03779         int old_next_eqn = next_eqn;
03780
03781         // find optimizations

```

```

03799     vector<optimization> optim = find_optimizations(instr);
03800
03801     // add_eqnidxs contains modified equations, which might have to be updated later again
03802     vector<int> add_eqnidxs;
03803
03804     // number of optimizations to do
03805     int num_do_optims = max(AO.Optimize.greedyminnum,
03806 (int)optim.size()*AO.Optimize.greedymaxperc/100);
03807
03808     // if best improvement is one, do all optimizations
03809     int best_improve=0;
03810     for (int i=0; i<(int)optim.size(); i++)
03811         best_improve = max(best_improve, optim[i].improve);
03812     if (best_improve <= 1)
03813         num_do_optims = MAXPOSITIVE;
03814     // do a number of optimizations
03815     while (optim.size() > 0 && num_do_optims-- > 0) {
03816
03817         // find best optimization
03818         int best=0;
03819         best_improve=0;
03820         for (int i=0; i<(int)optim.size(); i++)
03821             if (optim[i].improve > best_improve) {
03822                 best=i;
03823                 best_improve=optim[i].improve;
03824             }
03825
03826         // add extra equations
03827         for (int i=0; i<(int)add_eqnidxs.size(); i++)
03828             optim[best].eqnidxs.push_back(add_eqnidxs[i]);
03829
03830         // do optimization, update next_eqn if successful
03831         int next_idx = instr.size();
03832         if (do_optimization(optim[best], instr, next_eqn)) {
03833             next_eqn++;
03834             add_eqnidxs.push_back(next_idx);
03835         }
03836
03837         optim.erase(optim.begin()+best);
03838     }
03839
03840     // partially factorize with improve >= best_improve
03841     next_eqn = partial_factorize(instr, next_eqn, best_improve);
03842
03843     // check whether nothing has changed
03844     if (next_eqn == old_next_eqn) break;
03845 }
03846
03847 // add final equation to the back (must be by definition)
03848 instr.push_back(next_eqn);
03849 instr.insert(instr.end(), instr.begin()+final_eqn_idx+1,
03850 instr.begin()+final_eqn_idx+instr[final_eqn_idx+2]);
03851
03852 // removed original final equation
03853 instr[final_eqn_idx+3] = 0;
03854
03855 // remove extra zeroes
03856 WORD *t = &instr[0];
03857
03858 ebegin = &*instr.begin();
03859 eend = ebegin+instr.size();
03860 int de=0;
03861
03862 for (WORD *e=ebegin; e!=eend; e+=de) {
03863     de = *(e+2);
03864     int n=3;
03865     while (*(e+n) != 0) n+=*(e+n);
03866     n++;
03867     memmove(t, e, n*sizeof(WORD));
03868     *(t+2) = n;
03869     t += n;
03870 }
03871
03872 instr.resize(t - &instr[0]);
03873
03874 #ifdef DEBUG
03875 MesPrint ("*** [%s, w=%w] DONE: optimize_greedy(numoper=%d) : numoper=%d",
03876         thetime_str().c_str(), old_num_oper, count_operators(instr));
03877 #endif
03878
03879 return instr;
03880 }
03881
03882 /*
03883 #] optimize_greedy :
03884 #[ recycle_variables :
03885 */

```

```

03884
03913 vector<WORD> recycle_variables (const vector<WORD> &all_instr) {
03914
03915 #ifdef DEBUG_MORE
03916     MesPrint ("*** [%s, w=%w] CALL: recycle_variables", thetime_str().c_str());
03917 #endif
03918
03919     // get starting positions of instructions
03920     vector<const WORD *> instr;
03921     const WORD *tbegin = &all_instr.begin();
03922     const WORD *tend = tbegin+all_instr.size();
03923     for (const WORD *t=tbegin; t!=tend; t+=*(t+2))
03924         instr.push_back(t);
03925     int n = instr.size();
03926
03927     // determine with expressions are connected, how many intermediate
03928     // are needed (assuming it's a expression tree instead of a DAG) and
03929     // sort the leaves such that you need a minimal number of variables
03930     vector<int> vars_needed(n);
03931     vector<bool> vis(n,false);
03932     vector<vector<int>> > conn(n);
03933
03934     stack<int> s;
03935     s.push(n);
03936
03937     while (!s.empty()) {
03938         int i=s.top(); s.pop();
03939         if (i>0) {
03940             i--;
03941             if (vis[i]) continue;
03942             vis[i]=true;
03943             s.push(-(i+1));
03944
03945             // find all connections
03946             for (const WORD *t=instr[i]+3; *t!=0; t+=*t)
03947                 if (*t!=1+ABS(*(t+*t-1)) && *(t+1)==EXTRASYMBOL) {
03948                     int k = *(t+3);
03949                     conn[i].push_back(k);
03950                     s.push(k+1);
03951                 }
03952         }
03953         else {
03954             i=-i-1;
03955
03956             // sort the childs w.r.t. needed variables
03957             vector<pair<int,int>> need;
03958             for (int j=0; j<(int)conn[i].size(); j++)
03959                 need.push_back(make_pair(vars_needed[conn[i][j]], conn[i][j]));
03960
03961             // keep the comma expression in proper order
03962             if (*(instr[i]+1) != OPER_COMMA)
03963                 sort(need.rbegin(), need.rend());
03964
03965             vars_needed[i] = 1;
03966             for (int j=0; j<(int)need.size(); j++) {
03967                 vars_needed[i] = max(vars_needed[i], need[j].first+j);
03968                 conn[i][j] = need[j].second;
03969             }
03970         }
03971     }
03972
03973     // order the instructions in depth-first order and determine the first
03974     // and last occurrences of variables
03975     vector<int> order, first(n,0), last(n,0);
03976     vis = vector<bool>(n,false);
03977     s.push(n);
03978
03979     while (!s.empty()) {
03980         int i=s.top(); s.pop();
03981
03982         if (i>0) {
03983             i--;
03984             if (vis[i]) continue;
03985             vis[i]=true;
03986             s.push(-(i+1));
03987             for (int j=(int)conn[i].size()-1; j>=0; j--)
03988                 s.push(conn[i][j]+1);
03989         }
03990         else {
03991             i=-i-1;
03992             first[i] = last[i] = order.size();
03993             order.push_back(i);
03994             for (int j=0; j<(int)conn[i].size(); j++) {
03995                 int k = conn[i][j];
03996                 last[k] = max(last[k], first[i]);
03997             }
03998         }
03999     }

```

```

03999     }
04000 }
04001
04002 // find the renumbering to recycled variables, where at any time the
04003 // lowest-indexed variable that can be used is chosen
04004 int numvar=0;
04005 set<int> var;
04006 vector<int> renum(n);
04007
04008 for (int i=0; i<(int)order.size(); i++) {
04009     for (int j=0; j<(int)conn[order[i]].size(); j++) {
04010         int k = conn[order[i]][j];
04011         if (last[k] == i) var.insert(renum[k]);
04012     }
04013
04014     if (var.empty()) var.insert(numvar++);
04015     renum[order[i]] = *var.begin(); var.erase(var.begin());
04016 }
04017
04018 // put the number of variables used in a preprocessor variable
04019
04020 // generate new instructions with the renumbering
04021 vector<WORD> newinstr;
04022
04023 for (int i=0; i<(int)order.size(); i++) {
04024     int x = order[i];
04025     int j = newinstr.size();
04026     newinstr.insert(newinstr.end(), instr[x], instr[x]+(instr[x]+2));
04027
04028     newinstr[j] = renum[newinstr[j]];
04029
04030     for (WORD *t=&newinstr[j+3]; *t!=0; t+=*t)
04031         if (*t!=1+ABS(*t+*t-1)) && *(t+1)==EXTRASymbol
04032             *(t+3) = renum[*t+3];
04033 }
04034
04035 #ifdef DEBUG_MORE
04036     MesPrint ("*** [%s, w=%w] DONE: recycle_variables", thetime_str().c_str());
04037 #endif
04038
04039     return newinstr;
04040 }
04041
04042 /*
04043 #] recycle_variables :
04044 #[ optimize_expression_given_Horner :
04045 */
04046
04060 void optimize_expression_given_Horner () {
04061
04062     #ifdef DEBUG
04063         MesPrint ("*** [%s, w=%w] CALL: optimize_expression_given_Horner", thetime_str().c_str());
04064     #endif
04065
04066     GETIDENTITY;
04067
04068     // initialize timer
04069     LONG start_time = TimeWallClock(1);
04070     LONG time_limit = 100 * AO.Optimize.greedytimelimit / (AO.Optimize.horner == O_MCTS ?
AO.Optimize.mctsnmkeep : 1);
04071     if (time_limit == 0) time_limit=MAXPOSITIVE;
04072
04073     // pick a Horner scheme from the list
04074     LOCK(optimize_lock);
04075     vector<WORD> Horner_scheme = optimize_best_Horner_schemes.back();
04076     optimize_best_Horner_schemes.pop_back();
04077     UNLOCK(optimize_lock);
04078
04079     // if ( ( AO.Optimize.debugflags&2 ) == 2 ) {
04080     //     MesPrint ("Scheme: %a", Horner_scheme.size(), &(Horner_scheme[0]));
04081     // }
04082
04083     // apply Horner scheme
04084     vector<WORD> tree = Horner_tree(optimize_expr, Horner_scheme);
04085
04086     // generate instructions, eventually with CSE
04087     vector<WORD> instr;
04088
04089     if (AO.Optimize.method == O_CSE || AO.Optimize.method == O_CSEGREEDY)
04090         instr = generate_instructions(tree, true);
04091     else
04092         instr = generate_instructions(tree, false);
04093     if (AO.Optimize.method == O_CSEGREEDY || AO.Optimize.method == O_GREEDY) {
04094         instr = merge_operators(instr, false);
04095         instr = optimize_greedy(instr, time_limit-(TimeWallClock(1)-start_time));
04096         instr = merge_operators(instr, true);
04097         instr = optimize_greedy(instr, time_limit-(TimeWallClock(1)-start_time));
04098     }

```

```

04099     }
04100     instr = merge_operators(instr, true);
04101
04102     // recycle the temporary variables
04103     instr = recycle_variables(instr);
04104
04105     // determine the quality of the code and possibly update the best code
04106     int num_oper = count_operators(instr);
04107
04108     LOCK(optimize_lock);
04109     if (num_oper < optimize_best_num_oper) {
04110         optimize_num_vars = Horner_scheme.size();
04111         optimize_best_num_oper = num_oper;
04112         optimize_best_instr = instr;
04113         optimize_best_vars = vector<WORD>(AN.poly_vars, AN.poly_vars+AN.poly_num_vars);
04114     }
04115     UNLOCK(optimize_lock);
04116
04117     // clean poly_vars, that are allocated by Horner_tree
04118     AN.poly_num_vars = 0;
04119     M_free(AN.poly_vars, "poly_vars");
04120
04121 #ifdef DEBUG
04122     MesPrint ("*** [%s, w=%w] DONE: optimize_expression_given_Horner", thetime_str().c_str());
04123 #endif
04124 }
04125
04126 /*
04127     #] optimize_expression_given_Horner :
04128     #[ PF_optimize_expression_given_Horner :
04129 */
04130 #ifdef WITHMPI
04131
04132 // Initialization.
04133 void PF_optimize_expression_given_Horner_master_init () {
04134     // Nothing to do for now.
04135 }
04136
04137 // Wait for an idle slave and return the process number.
04138 int PF_optimize_expression_given_Horner_master_next () {
04139     // Find an idle slave.
04140     int next;
04141     PF_LongSingleReceive(PF_ANY_SOURCE, PF_OPT_HORNER_MSGTAG, &next, NULL);
04142     return next;
04143 }
04144
04145 // The main function on the master.
04146 void PF_optimize_expression_given_Horner_master () {
04147 #ifdef DEBUG
04148     MesPrint ("*** [%s, w=%w] CALL: PF_optimize_expression_given_Horner_master",
04149         thetime_str().c_str());
04150 #endif
04151
04152     // pick a Horner scheme from the list
04153     vector<WORD> Horner_scheme = optimize_best_Horner_schemes.back();
04154     optimize_best_Horner_schemes.pop_back();
04155
04156     // Find an idle slave.
04157     int next = PF_optimize_expression_given_Horner_master_next();
04158
04159     // Send a new task to the slave.
04160     PF_PrepareLongSinglePack();
04161     int len = Horner_scheme.size();
04162     PF_LongSinglePack(&len, 1, PF_INT);
04163     PF_LongSinglePack(&Horner_scheme[0], len, PF_WORD);
04164     PF_LongSingleSend(next, PF_OPT_HORNER_MSGTAG);
04165
04166 #ifdef DEBUG
04167     MesPrint ("*** [%s, w=%w] DONE: PF_optimize_expression_given_Horner_master",
04168         thetime_str().c_str());
04169 #endif
04170 }
04171
04172 // Wait for all the slaves to finish their tasks.
04173 void PF_optimize_expression_given_Horner_master_wait () {
04174 #ifdef DEBUG
04175     MesPrint ("*** [%s, w=%w] CALL: PF_optimize_expression_given_Horner_master_wait",
04176         thetime_str().c_str());
04177 #endif
04178
04179     // Wait for all the slaves.
04180     for (int i = 1; i < PF.numtasks; i++) {

```

```

04183     int next = PF_optimize_expression_given_Horner_master_next();
04184     // Send a null task.
04185     PF_PrepareLongSinglePack();
04186     int len = 0;
04187     PF_LongSinglePack(&len, 1, PF_INT);
04188     PF_LongSingleSend(next, PF_OPT_HORNER_MSGTAG);
04189 }
04190
04191 // Combine the result from all the slaves.
04192 optimize_best_num_oper = INT_MAX;
04193 for (int i = 1; i < PF.numtasks; i++) {
04194     PF_LongSingleReceive(PF_ANY_SOURCE, PF_OPT_COLLECT_MSGTAG, NULL, NULL);
04195
04196     int len;
04197
04198     // The first integer is the number of operations.
04199     PF_LongSingleUnpack(&len, 1, PF_INT);
04200
04201     if (len >= optimize_best_num_oper) continue;
04202
04203     // Update the best result.
04204     optimize_best_num_oper = len;
04205     PF_LongSingleUnpack(&len, 1, PF_INT);
04206     optimize_best_instr.resize(len);
04207     PF_LongSingleUnpack(&optimize_best_instr[0], len, PF_WORD);
04208     PF_LongSingleUnpack(&len, 1, PF_INT);
04209     optimize_best_vars.resize(len);
04210     PF_LongSingleUnpack(&optimize_best_vars[0], len, PF_WORD);
04211
04212     optimize_num_vars = optimize_best_vars.size(); // TODO
04213 }
04214
04215 #ifdef DEBUG
04216     MesPrint ("*** [%s, w=%w] DONE: PF_optimize_expression_given_Horner_master_wait",
04217             thetime_str().c_str());
04218 #endif
04219 }
04220
04221 // The main function on the slaves.
04222 void PF_optimize_expression_given_Horner_slave () {
04223
04224     #ifdef DEBUG
04225         MesPrint ("*** [%s, w=%w] CALL: PF_optimize_expression_given_Horner_slave",
04226                 thetime_str().c_str());
04227     #endif
04228
04229     optimize_best_Horner_schemes.clear();
04230     optimize_best_num_oper = INT_MAX;
04231
04232     int dummy = 0;
04233     int len;
04234
04235     for (;;) {
04236         // Ask the master for the next task.
04237         PF_PrepareLongSinglePack();
04238         PF_LongSinglePack(&dummy, 1, PF_INT);
04239         PF_LongSingleSend(MASTER, PF_OPT_HORNER_MSGTAG);
04240
04241         // Get a task from the master.
04242         PF_LongSingleReceive(MASTER, PF_OPT_HORNER_MSGTAG, NULL, NULL);
04243
04244         // Length of the task.
04245         PF_LongSingleUnpack(&len, 1, PF_INT);
04246
04247         // No task remains.
04248         if (len == 0) break;
04249
04250         // Perform the given task.
04251         optimize_best_Horner_schemes.push_back(vector<WORD>());
04252         vector<WORD> &Horner_scheme = optimize_best_Horner_schemes.back();
04253         Horner_scheme.resize(len);
04254         PF_LongSingleUnpack(&Horner_scheme[0], len, PF_WORD);
04255         optimize_expression_given_Horner ();
04256     }
04257
04258     // Send the result to the master.
04259     PF_PrepareLongSinglePack();
04260     PF_LongSinglePack(&optimize_best_num_oper, 1, PF_INT);
04261     if (optimize_best_num_oper != INT_MAX) {
04262         len = optimize_best_instr.size();
04263         PF_LongSinglePack(&len, 1, PF_INT);
04264         PF_LongSinglePack(&optimize_best_instr[0], len, PF_WORD);
04265         len = optimize_best_vars.size();
04266         PF_LongSinglePack(&len, 1, PF_INT);
04267         PF_LongSinglePack(&optimize_best_vars[0], len, PF_WORD);
04268     }

```

```

04268     PF_LongSingleSend(MASTER, PF_OPT_COLLECT_MSGTAG);
04269
04270 #ifdef DEBUG
04271     MesPrint ("*** [%s, w=%w] DONE: PF_optimize_expression_given_Horner_slave",
04272             thetime_str().c_str());
04273 #endif
04274 }
04275
04276 #endif
04277 /*
04278     #] PF_optimize_expression_given_Horner :
04279     #[ generate_output :
04280     */
04281
04291 void generate_output (const vector<WORD> &instr, int exprnr, int extraoffset, const
    vector<vector<WORD> > &brackets) {
04292
04293 #ifdef DEBUG
04294     MesPrint ("*** [%s, w=%w] CALL: generate_output", thetime_str().c_str());
04295 #endif
04296
04297     GETIDENTITY;
04298     vector<WORD> output;
04299
04300     // one-indexed instead of zero-indexed
04301     extraoffset++;
04302     int num = 0;
04303     int maxsize = (int)instr.size();
04304     for (int i=0; i<maxsize; i+=instr[i+2]) num++;
04305     int *tstart = (int *)Malloca1(num*sizeof(int), "nplaces");
04306     num = 0;
04307     for (int i=0; i<maxsize; i+=instr[i+2]) tstart[num++] = i;
04308     for (int j=0; j<num; j++) {
04309         int i = tstart[j];
04310
04311         // copy arguments
04312         WCOPY(AT.WorkPointer, &instr[i+3], (instr[i+2]-3));
04313
04314         // update maximal variable number
04315         AO.OptimizeResult.maxvar = MAX(AO.OptimizeResult.maxvar, instr[i]+extraoffset);
04316
04317         // renumber symbols and extrasymbols to correct values
04318         for (WORD *t=AT.WorkPointer; *t!=0; t+=*t) {
04319             if (*t == ABS>(*t+*t-1)+1) continue;
04320             if (*t+1==SYMBOL) *t+3 = optimize_best_vars[*t+3];
04321             if (*t+1==EXTRASymbol) {
04322                 *t+1 = SYMBOL;
04323                 *t+3 = MAXVARIABLES-*t+3-extraoffset;
04324             }
04325         }
04326
04327         // reformat multiplication instructions, since their current
04328         // format is "expr.nr OPER_MUL length arguments", which differs
04329         // from Form's format for a product of symbols
04330         if (instr[i+1]==OPER_MUL) {
04331
04332             WORD *now=AT.WorkPointer+1;
04333             int dt;
04334             bool coeff=false;
04335             int coeff_sign=1;
04336
04337             for (WORD *t=AT.WorkPointer; *t!=0; t+=dt) {
04338                 dt = *t;
04339                 if (*t == ABS>(*t+*t-1)+1) {
04340                     // copy coefficient
04341                     memmove(AT.WorkPointer+instr[i+2], t, *t*sizeof(WORD));
04342                     coeff = true;
04343                 }
04344                 else {
04345                     // move symbol
04346                     int n = *(t+2);
04347                     memmove(now, t+1, n*sizeof(WORD));
04348                     now += n;
04349                     coeff_sign *= SGN(*t+dt-1);
04350                 }
04351             }
04352
04353             if (coeff) {
04354                 // add existing coefficient
04355                 int n = *(AT.WorkPointer + instr[i+2]) - 1;
04356                 memmove(now, AT.WorkPointer+instr[i+2]+1, n*sizeof(WORD));
04357                 now += n;
04358             }
04359             else {
04360                 // add coefficient of one
04361                 *now++=1;

```



```

04362         *now++=1;
04363         *now++=3;
04364     }
04365
04366     *(now-1) *= coeff_sign;
04367
04368     *AT.WorkPointer = now - AT.WorkPointer;
04369     *now++ = 0;
04370 }
04371
04372 // in the case of simultaneous optimization of expressions, add the
04373 // brackets to the final expression
04374 if (instr[i+1]==OPER_COMMA) {
04375     WORD *start = AT.WorkPointer + instr[i+2];
04376     WORD *now = start;
04377     int b=0;
04378     for (const WORD *t=AT.WorkPointer; *t!=0; t+=*t) {
04379         if ( ( brackets[b].size() != 0 ) && ( brackets[b][0] == 0 ) ) break;
04380         *now++ = *t + brackets[b].size();
04381         if (brackets[b].size() != 0) {
04382             memcpy(now, &brackets[b][0], brackets[b].size()*sizeof(WORD));
04383             now += brackets[b].size();
04384         }
04385         memcpy(now, t+1, (*t-1)*sizeof(WORD));
04386         now += *t-1;
04387         b++;
04388     }
04389     *now++ = 0;
04390     memmove(AT.WorkPointer, start, (now-start)*sizeof(WORD));
04391 }
04392
04393 // add the number of the extra symbol; if it is the last one,
04394 // replace the extra symbol number with the expression number
04395 if (i+instr[i+2]<(int)instr.size())
04396     output.push_back(instr[i]+extraoffset);
04397 else {
04398     output.push_back(-(exprnr+1));
04399 }
04400
04401 // add code for this symbol
04402 int n=0;
04403 while (* (AT.WorkPointer+n)!=0)
04404     n += * (AT.WorkPointer+n);
04405 n++;
04406 output.insert(output.end(), AT.WorkPointer, AT.WorkPointer+n);
04407 }
04408
04409 // trailing zero
04410 output.push_back(0);
04411
04412 // clear buffer
04413 if (AO.OptimizeResult.code != NULL)
04414     M_free(AO.OptimizeResult.code, "optimize output");
04415
04416 M_free(tstart,"nplaces");
04417
04418 // copy buffer
04419 AO.OptimizeResult.codesize = output.size();
04420 AO.OptimizeResult.code = (WORD *)Malloc1(output.size()*sizeof(WORD), "optimize output");
04421 memcpy(AO.OptimizeResult.code, &output[0], output.size()*sizeof(WORD));
04422
04423 #ifdef DEBUG
04424     MesPrint ("*** [%s, w=%w] DONE: generate_output", thetime_str().c_str());
04425 #endif
04426 }
04427
04428 /*
04429     #] generate_output :
04430     #[ generate_expression :
04431 */
04432
04440 WORD generate_expression (WORD exprnr) {
04441
04442     #ifdef DEBUG
04443         MesPrint ("*** [%s, w=%w] CALL: generate_expression", thetime_str().c_str());
04444     #endif
04445
04446     GETIDENTITY;
04447
04448     WORD *oldWorkPointer = AT.WorkPointer;
04449
04450     CBUF *C = cbuf+AC.cbufnum;
04451     WORD *term = AT.WorkPointer, oldcurexpr = AR.CurExpr;
04452     POSITION position;
04453     EXPRESSIONS e = Expressions+exprnr;
04454     SetScratch(AR.infile,&(e->onfile));
04455

```

```

04456     if ( GetTerm(BHEAD term) <= 0 ) {
04457         MesPrint("Expression %d has problems in scratchfile",exprnr);
04458         Terminate(-1);
04459     }
04460     SeekScratch(AR.outfile,&position);
04461     e->onfile = position;
04462     if ( PutOut(BHEAD term,&position,AR.outfile,0) < 0 ) {
04463         MesPrint("Expression %d has problems in output scratchfile",exprnr);
04464         Terminate(-1);
04465     }
04466
04467     AR.CurExpr = exprnr;
04468     NewSort(BHEAD0);
04469
04470     // scan for the original expression (marked by *t<0) and give the
04471     // terms to Generator
04472     WORD *t = AO.OptimizeResult.code;
04473     {
04474         WORD old = AR.Cnumlhs; AR.Cnumlhs = 0;
04475         // We can use the remaining part of the WorkSpace for every term that
04476         // goes through Generator. We have WorkSpace problems here if we use
04477         // the (modified) AT.WorkPointer every time in the loop.
04478         WORD* currentWorkPointer = AT.WorkPointer;
04479         while (*t!=0) {
04480             bool is_expr = *t < 0;
04481             t++;
04482             while (*t!=0) {
04483                 if (is_expr) {
04484                     memcpy(currentWorkPointer, t, *t*sizeof(WORD));
04485                     Generator(BHEAD currentWorkPointer, C->numlhs);
04486                 }
04487                 t+=*t;
04488             }
04489             t++;
04490         }
04491         AR.Cnumlhs = old;
04492     }
04493
04494     // final sorting
04495     if (EndSort(BHEAD NULL,0) < 0) {
04496         LowerSortLevel();
04497         Terminate(-1);
04498     }
04499
04500     AT.WorkPointer = oldWorkPointer;
04501     AR.CurExpr = oldcurexpr;
04502
04503     #ifdef DEBUG
04504     MesPrint ("*** [%s, w=%w] DONE: generate_expression", thetime_str().c_str());
04505     #endif
04506
04507     return 0;
04508 }
04509
04510 /*
04511 #] generate_expression :
04512 #[ optimize_print_code :
04513 */
04514
04524 void optimize_print_code (int print_expr) {
04525
04526     #ifdef DEBUG
04527     MesPrint ("*** [%s, w=%w] CALL: optimize_print_code", thetime_str().c_str());
04528     #endif
04529     if ( ( AO.Optimize.debugflags & 1 ) != 0 ) {
04530         /*
04531          * The next code is for debugging purposes. We may want the statements
04532          * in reverse order to substitute them all back.
04533          * Jan used a Mathematica program to do this. Here we make that
04534          * Format Ox,debugflag=1;
04535          * Creates reverse order during printing.
04536          * All we have to do is put id in front of the statements. This is done
04537          * in PrintExtraSymbol.
04538          */
04539         WORD *t = AO.OptimizeResult.code;
04540         WORD num = 0; // First we count the number of objects.
04541         while (*t!=0) {
04542             num++;
04543             t++; while (*t!=0) t+=*t; t++;
04544         }
04545         WORD **tstart = (WORD **)Malloc1(num*sizeof(WORD *),"print objects");
04546         t = AO.OptimizeResult.code; num = 0; // Now we get the addresses
04547         while (*t!=0) {
04548             tstart[num++] = t;
04549             t++; while (*t!=0) t+=*t; t++;
04550         }
04551         // Flip the addresses

```

```

04552         int halfnum = num/2;
04553         for (int i=0; i<halfnum; i++) { swap(tstart[i],tstart[num-1-i]); }
04554         for ( int i = 0; i < num; i++ ) {
04555             t = tstart[i];
04556             if (*t > 0)
04557                 PrintExtraSymbol(*t, t+1, EXTRASYMBOL);
04558             else if (print_expr)
04559                 PrintExtraSymbol(-*t-1, t+1, EXPRESSIONNUMBER);
04560         }
04561         CBUF *C = cbuf + AM.sbufnum;
04562         if (C->numrhs >= AO.OptimizeResult.minvar)
04563             PrintSubtermList(AO.OptimizeResult.minvar, C->numrhs);
04564     }
04565     else {
04566         // print extra symbols from ConvertToPoly in optimization
04567         CBUF *C = cbuf + AM.sbufnum;
04568         if (C->numrhs >= AO.OptimizeResult.minvar)
04569             PrintSubtermList(AO.OptimizeResult.minvar, C->numrhs);
04570
04571         WORD *t = AO.OptimizeResult.code;
04572         while (*t!=0) {
04573             if (*t > 0) {
04574                 PrintExtraSymbol(*t, t+1, EXTRASYMBOL);
04575             }
04576             else if (print_expr)
04577                 PrintExtraSymbol(-*t-1, t+1, EXPRESSIONNUMBER);
04578             t++;
04579             while (*t!=0) t+=*t;
04580             t++;
04581         }
04582     }
04583
04584 #ifdef DEBUG
04585     MesPrint ("*** [%s, w=%w] DONE: optimize_print_code", thetime_str().c_str());
04586 #endif
04587 }
04588
04589 /*
04590 #] optimize_print_code :
04591 #[ Optimize :
04592 */
04593
04637 int Optimize (WORD exprnr, int do_print) {
04638
04639     #if defined(WITHMPI) && (defined(DEBUG) || defined(DEBUG_MORE) || defined(DEBUG_MCTS) ||
04640         defined(DEBUG_GREEDY))
04641         // set AS.printflag negative temporary.
04642         struct save_printflag {
04643             save_printflag() {
04644                 oldprintflag = AS.printflag;
04645                 AS.printflag = -1;
04646             }
04647             ~save_printflag() {
04648                 AS.printflag = oldprintflag;
04649             }
04650             int oldprintflag;
04651         } save_printflag_;
04652     #endif
04653
04654     #ifdef DEBUG
04655         MesPrint ("*** [%s, w=%w] CALL: Optimize", thetime_str().c_str());
04656         MesPrint ("*** %"); PrintRunningTime();
04657     #endif
04658     #ifdef WITHPTHREADS
04659         optimize_lock = dummylock;
04660     #endif
04661
04662     AO.OptimizeResult.minvar = (cbuf + AM.sbufnum)->numrhs + 1;
04663
04664     if (get_expression(exprnr) < 0) return -1;
04665     vector<vector<WORD> > brackets = get_brackets();
04666
04667     #ifdef DEBUG
04668     #ifdef WITHMPI
04669         if (PF.me == MASTER)
04670     #endif
04671         MesPrint ("*** runtime after preparing the expression: %"); PrintRunningTime();
04672     #endif
04673
04674     if (optimize_expr[0]==0 ||
04675         (optimize_expr[optimize_expr[0]]==0 &&
04676          optimize_expr[0]==ABS(optimize_expr[optimize_expr[0]-1])+1) ||
04677         (optimize_expr[optimize_expr[0]]==0 && optimize_expr[0]==8 &&
04678          optimize_expr[5]==1 && optimize_expr[6]==1 && ABS(optimize_expr[7])==3)) {
04679         if (AO.OptimizeResult.code != NULL)
04679             M_free(AO.OptimizeResult.code, "optimize output");

```

```

04680
04681 // zero terms or one trivial term (number or +/-variable), so no
04682 // optimization, so copy expression; special case because without
04683 // operators the optimization crashes
04684 AO.OptimizeResult.code = (WORD *)Malloc1((optimize_expr[0]+3)*sizeof(WORD), "optimize
output");
04685 AO.OptimizeResult.code[0] = -(exprnr+1);
04686 memcpy(AO.OptimizeResult.code+1, optimize_expr, (optimize_expr[0]+1)*sizeof(WORD));
04687 AO.OptimizeResult.code[optimize_expr[0]+2] = 0;
04688 }
04689 else {
04690 // find Horner scheme(s)
04691 optimize_best_Horner_schemes.clear();
04692 if (AO.Optimize.horner == O_OCCURRENCE) {
04693     if (AO.Optimize.hornerdirection != O_BACKWARD)
04694         optimize_best_Horner_schemes.push_back(occurrence_order(optimize_expr, false));
04695     if (AO.Optimize.hornerdirection != O_FORWARD)
04696         optimize_best_Horner_schemes.push_back(occurrence_order(optimize_expr, true));
04697 }
04698 else {
04699     if (AO.Optimize.horner == O_SIMULATED_ANNEALING) {
04700         optimize_best_Horner_schemes.push_back(simulated_annealing());
04701     }
04702     else {
04703         mcts_best_schemes.clear();
04704         if (AO.inscheme) {
04705             optimize_best_Horner_schemes.push_back(vector<WORD>(AO.schemenum));
04706             for (int i=0; i < AO.schemenum; i++)
04707                 optimize_best_Horner_schemes[0][i] = AO.inscheme[i];
04708         }
04709         else {
04710             for (int i = 0; i < AO.Optimize.mctsnmrepeat; i++)
04711                 find_Horner_MCTS();
04712             // generate results
04713             for (set<pair<int, vector<WORD>>>::iterator i=mcts_best_schemes.begin();
i!=mcts_best_schemes.end(); i++) {
04714                 optimize_best_Horner_schemes.push_back(i->second);
04715 #ifdef DEBUG_MCTS
04716                 MesPrint ("{%a} -> %d", i->second.size(), &i->second[0], i->first);
04717 #endif
04718             }
04719             // clear the tree by making a new empty one.
04720             mcts_root = tree_node();
04721         }
04722     }
04723 }
04724 #ifdef DEBUG
04725 #ifdef WITHMPI
04726     if (PF.me == MASTER)
04727 #endif
04728     MesPrint ("*** runtime after Horner: %"); PrintRunningTime();
04729 #endif
04730
04731 #ifdef WITHMPI
04732     if (PF.me == MASTER) {
04733         PF_optimize_expression_given_Horner_master_init();
04734     }
04735 #endif
04736 // find best Horner scheme and results
04737 optimize_best_num_oper = INT_MAX;
04738
04739 int imax = (int)optimize_best_Horner_schemes.size();
04740
04741 for (int i=0; i<imax; i++) {
04742 #if defined(WITHPTHREADS)
04743     if (AM.totalnumberofthreads > 1)
04744         optimize_expression_given_Horner_threaded();
04745     else
04746 #elif defined(WITHMPI)
04747         if (PF.numtasks > 1)
04748             PF_optimize_expression_given_Horner_master();
04749         else
04750 #endif
04751             optimize_expression_given_Horner();
04752 }
04753
04754 #ifdef WITHMPI
04755     PF_optimize_expression_given_Horner_master_wait();
04756 }
04757 else {
04758     if (PF.numtasks > 1)
04759         PF_optimize_expression_given_Horner_slave();
04760 }
04761 #endif
04762
04763 #ifdef WITHPTHREADS
04764     MasterWaitAll();

```

```

04765 #endif
04766 // format results, then print it (for "Print") or modify
04767 // expression (for "#Optimize")
04768 #ifdef WITHMPI
04769     if (PF.me == MASTER)
04770 #endif
04771     generate_output(optimize_best_instr, exprnr, cbuf[AM.sbufnum].numrhs, brackets);
04772 #ifdef WITHMPI
04773     else {
04774         // non-null dummy code for slaves
04775         if (AO.OptimizeResult.code == NULL) {
04776             AO.OptimizeResult.code = (WORD *)Malloc1(sizeof(WORD), "optimize output");
04777         }
04778     }
04779 #endif
04780 }
04781
04782 #ifdef WITHMPI
04783     if (PF.me == MASTER) {
04784         PF_PrepPack();
04785         PF_Pack(&AO.OptimizeResult.minvar, 1, PF_WORD);
04786         PF_Pack(&AO.OptimizeResult.maxvar, 1, PF_WORD);
04787     }
04788     PF_Broadcast();
04789     if (PF.me != MASTER) {
04790         PF_Unpack(&AO.OptimizeResult.minvar, 1, PF_WORD);
04791         PF_Unpack(&AO.OptimizeResult.maxvar, 1, PF_WORD);
04792     }
04793 #endif
04794
04795 // set preprocessor variables
04796 char str[100];
04797 snprintf(str, 100, "%d", AO.OptimizeResult.minvar);
04798 PutPreVar((UBYTE *) "optimminvar_", (UBYTE *) str, 0, 1);
04799 snprintf(str, 100, "%d", AO.OptimizeResult.maxvar);
04800 PutPreVar((UBYTE *) "optimmaxvar_", (UBYTE *) str, 0, 1);
04801
04802 if (do_print) {
04803 #ifdef WITHMPI
04804     if (PF.me == MASTER)
04805 #endif
04806     optimize_print_code(1);
04807     ClearOptimize();
04808 }
04809 else {
04810 #ifdef WITHMPI
04811     if (PF.me == MASTER)
04812 #endif
04813     generate_expression(exprnr);
04814 }
04815
04816 #ifdef WITHMPI
04817     if (PF.me == MASTER) {
04818 #endif
04819
04820     if (AO.Optimize.printstats > 0) {
04821         char str[20];
04822         MesPrint("");
04823         count_operators(optimize_expr, true);
04824         int numop = count_operators(optimize_best_instr, true);
04825         snprintf(str, 20, "%d", numop);
04826         PutPreVar((UBYTE *) "optimvalue_", (UBYTE *) str, 0, 1);
04827     }
04828     else {
04829         char str[20];
04830         int numop = count_operators(optimize_best_instr, false);
04831         snprintf(str, 20, "%d", numop);
04832         PutPreVar((UBYTE *) "optimvalue_", (UBYTE *) str, 0, 1);
04833     }
04834
04835     if ( ( AO.Optimize.schemeflags & 1 ) == 1 ) {
04836         GETIDENTITY
04837         UBYTE *OutScr, *Out, *old1 = AO.OutputLine, *old2 = AO.OutFill;
04838         int i, sym;
04839         AO.OutputLine = AO.OutFill = (UBYTE *) AT.WorkPointer;
04840         FiniLine();
04841         OutScr = (UBYTE *) AT.WorkPointer + ( TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer) ) / 2;
04842         TokenToLine((UBYTE *) " Scheme selected: ");
04843         for ( i = 0; i < optimize_num_vars; i++ ) {
04844             Out = OutScr;
04845             sym = optimize_best_vars[i];
04846             if ( i > 0 ) TokenToLine((UBYTE *) ",");
04847             if ( sym < NumSymbols ) {
04848                 StrCopy(FindSymbol(sym), OutScr);
04849             /* StrCopy(VARNAME(symbols, sym), OutScr); */
04850             }
04851         }
04852     }
04853 #endif

```

```

04852         Out = StrCopy((UBYTE *)AC.extrasym,Out);
04853         if ( AC.extrasymbols == 0 ) {
04854             Out = NumCopy((MAXVARIABLES-sym),Out);
04855             Out = StrCopy((UBYTE *)"_",Out);
04856         }
04857         else if ( AC.extrasymbols == 1 ) {
04858             Out = AddArrayIndex((MAXVARIABLES-sym),Out);
04859         }
04860     }
04861     TokenToLine(OutScr);
04862 }
04863 TokenToLine((UBYTE *)");");
04864 FiniLine();
04865 AO.OutFill = old2;
04866 AO.OutputLine = old1;
04867 }
04868
04869 {
04870     GETIDENTITY
04871     UBYTE *OutScr, *Out, *outstring = 0;
04872     int i, sym;
04873     AO.OutputLine = AO.OutFill = (UBYTE *)AT.WorkPointer;
04874     OutScr = (UBYTE *)AT.WorkPointer + ( TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer) ) /2;
04875     for ( i = 0; i < optimize_num_vars; i++ ) {
04876         Out = OutScr;
04877         sym = optimize_best_vars[i];
04878         if ( sym < NumSymbols ) {
04879             StrCopy(FindSymbol(sym),OutScr);
04880             /* StrCopy(VARNAME(symbols,sym),OutScr); */
04881         }
04882         else {
04883             Out = StrCopy((UBYTE *)AC.extrasym,Out);
04884             if ( AC.extrasymbols == 0 ) {
04885                 Out = NumCopy((MAXVARIABLES-sym),Out);
04886                 Out = StrCopy((UBYTE *)"_",Out);
04887             }
04888             else if ( AC.extrasymbols == 1 ) {
04889                 Out = AddArrayIndex((MAXVARIABLES-sym),Out);
04890             }
04891         }
04892         outstring = AddToString(outstring,OutScr,1);
04893     }
04894     if ( outstring == 0 ) {
04895         PutPreVar((UBYTE *)"optimscheme_", (UBYTE *)"",0,1);
04896     }
04897     else {
04898         PutPreVar((UBYTE *)"optimscheme_", (UBYTE *)outstring,0,1);
04899         M_free(outstring,"AddToString");
04900     }
04901 }
04902
04903 #ifdef WITHMPI
04904 }
04905
04906 // synchronize optimvalue_ and optimscheme_
04907 if ( PF.me == MASTER ) {
04908     UBYTE *value;
04909     int bytes;
04910
04911     PF_PrepareLongMultiPack();
04912
04913     value = GetPreVar((UBYTE *)"optimvalue_", WITHERROR);
04914     bytes = strlen((char *)value);
04915     PF_LongMultiPack(&bytes, 1, PF_INT);
04916     PF_LongMultiPack(value, bytes, PF_BYTE);
04917
04918     value = GetPreVar((UBYTE *)"optimscheme_", WITHERROR);
04919     bytes = strlen((char *)value);
04920     PF_LongMultiPack(&bytes, 1, PF_INT);
04921     PF_LongMultiPack(value, bytes, PF_BYTE);
04922 }
04923 PF_LongMultiBroadcast();
04924 if ( PF.me != MASTER ) {
04925     static vector<UBYTE> prevarbuf;
04926     UBYTE *value;
04927     int bytes;
04928
04929     PF_LongMultiUnpack(&bytes, 1, PF_INT);
04930     prevarbuf.reserve(bytes + 1);
04931     value = &prevarbuf[0];
04932     PF_LongMultiUnpack(value, bytes, PF_BYTE);
04933     value[bytes] = '\0'; // null terminator
04934     PutPreVar((UBYTE *)"optimvalue_", value, NULL, 1);
04935
04936     PF_LongMultiUnpack(&bytes, 1, PF_INT);
04937     prevarbuf.reserve(bytes + 1);
04938     value = &prevarbuf[0];

```

```

04939     PF_LongMultiUnpack(value, bytes, PF_BYTE);
04940     value[bytes] = '\0'; // null terminator
04941     PutPreVar((UBYTE *)"optimscheme_", value, NULL, 1);
04942 }
04943 #endif
04944
04945 // cleanup
04946 M_free(optimize_expr, "LoadOptim");
04947
04948 #ifdef DEBUG
04949     MesPrint ("*** [%s, w=%w] DONE: Optimize", thetime_str().c_str());
04950 #endif
04951
04952     return 0;
04953 }
04954
04955 /*
04956 #] Optimize :
04957 #[ ClearOptimize :
04958 */
04959
04974 int ClearOptimize()
04975 {
04976     char str[20];
04977     WORD numexpr, *w;
04978     int error = 0;
04979     if ( AO.OptimizeResult.code != NULL ) {
04980         M_free(AO.OptimizeResult.code, "optimize output");
04981         AO.OptimizeResult.code = NULL;
04982         AO.OptimizeResult.codesize = 0;
04983         PutPreVar((UBYTE *)"optimminvar_", (UBYTE *)("0"), 0, 1);
04984         PutPreVar((UBYTE *)"optimmaxvar_", (UBYTE *)("0"), 0, 1);
04985         PruneExtraSymbols(AO.OptimizeResult.minvar-1);
04986         cbuf[AM.sbufnum].numrhs = AO.OptimizeResult.minvar-1;
04987         AO.OptimizeResult.minvar = AO.OptimizeResult.maxvar = 0;
04988     }
04989     if ( AO.OptimizeResult.nameofexpr != NULL ) {
04990 /*
04991         We have to pick up the expression by its name. Numbers may change.
04992         Note that this requires that when we overwrite an expression, we
04993         check that it is not an optimized expression. See execute.c and
04994         comexpr.c
04995 */
04996         if ( GetName(AC.exprnames, AO.OptimizeResult.nameofexpr, &numexpr, NOAUTO) != CEXPRESSION ) {
04997             MesPrint("@Internal error while clearing optimized expression %s",
04998 AO.OptimizeResult.nameofexpr);
04999             Terminate(-1);
05000         }
05001         M_free(AO.OptimizeResult.nameofexpr, "optimize expression name");
05002         AO.OptimizeResult.nameofexpr = NULL;
05003         w = &(Expressions[numexpr].status);
05004         *w = SetExprCases(DROP, 1, *w);
05005         if ( *w < 0 ) error = 1;
05006     }
05007     sprintf (str, 20, "%d", cbuf[AM.sbufnum].numrhs);
05008     PutPreVar(AM.oldnumextrasyms, (UBYTE *)str, 0, 1);
05009     PutPreVar((UBYTE *)"optimvalue_", (UBYTE *)("0"), 0, 1);
05010     PutPreVar((UBYTE *)"optimscheme_", (UBYTE *)("0"), 0, 1);
05011     return(error);
05012 }
05013 /*
05014 #] ClearOptimize :
05015 */

```

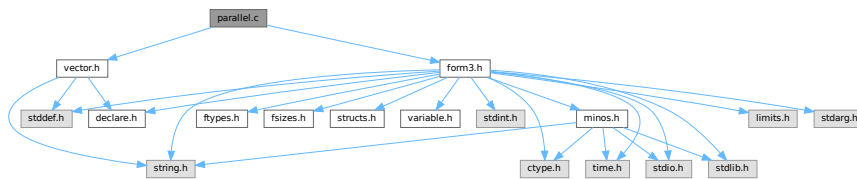
4.95 parallel.c File Reference

```

#include "form3.h"
#include "vector.h"

```

Include dependency graph for parallel.c:



Data Structures

- struct [NoDe](#)
- struct [dollar_buf](#)
- struct [bufIPstruct](#)

Macros

- `#define PRINTFBUF(TEXT, TERM, SIZE) {}`
- `#define SWAP(x, y)`
- `#define PACK\_LONG(p, n)`
- `#define UNPACK\_LONG(p, n)`
- `#define CHECK(condition) __CHECK(condition, __FILE__, __LINE__)`
- `#define \_\_CHECK(condition, file, line) __CHECK(condition, file, line)`
- `#define \_\_CHECK(condition, file, line)`
- `#define DBGOUT(lv1, lv2, a) do { if ( lv1 >= lv2 ) { printf a; fflush(stdout); } } while (0)`
- `#define DBGOUT\_NINTERMS(lv, a)`
- `#define PF\_STATS\_SIZE 5`
- `#define PF\_SNDFILEBUFSIZE 4096`
- `#define recvBuffer logBuffer /* (master) The buffer for receiving messages. */`

Typedefs

- typedef struct [NoDe](#) [NODE](#)
- typedef struct [bufIPstruct](#) [bufIPstruct_t](#)

Functions

- LONG [PF_RealTime](#) (int)
- int [PF_LibInit](#) (int *, char ***)
- int [PF_LibTerminate](#) (int)
- int [PF_Probe](#) (int *)
- int [PF_RecvWbuf](#) (WORD *, LONG *, int *)
- int [PF_IRecvRbuf](#) ([PF_BUFFER](#) *, int, int)
- int [PF_WaitRbuf](#) ([PF_BUFFER](#) *, int, LONG *)
- int [PF_RawSend](#) (int dest, void *buf, LONG l, int tag)
- LONG [PF_RawRecv](#) (int *src, void *buf, LONG thesize, int *tag)
- int [PF_RawProbe](#) (int *src, int *tag, int *bytesize)
- int [PF_EndSort](#) (void)
- WORD [PF_Deferred](#) (WORD *term, WORD level)

- int [PF_Processor](#) ([EXPRESSIONS](#) e, WORD i, WORD LastExpression)
- int [PF_Init](#) (int *argc, char ***argv)
- int [PF_Terminate](#) (int errorcode)
- LONG [PF_GetSlaveTimes](#) (void)
- LONG [PF_BroadcastNumber](#) (LONG x)
- void [PF_BroadcastBuffer](#) (WORD **buffer, LONG *length)
- int [PF_BroadcastString](#) (UBYTE *str)
- int [PF_BroadcastPreDollar](#) (WORD **dbuffer, LONG *newsize, int *numterms)
- int [PF_CollectModifiedDollars](#) (void)
- int [PF_BroadcastModifiedDollars](#) (void)
- int [PF_BroadcastRedefinedPreVars](#) (void)
- int [PF_StoreInsideInfo](#) (void)
- int [PF_RestoreInsideInfo](#) (void)
- int [PF_BroadcastCBuf](#) (int bufnum)
- int [PF_BroadcastExpFlags](#) (void)
- int [PF_BroadcastExpr](#) ([EXPRESSIONS](#) e, [FILEHANDLE](#) *file)
- int [PF_BroadcastRHS](#) (void)
- int [PF_InParallelProcessor](#) (void)
- int [PF_SendFile](#) (int to, FILE *fd)
- int [PF_RecvFile](#) (int from, FILE *fd)
- void [PF_MLock](#) (void)
- void [PF_MUnlock](#) (void)
- LONG [PF_WriteFileToFile](#) (int handle, UBYTE *buffer, LONG size)
- void [PF_FlushStdOutBuffer](#) (void)
- void [PF_FreeErrorMessageBuffers](#) (void)

Variables

- [PARALLELVARS](#) PF

4.95.1 Detailed Description

Message passing library independent functions of parform

This file contains functions needed for the parallel version of form3 these functions need no real link to the message passing libraries, they only need some interface dependent preprocessor definitions (check [parallel.h](#)). So there still need two different objectfiles to be compiled for mpi and pvm!

Definition in file [parallel.c](#).

4.95.2 Macro Definition Documentation

4.95.2.1 PRINTFBUF

```
#define PRINTFBUF(
    TEXT,
    TERM,
    SIZE ) {}
```

Definition at line 117 of file [parallel.c](#).

4.95.2.2 SWAP

```
#define SWAP(
    x,
    y )
```

Value:

```
do { \
    char swap_tmp__[sizeof(x) == sizeof(y) ? (int)sizeof(x) : -1]; \
    memcpy(swap_tmp__, &y, sizeof(x)); \
    memcpy(&y, &x, sizeof(x)); \
    memcpy(&x, swap_tmp__, sizeof(x)); \
} while (0)
```

Swaps the variables *x* and *y*. If `sizeof(x) != sizeof(y)` then a compilation error will occur. A set of `memcpy` calls with constant sizes is expected to be inlined by the optimisation.

Definition at line 124 of file [parallel.c](#).

4.95.2.3 PACK_LONG

```
#define PACK_LONG(
    p,
    n )
```

Value:

```
do { \
    *p)++ = (UWORD)((ULONG)(n) & (ULONG)WORDMASK); \
    *p)++ = (UWORD)((ULONG)(n) >> BITSINWORD) & (ULONG)WORDMASK); \
} while (0)
```

Packs a LONG value *n* to a WORD buffer *p*.

Definition at line 135 of file [parallel.c](#).

4.95.2.4 UNPACK_LONG

```
#define UNPACK_LONG(
    p,
    n )
```

Value:

```
do { \
    (n) = (LONG)((((ULONG)(p)[1] & (ULONG)WORDMASK) << BITSINWORD) | ((ULONG)(p)[0] & (ULONG)WORDMASK)); \
    (p) += 2; \
} while (0)
```

Unpacks a LONG value *n* from a WORD buffer *p*.

Definition at line 144 of file [parallel.c](#).

4.95.2.5 CHECK

```
#define CHECK(
    condition ) _CHECK(condition, __FILE__, __LINE__)
```

A simple check for unrecoverable errors.

Definition at line 153 of file [parallel.c](#).

4.95.2.6 _CHECK

```
#define _CHECK(  
    condition,  
    file,  
    line ) __CHECK(condition, file, line)
```

Definition at line 154 of file [parallel.c](#).

4.95.2.7 __CHECK

```
#define __CHECK(  
    condition,  
    file,  
    line )
```

Value:

```
do { \  
    if ( !(condition) ) { \  
        Error0("Fatal error at " file ":" #line); \  
        Terminate(-1); \  
    } \  
} while (0)
```

Definition at line 155 of file [parallel.c](#).

4.95.2.8 DBGOUT

```
#define DBGOUT(  
    lv1,  
    lv2,  
    a ) do { if ( lv1 >= lv2 ) { printf a; fflush(stdout); } } while (0)
```

Definition at line 166 of file [parallel.c](#).

4.95.2.9 DBGOUT_NINTERMS

```
#define DBGOUT_NINTERMS(  
    lv,  
    a )
```

Definition at line 169 of file [parallel.c](#).

4.95.2.10 PF_STATS_SIZE

```
#define PF_STATS_SIZE 5
```

Definition at line 178 of file [parallel.c](#).

4.95.2.11 PF_SNDFILEBUFSIZE

```
#define PF_SNDFILEBUFSIZE 4096
```

Definition at line 4213 of file [parallel.c](#).

4.95.2.12 recvBuffer

```
#define recvBuffer logBuffer /* (master) The buffer for receiving messages. */
```

Definition at line 4321 of file [parallel.c](#).

4.95.3 Typedef Documentation

4.95.3.1 NODE

```
typedef struct NoDe NODE
```

A node for the tree of losers in the final sorting on the master.

4.95.4 Function Documentation

4.95.4.1 PF_RealTime()

```
LONG PF_RealTime (
    int i )
```

Returns the realtime in 1/100 sec. as a LONG.

Parameters

<i>i</i>	the timer will be reset if <code>i == 0</code> .

Returns

the real elapsed time in 1/100 second.

Definition at line 101 of file [mpi.c](#).

Referenced by [PF_Init\(\)](#).

4.95.4.2 PF_LibInit()

```
int PF_LibInit (
    int * argcp,
    char *** argvp )
```

Performs all library dependent initializations.

Parameters

<i>argcp</i>	pointer to the number of arguments.
<i>argvp</i>	pointer to the arguments.

Returns

0 if OK, nonzero on error.

Definition at line 123 of file [mpi.c](#).

Referenced by [PF_Init\(\)](#).

4.95.4.3 PF_LibTerminate()

```
int PF_LibTerminate (
    int error )
```

Exits mpi, when there is an error either indicated or happening, returnvalue is 1, else returnvalue is 0.

Parameters

<i>error</i>	an error code.

Returns

0 if OK, nonzero on error.

Definition at line 209 of file [mpi.c](#).

Referenced by [PF_Terminate\(\)](#).

4.95.4.4 PF_Probe()

```
int PF_Probe (
    int * src )
```

Probes the next incoming message. If `src == PF_ANY_SOURCE` this operation is blocking, otherwise nonblocking.

Parameters

<i>in, out</i>	<i>src</i>	the source process number. In output, the process number of actual found source.

Returns

the tag value of the next incoming message if found, 0 if a nonblocking probe (input `src != PF_ANY_SOURCE`) did not find any messages. A negative returned value indicates an error.

Definition at line 230 of file [mpi.c](#).

4.95.4.5 PF_RecvWbuf()

```
int PF_RecvWbuf (
    WORD * b,
    LONG * s,
    int * src )
```

Blocking receive of a WORD buffer.

Parameters

out	<i>b</i>	the buffer to store the received data.
in, out	<i>s</i>	the size of the buffer. The output value is the actual size of the received data.
in, out	<i>src</i>	the source process number. The output value is the process number of actual source.

Returns

the received message tag. A negative value indicates an error.

Definition at line 337 of file [mpi.c](#).

4.95.4.6 PF_IRecvRbuf()

```
int PF_IRecvRbuf (
    PF_BUFFER * r,
    int bn,
    int from )
```

Posts nonblocking receive for the active receive buffer. The buffer is filled from `full` to `stop`.

Parameters

<i>r</i>	the PF_BUFFER struct for the nonblocking receive.
<i>bn</i>	the index of the cyclic buffer.
<i>from</i>	the source process number.

Returns

0 if OK, nonzero on error.

Definition at line 366 of file [mpi.c](#).

4.95.4.7 PF_WaitRbuf()

```
int PF_WaitRbuf (
    PF_BUFFER * r,
    int bn,
    LONG * size )
```

Waits for the buffer *bn* to finish a pending nonblocking receive. It returns the received tag and in **size* the number of field received. If the receive is already finished, just return the flag and size, else wait for it to finish, but also check for other pending receives.

Parameters

	<i>r</i>	the PF_BUFFER struct for the pending nonblocking receive.
	<i>bn</i>	the index of the cyclic buffer.
out	<i>size</i>	the actual size of received data.

Returns

the received message tag. A negative value indicates an error.

Definition at line [400](#) of file [mpi.c](#).

4.95.4.8 PF_RawSend()

```
int PF_RawSend (
    int dest,
    void * buf,
    LONG l,
    int tag )
```

Sends *l* bytes from *buf* to *dest*. Returns 0 on success, or -1.

Parameters

<i>dest</i>	the destination process number.
<i>buf</i>	the send buffer.
<i>l</i>	the size of data in the send buffer in bytes.
<i>tag</i>	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line [463](#) of file [mpi.c](#).

Referenced by [PF_MUnlock\(\)](#), and [PF_SendFile\(\)](#).

4.95.4.9 PF_RawRecv()

```
LONG PF_RawRecv (
    int * src,
    void * buf,
    LONG thesize,
    int * tag )
```

Receives not more than *thesize* bytes from *src*, returns the actual number of received bytes, or -1 on failure.

Parameters

in, out	<i>src</i>	the source process number. In output, that of the actual received message.
out	<i>buf</i>	the receive buffer.
	<i>thesize</i>	the size of the receive buffer in bytes.
out	<i>tag</i>	the message tag of the actual received message.

Returns

the actual sizeof received data in bytes, or -1 on failure.

Definition at line 484 of file [mpi.c](#).

Referenced by [PF_RecvFile\(\)](#).

4.95.4.10 PF_RawProbe()

```
int PF_RawProbe (
    int * src,
    int * tag,
    int * bytesize )
```

Probes an incoming message.

Parameters

in, out	<i>src</i>	the source process number. In output, that of the actual received message.
in, out	<i>tag</i>	the message tag. In output, that of the actual received message.
out	<i>bytesize</i>	the size of incoming data in bytes.

Returns

0 if OK, nonzero on error.

Definition at line 508 of file [mpi.c](#).

4.95.4.11 PF_EndSort()

```
int PF_EndSort (
    void )
```

Finishes a master sorting with collecting terms from slaves. Called by [EndSort\(\)](#).

If this is not the masterprocess, just initialize the sendbuffers and return 0, else [PF_EndSort\(\)](#) sends the rest of the terms in the sendbuffer to the next slave and a dummy message to all slaves with tag PF_ENDSORT_MSGTAG. Then it receives the sorted terms, sorts them using a recursive 'tree of losers' ([PF_GetLoser\(\)](#)) and writes them to the outputfile.

Returns

1 if the sorting on the master was done. 0 if [EndSort\(\)](#) still must perform a regular sorting because it is not at the ground level or not on the master or in the sequential mode or in the InParallel mode. -1 if an error occurred.

Remarks

The slaves will send the sorted terms back to the master in the regular sorting (after the initialization of the send buffer in [PF_EndSort\(\)](#)). See [PutOut\(\)](#) and [FlushOut\(\)](#).

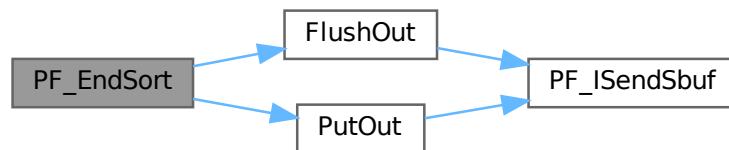
This function has been changed such that when it returns 1, AM.S0->TermsLeft is set correctly. But AM.S0->GenTerms is not set: it will be set after collecting the statistics from the slaves at the end of [PF_Processor\(\)](#). (TU 30 Jun 2011)

Definition at line 864 of file [parallel.c](#).

References [FlushOut\(\)](#), and [PutOut\(\)](#).

Referenced by [EndSort\(\)](#).

Here is the call graph for this function:

**4.95.4.12 PF_Deferred()**

```
WORD PF_Deferred (
    WORD * term,
    WORD level )
```

Replaces [Deferred\(\)](#) on the slaves.

Parameters

<i>term</i>	the term that must be multiplied by the contents of the current bracket.
<i>level</i>	the compiler level.

Returns

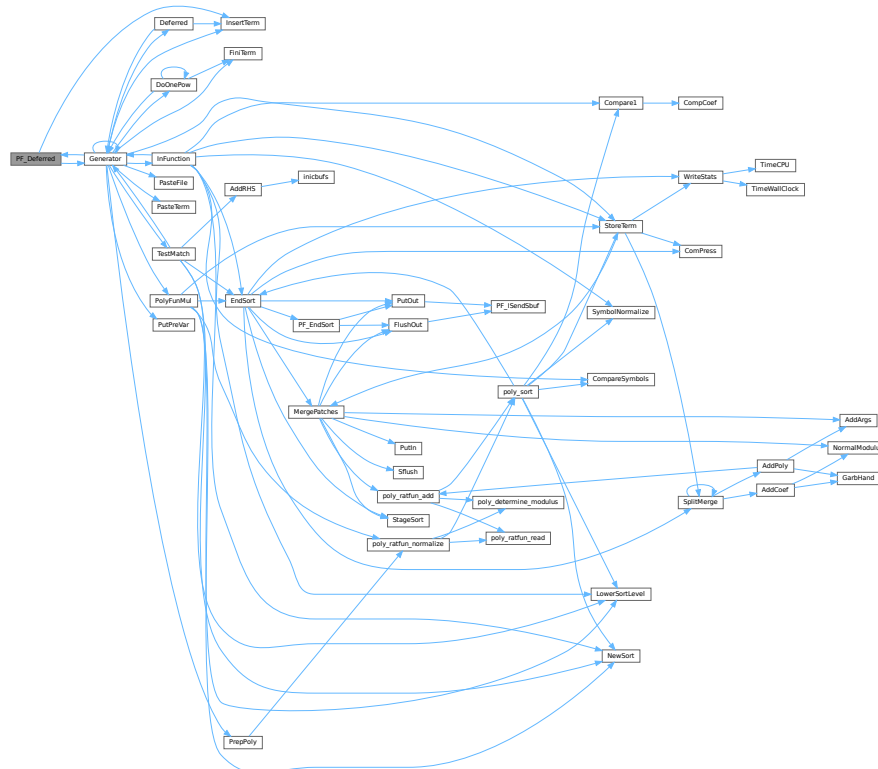
0 if OK, nonzero on error.

Definition at line 1208 of file [parallel.c](#).

References [Generator\(\)](#), and [InsertTerm\(\)](#).

Referenced by [Generator\(\)](#).

Here is the call graph for this function:



4.95.4.13 PF_Processor()

```
int PF_Processor (
    EXPRESSIONS e,
    WORD i,
    WORD LastExpression )
```

Replaces parts of [Processor\(\)](#) on the masters and slaves. On the master [PF_Processor\(\)](#) is responsible for proper distribution of terms from the input file to the slaves. On the slaves it calls [Generator\(\)](#) for all the terms that this process gets, but [PF_GetTerm\(\)](#) gets terms from the master (not directly from infile).

Parameters

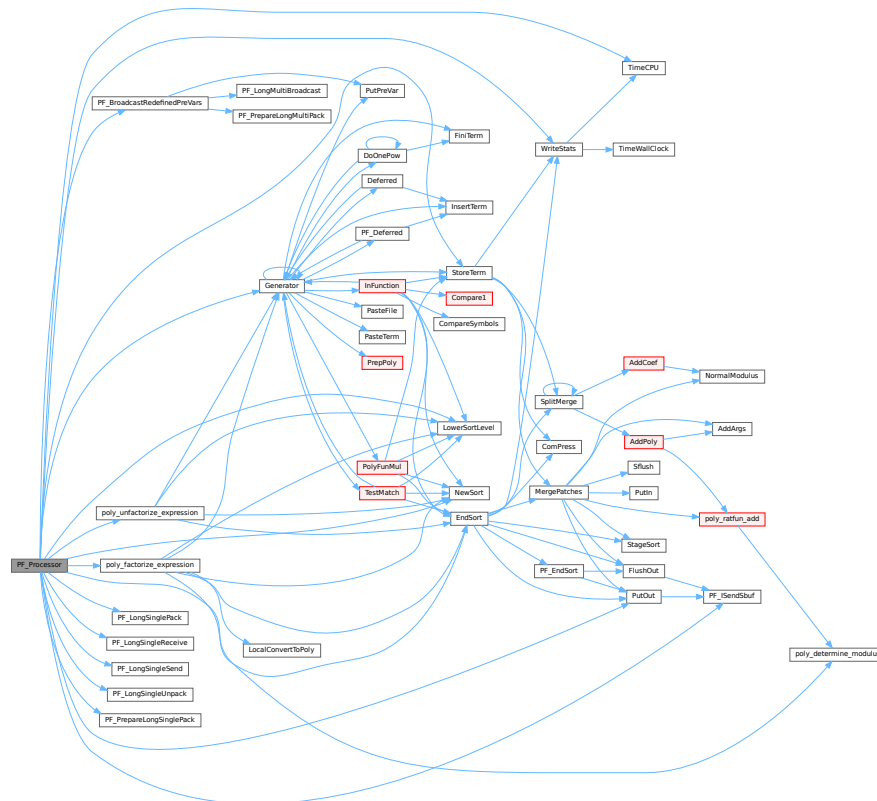
<i>e</i>	The pointer to the current expression.
<i>i</i>	The index for the current expression.
<i>LastExpression</i>	The flag indicating whether it is the last expression.

Returns

0 if OK, nonzero on error.

References [EndSort\(\)](#), [Generator\(\)](#), [LowerSortLevel\(\)](#), [NewSort\(\)](#), [PACK_LONG](#), [PF_BroadcastRedefinedPreVars\(\)](#), [PF_ISendSbuf\(\)](#), [PF_LongSinglePack\(\)](#), [PF_LongSingleReceive\(\)](#), [PF_LongSingleSend\(\)](#), [PF_LongSingleUnpack\(\)](#), [PF_PrepareLongSinglePack\(\)](#), [poly_factorize_expression\(\)](#), [poly_unfactorize_expression\(\)](#), [PutOut\(\)](#), [StoreTerm\(\)](#), [SWAP](#), [TimeCPU\(\)](#), and [WriteStats\(\)](#).

Here is the call graph for this function:



```
int PF_Init (
    int * argc,
    char *** argv )
```

Parameters

<i>argc</i>	pointer to the number of arguments.
<i>argv</i>	pointer to the arguments.

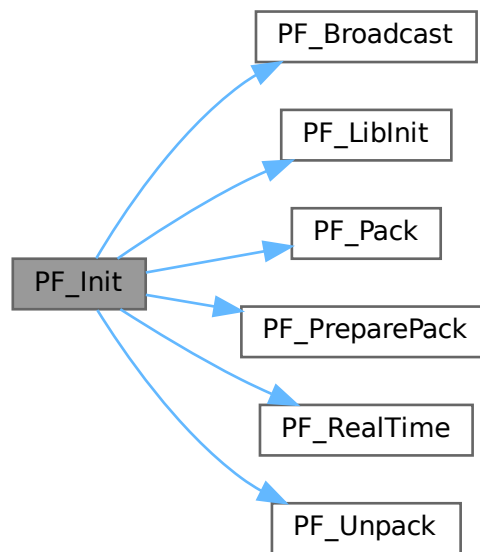
Returns

0 if OK, nonzero on error.

Definition at line 1965 of file [parallel.c](#).

References [PF_Broadcast\(\)](#), [PF_LibInit\(\)](#), [PF_Pack\(\)](#), [PF_PreparePack\(\)](#), [PF_RealTime\(\)](#), and [PF_Unpack\(\)](#).

Here is the call graph for this function:

**4.95.4.15 PF_Terminate()**

```
int PF_Terminate (
    int errorcode )
```

Performs the finalization of ParFORM. To be called by `Terminate()`.

Parameters

<i>error</i>	an error code.

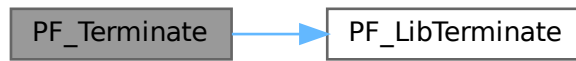
Returns

0 if OK, nonzero on error.

Definition at line 2060 of file [parallel.c](#).

References [PF_LibTerminate\(\)](#).

Here is the call graph for this function:



4.95.4.16 PF_GetSlaveTimes()

```
LONG PF_GetSlaveTimes (
    void )
```

Returns the total CPU time of all slaves together. This function must be called on the master and all slaves.

Returns

on the master, the sum of CPU times on all slaves.

Definition at line 2076 of file [parallel.c](#).

References [TimeCPU\(\)](#).

Here is the call graph for this function:



4.95.4.17 PF_BroadcastNumber()

```
LONG PF_BroadcastNumber (
    LONG x )
```

Broadcasts a LONG value from the master to the all slaves.

Parameters

x	the number to be broadcast (set on the master).

Returns

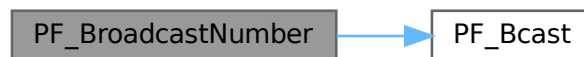
the synchronised result.

Definition at line 2096 of file [parallel.c](#).

References [PF_Bcast\(\)](#).

Referenced by [DoCheckpoint\(\)](#), [PF_BroadcastBuffer\(\)](#), and [StartVariables\(\)](#).

Here is the call graph for this function:

**4.95.4.18 PF_BroadcastBuffer()**

```
void PF_BroadcastBuffer (
    WORD ** buffer,
    LONG * length )
```

Broadcasts a buffer from the master to all the slaves.

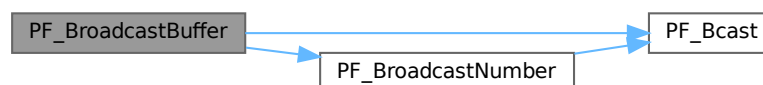
Parameters

<code>in, out</code>	<code>buffer</code>	on the master, the buffer to be broadcast. On the slaves, the buffer will be allocated if the length is greater than 0. The caller must free it.
<code>in, out</code>	<code>length</code>	on the master, the length of the buffer to be broadcast. On the slaves, it receives the length of transferred buffer. The actual transfer occurs only if the length is greater than 0.

Definition at line 2123 of file [parallel.c](#).

References [PF_Bcast\(\)](#), and [PF_BroadcastNumber\(\)](#).

Here is the call graph for this function:



4.95.4.19 PF_BroadcastString()

```
int PF_BroadcastString (
    UBYTE * str )
```

Broadcasts a string from the master to all slaves.

Parameters

<i>in, out</i>	<i>str</i>	The pointer to a null-terminated string.

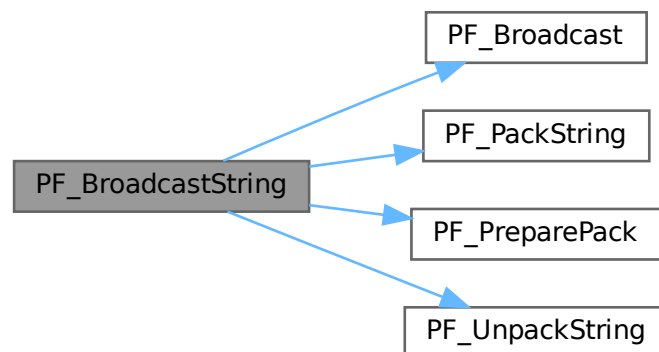
Returns

0 if OK, nonzero on error.

Definition at line 2165 of file [parallel.c](#).

References [PF_Broadcast\(\)](#), [PF_PackString\(\)](#), [PF_PreparePack\(\)](#), and [PF_UnpackString\(\)](#).

Here is the call graph for this function:



4.95.4.20 PF_BroadcastPreDollar()

```
int PF_BroadcastPreDollar (
    WORD ** dbuffer,
    LONG * newsize,
    int * numterms )
```

Broadcasts dollar variables set as a preprocessor variables. Only the master is able to make an assignment like `#$a=g;` where `g` is an expression: only the master has an access to the expression. So, the master broadcasts the result to slaves.

The result is in `*dbuffer` of the size is `*newsize` (in number of WORDs), +1 for trailing zero. For slave `newsize` and `numterms` are output parameters.

Parameters

<i>in, out</i>	<i>dbuffer</i>	the buffer for a dollar variable.
<i>in, out</i>	<i>newsize</i>	the size of the dollar variable in WORDs.
<i>in, out</i>	<i>numterms</i>	the number of terms in the dollar variable.

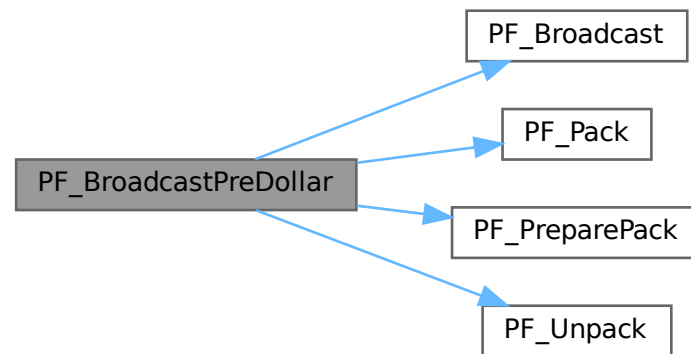
Returns

0 if OK, nonzero on error.

Definition at line 2220 of file [parallel.c](#).

References [PF_Broadcast\(\)](#), [PF_Pack\(\)](#), [PF_PreparePack\(\)](#), and [PF_Unpack\(\)](#).

Here is the call graph for this function:

**4.95.4.21 PF_CollectModifiedDollars()**

```
int PF_CollectModifiedDollars (
    void )
```

Combines modified dollar variables on the all slaves, and store them into those on the master.

The potentially modified dollar variables are given in `PotModdollars`, and the number of them is given by `NumPotModdollars`.

The current module could be executed in parallel only if all potentially modified variables are listed in `ModOptdollars`, otherwise the module was switched to the sequential mode.

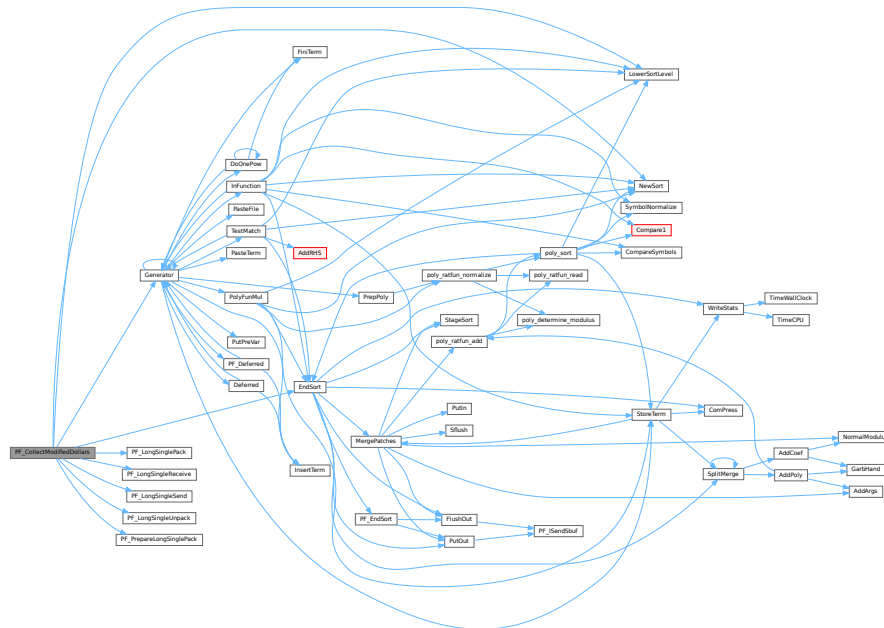
Returns

0 if OK, nonzero on error.

Definition at line 2507 of file [parallel.c](#).

References [EndSort\(\)](#), [Generator\(\)](#), [LowerSortLevel\(\)](#), [NewSort\(\)](#), [PF_LongSinglePack\(\)](#), [PF_LongSingleReceive\(\)](#), [PF_LongSingleSend\(\)](#), [PF_LongSingleUnpack\(\)](#), [PF_PrepareLongSinglePack\(\)](#), [VectorInit](#), [VectorPtr](#), [VectorReserve](#), and [VectorSize](#).

Here is the call graph for this function:



4.95.4.22 PF_BroadcastModifiedDollars()

```
int PF_BroadcastModifiedDollars (
    void )
```

Broadcasts modified dollar variables on the master to the all slaves.

The potentially modified dollar variables are given in PotModdollars, and the number of them is given by NumPotModdollars.

The current module could be executed in parallel only if all potentially modified variables are listed in ModOptdollars, otherwise the module was switched to the sequential mode. In either cases, we need to broadcast them.

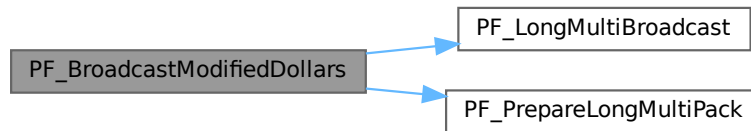
Returns

0 if OK, nonzero on error.

Definition at line 2786 of file [parallel.c](#).

References [PF_LongMultiBroadcast\(\)](#), and [PF_PrepareLongMultiPack\(\)](#).

Here is the call graph for this function:

**4.95.4.23 PF_BroadcastRedefinedPreVars()**

```
int PF_BroadcastRedefinedPreVars (
    void )
```

Broadcasts preprocessor variables, which were changed by the Redefine statements in the current module, from the master to the all slaves.

The potentially redefined preprocessor variables are given in `AC.pfirstnum`, and the number of them is given by `AC.numpfirstnum`. For an actually redefined variable, the corresponding value in `AC.inputnumbers` is non-negative.

Returns

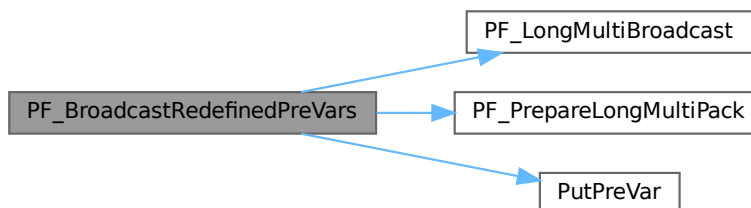
0 if OK, nonzero on error.

Definition at line 3003 of file [parallel.c](#).

References [PF_LongMultiBroadcast\(\)](#), [PF_PrepareLongMultiPack\(\)](#), [PutPreVar\(\)](#), [VectorPtr](#), and [VectorReserve](#).

Referenced by [PF_Processor\(\)](#).

Here is the call graph for this function:



4.95.4.24 PF_StoreInsideInfo()

```
int PF_StoreInsideInfo (
    void )
```

Definition at line 3093 of file [parallel.c](#).

4.95.4.25 PF_RestoreInsideInfo()

```
int PF_RestoreInsideInfo (
    void )
```

Definition at line 3119 of file [parallel.c](#).

4.95.4.26 PF_BroadcastCBuf()

```
int PF_BroadcastCBuf (
    int bufnum )
```

Broadcasts a compiler buffer specified by *bufnum* from the master to the all slaves.

Parameters

<i>bufnum</i>	The index of the compiler buffer to be broadcast.

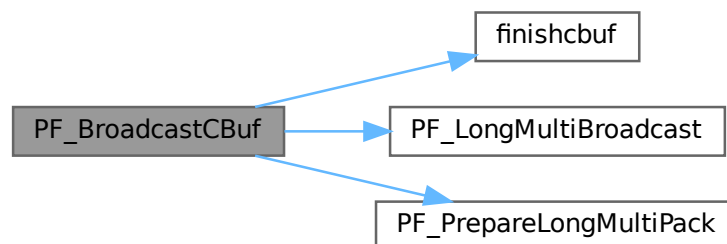
Returns

0 if OK, nonzero on error.

Definition at line 3145 of file [parallel.c](#).

References [CbUf::boomlijst](#), [CbUf::Buffer](#), [CbUf::BufferSize](#), [CbUf::CanCommu](#), [CbUf::dimension](#), [finishcbuf\(\)](#), [CbUf::lhs](#), [CbUf::numdum](#), [CbUf::NumTerms](#), [PF_LongMultiBroadcast\(\)](#), [PF_PrepereLongMultiPack\(\)](#), [CbUf::Pointer](#), [CbUf::rhs](#), and [CbUf::Top](#).

Here is the call graph for this function:



4.95.4.27 PF_BroadcastExpFlags()

```
int PF_BroadcastExpFlags (
    void )
```

Broadcasts AR.expflags and several properties of each expression, e.g., e->vflags, from the master to all slaves.

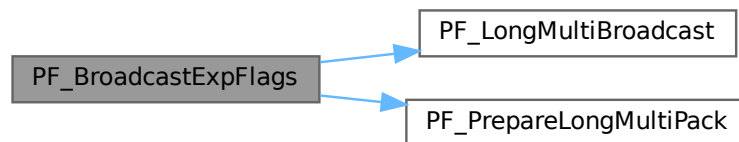
Returns

0 if OK, nonzero on error.

Definition at line 3256 of file [parallel.c](#).

References [PF_LongMultiBroadcast\(\)](#), and [PF_PrepareLongMultiPack\(\)](#).

Here is the call graph for this function:



4.95.4.28 PF_BroadcastExpr()

```
int PF_BroadcastExpr (
    EXPRESSIONS e,
    FILEHANDLE * file )
```

Broadcasts an expression from the master to the all slaves.

Parameters

<i>e</i>	The expression to be broadcast.
<i>file</i>	The file in which the expression is sitting.

Returns

0 if OK, nonzero on error.

Definition at line 3550 of file [parallel.c](#).

Referenced by [PF_BroadcastRHS\(\)](#).

4.95.4.29 PF_BroadcastRHS()

```
int PF_BroadcastRHS (  
    void )
```

Broadcasts expressions appearing in the right-hand side from the master to the all slaves.

Returns

0 if OK, nonzero on error.

Definition at line 3578 of file [parallel.c](#).

References [PF_BroadcastExpr\(\)](#).

Referenced by [Processor\(\)](#).

Here is the call graph for this function:



4.95.4.30 PF_InParallelProcessor()

```
int PF_InParallelProcessor (  
    void )
```

Processes expressions in the InParallel mode, i.e., dividing expressions marked by partodo over the slaves.

Returns

0 if OK, nonzero on error.

Definition at line 3625 of file [parallel.c](#).

Referenced by [Processor\(\)](#).

4.95.4.31 PF_SendFile()

```
int PF_SendFile (  
    int to,  
    FILE * fd )
```

Sends a file to the process specified by *to*.

Parameters

<i>to</i>	the destination process number.
<i>fd</i>	the file to be sent.

Returns

the size of sent data in bytes, or -1 on error.

Definition at line 4222 of file [parallel.c](#).

References [PF_RawSend\(\)](#).

Referenced by [DoCheckpoint\(\)](#).

Here is the call graph for this function:

**4.95.4.32 PF_RecvFile()**

```
int PF_RecvFile (
    int from,
    FILE * fd )
```

Receives a file from the process specified by *from*.

Parameters

<i>from</i>	the source process number.
<i>fd</i>	the file to save the received data.

Returns

the size of received data in bytes, or -1 on error.

Definition at line 4260 of file [parallel.c](#).

References [PF_RawRecv\(\)](#).

Referenced by [DoCheckpoint\(\)](#).

Here is the call graph for this function:



4.95.4.33 PF_MLock()

```
void PF_MLock (
    void )
```

A function called by MLOCK(ErrorMessageLock) for slaves.

Definition at line 4341 of file [parallel.c](#).

References [VectorClear](#).

4.95.4.34 PF_MUnlock()

```
void PF_MUnlock (
    void )
```

A function called by MUNLOCK(ErrorMessageLock) for slaves.

Definition at line 4357 of file [parallel.c](#).

References [PF_RawSend\(\)](#), [VectorEmpty](#), [VectorPtr](#), and [VectorSize](#).

Here is the call graph for this function:



4.95.4.35 PF_WriteFileToFile()

```
LONG PF_WriteFileToFile (
    int handle,
    UBYTE * buffer,
    LONG size )
```

Replaces WriteFileToFile() on the master and slaves.

It copies the given buffer into internal buffers if called between MLOCK(ErrorMessageLock) and MUNLOCK(ErrorMessageLock) for slaves and handle is StdOut or LogHandle, otherwise calls WriteFileToFile().

Parameters

<i>handle</i>	a file handle that specifies the output.
<i>buffer</i>	a pointer to the source buffer containing the data to be written.
<i>size</i>	the size of data to be written in bytes.

Returns

the actual size of data written to the output in bytes.

Definition at line [4386](#) of file [parallel.c](#).

References [VectorClear](#), [VectorPtr](#), [VectorPushBacks](#), and [VectorSize](#).

4.95.4.36 PF_FlushStdOutBuffer()

```
void PF_FlushStdOutBuffer (
    void )
```

Explicitly Flushes the buffer for the standard output on the master, which is used if PF_ENABLE_STDOUT_↵
BUFFERING is defined.

Definition at line [4480](#) of file [parallel.c](#).

References [VectorClear](#), [VectorPtr](#), and [VectorSize](#).

4.95.4.37 PF_FreeErrorMessageBuffers()

```
void PF_FreeErrorMessageBuffers (
    void )
```

Frees the buffers allocated for the synchronized output.

Currently, not used anywhere, but could be used in [PF_Terminate\(\)](#).

Definition at line [4616](#) of file [parallel.c](#).

References [VectorFree](#).

4.95.5 Variable Documentation**4.95.5.1 PF**

[PARALLELVARS](#) PF

Definition at line [90](#) of file [parallel.c](#).

4.96 parallel.c

[Go to the documentation of this file.](#)

```

00001
00011 /* #[ License : */
00012 /*
00013 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00014 *   When using this file you are requested to refer to the publication
00015 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00016 *   This is considered a matter of courtesy as the development was paid
00017 *   for by FOM the Dutch physics granting agency and we would like to
00018 *   be able to track its scientific use to convince FOM of its value
00019 *   for the community.
00020 *
00021 *   This file is part of FORM.
00022 *
00023 *   FORM is free software: you can redistribute it and/or modify it under the
00024 *   terms of the GNU General Public License as published by the Free Software
00025 *   Foundation, either version 3 of the License, or (at your option) any later
00026 *   version.
00027 *
00028 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00029 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00030 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00031 *   details.
00032 *
00033 *   You should have received a copy of the GNU General Public License along
00034 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00035 */
00036 /* #[ License : */
00037 /*
00038 *   #[ includes :
00039 */
00040 #include "form3.h"
00041 #include "vector.h"
00042
00043 /*
00044 #define PF_DEBUG_BCAST_LONG
00045 #define PF_DEBUG_BCAST_BUF
00046 #define PF_DEBUG_BCAST_PREDOLLAR
00047 #define PF_DEBUG_BCAST_RHSEXP
00048 #define PF_DEBUG_BCAST_DOLLAR
00049 #define PF_DEBUG_BCAST_PREVAR
00050 #define PF_DEBUG_BCAST_CBUF
00051 #define PF_DEBUG_BCAST_EXPRFLAGS
00052 #define PF_DEBUG_REDUCE_DOLLAR
00053 */
00054
00055 /* mpi.c */
00056 LONG PF_RealTime(int);
00057 int PF_LibInit(int*, char**);
00058 int PF_LibTerminate(int);
00059 int PF_Probe(int*);
00060 int PF_RecvWbuf(WORD*, LONG*, int*);
00061 int PF_IRecvRbuf(PF_BUFFER*, int, int);
00062 int PF_WaitRbuf(PF_BUFFER *, int, LONG *);
00063 int PF_RawSend(int dest, void *buf, LONG l, int tag);
00064 LONG PF_RawRecv(int *src, void *buf, LONG thesize, int *tag);
00065 int PF_RawProbe(int *src, int *tag, int *bytesize);
00066
00067 /* Private functions */
00068
00069 static int PF_WaitAllSlaves(void);
00070
00071 static void PF_PackRedefinedPreVars(void);
00072 static void PF_UnpackRedefinedPreVars(void);
00073
00074 static int PF_Wait4MasterIP(int tag);
00075 static int PF_DoOneExpr(void);
00076 static int PF_ReadMaster(void); /*reads directly to its scratch!*/
00077 static int PF_Slave2MasterIP(int src); /*both master and slave*/
00078 static int PF_Master2SlaveIP(int dest, EXPRESSIONS e);
00079 static int PF_WalkThrough(WORD *t, LONG l, LONG chunk, LONG *count);
00080 static int PF_SendChunkIP(FILEHANDLE *curfile, POSITION *position, int to, LONG thesize);
00081 static int PF_RecvChunkIP(FILEHANDLE *curfile, int from, LONG thesize);
00082
00083 static void PF_ReceiveErrorMessage(int src, int tag);
00084 static void PF_CatchErrorMessages(int *src, int *tag);
00085 static void PF_CatchErrorMessagesForAll(void);
00086 static int PF_ProbeWithCatchingErrorMessages(int *src);
00087
00088 /* Variables */
00089
00090 PARALLELVARS PF;
00091 #ifdef MPI2

```

```

00092 WORD *PF_shared_buff;
00093 #endif
00094
00095 static LONG PF_goutterms; /* (master) Total out terms at PF_EndSort(), used in PF_Statistics().
    */
00096 static POSITION PF_exprsize; /* (master) The size of the expression at PF_EndSort(), used in
    PF_Processor(). */
00097
00098 /*
00099 This will work well only under Linux, see
00100 #ifdef PF_WITH_SCHED_YIELD
00101 below in PF_WaitAllSlaves().
00102 */
00103 #ifdef PF_WITH_SCHED_YIELD
00104 #include <sched.h>
00105 #endif
00106
00107 #ifdef PF_WITHLOG
00108 #define PRINTFBUF(TEXT,TERM,SIZE) { UBYTE lbuf[24]; if(PF.log){ WORD iii;\
00109 NumToStr(lbuf,AC.CModule); \
00110 fprintf(stderr,"%d|s] %s : ",PF.me,lbuf,(char*)TEXT);\
00111 if(TERM){ fprintf(stderr,"%d] ",(int)(*TERM));\
00112 if((SIZE)<500 && (SIZE)>0) for(iii=1;iii<(SIZE);iii++)\
00113 fprintf(stderr,"%d ",TERM[iii]); }\
00114 fprintf(stderr,"\n");\
00115 fflush(stderr); } }
00116 #else
00117 #define PRINTFBUF(TEXT,TERM,SIZE) {}
00118 #endif
00119
00124 #define SWAP(x, y) \
00125 do { \
00126 char swap_tmp__[sizeof(x) == sizeof(y) ? (int)sizeof(x) : -1]; \
00127 memcpy(swap_tmp__, &y, sizeof(x)); \
00128 memcpy(&y, &x, sizeof(x)); \
00129 memcpy(&x, swap_tmp__, sizeof(x)); \
00130 } while (0)
00131
00135 #define PACK_LONG(p, n) \
00136 do { \
00137 * (p)++ = (UWORD)((ULONG)(n) & (ULONG)WORDMASK); \
00138 * (p)++ = (UWORD)((ULONG)(n) >> BITSINWORD & (ULONG)WORDMASK); \
00139 } while (0)
00140
00144 #define UNPACK_LONG(p, n) \
00145 do { \
00146 (n) = (LONG)((((ULONG)(p)[1] & (ULONG)WORDMASK) << BITSINWORD) | ((ULONG)(p)[0] &
    (ULONG)WORDMASK)); \
00147 (p) += 2; \
00148 } while (0)
00149
00153 #define CHECK(condition) _CHECK(condition, __FILE__, __LINE__)
00154 #define _CHECK(condition, file, line) __CHECK(condition, file, line)
00155 #define __CHECK(condition, file, line) \
00156 do { \
00157 if ( !(condition) ) { \
00158 Error0("Fatal error at " file ":" #line); \
00159 Terminate(-1); \
00160 } \
00161 } while (0)
00162
00163 /*
00164 * For debugging.
00165 */
00166 #define DBGOUT(lv1, lv2, a) do { if ( lv1 >= lv2 ) { printf a; fflush(stdout); } } while (0)
00167
00168 /* (AN.ninterms of master) == max(AN.ninterms of slaves) == sum(PF_linterms of slaves) at EndSort().
    */
00169 #define DBGOUT_NINTERMS(lv, a)
00170 /* #define DBGOUT_NINTERMS(lv, a) DBGOUT(1, lv, a) */
00171
00172 /*
00173 #] includes :
00174 #[ statistics :
00175 #] variables : (should be part of a struct?)
00176 */
00177 static LONG PF_linterms; /* local interms on this proces: PF_Proces */
00178 #define PF_STATS_SIZE 5
00179 static LONG **PF_stats = NULL; /* space for collecting statistics of all procs */
00180 static LONG PF_laststat; /* last realtime when statistics were printed */
00181 static LONG PF_statsinterval; /* timeinterval for printing statistics */
00182 /*
00183 #] variables :
00184 #[ PF_Statistics :
00185 */
00186
00195 static int PF_Statistics(LONG **stats, int proc)

```

```

00196 {
00197     GETIDENTITY
00198     LONG real, cpu;
00199     WORD rpart, cpart;
00200     int i, j;
00201
00202     if ( AT.SS == AM.S0 && PF.me == MASTER ) {
00203         real = PF_RealTime(PF_TIME); rpart = (WORD)(real%100); real /= 100;
00204
00205         if ( PF_stats == NULL ) {
00206             PF_stats = (LONG**)Malloc1(PF.numtasks*sizeof(LONG*), "PF_stats 1");
00207             for ( i = 0; i < PF.numtasks; i++ ) {
00208                 PF_stats[i] = (LONG*)Malloc1(PF_STATS_SIZE*sizeof(LONG), "PF_stats 2");
00209                 for ( j = 0; j < PF_STATS_SIZE; j++ ) PF_stats[i][j] = 0;
00210             }
00211         }
00212         if ( proc > 0 ) for ( i = 0; i < PF_STATS_SIZE; i++ ) PF_stats[proc][i] = stats[0][i];
00213
00214         if ( real >= PF_laststat + PF_statsinterval || proc == 0 ) {
00215             LONG sum[PF_STATS_SIZE];
00216
00217             for ( i = 0; i < PF_STATS_SIZE; i++ ) sum[i] = 0;
00218             sum[0] = cpu = TimeCPU(1);
00219             cpart = (WORD)(cpu%1000);
00220             cpu /= 1000;
00221             cpart /= 10;
00222             if ( AC.OldParallelStats ) MesPrint("");
00223             if ( proc > 0 && AC.StatsFlag && AC.OldParallelStats ) {
00224                 MesPrint("proc      CPU      in      gen      left      byte");
00225                 MesPrint("%3d : %7l.%2i %10l", 0, cpu, cpart, AN.ninterms);
00226             }
00227             else if ( AC.StatsFlag && AC.OldParallelStats ) {
00228                 MesPrint("proc      CPU      in      gen      out      byte");
00229                 MesPrint("%3d : %7l.%2i %10l %10l %10l", 0, cpu, cpart, AN.ninterms, 0, PF_goutterms);
00230             }
00231
00232             for ( i = 1; i < PF.numtasks; i++ ) {
00233                 cpart = (WORD)(PF_stats[i][0]%1000);
00234                 cpu = PF_stats[i][0] / 1000;
00235                 cpart /= 10;
00236                 if ( AC.StatsFlag && AC.OldParallelStats )
00237                     MesPrint("%3d : %7l.%2i %10l %10l %10l", i, cpu, cpart,
00238                             PF_stats[i][2], PF_stats[i][3], PF_stats[i][4]);
00239                 for ( j = 0; j < PF_STATS_SIZE; j++ ) sum[j] += PF_stats[i][j];
00240             }
00241             cpart = (WORD)(sum[0]%1000);
00242             cpu = sum[0] / 1000;
00243             cpart /= 10;
00244             if ( AC.StatsFlag && AC.OldParallelStats ) {
00245                 MesPrint("Sum = %7l.%2i %10l %10l %10l", cpu, cpart, sum[2], sum[3], sum[4]);
00246                 MesPrint("Real = %7l.%2i %20s (%1) %16s",
00247                         real, rpart, AC.Commercial, AC.CModule, EXPRNAME(AR.CurExpr));
00248                 MesPrint("");
00249             }
00250             PF_laststat = real;
00251         }
00252     }
00253     return(0);
00254 }
00255 /*
00256     #] PF_Statistics :
00257     #] statistics :
00258     #[ sort.c :
00259     #[ sort variables :
00260 */
00261
00262 typedef struct NoDe {
00263     struct NoDe *left;
00264     struct NoDe *rghet;
00265     int lloser;
00266     int rloser;
00267     int lsrc;
00268     int rsrc;
00269 } NODE;
00270
00271 /*
00272     should/could be put in one struct
00273 */
00274 static NODE *PF_root; /* root of tree of losers */
00275 static WORD PF_loser; /* this is the last loser */
00276 static WORD **PF_term; /* these point to the active terms */
00277 static WORD **PF_newcpo; /* new coeffs of merged terms */
00278 static WORD *PF_newclen; /* length of new coefficients */
00279
00280 /*
00281     preliminary: could also write somewhere else?
00282 */

```

```

00286
00287 static WORD *PF_WorkSpace; /* used in PF_EndSort() */
00288 static UWORD *PF_ScratchSpace; /* used in PF_GetLoser() */
00289
00290 /*
00291     #] sort variables :
00292     #[ PF_AllocBuf :
00293 */
00294
00311 static PF_BUFFER *PF_AllocBuf(int nbufs, LONG bsize, WORD free)
00312 {
00313     PF_BUFFER *buf;
00314     UBYTE *p, *stop;
00315     LONG allocsize;
00316     int i;
00317
00318     allocsize =
00319         (LONG)(sizeof(PF_BUFFER) + 4*nbufs*sizeof(WORD*) + (nbufs-free)*bsize);
00320
00321     allocsize +=
00322         (LONG)( nbufs * ( 2 * sizeof(MPI_Status)
00323                         + sizeof(MPI_Request)
00324                         + sizeof(MPI_Datatype)
00325                     ) );
00326     allocsize += (LONG)( nbufs * 3 * sizeof(int) );
00327
00328     if ( ( buf = (PF_BUFFER*)Malloc1(allocsize,"PF_AllocBuf") ) == NULL ) return(NULL);
00329
00330     p = ((UBYTE *)buf) + sizeof(PF_BUFFER);
00331     stop = ((UBYTE *)buf) + allocsize;
00332
00333     buf->nbufs = nbufs;
00334     buf->active = 0;
00335
00336     buf->buff = (WORD**)p; p += buf->nbufs*sizeof(WORD*);
00337     buf->fill = (WORD**)p; p += buf->nbufs*sizeof(WORD*);
00338     buf->full = (WORD**)p; p += buf->nbufs*sizeof(WORD*);
00339     buf->stop = (WORD**)p; p += buf->nbufs*sizeof(WORD*);
00340     buf->status = (MPI_Status *)p; p += buf->nbufs*sizeof(MPI_Status);
00341     buf->retstat = (MPI_Status *)p; p += buf->nbufs*sizeof(MPI_Status);
00342     buf->request = (MPI_Request *)p; p += buf->nbufs*sizeof(MPI_Request);
00343     buf->type = (MPI_Datatype *)p; p += buf->nbufs*sizeof(MPI_Datatype);
00344     buf->index = (int *)p; p += buf->nbufs*sizeof(int);
00345
00346     for ( i = 0; i < buf->nbufs; i++ ) buf->request[i] = MPI_REQUEST_NULL;
00347     buf->tag = (int *)p; p += buf->nbufs*sizeof(int);
00348     buf->from = (int *)p; p += buf->nbufs*sizeof(int);
00349 /*
00350     and finally the real bufferspace
00351 */
00352     for ( i = free; i < buf->nbufs; i++ ) {
00353         buf->buff[i] = (WORD*)p; p += bsize;
00354         buf->stop[i] = (WORD*)p;
00355         buf->fill[i] = buf->full[i] = buf->buff[i];
00356     }
00357     if ( p != stop ) {
00358         MesPrint("Error in PF_AllocBuf p = %x stop = %x\n",p,stop);
00359         return(NULL);
00360     }
00361     return(buf);
00362 }
00363
00364 /*
00365     #] PF_AllocBuf :
00366     #[ PF_InitTree :
00367 */
00368
00380 static int PF_InitTree(void)
00381 {
00382     GETIDENTITY
00383     PF_BUFFER **rbuf = PF.rbufs;
00384     UBYTE *p, *stop;
00385     int numrbufs,numtasks = PF.numtasks;
00386     int i, j, src, numnodes;
00387     int numslaves = numtasks - 1;
00388     LONG size;
00389 /*
00390     #] the buffers : for the new coefficients and the terms
00391     we need one for each slave
00392 */
00393     if ( PF_term == NULL ) {
00394         size = 2*numtasks*sizeof(WORD*) + sizeof(WORD)*
00395             ( numtasks*(1 + AM.MaxTal) + (AM.MaxTer/sizeof(WORD)+1) + 2*(AM.MaxTal+2) );
00396
00397         PF_term = (WORD **)Malloc1(size,"PF_term");
00398         stop = ((UBYTE*)PF_term) + size;
00399         p = ((UBYTE*)PF_term) + numtasks*sizeof(WORD*);

```

```

00400
00401     PF_newcpos = (WORD **)p; p += sizeof(WORD*) * numtasks;
00402     PF_newclen = (WORD *)p; p += sizeof(WORD) * numtasks;
00403     for ( i = 0; i < numtasks; i++ ) {
00404         PF_newcpos[i] = (WORD *)p; p += sizeof(WORD)*AM.MaxTal;
00405         PF_newclen[i] = 0;
00406     }
00407     PF_WorkSpace = (WORD *)p; p += AM.MaxTer+sizeof(WORD);
00408     PF_ScratchSpace = (UWORD*)p; p += 2*(AM.MaxTal+2)*sizeof(UWORD);
00409
00410     if ( p != stop ) { MesPrint("error in PF_InitTree"); return(-1); }
00411 }
00412 /*
00413     #] the buffers :
00414     #[ the receive buffers :
00415 */
00416     numrbufs = PF.numrbufs;
00417 /*
00418     this is the size we have in the combined sortbufs for one slave
00419 */
00420     size = (AT.SS->sTop2 - AT.SS->lBuffer - 1)/(PF.numtasks - 1);
00421
00422     if ( rbuf == NULL ) {
00423         if ( ( rbuf = (PF_BUFFER**)Malloca1(numtasks*sizeof(PF_BUFFER*), "Master: rbufs") ) == NULL )
00424             return(-1);
00425         if ( ( rbuf[0] = PF_AllocBuf(1,0,1) ) == NULL ) return(-1);
00426         for ( i = 1; i < numtasks; i++ ) {
00427             if (!(rbuf[i] = PF_AllocBuf(numrbufs,sizeof(WORD)*size,1))) return(-1);
00428         }
00429         rbuf[0]->buff[0] = AT.SS->lBuffer;
00430         rbuf[0]->full[0] = rbuf[0]->fill[0] = rbuf[0]->buff[0];
00431         rbuf[0]->stop[0] = rbuf[1]->buff[0] = rbuf[0]->buff[0] + 1;
00432         rbuf[1]->full[0] = rbuf[1]->fill[0] = rbuf[1]->buff[0];
00433         for ( i = 2; i < numtasks; i++ ) {
00434             rbuf[i-1]->stop[0] = rbuf[i]->buff[0] = rbuf[i-1]->buff[0] + size;
00435             rbuf[i]->full[0] = rbuf[i]->fill[0] = rbuf[i]->buff[0];
00436         }
00437         rbuf[numtasks-1]->stop[0] = rbuf[numtasks-1]->buff[0] + size;
00438
00439         for ( i = 1; i < numtasks; i++ ) {
00440             for ( j = 0; j < rbuf[i]->numbufs; j++ ) {
00441                 rbuf[i]->full[j] = rbuf[i]->fill[j] = rbuf[i]->buff[j] + AM.MaxTer/sizeof(WORD) + 2;
00442             }
00443             PF_term[i] = rbuf[i]->fill[rbuf[i]->active];
00444             *PF_term[i] = 0;
00445             PF_IRcvRbuf(rbuf[i],rbuf[i]->active,i);
00446         }
00447         rbuf[0]->active = 0;
00448         PF_term[0] = rbuf[0]->buff[0];
00449         PF_term[0][0] = 0; /* PF_term[0] is used for a zero term. */
00450         PF.rbufs = rbuf;
00451 /*
00452         #] the receive buffers :
00453         #[ the actual tree :
00454
00455         calculate number of nodes in mergetree and allocate space for them
00456 */
00457         if ( numslaves < 3 ) numnodes = 1;
00458         else {
00459             numnodes = 2;
00460             while ( numnodes < numslaves ) numnodes *= 2;
00461             numnodes -= 1;
00462         }
00463
00464         if ( PF_root == NULL )
00465             if ( ( PF_root = (NODE*)Malloca1(sizeof(NODE)*numnodes,"nodes in mergetree") ) == NULL )
00466                 return(-1);
00467 /*
00468         then initialize all the nodes
00469 */
00470         src = 1;
00471         for ( i = 0; i < numnodes; i++ ) {
00472             if ( 2*(i+1) <= numnodes ) {
00473                 PF_root[i].left = &(PF_root[2*(i+1)-1]);
00474                 PF_root[i].lsrc = 0;
00475             }
00476             else {
00477                 PF_root[i].left = 0;
00478                 if ( src < numtasks ) PF_root[i].lsrc = src++;
00479                 else PF_root[i].lsrc = 0;
00480             }
00481             PF_root[i].lloser = 0;
00482         }
00483         for ( i = 0; i < numnodes; i++ ) {
00484             if ( 2*(i+1)+1 <= numnodes ) {
00485                 PF_root[i].rght = &(PF_root[2*(i+1)]);

```

```

00486         PF_root[i].rsrc = 0;
00487     }
00488     else {
00489         PF_root[i].rghet = 0;
00490         if (src<numtasks) PF_root[i].rsrc = src++;
00491         else PF_root[i].rsrc = 0;
00492     }
00493     PF_root[i].rloser = 0;
00494 }
00495 /*
00496     #] the actual tree :
00497 */
00498 return(numnodes);
00499 }
00500
00501 /*
00502     #] PF_InitTree :
00503     #[ PF_PutIn :
00504 */
00505
00524 static WORD *PF_PutIn(int src)
00525 {
00526     int tag;
00527     WORD im, r;
00528     WORD *m1, *m2;
00529     LONG size;
00530     PF_BUFFER *rbuf = PF.rbufs[src];
00531     int a = rbuf->active;
00532     int next = a+1 >= rbuf->numbufs ? 0 : a+1 ;
00533     WORD *lastterm = PF_term[src];
00534     WORD *term = rbuf->fill[a];
00535
00536     if ( src <= 0 ) return(PF_term[0]);
00537
00538     if ( rbuf->full[a] == rbuf->buff[a] + AM.MaxTer/sizeof(WORD) + 2 ) {
00539 /*
00540         very first term from this src
00541 */
00542         tag = PF_WaitRbuf(rbuf,a,&size);
00543         rbuf->full[a] += size;
00544         if ( tag == PF_ENDBUFFER_MSGTAG ) *rbuf->full[a]++ = 0;
00545         else if ( rbuf->numbufs > 1 ) {
00546 /*
00547             post a nonblock. recv. for the next buffer
00548 */
00549             rbuf->full[next] = rbuf->buff[next] + AM.MaxTer/sizeof(WORD) + 2;
00550             size = (LONG)(rbuf->stop[next] - rbuf->full[next]);
00551             PF_IRecvRbuf(rbuf,next,src);
00552         }
00553     }
00554     if ( *term == 0 && term != rbuf->full[a] ) return(PF_term[0]);
00555 /*
00556     exception is for rare cases when the terms fitted exactly into buffer
00557 */
00558     if ( term + *term > rbuf->full[a] || term + 1 >= rbuf->full[a] ) {
00559 newterms:
00560         m1 = rbuf->buff[next] + AM.MaxTer/sizeof(WORD) + 1;
00561         if ( *term < 0 || term == rbuf->full[a] ) {
00562 /*
00563             copy term and lastterm to the new buffer, so that they end at m1
00564 */
00565             m2 = rbuf->full[a] - 1;
00566             while ( m2 >= term ) *m1-- = *m2--;
00567             rbuf->fill[next] = term = m1 + 1;
00568             m2 = lastterm + *lastterm - 1;
00569             while ( m2 >= lastterm ) *m1-- = *m2--;
00570             lastterm = m1 + 1;
00571         }
00572         else {
00573 /*
00574             copy beginning of term to the next buffer so that it ends at m1
00575 */
00576             m2 = rbuf->full[a] - 1;
00577             while ( m2 >= term ) *m1-- = *m2--;
00578             rbuf->fill[next] = term = m1 + 1;
00579         }
00580         if ( rbuf->numbufs == 1 ) {
00581             rbuf->full[a] = rbuf->buff[a] + AM.MaxTer/sizeof(WORD) + 2;
00582             size = (LONG)(rbuf->stop[a] - rbuf->full[a]);
00583             PF_IRecvRbuf(rbuf,a,src);
00584         }
00585 /*
00586         wait for new terms in the next buffer
00587 */
00588         rbuf->full[next] = rbuf->buff[next] + AM.MaxTer/sizeof(WORD) + 2;
00589         tag = PF_WaitRbuf(rbuf,next,&size);
00590         rbuf->full[next] += size;

```

```

00591     if ( tag == PF_ENDBUFFER_MSGTAG ) {
00592         *rbuf->full[next]++ = 0;
00593     }
00594     else if ( rbuf->numbufs > 1 ) {
00595         /*
00596             post a nonblock. recv. for active buffer, it is not needed anymore
00597         */
00598         rbuf->full[a] = rbuf->buff[a] + AM.MaxTer/sizeof(WORD) + 2;
00599         size = (LONG)(rbuf->stop[a] - rbuf->full[a]);
00600         PF_IRecvRbuf(rbuf,a,src);
00601     }
00602     /*
00603         now safely make next buffer active
00604     */
00605     a = rbuf->active = next;
00606 }
00607
00608 if ( *term < 0 ) {
00609     /*
00610         We need to decompress the term
00611     */
00612     im = *term;
00613     r = term[1] - im + 1;
00614     m1 = term + 2;
00615     m2 = lastterm - im + 1;
00616     while ( ++im <= 0 ) *--m1 = *--m2;
00617     *--m1 = r;
00618     rbuf->fill[a] = term = m1;
00619     if ( term + *term > rbuf->full[a] ) goto newterms;
00620 }
00621 rbuf->fill[a] += *term;
00622 return(term);
00623 }
00624
00625 /*
00626     #] PF_PutIn :
00627     #[ PF_GetLoser :
00628 */
00629
00648 static int PF_GetLoser(NODE *n)
00649 {
00650     GETIDENTITY
00651     WORD comp;
00652
00653     if ( PF_loser == 0 ) {
00654         /*
00655             this is for the right initialization of the tree only
00656         */
00657         if ( n->left ) n->lloser = PF_GetLoser(n->left);
00658         else {
00659             n->lloser = n->lsrc;
00660             if ( *(PF_term[n->lsrc] = PF_PutIn(n->lsrc)) == 0 ) n->lloser = 0;
00661         }
00662         PF_loser = 0;
00663         if ( n->right ) n->rloser = PF_GetLoser(n->right);
00664         else {
00665             n->rloser = n->rsrc;
00666             if ( *(PF_term[n->rsrc] = PF_PutIn(n->rsrc)) == 0 ) n->rloser = 0;
00667         }
00668         PF_loser = 0;
00669     }
00670     else if ( PF_loser == n->lloser ) {
00671         if ( n->left ) n->lloser = PF_GetLoser(n->left);
00672         else {
00673             n->lloser = n->lsrc;
00674             if ( *(PF_term[n->lsrc] = PF_PutIn(n->lsrc)) == 0 ) n->lloser = 0;
00675         }
00676     }
00677     else if ( PF_loser == n->rloser ) {
00678 newright:
00679         if ( n->right ) n->rloser = PF_GetLoser(n->right);
00680         else {
00681             n->rloser = n->rsrc;
00682             if ( *(PF_term[n->rsrc] = PF_PutIn(n->rsrc)) == 0 ) n->rloser = 0;
00683         }
00684     }
00685     if ( n->lloser > 0 && n->rloser > 0 ) {
00686         comp = CompareTerms(BHEAD PF_term[n->lloser],PF_term[n->rloser],(WORD)0);
00687         if ( comp > 0 ) return(n->lloser);
00688         else if (comp < 0 ) return(n->rloser);
00689         else {
00690             /*
00691                 #[ terms are equal :
00692             */
00693             WORD *lcpos, *rcpos;
00694             UWORD *newcpos;
00695             WORD lclen, rclen, newclen, newnlen;

```

```

00696         SORTING *S = AT.SS;
00697
00698         if ( S->PolyWise ) {
00699             /*
00700             #[ Here we work with PolyFun :
00701             */
00702                 WORD *ttl, *w;
00703                 WORD r1,r2;
00704                 WORD *ml = PF_term[n->lloser];
00705                 WORD *mr = PF_term[n->rloser];
00706
00707                 if ( ( r1 = (int)*PF_term[n->lloser] ) <= 0 ) r1 = 20;
00708                 if ( ( r2 = (int)*PF_term[n->rloser] ) <= 0 ) r2 = 20;
00709                 ttl = ml;
00710                 ml += S->PolyWise;
00711                 mr += S->PolyWise;
00712                 if ( S->PolyFlag == 2 ) {
00713                     w = poly_ratfun_add(BHEAD ml,mr);
00714                     if ( *ttl + w[1] - ml[1] > AM.MaxTer/((LONG)sizeof(WORD)) ) {
00715                         MesPrint("Term too complex in PolyRatFun addition. MaxTermSize of %10l is too
small",AM.MaxTer);
00716                         Terminate(-1);
00717                     }
00718                     AT.WorkPointer = w;
00719                 }
00720                 else {
00721                     w = AT.WorkPointer;
00722                     if ( w + ml[1] + mr[1] > AT.WorkTop ) {
00723                         MesPrint("A WorkSpace of %10l is too small",AM.WorkSize);
00724                         Terminate(-1);
00725                     }
00726                     AddArgs(BHEAD ml,mr,w);
00727                 }
00728                 r1 = w[1];
00729                 if ( r1 <= FUNHEAD || ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 ) ) {
00730                     goto cancelled;
00731                 }
00732                 if ( r1 == ml[1] ) {
00733                     NCOPY(ml,w,r1);
00734                 }
00735                 else if ( r1 < ml[1] ) {
00736                     r2 = ml[1] - r1;
00737                     mr = w + r1;
00738                     ml += ml[1];
00739                     while ( --r1 >= 0 ) *--ml = *--mr;
00740                     mr = ml - r2;
00741                     r1 = S->PolyWise;
00742                     while ( --r1 >= 0 ) *--ml = *--mr;
00743                     *ml -= r2;
00744                     PF_term[n->lloser] = ml;
00745                 }
00746                 else {
00747                     r2 = r1 - ml[1];
00748                     if ( r2 > 2*AM.MaxTal )
00749                         MesPrint("warning: new term in polyfun is large");
00750                     mr = ttl - r2;
00751                     r1 = S->PolyWise;
00752                     ml = ttl;
00753                     *ml += r2;
00754                     PF_term[n->lloser] = mr;
00755                     NCOPY(mr,ml,r1);
00756                     r1 = w[1];
00757                     NCOPY(mr,w,r1);
00758                 }
00759                 PF_newclen[n->rloser] = 0;
00760                 PF_loser = n->rloser;
00761                 goto newright;
00762             /*
00763             #[ Here we work with PolyFun :
00764             */
00765                 }
00766                 if ( ( lclen = PF_newclen[n->lloser] ) != 0 ) lcpos = PF_newcpos[n->lloser];
00767                 else {
00768                     lcpos = PF_term[n->lloser];
00769                     lclen = *(lcpos += *lcpos - 1);
00770                     lcpos -= ABS(lclen) - 1;
00771                 }
00772                 if ( ( rclen = PF_newclen[n->rloser] ) != 0 ) rcpos = PF_newcpos[n->rloser];
00773                 else {
00774                     rcpos = PF_term[n->rloser];
00775                     rclen = *(rcpos += *rcpos - 1);
00776                     rcpos -= ABS(rclen) - 1;
00777                 }
00778                 lclen = ( (lclen > 0) ? (lclen-1) : (lclen+1) ) >> 1;
00779                 rclen = ( (rclen > 0) ? (rclen-1) : (rclen+1) ) >> 1;
00780                 newcpos = PF_ScratchSpace;
00781                 if ( AddRat(BHEAD (UWORD *)lcpos,lclen,(UWORD *)rcpos,rclen,newcpos,&newnlen) )

```



```

return(-1);
00782     if ( AN.ncmod != 0 ) {
00783         if ( ( AC.modmode & POSNEG ) != 0 ) {
00784             NormalModulus(newcpos,&newnlen);
00785         }
00786         if ( BigLong(newcpos,newnlen,(UWORD *)AC.cmod,ABS(AN.ncmod)) >=0 ) {
00787             WORD ii;
00788             SubPLon(newcpos,newnlen,(UWORD *)AC.cmod,ABS(AN.ncmod),newcpos,&newnlen);
00789             newcpos[newnlen] = 1;
00790             for ( ii = 1; ii < newnlen; ii++ ) newcpos[newnlen+ii] = 0;
00791         }
00792     }
00793     if ( newnlen == 0 ) {
00794         /*
00795             terms cancel, get loser of left subtree and then of right subtree
00796         */
00797         cancelled:
00798             PF_loser = n->lloser;
00799             PF_newcpos[n->lloser] = 0;
00800             if ( n->left ) n->lloser = PF_GetLoser(n->left);
00801             else {
00802                 n->lloser = n->lsrc;
00803                 if ( *(PF_term[n->lsrc] = PF_PutIn(n->lsrc)) == 0 ) n->lloser = 0;
00804             }
00805             PF_loser = n->rloser;
00806             PF_newcpos[n->rloser] = 0;
00807             goto newright;
00808         }
00809         else {
00810             /*
00811                 keep the left term and get the loser of right subtree
00812             */
00813             newnlen *= 2;
00814             newcpos = ( newnlen > 0 ) ? ( newnlen + 1 ) : ( newnlen - 1 );
00815             if ( newnlen < 0 ) newnlen = -newnlen;
00816             PF_newcpos[n->lloser] = newcpos;
00817             lcp = PF_newcpos[n->lloser];
00818             if ( newcpos < 0 ) newcpos = -newcpos;
00819             while ( newcpos-- ) *lcp++ = *newcpos++;
00820             PF_loser = n->rloser;
00821             PF_newcpos[n->rloser] = 0;
00822             goto newright;
00823         }
00824         /*
00825             #] terms are equal :
00826         */
00827     }
00828 }
00829 if ( n->lloser > 0 ) return(n->lloser);
00830 if ( n->rloser > 0 ) return(n->rloser);
00831 return(0);
00832 }
00833 /*
00834     #] PF_GetLoser :
00835     #[ PF_EndSort :
00836 */
00837
00864 int PF_EndSort(void)
00865 {
00866     GETIDENTITY
00867     FILEHANDLE *fout = AR.outfile;
00868     PF_BUFFER *sbuf=PF.sbuf;
00869     SORTING *S = AT.SS;
00870     WORD *outterm,*pp;
00871     LONG size, noutterms;
00872     POSITION position, oldposition;
00873     WORD i,cc;
00874     int oldgzipCompress;
00875
00876     if ( AT.SS != AT.S0 || !PF.parallel ) return 0;
00877
00878     if ( PF.me != MASTER ) {
00879         /*
00880             #[ the slaves have to initialize their sendbuffer :
00881
00882             this is a slave and it's PObuffer should be the minimum of the
00883             sortiosize on the master and the PObuffer of our file.
00884             First save the original PObuffer and PObuffer of the outfile
00885         */
00886         size = (S->sTop2 - S->lBuffer - 1)/(PF.numtasks - 1);
00887         size -= (AM.MaxTer/sizeof(WORD) + 2);
00888         if ( fout->POsize < (LONG)(size*sizeof(WORD)) ) size = fout->POsize/sizeof(WORD);
00889         if ( sbuf == NULL ) {
00890             if ( (sbuf = PF_AllocBuf(PF.numtasks, size*sizeof(WORD), 1)) == NULL ) return -1;
00891             sbuf->active = 0;
00892             PF.sbuf = sbuf;
00893         }
00894     }

```

```

00894     sbuf->buff[0] = fout->PObuffer;
00895     sbuf->stop[0] = fout->PObuffer+size;
00896     if ( sbuf->stop[0] > fout->POstop ) return -1;
00897     for ( i = 0; i < PF.numsbufs; i++ )
00898         sbuf->fill[i] = sbuf->full[i] = sbuf->buff[i];
00899
00900     fout->PObuffer = sbuf->buff[sbuf->active];
00901     fout->POstop = sbuf->stop[sbuf->active];
00902     fout->POsize = size*sizeof(WORD);
00903     fout->POfill = fout->POfull = fout->PObuffer;
00904 /*
00905     #] the slaves have to initialize their sendbuffer :
00906 */
00907     return(0);
00908 }
00909 /*
00910     this waits for all slaves to be ready to send terms back
00911 */
00912 PF_WaitAllSlaves(); /* Note, the returned value should be 0 on success. */
00913 /*
00914     Now collect the terms of all slaves and merge them.
00915     PF_GetLoser gives the position of the smallest term, which is the real
00916     work. The smallest term needs to be copied to the outbuf: use PutOut.
00917 */
00918 PF_InitTree();
00919 if ( AR.PolyFun == 0 ) { S->PolyFlag = 0; }
00920 else if ( AR.PolyFunType == 1 ) { S->PolyFlag = 1; }
00921 else if ( AR.PolyFunType == 2 ) {
00922     if ( AR.PolyFunExp == 2
00923         || AR.PolyFunExp == 3 ) S->PolyFlag = 1;
00924     else
00925         S->PolyFlag = 2;
00926 }
00927 *AR.CompressPointer = 0;
00928 SeekScratch(fout, &position);
00929 oldposition = position;
00929 oldgzipCompress = AR.gzipCompress;
00930 AR.gzipCompress = 0;
00931
00932 noutterms = 0;
00933
00934 while ( PF_loser >= 0 ) {
00935     if ( (PF_loser = PF_GetLoser(PF_root)) == 0 ) break;
00936     outterm = PF_term[PF_loser];
00937     noutterms++;
00938
00939     if ( PF_newclen[PF_loser] != 0 ) {
00940 /*
00941         #[ this is only when new coeff was too long :
00942 */
00943         outterm = PF_WorkSpace;
00944         pp = PF_term[PF_loser];
00945         cc = *pp;
00946         while ( cc-- ) *outterm++ = *pp++;
00947         outterm = (outterm[-1] > 0) ? outterm-outterm[-1] : outterm+outterm[-1];
00948         if ( PF_newclen[PF_loser] > 0 ) cc = (WORD)PF_newclen[PF_loser] - 1;
00949         else
00950             cc = -(WORD)PF_newclen[PF_loser] - 1;
00951         pp = PF_newcpos[PF_loser];
00952         while ( cc-- ) *outterm++ = *pp++;
00953         *outterm++ = PF_newclen[PF_loser];
00954         *PF_WorkSpace = outterm - PF_WorkSpace;
00955         outterm = PF_WorkSpace;
00956         *PF_newcpos[PF_loser] = 0;
00957         PF_newclen[PF_loser] = 0;
00958 /*
00959         #] this is only when new coeff was too long :
00960 */
00961     }
00962     PRINTFBUF("PF_EndSort to PutOut: ",outterm,*outterm);
00963     PutOut(BHEAD outterm,&position,fout,1);
00964 }
00965 if ( FlushOut(&position,fout,0) ) {
00966     AR.gzipCompress = oldgzipCompress;
00967     return(-1);
00968 }
00969 S->TermsLeft = PF_goutterms = noutterms;
00970 DIFPOS(PF_exprsize, position, oldposition);
00971 AR.gzipCompress = oldgzipCompress;
00972 return(1);
00973 }
00974 /*
00975     #] PF_EndSort :
00976 #] sort.c :
00977 #[ proces.c :
00978     #[ variables :
00979 */
00980

```

```

00981 static WORD *PF_CurrentBracket;
00982
00983 /*
00984      #] variables :
00985      #[ PF_GetTerm :
00986 */
00987
01006 static WORD PF_GetTerm(WORD *term)
01007 {
01008     GETIDENTITY
01009     FILEHANDLE *fi = AC.RhsExprInModuleFlag && PF.rhsInParallel ? &PF.slavebuf : AR.infile;
01010     WORD i;
01011     WORD *next, *np, *last, *lp = 0, *nextstop, *tp=term;
01012
01013     /* Only on the slaves. */
01014
01015     AN.deferskipped = 0;
01016     if ( fi->POfill >= fi->POfull || fi->POfull == fi->PObuffer ) {
01017 ReceiveNew:
01018     {
01019     /*
01020         #[ receive new terms from master :
01021     */
01022         int src = MASTER, tag;
01023         int follow = 0;
01024         LONG size,cpu,space = 0;
01025
01026         if ( PF.log ) {
01027             fprintf(stderr,"%d] Starting to send to Master\n",PF.me);
01028             fflush(stderr);
01029         }
01030
01031         cpu = TimeCPU(1);
01032         PF_PrepPack();
01033         PF_Pack(&cpu, 1,PF_LONG);
01034         PF_Pack(&space, 1,PF_LONG);
01035         PF_Pack(&PF_linterms, 1,PF_LONG);
01036         PF_Pack(&(AM.S0->GenTerms), 1,PF_LONG);
01037         PF_Pack(&(AM.S0->TermsLeft), 1,PF_LONG);
01038         PF_Pack(&follow, 1,PF_INT );
01039
01040         if ( PF.log ) {
01041             fprintf(stderr,"%d] Now sending with tag = %d\n",PF.me,PF_READY_MSGTAG);
01042             fflush(stderr);
01043         }
01044
01045         PF_Send(MASTER, PF_READY_MSGTAG);
01046
01047         if ( PF.log ) {
01048             fprintf(stderr,"%d] returning from send\n",PF.me);
01049             fflush(stderr);
01050         }
01051
01052         size = fi->POstop - fi->PObuffer - 1;
01053 #ifdef AbsolutelyExtra
01054         PF_Receive(MASTER,PF_ANY_MSGTAG,&src,&tag);
01055 #ifdef MPI2
01056         if ( tag == PF_TERM_MSGTAG ) {
01057             PF_Unpack(&size, 1, PF_LONG);
01058             if ( PF_Put_target(src) == 0 ) {
01059                 printf("PF_Put_target error ...\n");
01060             }
01061         }
01062         else {
01063             PF_RecvWbuf(fi->PObuffer,&size,&src);
01064         }
01065 #else
01066         PF_RecvWbuf(fi->PObuffer,&size,&src);
01067 #endif
01068 #endif
01069         tag=PF_RecvWbuf(fi->PObuffer,&size,&src);
01070
01071         fi->POfill = fi->PObuffer;
01072         /* Get AN.ninterms which sits in the first 2 WORDs. */
01073         {
01074             LONG ninterms;
01075             UNPACK_LONG(fi->POfill, ninterms);
01076             if ( fi->POfill < fi->POfull ) {
01077                 DBGOUT_NINTERMS(2, ("PF.me=%d AN.ninterms=%d PF_linterms=%d ninterms=%d GET\n",
(int)PF.me, (int)AN.ninterms, (int)PF_linterms, (int)ninterms));
01078                 AN.ninterms = ninterms - 1;
01079             } else {
01080                 DBGOUT_NINTERMS(2, ("PF.me=%d AN.ninterms=%d PF_linterms=%d ninterms=%d GETEND\n",
(int)PF.me, (int)AN.ninterms, (int)PF_linterms, (int)ninterms));
01081             }
01082         }
01083         fi->POfull = fi->PObuffer + size;

```

```

01084         if ( tag == PF_ENDSORT_MSGTAG ) *fi->POfull++ = 0;
01085 /*
01086      #] receive new terms from master :
01087 */
01088     }
01089     if ( PF_CurrentBracket ) *PF_CurrentBracket = 0;
01090 }
01091 if ( *fi->POfill == 0 ) {
01092     fi->POfill = fi->POfull = fi->PObuffer;
01093     *term = 0;
01094     goto RegRet;
01095 }
01096 if ( AR.DeferFlag ) {
01097     if ( !PF_CurrentBracket ) {
01098 /*
01099      #[ alloc space :
01100 */
01101         PF_CurrentBracket =
01102             (WORD*)Malloc1(AM.MaxTer,"PF_CurrentBracket");
01103         *PF_CurrentBracket = 0;
01104 /*
01105      #] alloc space :
01106 */
01107     }
01108     while ( *PF_CurrentBracket ) { /* "for each term in the buffer" */
01109 /*
01110      #[ test : bracket & skip if it's equal to the last in PF_CurrentBracket
01111 */
01112         next = fi->POfill;
01113         nextstop = next + *next; nextstop -= ABS(nextstop[-1]);
01114         next++;
01115         last = PF_CurrentBracket+1;
01116         while ( next < nextstop ) {
01117 /*
01118             scan the next term and PF_CurrentBracket
01119 */
01120             if ( *last == HAAKJE && *next == HAAKJE ) {
01121 /*
01122                 the part outside brackets is equal => skip this term
01123 */
01124                 PRINTFBUF("PF_GetTerm skips",fi->POfill,*fi->POfill);
01125                 break;
01126             }
01127 /*
01128             check if the current subterms are equal
01129 */
01130             np = next; next += next[1];
01131             lp = last; last += last[1];
01132             while ( np < next ) if ( *lp++ != *np++ ) goto strip;
01133         }
01134 /*
01135         go on to next term
01136 */
01137         fi->POfill += *fi->POfill;
01138         AN.deferskipped++;
01139 /*
01140         the usual checks
01141 */
01142         if ( fi->POfill >= fi->POfull || fi->POfull == fi->PObuffer )
01143             goto ReceiveNew;
01144         if ( *fi->POfill == 0 ) {
01145             fi->POfill = fi->POfull = fi->PObuffer;
01146             *term = 0;
01147             goto RegRet;
01148         }
01149 /*
01150      #] test :
01151 */
01152     }
01153 /*
01154      #[ copy :
01155 */
01156     this term to CurrentBracket and the part outside of bracket
01157     to Workspace at term
01158 */
01159 strip:
01160     next = fi->POfill;
01161     nextstop = next + *next; nextstop -= ABS(nextstop[-1]);
01162     next++;
01163     tp++;
01164     lp = PF_CurrentBracket + 1;
01165     while ( next < nextstop ) {
01166         if ( *next == HAAKJE ) {
01167             fi->POfill += *fi->POfill;
01168             while ( next < fi->POfill ) *lp++ = *next++;
01169             *PF_CurrentBracket = lp - PF_CurrentBracket;
01170             *lp = 0;

```

```

01171             *tp++ = 1;
01172             *tp++ = 1;
01173             *tp++ = 3;
01174             *term = WORDDIFF(tp,term);
01175             PRINTFBUF("PF_GetTerm new brack",PF_CurrentBracket,*PF_CurrentBracket);
01176             PRINTFBUF("PF_GetTerm POfill",fi->POfill,*fi->POfill);
01177             goto RegRet;
01178         }
01179         np = next; next += next[1];
01180         while ( np < next ) *tp++ = *lp++ = *np++;
01181     }
01182     tp = term;
01183     /*
01184     #] copy :
01185     */
01186 }
01187
01188     i = *fi->POfill;
01189     while ( i-- ) *tp++ = *fi->POfill++;
01190 RegRet:
01191     PRINTFBUF("PF_GetTerm returns",term,*term);
01192     return(*term);
01193 }
01194
01195 /*
01196     #] PF_GetTerm :
01197     #[ PF_Deferred :
01198     */
01199
01200 WORD PF_Deferred(WORD *term, WORD level)
01201 {
01210     GETIDENTITY
01211     WORD *bra, *bstop;
01212     WORD *tstart;
01213     FILEHANDLE *fi = AC.RhsExprInModuleFlag && PF.rhsInParallel ? &PF.slavebuf : AR.infile;
01214     WORD *next = fi->POfill;
01215     WORD *termout = AT.WorkPointer;
01216     WORD *oldwork = AT.WorkPointer;
01217
01218     AT.WorkPointer = (WORD *) ((UBYTE *) (AT.WorkPointer) + AM.MaxTer);
01219     AR.DeferFlag = 0;
01220
01221     PRINTFBUF("PF_Deferred (Term) ",term,*term);
01222     PRINTFBUF("PF_Deferred (Bracket)",PF_CurrentBracket,*PF_CurrentBracket);
01223
01224     bra = bstop = PF_CurrentBracket;
01225     if ( *bstop > 0 ) {
01226         bstop += *bstop;
01227         bstop -= ABS(bstop[-1]);
01228     }
01229     bra++;
01230     while ( *bra != HAAKJE && bra < bstop ) bra += bra[1];
01231     if ( bra >= bstop ) { /* No deferred action! */
01232         AT.WorkPointer = term + *term;
01233         if ( Generator(BHEAD term,level) ) goto DefCall;
01234         AR.DeferFlag = 1;
01235         AT.WorkPointer = oldwork;
01236         return(0);
01237     }
01238     bstop = bra;
01239     tstart = bra + bra[1];
01240     bra = PF_CurrentBracket;
01241     tstart--;
01242     *tstart = bra + *bra - tstart;
01243     bra++;
01244     /*
01245         Status of affairs:
01246         First bracket content starts at tstart.
01247         Next term starts at next.
01248         The outside of the bracket runs from bra = PF_CurrentBracket to bstop.
01249     */
01250     for(;;) {
01251         if ( InsertTerm(BHEAD term,0,AM.rbufnum,tstart,termout,0) < 0 ) {
01252             goto DefCall;
01253         }
01254         /*
01255             call Generator with new composed term
01256         */
01257         AT.WorkPointer = termout + *termout;
01258         if ( Generator(BHEAD termout,level) ) goto DefCall;
01259         AT.WorkPointer = termout;
01260         tstart = next + 1;
01261         if ( tstart >= fi->POfull ) goto ThatsIt;
01262         next += *next;
01263         /*
01264             compare with current bracket
01265         */

```

```

01266         while ( bra <= bstop ) {
01267             if ( *bra != *tstart ) goto ThatsIt;
01268             bra++; tstart++;
01269         }
01270 /*
01271         now bra and tstart should both be a HAAKJE
01272 */
01273     bra--; tstart--;
01274     if ( *bra != HAAKJE || *tstart != HAAKJE ) goto ThatsIt;
01275     tstart += tstart[1];
01276     tstart--;
01277     *tstart = next - tstart;
01278     bra = PF_CurrentBracket + 1;
01279 }
01280
01281 ThatsIt:
01282 /*
01283     AT.WorkPointer = oldwork;
01284 */
01285     AR.DeferFlag = 1;
01286     return(0);
01287 DefCall:
01288     MesCall("PF_Deferred");
01289     SETERROR(-1);
01290 }
01291
01292 /*
01293     #] PF_Deferred :
01294     #[ PF_Wait4Slave :
01295 */
01296
01297 static LONG **PF_W4Sstats = 0;
01298
01305 static int PF_Wait4Slave(int src)
01306 {
01307     int j, tag, next;
01308
01309     tag = PF_ANY_MSGTAG;
01310     PF_CatchErrorMessages(&src, &tag);
01311     PF_Receive(src, tag, &next, &tag);
01312
01313     if ( tag != PF_READY_MSGTAG ) {
01314         MesPrint("[%d] PF_Wait4Slave: received MSGTAG %d", (WORD)PF.me, (WORD)tag);
01315         return(-1);
01316     }
01317     if ( PF_W4Sstats == 0 ) {
01318         PF_W4Sstats = (LONG**)Malloca(sizeof(LONG*), "");
01319         PF_W4Sstats[0] = (LONG*)Malloca(PF_STATS_SIZE*sizeof(LONG), "");
01320     }
01321     PF_Unpack(PF_W4Sstats[0], PF_STATS_SIZE, PF_LONG);
01322     PF_Statistics(PF_W4Sstats, next);
01323
01324     PF_Unpack(&j, 1, PF_INT);
01325
01326     if ( j ) {
01327 /*
01328         actions depending on rest of information in last message
01329 */
01330     }
01331     return(next);
01332 }
01333
01334 /*
01335     #] PF_Wait4Slave :
01336     #[ PF_Wait4SlaveIP :
01337 */
01338 /*
01339     array of expression numbers for PF_InParallel processor.
01340     Each time the master sends expression "i" to the slave
01341     "next" it sets partodoexr[next]=i:
01342 */
01343 static WORD *partodoexr=NULL;
01344
01352 static int PF_Wait4SlaveIP(int *src)
01353 {
01354     int j, tag, next;
01355
01356     tag = PF_ANY_MSGTAG;
01357     PF_CatchErrorMessages(src, &tag);
01358     PF_Receive(*src, tag, &next, &tag);
01359     *src=tag;
01360     if ( PF_W4Sstats == 0 ) {
01361         PF_W4Sstats = (LONG**)Malloca(sizeof(LONG*), "");
01362         PF_W4Sstats[0] = (LONG*)Malloca(PF_STATS_SIZE*sizeof(LONG), "");
01363     }
01364
01365     PF_Unpack(PF_W4Sstats[0], PF_STATS_SIZE, PF_LONG);

```

```

01366     if ( tag == PF_DATA_MSGTAG )
01367         AR.CurExpr = partodoexr[next];
01368     PF_Statistics(PF_W4Sstats,next);
01369
01370     PF_Unpack(&j,1,PF_INT);
01371
01372     if ( j ) {
01373         /* actions depending on rest of information in last message */
01374     }
01375
01376     return(next);
01377 }
01378 /*
01379     #] PF_Wait4SlaveIP :
01380     #[ PF_WaitAllSlaves :
01381 */
01382
01391 static int PF_WaitAllSlaves(void)
01392 {
01393     int i, readySlaves, tag, next = PF_ANY_SOURCE;
01394     UBYTE *has_sent = 0;
01395
01396     has_sent = (UBYTE*)Malloc1(sizeof(UBYTE)*(PF.numtasks + 1),"PF_WaitAllSlaves");
01397     for ( i = 0; i < PF.numtasks; i++ ) has_sent[i] = 0;
01398
01399     for ( readySlaves = 1; readySlaves < PF.numtasks; ) {
01400         if ( next != PF_ANY_SOURCE ) { /*Go to the next slave:*/
01401             do{ /*Note, here readySlaves<PF.numtasks, so this loop can't be infinite*/
01402                 if ( ++next >= PF.numtasks ) next = 1;
01403             } while ( has_sent[next] == 1 );
01404         }
01405         /*
01406             Here PF_ProbeWithCatchingErrorMessages() is BLOCKING function if next = PF_ANY_SOURCE:
01407         */
01408         tag = PF_ProbeWithCatchingErrorMessages(&next);
01409         /*
01410             Here next != PF_ANY_SOURCE
01411         */
01412         switch ( tag ) {
01413             case PF_BUFFER_MSGTAG:
01414             case PF_ENDBUFFER_MSGTAG:
01415                 /*
01416                     Slaves are ready to send their results back
01417                 */
01418                 if ( has_sent[next] == 0 ) {
01419                     has_sent[next] = 1;
01420                     readySlaves++;
01421                 }
01422                 else { /*error?*/
01423                     fprintf(stderr,"ERROR next=%d tag=%d\n",next,tag);
01424                 }
01425                 /*
01426                     Note, we do NOT read results here! Messages from these slaves will be read
01427                     only after all slaves are ready, further in caller function
01428                 */
01429                 break;
01430             case 0:
01431                 /*
01432                     The slave is not ready. Just go to the next slave.
01433                     It may appear that there are no more ready slaves, and the master
01434                     will wait them in infinite loop. Stupid situation - the master can
01435                     receive buffers from ready slaves!
01436                 */
01437                 #ifdef PF_WITH_SCHED_YIELD
01438                 /*
01439                     Relinquish the processor:
01440                 */
01441                 sched_yield();
01442                 #endif
01443                 break;
01444             case PF_DATA_MSGTAG:
01445                 tag=next;
01446                 next=PF_Wait4SlaveIP(&tag);
01447             /*
01448                 tag must be == PF_DATA_MSGTAG!
01449             */
01450             PF_Statistics(PF_stats,0);
01451             PF_Slave2MasterIP(next);
01452             PF_Master2SlaveIP(next,NULL);
01453             if ( has_sent[next] == 0 ) {
01454                 has_sent[next]=1;
01455                 readySlaves++;
01456             }else{
01457                 /*error?*/
01458                 fprintf(stderr,"ERROR next=%d tag=%d\n",next,tag);
01459             }/*if ( has_sent[next] == 0 )*/
01460             break;

```

```

01461         case PF_EMPTY_MSGTAG:
01462             tag=next;
01463             next=PF_Wait4SlaveIP(&tag);
01464     /*
01465     tag must be == PF_EMPTY_MSGTAG!
01466     */
01467         PF_Master2SlaveIP(next,NULL);
01468         if ( has_sent[next] == 0 ) {
01469             has_sent[next]=1;
01470             readySlaves++;
01471         }else{
01472             /*error?*/
01473             fprintf(stderr,"ERROR next=%d tag=%d\n",next,tag);
01474             /*if ( has_sent[next] == 0 )*/
01475             break;
01476         case PF_READY_MSGTAG:
01477     /*
01478         idle slave
01479         May be only PF_READY_MSGTAG:
01480     */
01481         next = PF_Wait4Slave(next);
01482         if ( next == -1 ) return(next); /*Cannot be!*/
01483         if ( has_sent[0] == 0 ) { /*Send the last chunk to the slave*/
01484             PF.sbuf->active = 0;
01485             has_sent[0] = 1;
01486         }
01487         else {
01488     /*
01489         Last chunk was sent, so just send to slave ENDSORT
01490         AN.ninterms must be sent because the slave expects it:
01491     */
01492             PACK_LONG(PF.sbuf->fill[next], AN.ninterms);
01493     /*
01494         This will tell to the slave that there are no more terms:
01495     */
01496             *(PF.sbuf->fill[next])++ = 0;
01497             PF.sbuf->active = next;
01498         }
01499     /*
01500         Send ENDSORT
01501     */
01502         PF_ISendSbuf(next,PF_ENDSORT_MSGTAG);
01503         break;
01504     default:
01505     /*
01506         Error?
01507         Indicates the error. This will force exit from the main loop:
01508     */
01509         MesPrint("!!!Unexpected MPI message src=%d tag=%d.", next, tag);
01510         readySlaves = PF.numtasks+1;
01511         break;
01512     }
01513 }
01514
01515 if ( has_sent ) M_free(has_sent,"PF_WaitAllSlaves");
01516 /*
01517 0 on success (exit from the main loop by loop condition), or -1 if fails
01518 (exit from the main loop since readySlaves=PF.numtasks+1):
01519 */
01520 return(PF.numtasks-readySlaves);
01521 }
01522
01523 /*
01524 #] PF_WaitAllSlaves :
01525 #[ PF_Processor :
01526 */
01527
01540 int PF_Processor(EXPRESSIONS e, WORD i, WORD LastExpression)
01541 {
01542     GETIDENTITY
01543     WORD *term = AT.WorkPointer;
01544     LONG dd = 0;
01545     PF_BUFFER *sb = PF.sbuf;
01546     WORD j, *s, next;
01547     LONG size, cpu;
01548     POSITION position;
01549     int k, src, tag;
01550     FILEHANDLE *oldoutfile = AR.outfile;
01551
01552 #ifndef MPI2
01553     if ( PF_shared_buff == NULL ) {
01554         if ( PF_SMWin_Init() == 0 ) {
01555             MesPrint("PF_SMWin_Init error");
01556             exit(-1);
01557         }
01558     }
01559 #endif

```



```

01560
01561     if ( ( (WORD *) ((UBYTE *) (AT.WorkPointer)) + AM.MaxTer ) > AT.WorkTop ) {
01562         MesWork();
01563     }
01564
01565     /* For redefine statements. */
01566     if ( AC.numpfirstnum > 0 ) {
01567         for ( j = 0; j < AC.numpfirstnum; j++ ) {
01568             AC.inputnumbers[j] = -1;
01569         }
01570     }
01571
01572     if ( AC.mparallelfalg != PARALLELFLAG ) return(0);
01573
01574     if ( PF.me == MASTER ) {
01575         /*
01576             #[] Master:
01577             #[] write prototype to outfile:
01578         */
01579         WORD oldBracketOn = AR.BracketOn;
01580         WORD *oldBrackBuf = AT.BrackBuf;
01581         WORD oldbracketindexflag = AT.bracketindexflag;
01582
01583         LONG maxinterms; /* the maximum number of terms in the bucket */
01584         int cmaxinterms; /* a variable controlling the transition of maxinterms */
01585         LONG termsinbucket; /* the number of filled terms in the bucket */
01586         LONG ProcessBucketSize = AC.mProcessBucketSize;
01587
01588         if ( PF.log && AC.CModule >= PF.log )
01589             MesPrint("[%d] working on expression %s in module %l",PF.me,EXPRNAME(i),AC.CModule);
01590         if ( GetTerm(BHEAD term) <= 0 ) {
01591             MesPrint("[%d] Expression %d has problems in scratchfile",PF.me,i);
01592             return(-1);
01593         }
01594         term[3] = i;
01595         if ( AR.outtohide ) {
01596             SeekScratch(AR.hidefile,&position);
01597             e->onfile = position;
01598             if ( PutOut (BHEAD term,&position,AR.hidefile,0) < 0 ) return(-1);
01599         }
01600         else {
01601             SeekScratch(AR.outfile,&position);
01602             e->onfile = position;
01603             if ( PutOut (BHEAD term,&position,AR.outfile,0) < 0 ) return(-1);
01604         }
01605         AR.DeferFlag = 0; /* The master leave the brackets!!! */
01606         AR.Eside = RHSIDE;
01607         if ( ( e->vflags & ISFACTORIZED ) != 0 ) {
01608             AR.BracketOn = 1;
01609             AT.BrackBuf = AM.BracketFactors;
01610             AT.bracketindexflag = 1;
01611         }
01612         if ( AT.bracketindexflag > 0 ) OpenBracketIndex(i);
01613         /*
01614             #[] write prototype to outfile:
01615             #[] initialize sendbuffer if necessary:
01616
01617             the size of the sendbufs is:
01618             MIN(1/PF.numtasks*(AT.SS->sBufsize+AT.SS->lBufsize),AR.infile->POsize)
01619             No allocation for extra buffers necessary, just make sb->buf... point
01620             to the right places in the sortbuffers.
01621         */
01622         NewSort(BHEAD0); /* we need AT.SS to be set for this!!! */
01623         if ( sb == 0 || sb->buff[0] != AT.SS->lBuffer ) {
01624             size = (LONG) ((AT.SS->sTop2 - AT.SS->lBuffer)/(PF.numtasks));
01625             if ( size > (LONG) (AR.infile->POsize/sizeof(WORD) - 1) )
01626                 size = AR.infile->POsize/sizeof(WORD) - 1;
01627             if ( sb == 0 ) {
01628                 if ( ( sb = PF_AllocBuf (PF.numtasks,size*sizeof(WORD),PF.numtasks) ) == NULL )
01629                     return(-1);
01630             }
01631             sb->buff[0] = AT.SS->lBuffer;
01632             sb->full[0] = sb->fill[0] = sb->buff[0];
01633             for ( j = 1; j < PF.numtasks; j++ ) {
01634                 sb->stop[j-1] = sb->buff[j] = sb->buff[j-1] + size;
01635             }
01636             sb->stop[PF.numtasks-1] = sb->buff[PF.numtasks-1] + size;
01637             PF.sbuf = sb;
01638         }
01639         for ( j = 0; j < PF.numtasks; j++ ) {
01640             sb->full[j] = sb->fill[j] = sb->buff[j];
01641         }
01642         /*
01643             #[] initialize sendbuffer if necessary:
01644             #[] loop for all terms in infile:
01645         */
01646         /*

```

```

01647      * The initial value of maxinterms is determined by the user given
01648      * ProcessBucketSize and the number of terms in the current expression.
01649      * We make the initial maxinterms smaller, so that we get the all
01650      * workers busy as soon as possible.
01651      */
01652      maxinterms = ProcessBucketSize / 100;
01653      if ( maxinterms > e->counter / (PF.numtasks - 1) / 4 )
01654          maxinterms = e->counter / (PF.numtasks - 1) / 4;
01655      if ( maxinterms < 1 ) maxinterms = 1;
01656      cmaxinterms = 0;
01657      /*
01658      * Copy them always to sb->buff[0]. When that is full, wait for
01659      * the next slave to accept terms, exchange sb->buff[0] and
01660      * sb->buff[next], send sb->buff[next] to next slave and go on
01661      * filling the now empty sb->buff[0].
01662      */
01663      AN.ninterms = 0;
01664      termsinbucket = 0;
01665      PACK_LONG(sb->fill[0], 1);
01666      while ( GetTerm(BHEAD term) ) {
01667          AN.ninterms++; dd = AN.deferskipped;
01668          if ( AC.CollectFun && *term <= (LONG)(AM.MaxTer/(2*sizeof(WORD))) ) {
01669              if ( GetMoreTerms(term) < 0 ) {
01670                  LowerSortLevel(); return(-1);
01671              }
01672          }
01673          PRINTFBUF("PF_Processor gets",term,*term);
01674          if ( termsinbucket >= maxinterms || sb->fill[0] + *term >= sb->stop[0] ) {
01675              next = PF_Wait4Slave(PF_ANY_SOURCE);
01676
01677              sb->fill[next] = sb->fill[0];
01678              sb->full[next] = sb->full[0];
01679              SWAP(sb->stop[next], sb->stop[0]);
01680              SWAP(sb->buff[next], sb->buff[0]);
01681              sb->fill[0] = sb->full[0] = sb->buff[0];
01682              sb->active = next;
01683
01684          #ifdef MPI2
01685              if ( PF_Put_origin(next) == 0 ) {
01686                  printf("PF_Put_origin error...\n");
01687              }
01688          #else
01689              PF_ISendSbuf(next,PF_TERM_MSGTAG);
01690          #endif
01691
01692          /* Initialize the next bucket. */
01693          termsinbucket = 0;
01694          PACK_LONG(sb->fill[0], AN.ninterms);
01695          /*
01696          * For the "slow startup". We double maxinterms up to ProcessBucketSize
01697          * after (hopefully) the all workers got some terms.
01698          */
01699          if ( cmaxinterms >= PF.numtasks - 2 ) {
01700              maxinterms *= 2;
01701              if ( maxinterms >= ProcessBucketSize ) {
01702                  cmaxinterms = -1;
01703                  maxinterms = ProcessBucketSize;
01704              }
01705              else if ( cmaxinterms >= 0 ) {
01706                  cmaxinterms++;
01707              }
01708          }
01709          j = *(s = term);
01710          NCOPY(sb->fill[0], s, j);
01711          termsinbucket++;
01712      }
01713      /* NOTE: The last chunk will be sent to a slave at EndSort() => PF_EndSort()
01714      *      => PF_WaitAllSlaves(). */
01715      AN.ninterms += dd;
01716      /*
01717      #] loop for all terms in infile:
01718      #[ Clean up & EndSort:
01719      */
01720      if ( LastExpression ) {
01721          UpdateMaxSize();
01722          if ( AR.infile->handle >= 0 ) {
01723              CloseFile(AR.infile->handle);
01724              AR.infile->handle = -1;
01725              remove(AR.infile->name);
01726              PUTZERO(AR.infile->POposition);
01727          }
01728          AR.infile->POfill = AR.infile->POfull = AR.infile->PObuffer;
01729      }
01730      if ( AR.outtohide ) AR.outfile = AR.hidefile;
01731      PF.parallel = 1;
01732      if ( EndSort(BHEAD AM.S0->sBuffer,0) < 0 ) return(-1);
01733      PF.parallel = 0;

```

```

01734         if ( AR.outtohide ) {
01735             AR.outfile = oldoutfile;
01736             AR.hidefile->POfull = AR.hidefile->POfill;
01737         }
01738         UpdateMaxSize();
01739         AR.BraceOn = oldBracketOn;
01740         AT.BrackBuf = oldBrackBuf;
01741         if ( ( e->vflags & TOBEFACTORED ) != 0 )
01742             poly_factorize_expression(e);
01743         else if ( ( ( e->vflags & TOBEUNFACTORED ) != 0 )
01744             && ( ( e->vflags & ISFACTORIZED ) != 0 ) )
01745             poly_unfactorize_expression(e);
01746         AT.bracketindexflag = oldbracketindexflag;
01747         AR.GetFile = 0;
01748         AR.outtohide = 0;
01749         /*
01750          * NOTE: e->numdummies, e->vflags and AR.exprflags will be updated
01751          *       after gathering the information from all slaves.
01752          */
01753     /*
01754         #] Clean up & EndSort:
01755         #[ Collect (stats,prepro,...):
01756     */
01757     DBGOUT_NINTERMS(1, ("PF.me=%d AN.ninterms=%d ENDSORT\n", (int)PF.me, (int)AN.ninterms));
01758     PF_CatchErrorMessagesForAll();
01759     e->numdummies = 0;
01760     for ( k = 1; k < PF.numtasks; k++ ) {
01761         PF_LongSingleReceive(PF_ANY_SOURCE, PF_ENDSORT_MSGTAG, &src, &tag);
01762         PF_LongSingleUnpack(PF_stats[src], PF_STATS_SIZE, PF_LONG);
01763         {
01764             WORD numdummies, expchanged;
01765             PF_LongSingleUnpack(&numdummies, 1, PF_WORD);
01766             PF_LongSingleUnpack(&expchanged, 1, PF_WORD);
01767             if ( e->numdummies < numdummies ) e->numdummies = numdummies;
01768             AR.expchanged |= expchanged;
01769         }
01770         /* Now handle redefined preprocessor variables. */
01771         if ( AC.numpfirstnum > 0 ) PF_UnpackRedefinedPreVars();
01772     }
01773     /* Broadcast redefined preprocessor variables. */
01774     if ( AC.numpfirstnum > 0 ) {
01775         int RetCode = PF_BroadcastRedefinedPreVars();
01776         if ( RetCode ) return RetCode;
01777     }
01778     if ( ! AC.OldParallelStats ) {
01779         /* Now we can calculate AT.SS->GenTerms from the statistics of the slaves. */
01780         LONG genterms = 0;
01781         for ( k = 1; k < PF.numtasks; k++ ) {
01782             genterms += PF_stats[k][3];
01783         }
01784         AT.SS->GenTerms = genterms;
01785         WriteStats(&PF_exprsize, STATSPOSTSORT, NOCHECKLOGTYPE);
01786         Expressions[AR.CurExpr].size = PF_exprsize;
01787     }
01788     PF_Statistics(PF_stats,0);
01789     /*
01790         #] Collect (stats,prepro,...):
01791         #[ Update flags :
01792     */
01793     if ( AM.S0->TermsLeft ) e->vflags &= ~ISZERO;
01794     else e->vflags |= ISZERO;
01795     if ( AR.expchanged == 0 ) e->vflags |= ISUNMODIFIED;
01796     if ( AM.S0->TermsLeft ) AR.expflags |= ISZERO;
01797     if ( AR.expchanged ) AR.expflags |= ISUNMODIFIED;
01798     /*
01799         #] Update flags :
01800         #] Master:
01801     */
01802     }
01803     else {
01804     /*
01805         #] Slave :
01806     */
01807     /*
01808         #] Generator Loop & EndSort :
01809
01810         loop for all terms to get from master, call Generator for each of them
01811         then call EndSort and do cleanup (to be implemented)
01812     */
01813         WORD oldBracketOn = AR.BraceOn;
01814         WORD *oldBrackBuf = AT.BrackBuf;
01815         WORD oldbracketindexflag = AT.bracketindexflag;
01816
01817         /* For redefine statements. */
01818         if ( AC.numpfirstnum > 0 ) {
01819             for ( j = 0; j < AC.numpfirstnum; j++ ) {
01820                 AC.inputnumbers[j] = -1;

```

```

01821     }
01822 }
01823
01824 SeekScratch(AR.outfile,&position);
01825 e->onfile = position;
01826 AR.DeferFlag = AC.ComDefer;
01827 AR.Eside = RHSE;
01828 if ( ( e->vflags & ISFACTORIZED ) != 0 ) {
01829     AR.BracketOn = 1;
01830     AT.BrackBuf = AM.BracketFactors;
01831     AT.bracketindexflag = 1;
01832 }
01833 NewSort(BHEAD0);
01834 AR.MaxDum = AM.IndDum;
01835 AN.ninterms = 0;
01836 PF_linterms = 0;
01837 PF.parallel = 1;
01838 #ifdef MPI2
01839 AR.infile->POfull = AR.infile->POfill = AR.infile->PObuffer = PF_shared_buff;
01840 #endif
01841 {
01842     FILEHANDLE *fi = AC.RhsExprInModuleFlag && PF.rhsInParallel ? &PF.slavebuf : AR.infile;
01843     fi->POfull = fi->POfill = fi->PObuffer;
01844 }
01845 /* FIXME: AN.ninterms is still broken when AN.deferskipped is non-zero.
01846  * It still needs some work, also in PF_GetTerm(). (TU 30 Aug 2011) */
01847 while ( PF_GetTerm(term) ) {
01848     PF_linterms++; AN.ninterms++; dd = AN.deferskipped;
01849     AT.WorkPointer = term + *term;
01850     AN.RepPoint = AT.RepCount + 1;
01851     if ( ( e->vflags & ISFACTORIZED ) != 0 && term[1] == HAAKJE ) {
01852         StoreTerm(BHEAD term);
01853         continue;
01854     }
01855     if ( AR.DeferFlag ) {
01856         AR.CurDum = AN.IndDum = Expressions[AR.CurExpr].numdummies + AM.IndDum;
01857     }
01858     else {
01859         AN.IndDum = AM.IndDum;
01860         AR.CurDum = ReNumber(BHEAD term);
01861     }
01862     if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMLAG);
01863     if ( AN.ncmod ) {
01864         if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
01865         else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
01866     }
01867     else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
01868     if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
01869         && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
01870         PolyFunClean(BHEAD term);
01871     }
01872     if ( Generator(BHEAD term,0) ) {
01873         MesPrint("[%d] PF_Processor: Error in Generator",PF.me);
01874         LowerSortLevel(); return(-1);
01875     }
01876     PF_linterms += dd; AN.ninterms += dd;
01877 }
01878 PF_linterms += dd; AN.ninterms += dd;
01879 {
01880     /*
01881     * EndSort() overrides AR.outfile->PObuffer etc. (See also PF_EndSort()),
01882     * but it causes a problem because
01883     * (1) PF_EndSort() sets AR.outfile->PObuffer to a send-buffer.
01884     * (2) RevertScratch() clears AR.infile, but then swaps buffers of AR.infile
01885     *     and AR.outfile.
01886     * (3) RHS expressions are stored to AR.infile->PObuffer.
01887     * (4) Again, PF_EndSort() sets AR.outfile->PObuffer, but now AR.outfile->PObuffer
01888     *     == AR.infile->PObuffer because of (1) and (2).
01889     * (5) The result goes to AR.outfile. This breaks the RHS expressions,
01890     *     which may be needed for the next expression.
01891     * Solution: backup & restore AR.outfile->PObuffer etc. (TU 14 Sep 2011)
01892     */
01893     FILEHANDLE *fout = AR.outfile;
01894     WORD *oldbuff = fout->PObuffer;
01895     WORD *oldstop = fout->POstop;
01896     LONG oldsize = fout->POsize;
01897     if ( EndSort(BHEAD AM.S0->sBuffer, 0) < 0 ) return -1;
01898     fout->PObuffer = oldbuff;
01899     fout->POstop = oldstop;
01900     fout->POsize = oldsize;
01901     fout->POfill = fout->POfull = fout->PObuffer;
01902 }
01903 AR.BracketOn = oldBracketOn;
01904 AT.BrackBuf = oldBrackBuf;
01905 AT.bracketindexflag = oldbracketindexflag;
01906 /*
01907     #] Generator Loop & EndSort :

```

```

01908         #] Collect (stats,prepro...) :
01909     /*
01910         DBGOUT_NINTERMS(1, ("PF.me=%d AN.ninterms=%d PF_linterms=%d ENDSORT\n", (int)PF.me,
(int)AN.ninterms, (int)PF_linterms));
01911         PF_PrepareLongSinglePack();
01912         cpu = TimeCPU(1);
01913         size = 0;
01914         PF_LongSinglePack(&cpu, 1, PF_LONG);
01915         PF_LongSinglePack(&size, 1, PF_LONG);
01916         PF_LongSinglePack(&PF_linterms, 1, PF_LONG);
01917         PF_LongSinglePack(&AM.S0->GenTerms, 1, PF_LONG);
01918         PF_LongSinglePack(&AM.S0->TermsLeft, 1, PF_LONG);
01919         {
01920             WORD numdummies = AR.MaxDum - AM.IndDum;
01921             PF_LongSinglePack(&numdummies, 1, PF_WORD);
01922             PF_LongSinglePack(&AR.expchanged, 1, PF_WORD);
01923         }
01924         /* Now handle redefined preprocessor variables. */
01925         if ( AC.numpfirstnum > 0 ) PF_PackRedefinedPreVars();
01926         PF_LongSingleSend(MASTER, PF_ENDSORT_MSGTAG);
01927         /* Broadcast redefined preprocessor variables. */
01928         if ( AC.numpfirstnum > 0 ) {
01929             int RetCode = PF_BroadcastRedefinedPreVars();
01930             if ( RetCode ) return RetCode;
01931         }
01932     /*
01933         #] Collect (stats,prepro...) :
01934
01935         This operation is moved to the beginning of each block, see PreProcessor
01936         in pre.c.
01937
01938         #] Slave :
01939     */
01940         if ( PF.log ) {
01941             UBYTE lbuf[24];
01942             NumToStr(lbuf,AC.CModule);
01943             fprintf(stderr,"%d[%s] Endsort,Collect,Broadcast done\n",PF.me,lbuf);
01944             fflush(stderr);
01945         }
01946     }
01947     return(0);
01948 }
01949
01950 /*
01951     #] PF_Processor :
01952     #] proces.c :
01953     #[ startup :, prepro & compile
01954     #[ PF_Init :
01955 */
01956
01965 int PF_Init(int argc, char ***argv)
01966 {
01967     /*
01968         this should definitely be somewhere else ...
01969     */
01970     PF_CurrentBracket = 0;
01971
01972     PF.numtasks = 0; /* number of tasks, is determined in PF_LibInit ! */
01973     PF.numbufs = 2; /* might be changed by the environment variable on the master ! */
01974     PF.numrbufs = 2; /* might be changed by the environment variable on the master ! */
01975
01976     int ret = PF_LibInit(argc,argv);
01977     if (ret) { return ret; }
01978     PF_RealTime(PF_RESET);
01979
01980     PF.log = 0;
01981     PF.parallel = 0;
01982     PF_statsinterval = 10;
01983     PF.rhsInParallel=1;
01984     PF.exprbufsize=4096; /*in WORDs*/
01985
01986 #ifdef PF_WITHGETENV
01987     if ( PF.me == MASTER ) {
01988         char *c;
01989     /*
01990         get these from the environment at the moment should be in setfile/tail
01991     */
01992     if ( ( c = getenv("PF_LOG") ) != 0 ) {
01993         if ( *c ) PF.log = (int)atoi(c);
01994         else PF.log = 1;
01995         fprintf(stderr,"%d] changing PF.log to %d\n",PF.me,PF.log);
01996         fflush(stderr);
01997     }
01998     if ( ( c = (char*)getenv("PF_RBUFS") ) != 0 ) {
01999         PF.numrbufs = (int)atoi(c);
02000         fprintf(stderr,"%d] changing numrbufs to: %d\n",PF.me,PF.numrbufs);
02001         fflush(stderr);

```

```

02002     }
02003     if ( ( c = (char*)getenv("PF_SBUFS") ) != 0 ) {
02004         PF.numsbufs = (int)atoi(c);
02005         fprintf(stderr, "[%d] changing numsbufs to: %d\n", PF.me, PF.numsbufs);
02006         fflush(stderr);
02007     }
02008     if ( PF.numsbufs > 10 ) PF.numsbufs = 10;
02009     if ( PF.numsbufs < 1 ) PF.numsbufs = 1;
02010     if ( PF.numrbufs > 2 ) PF.numrbufs = 2;
02011     if ( PF.numrbufs < 1 ) PF.numrbufs = 1;
02012
02013     if ( ( c = getenv("PF_STATS") ) ) {
02014         UBYTE lbuf[24];
02015         PF_statsinterval = (int)atoi(c);
02016         NumToStr(lbuf, PF_statsinterval);
02017         fprintf(stderr, "[%d] changing PF_statsinterval to %s\n", PF.me, lbuf);
02018         fflush(stderr);
02019         if ( PF_statsinterval < 1 ) PF_statsinterval = 10;
02020     }
02021 }
02022 #endif
02023 /*
02024  #] Broadcast settings from getenv: could also be done in PF_DoSetup
02025  */
02026 if ( PF.me == MASTER ) {
02027     PF_PrepPack();
02028     PF_Pack(&PF.log, 1, PF_INT);
02029     PF_Pack(&PF.numrbufs, 1, PF_WORD);
02030     PF_Pack(&PF.numsbufs, 1, PF_WORD);
02031 }
02032 PF_Broadcast();
02033 if ( PF.me != MASTER ) {
02034     PF_Unpack(&PF.log, 1, PF_INT);
02035     PF_Unpack(&PF.numrbufs, 1, PF_WORD);
02036     PF_Unpack(&PF.numsbufs, 1, PF_WORD);
02037     if ( PF.log ) {
02038         fprintf(stderr, "[%d] log=%d rbufs=%d sbufs=%d\n",
02039             PF.me, PF.log, PF.numrbufs, PF.numsbufs);
02040         fflush(stderr);
02041     }
02042 }
02043 /*
02044  #] Broadcast settings from getenv:
02045  */
02046 return(0);
02047 }
02048 /*
02049     #] PF_Init :
02050     #] PF_Terminate :
02051  */
02052
02060 int PF_Terminate(int errorcode)
02061 {
02062     return PF_LibTerminate(errorcode);
02063 }
02064
02065 /*
02066     #] PF_Terminate :
02067     #] PF_GetSlaveTimes :
02068  */
02069
02076 LONG PF_GetSlaveTimes(void)
02077 {
02078     LONG slavetimes = 0;
02079     LONG t = PF.me == MASTER ? 0 : AM.SumTime + TimeCPU(1);
02080     MPI_Reduce(&t, &slavetimes, 1, PF_LONG, MPI_SUM, MASTER, PF_COMM);
02081     return slavetimes;
02082 }
02083
02084 /*
02085     #] PF_GetSlaveTimes :
02086     #] startup :
02087     #] PF_BroadcastNumber :
02088  */
02089
02096 LONG PF_BroadcastNumber(LONG x)
02097 {
02098     #ifdef PF_DEBUG_BCAST_LONG
02099     if ( PF.me == MASTER ) {
02100         MesPrint("» Broadcast LONG: %1", x);
02101     }
02102     #endif
02103     PF_Bcast(&x, sizeof(LONG));
02104     return x;
02105 }
02106
02107 /*

```

```

02108     #] PF_BroadcastNumber :
02109     #[ PF_BroadcastBuffer :
02110 */
02111
02123 void PF_BroadcastBuffer(WORD **buffer, LONG *length)
02124 {
02125     WORD *p;
02126     LONG rest;
02127     #ifndef PF_DEBUG_BCAST_BUF
02128     if ( PF.me == MASTER ) {
02129         MesPrint(">> Broadcast Buffer: length=%l", *length);
02130     }
02131     #endif
02132     /* Initialize the buffer on the slaves. */
02133     if ( PF.me != MASTER ) {
02134         *buffer = NULL;
02135     }
02136     /* Broadcast the length of the buffer. */
02137     *length = PF_BroadcastNumber(*length);
02138     if ( *length <= 0 ) return;
02139     /* Allocate the buffer on the slaves. */
02140     if ( PF.me != MASTER ) {
02141         *buffer = (WORD *)Malloc1(*length * sizeof(WORD), "PF_BroadcastBuffer");
02142     }
02143     /* Broadcast the data in the buffer. */
02144     p = *buffer;
02145     rest = *length;
02146     while ( rest > 0 ) {
02147         int l = rest < (LONG)PF.exprbufsize ? (int)rest : PF.exprbufsize;
02148         PF_Bcast(p, l * sizeof(WORD));
02149         p += l;
02150         rest -= l;
02151     }
02152 }
02153
02154 /*
02155 #] PF_BroadcastBuffer :
02156 #[ PF_BroadcastString :
02157 */
02158
02165 int PF_BroadcastString(UBYTE *str)
02166 {
02167     int clength = 0;
02168     /*
02169         If string does not fit to the PF_buffer, it
02170         will be split into chunks. Next chunk is started at str+clength
02171     */
02172     UBYTE *cstr=str;
02173     /*
02174         Note, compilation is performed INDEPENDENTLY on AC.mparallelfalg!
02175         No if ( AC.mparallelfalg == PARALLELFLAG ) !!
02176     */
02177     do {
02178         cstr += clength; /*at each step for all slaves and master */
02179         if ( MASTER == PF.me ) { /*Pack str*/
02180             /*
02181                 initialize buffers
02182             */
02183             if ( PF_PreparePack() != 0 ) Terminate(-1);
02184             if ( ( clength = PF_PackString(cstr) ) < 0 ) Terminate(-1);
02185         }
02186         PF_Broadcast();
02187         if ( MASTER != PF.me ) {
02188             /*
02189                 Slave - unpack received string
02190                 For slaves buffers are initialised automatically.
02191             */
02192             if ( ( clength = PF_UnpackString(cstr) ) < 0 ) Terminate(-1);
02193         }
02194     } while ( cstr[clength-1] != '\0' );
02195     return (0);
02196 }
02197
02198 /*
02199 #] PF_BroadcastString :
02200 #[ PF_BroadcastPreDollar :
02201 */
02202
02220 int PF_BroadcastPreDollar(WORD **dbuffer, LONG *newsize, int *numterms)
02221 {
02222     int err = 0;
02223     LONG i;
02224     /*
02225         Note, compilation is performed INDEPENDENTLY on AC.mparallelfalg!
02226         No if(AC.mparallelfalg==PARALLELFLAG) !!

```

```

02227 */
02228     if ( MASTER == PF.me ) {
02229 /*
02230         The problem is that sometimes dollar variables are longer
02231         than PF_packbuf! So we split long expression into chunks.
02232         There are n filled chunks and one partially filled chunk:
02233 */
02234         LONG n = ((*newsize)+1)/PF_maxDollarChunkSize;
02235 /*
02236         ...and one more chunk for the rest; if the expression fits to
02237         the buffer without splitting, the latter will be the only one.
02238
02239         PF_maxDollarChunkSize is the maximal number of items fitted to
02240         the buffer. It is calculated in PF_LibInit() in mpi.c.
02241         PF_maxDollarChunkSize is calculated for the first step, when
02242         two fields (numterms and newsize, see below) are already packed.
02243         For simplicity, this value is used also for all steps, in
02244         despite of it is a bit less than maximally available space.
02245 */
02246         WORD *thechunk = *dbuffer;
02247
02248         err = PF_PreparePack(); /* initialize buffers */
02249         err |= PF_Pack(numterms,1,PF_INT);
02250         err |= PF_Pack(newsize,1,PF_LONG); /* pack the size */
02251 /*
02252         Pack and broadcast completely filled chunks.
02253         It may happen, this loop is not entered at all:
02254 */
02255         for ( i = 0; i < n; i++ ) {
02256             err |= PF_Pack(thechunk,PF_maxDollarChunkSize,PF_WORD);
02257             err |= PF_Broadcast();
02258             thechunk +=PF_maxDollarChunkSize;
02259             PF_PreparePack();
02260         }
02261 /*
02262         Pack and broadcast the rest:
02263 */
02264         if ( ( n = ( (*newsize)+1)%PF_maxDollarChunkSize ) != 0 ) {
02265             err |= PF_Pack(thechunk,n,PF_WORD);
02266             err |= PF_Broadcast();
02267         }
02268 #ifdef PF_DEBUG_BCAST_PREDOLLAR
02269         MesPrint("> Broadcast PreDollar: newsize=%d numterms=%d", (int)*newsize, *numterms);
02270 #endif
02271     }
02272     if ( MASTER != PF.me ) { /* Slave - unpack received buffer */
02273         WORD *thechunk;
02274         LONG n, therest, thesize;
02275         err |= PF_Broadcast();
02276         err |=PF_Unpack(numterms,1,PF_INT);
02277         err |=PF_Unpack(newsize,1,PF_LONG);
02278 /*
02279         Now we know the buffer size.
02280 */
02281         thesize = (*newsize)+1;
02282 /*
02283         Evaluate the number of completely filled chunks. The last step must be
02284         treated separately, so -1:
02285 */
02286         n = (thesize/PF_maxDollarChunkSize) - 1;
02287 /*
02288         Note, here n can be <0, this is ok.
02289 */
02290         therest = thesize % PF_maxDollarChunkSize;
02291         thechunk = *dbuffer =
02292             (WORD*)Mallocl( thesize * sizeof(WORD), "$-buffer slave");
02293         if ( thechunk == NULL ) return(err|4);
02294 /*
02295         Unpack completely filled chunks and receive the next portion.
02296         It may happen, this loop is not entered at all:
02297 */
02298         for ( i = 0; i < n; i++ ) {
02299             err |= PF_Unpack(thechunk,PF_maxDollarChunkSize,PF_WORD);
02300             thechunk += PF_maxDollarChunkSize;
02301             err |= PF_Broadcast();
02302         }
02303 /*
02304         Now the last completely filled chunk:
02305 */
02306         if ( n >= 0 ) {
02307             err |= PF_Unpack(thechunk,PF_maxDollarChunkSize,PF_WORD);
02308             thechunk += PF_maxDollarChunkSize;
02309             if ( therest != 0 ) err |= PF_Broadcast();
02310         }
02311 /*
02312         Unpack the rest (it is already received!):
02313 */

```



```

02314         if ( therest != 0 ) err |= PF_Unpack(thechunk,therest,PF_WORD);
02315     }
02316     return (err);
02317 }
02318
02319 /*
02320  #] PF_BroadcastPreDollar :
02321  #[ Synchronization of modified dollar variables :
02322  #[ Helper functions :
02323  #[ dollarlen :
02324  */
02325
02329 static inline LONG dollarlen(const WORD *terms)
02330 {
02331     const WORD *p = terms;
02332     while ( *p ) p += *p;
02333     return p - terms; /* Not including the null terminator. */
02334 }
02335
02336 /*
02337  #] dollarlen :
02338  #[ dollar_mod_type :
02339  */
02340
02345 static inline WORD dollar_mod_type(WORD index)
02346 {
02347     int i;
02348     for ( i = 0; i < NumModOptdollars; i++ )
02349         if ( ModOptdollars[i].number == index ) break;
02350     if ( i >= NumModOptdollars ) return -1;
02351     return ModOptdollars[i].type;
02352 }
02353
02354
02355 /*
02356  #] dollar_mod_type :
02357  #] Helper functions :
02358  #[ PF_CollectModifiedDollars :
02359  */
02360
02361 /*
02362  #[ dollar_to_be_collected :
02363  */
02364
02369 static inline int dollar_to_be_collected(WORD index)
02370 {
02371     switch ( dollar_mod_type(index) ) {
02372     case MODSUM:
02373     case MODMAX:
02374     case MODMIN:
02375         return 1;
02376     default:
02377         return 0;
02378     }
02379 }
02380
02381 /*
02382  #] dollar_to_be_collected :
02383  #[ copy_dollar :
02384  */
02385
02390 static inline void copy_dollar(WORD index, WORD type, const WORD *where, LONG size)
02391 {
02392     DOLLARS d = Dollars + index;
02393
02394     CleanDollarFactors(d);
02395
02396     if ( type != DOLZERO && where != NULL && where != &AM.dollarzero && where[0] != 0 && size > 0 ) {
02397         if ( size > d->size || size < d->size / 4 ) { /* Reallocate if not enough or too much. */
02398             if ( d->where && d->where != &AM.dollarzero )
02399                 M_free(d->where, "old content of dollar");
02400             d->where = Malloc1(sizeof(WORD) * size, "copy buffer to dollar");
02401             d->size = size;
02402         }
02403         d->type = type;
02404         WCOPY(d->where, where, size);
02405     }
02406     else {
02407         if ( d->where && d->where != &AM.dollarzero )
02408             M_free(d->where, "old content of dollar");
02409         d->type = DOLZERO;
02410         d->where = &AM.dollarzero;
02411         d->size = 0;
02412     }
02413 }
02414
02415 /*

```

```

02416         #] copy_dollar :
02417         #[ compare_two_expressions :
02418     */
02419
02424 static inline int compare_two_expressions(const WORD *e1, const WORD *e2)
02425 {
02426     GETIDENTITY
02427     /*
02428      * We consider the cases that
02429      *   (1) the expression has no term,
02430      *   (2) the expression has only one term and it is a number,
02431      *   (3) otherwise.
02432      * Assume that the expressions are sorted and all terms are normalized.
02433      * The numerators of the coefficients must never be zero.
02434      *
02435      * Note that TwoExprCompare() is not adequate for our purpose
02436      * (as of 6 Aug. 2013), e.g., TwoExprCompare({0}, {4, 1, 1, -1}, LESS)
02437      * returns TRUE.
02438      */
02439     if ( e1[0] == 0 ) {
02440         if ( e2[0] == 0 ) {
02441             return(0);
02442         }
02443         else if ( e2[e2[0]] == 0 && e2[0] == ABS(e2[e2[0] - 1]) + 1 ) {
02444             if ( e2[e2[0] - 1] > 0 )
02445                 return(-1);
02446             else
02447                 return(+1);
02448         }
02449     }
02450     else if ( e1[e1[0]] == 0 && e1[0] == ABS(e1[e1[0] - 1]) + 1 ) {
02451         if ( e2[0] == 0 ) {
02452             if ( e1[e1[0] - 1] > 0 )
02453                 return(+1);
02454             else
02455                 return(-1);
02456         }
02457         else if ( e2[e2[0]] == 0 && e2[0] == ABS(e2[e2[0] - 1]) + 1 ) {
02458             return(CompCoef((WORD *)e1, (WORD *)e2));
02459         }
02460     }
02461     /* The expressions are not so simple. Define the order by each term. */
02462     while ( e1[0] && e2[0] ) {
02463         int c = CompareTerms(BHEAD (WORD *)e1, (WORD *)e2, 1);
02464         if ( c < 0 )
02465             return(-1);
02466         else if ( c > 0 )
02467             return(+1);
02468         e1 += e1[0];
02469         e2 += e2[0];
02470     }
02471     if ( e1[0] ) return(+1);
02472     if ( e2[0] ) return(-1);
02473     return(0);
02474 }
02475
02476 /*
02477         #] compare_two_expressions :
02478         #[ Variables :
02479     */
02480
02481 typedef struct {
02482     VectorStruct(WORD) buf;
02483     LONG size;
02484     WORD type;
02485 } dollar_buf;
02486
02487 /* Buffers used to store data for each variable from each slave. */
02488 static Vector(dollar_buf, dollar_slave_bufs);
02489
02490 /*
02491         #] Variables :
02492     */
02493
02507 int PF_CollectModifiedDollars(void)
02508 {
02509     int i, j, ndollars;
02510     /*
02511      * If the current module was executed in the sequential mode,
02512      * there are no modified module on the slaves.
02513      */
02514     if ( AC.mparallelflag != PARALLELFLAG && !AC.partodoflag ) return 0;
02515     /*
02516      * Count the number of (potentially) modified dollar variables, which we need to collect.
02517      * Here we need to collect all max/min/sum variables.
02518      */
02519     ndollars = 0;

```

```

02520     for ( i = 0; i < NumPotModdollars; i++ ) {
02521         WORD index = PotModdollars[i];
02522         if ( dollar_to_be_collected(index) ) ndollars++;
02523     }
02524     if ( ndollars == 0 ) return 0; /* No dollars to be collected. */
02525
02526     if ( PF.me == MASTER ) {
02527         /*
02528             # [ Master :
02529         */
02530         int nslaves, nvars;
02531         /* Prepare receive buffers. We need ndollars*(PF.numtasks-1) buffers. */
02532         int nbufs = ndollars * (PF.numtasks - 1);
02533         VectorReserve(dollar_slave_bufs, nbufs);
02534         for ( i = VectorSize(dollar_slave_bufs); i < nbufs; i++ ) {
02535             VectorInit(VectorPtr(dollar_slave_bufs)[i].buf);
02536         }
02537         VectorSize(dollar_slave_bufs) = nbufs;
02538         /* Receive data from each slave. */
02539         for ( nslaves = 1; nslaves < PF.numtasks; nslaves++ ) {
02540             int src;
02541             PF_LongSingleReceive(PF_ANY_SOURCE, PF_DOLLAR_MSGTAG, &src, NULL);
02542             nvars = 0;
02543             for ( i = 0; i < NumPotModdollars; i++ ) {
02544                 WORD index = PotModdollars[i];
02545                 dollar_buf *b;
02546                 if ( !dollar_to_be_collected(index) ) continue;
02547                 b = &VectorPtr(dollar_slave_bufs)[(PF.numtasks - 1) * nvars + (src - 1)];
02548                 PF_LongSingleUnpack(&b->type, 1, PF_WORD);
02549                 if ( b->type != DOLZERO ) {
02550                     LONG size;
02551                     WORD *where;
02552                     PF_LongSingleUnpack(&size, 1, PF_LONG);
02553                     VectorReserve(b->buf, size + 1);
02554                     where = VectorPtr(b->buf);
02555                     PF_LongSingleUnpack(where, size, PF_WORD);
02556                     where[size] = 0; /* The null terminator is needed. */
02557                     b->size = size + 1; /* Including the null terminator. */
02558                     /* Note that we don't collect factored stuff for max/min/sum variables. */
02559                 }
02560                 else {
02561                     VectorReserve(b->buf, 1);
02562                     VectorPtr(b->buf)[0] = 0;
02563                     b->size = 0;
02564                 }
02565                 nvars++;
02566             }
02567         }
02568         /*
02569         * Combine received dollars. The FORM reference manual says maximum/minimum/sum
02570         * $-variables must have a numerical value, however, this routine should work also
02571         * for non-numerical cases, although the maximum/minimum value for non-numerical
02572         * terms has ambiguity.
02573         */
02574         nvars = 0;
02575         for ( i = 0; i < NumPotModdollars; i++ ) {
02576             WORD index = PotModdollars[i];
02577             WORD dtype;
02578             DOLLARS d;
02579             dollar_buf *b;
02580             if ( !dollar_to_be_collected(index) ) continue;
02581             d = Dollars + index;
02582             b = &VectorPtr(dollar_slave_bufs)[(PF.numtasks - 1) * nvars];
02583             dtype = dollar_mod_type(index);
02584             switch ( dtype ) {
02585                 case MODMAX:
02586                 case MODMIN: {
02587                     /*
02588                     # [ MODMAX & MODMIN :
02589                     */
02590                     int selected = 0;
02591                     for ( j = 1; j < PF.numtasks - 1; j++ ) {
02592                         int c = compare_two_expressions(VectorPtr(b[j].buf),
02593                             VectorPtr(b[selected].buf));
02594                         if ( (dtype == MODMAX && c > 0) || (dtype == MODMIN && c < 0) )
02595                             selected = j;
02596                     }
02597                     b = b + selected;
02598                     copy_dollar(index, b->type, VectorPtr(b->buf), b->size);
02599                     #] MODMAX & MODMIN :
02600                 */
02601                     break;
02602                 }
02603                 case MODSUM: {
02604                     /*
02605                     # [ MODSUM :

```

```

02606 */
02607 GETIDENTITY
02608 int err = 0;
02609
02610 CBUF *C = cbuf + AM.rbufnum;
02611 WORD *oldwork = AT.WorkPointer, *oldcterm = AN.cTerm;
02612 WORD olddefer = AR.DeferFlag, oldnumlhs = AR.Cnumlhs, oldnumrhs = C->numrhs;
02613
02614 LONG size;
02615 WORD type, *dbuf;
02616
02617 AN.cTerm = 0;
02618 AR.DeferFlag = 0;
02619
02620 if ( ((WORD *) ((UBYTE *) AT.WorkPointer + AM.MaxTer)) > AT.WorkTop ) {
02621     err = -1;
02622     goto cleanup;
02623     MesWork();
02624 }
02625
02626 if ( NewSort(BHEAD0) ) {
02627     err = -1;
02628     goto cleanup;
02629 }
02630 if ( NewSort(BHEAD0) ) {
02631     LowerSortLevel();
02632     err = -1;
02633     goto cleanup;
02634 }
02635
02636 /*
02637  * Sum up the original $-variable in the master and $-variables on all slaves.
02638  * Note that $-variables on the slaves are set to zero at the beginning of
02639  * the module (See also DoExecute()).
02640  */
02641 for ( j = 0; j < PF.numtasks; j++ ) {
02642     const WORD *r;
02643     for ( r = j == 0 ? Dollars[index].where : VectorPtr(b[j - 1].buf); *r; r += *r
02644 ) {
02645         WCOPY(AT.WorkPointer, r, *r);
02646         AT.WorkPointer += *r;
02647         AR.Cnumlhs = 0;
02648         if ( Generator(BHEAD oldwork, 0) ) {
02649             LowerSortLevel(); LowerSortLevel();
02650             err = -1;
02651             goto cleanup;
02652         }
02653         AT.WorkPointer = oldwork;
02654     }
02655 }
02656 size = EndSort(BHEAD (WORD *)&dbuf, 2);
02657 if ( size < 0 ) {
02658     LowerSortLevel();
02659     err = -1;
02660     goto cleanup;
02661 }
02662 LowerSortLevel();
02663
02664 /* Find special cases. */
02665 type = DOLTERMS;
02666 if ( dbuf[0] == 0 ) {
02667     type = DOLZERO;
02668 }
02669 else if ( dbuf[dbuf[0]] == 0 ) {
02670     const WORD *t = dbuf, *w;
02671     WORD n, nsize;
02672     n = *t;
02673     nsize = t[n - 1];
02674     if ( nsize < 0 ) nsize = -nsize;
02675     if ( nsize == n - 1 ) {
02676         nsize = (nsize - 1) / 2;
02677         w = t + 1 + nsize;
02678         if ( *w == 1 ) {
02679             w++; while ( w < t + n - 1 ) { if ( *w ) break; w++; }
02680             if ( w >= t + n - 1 ) type = DOLNUMBER;
02681         }
02682         else if ( n == 7 && t[6] == 3 && t[5] == 1 && t[4] == 1 && t[1] == INDEX
&& t[2] == 3 ) {
02683             type = DOLINDEX;
02684             d->index = t[3];
02685         }
02686     }
02687 }
02688 copy_dollar(index, type, dbuf, dollarlen(dbuf) + 1);
02689 M_free(dbuf, "temporary dollar buffer");
02690 cleanup:

```

```

02691         AR.Cnumlhs = oldnumlhs;
02692         C->numrhs = oldnumrhs;
02693         AR.DeferFlag = olddefer;
02694         AN.cTerm = oldcTerm;
02695         AT.WorkPointer = oldwork;
02696
02697         if ( err ) return err;
02698 /*
02699         #] MODSUM :
02700 */
02701         break;
02702     }
02703 }
02704 if ( d->type == DOLTERMS )
02705     cbuf[AM.dbufnum].CanCommu[index] = numcommute(d->where,
02706     &cbuf[AM.dbufnum].NumTerms[index]);
02707     cbuf[AM.dbufnum].rhs[index] = d->where;
02708     nvars++;
02709 #ifdef PF_DEBUG_REDUCE_DOLLAR
02710     MesPrint("< Reduce $-var: %s", AC.dollarname->namebuffer + d->name);
02711 #endif
02712 }
02713 /*
02714         #] Master :
02715 */
02716     }
02717 else {
02718 /*
02719         #[ Slave :
02720 */
02721     PF_PreparesLongSinglePack();
02722     /* Pack each variable. */
02723     for ( i = 0; i < NumPotModdollars; i++ ) {
02724         WORD index = PotModdollars[i];
02725         DOLLARS d;
02726         if ( !dollar_to_be_collected(index) ) continue;
02727         d = Dollars + index;
02728         PF_LongSinglePack(&d->type, 1, PF_WORD);
02729         if ( d->type != DOLZERO ) {
02730             /*
02731              * NOTE: d->size is the allocated buffer size for d->where in WORDs.
02732              *       So dollarlen(d->where) can be < d->size-1. (TU 15 Dec 2011)
02733              */
02734             LONG size = dollarlen(d->where);
02735             PF_LongSinglePack(&size, 1, PF_LONG);
02736             PF_LongSinglePack(d->where, size, PF_WORD);
02737             /* Note that we don't collect factored stuff for max/min/sum variables. */
02738         }
02739     }
02740     PF_LongSingleSend(MASTER, PF_DOLLAR_MSGTAG);
02741 /*
02742         #] Slave :
02743 */
02744     }
02745     return 0;
02746 }
02747 /*
02748         #] PF_CollectModifiedDollars :
02749         #[ PF_BroadcastModifiedDollars :
02750 */
02751
02752 /*
02753         #[ dollar_to_be_broadcast :
02754 */
02755
02756 static inline int dollar_to_be_broadcast(WORD index)
02757 {
02758     switch ( dollar_mod_type(index) ) {
02759     case MODLOCAL:
02760         return 0;
02761     default:
02762         return 1;
02763     }
02764 }
02765
02766 /*
02767         #] dollar_to_be_broadcast :
02768 */
02769
02770 int PF_BroadcastModifiedDollars(void)
02771 {
02772     int i, j, ndollars;
02773     /*
02774      * Count the number of (potentially) modified dollar variables, which we need to broadcast.
02775      * Here we need to broadcast all non-local variables.
02776      */

```

```

02793     ndollars = 0;
02794     for ( i = 0; i < NumPotModdollars; i++ ) {
02795         WORD index = PotModdollars[i];
02796         if ( dollar_to_be_broadcast(index) ) ndollars++;
02797     }
02798     if ( ndollars == 0 ) return 0; /* No dollars to be broadcast. */
02799
02800     if ( PF.me == MASTER ) {
02801 /*
02802         #[] Master :
02803 */
02804         PF_PrepareLongMultiPack();
02805         /* Pack each variable. */
02806         for ( i = 0; i < NumPotModdollars; i++ ) {
02807             WORD index = PotModdollars[i];
02808             DOLLARS d;
02809             if ( !dollar_to_be_broadcast(index) ) continue;
02810             d = Dollars + index;
02811             PF_LongMultiPack(&d->type, 1, PF_WORD);
02812             if ( d->type != DOLZERO ) {
02813                 /*
02814                  * NOTE: d->size is the allocated buffer size for d->where in WORDs.
02815                  *       So dollarlen(d->where) can be < d->size-1. (TU 15 Dec 2011)
02816                  */
02817                 LONG size = dollarlen(d->where);
02818                 PF_LongMultiPack(&size, 1, PF_LONG);
02819                 PF_LongMultiPack(d->where, size, PF_WORD);
02820                 /* ...and the factored stuff. */
02821                 PF_LongMultiPack(&d->nfactors, 1, PF_WORD);
02822                 if ( d->nfactors > 1 ) {
02823                     for ( j = 0; j < d->nfactors; j++ ) {
02824                         FACDOLLAR *f = &d->factors[j];
02825                         PF_LongMultiPack(&f->type, 1, PF_WORD);
02826                         PF_LongMultiPack(&f->size, 1, PF_LONG);
02827                         if ( f->size > 0 )
02828                             PF_LongMultiPack(f->where, f->size, PF_WORD);
02829                         else
02830                             PF_LongMultiPack(&f->value, 1, PF_WORD);
02831                     }
02832                 }
02833             }
02834 #ifdef PF_DEBUG_BCAST_DOLLAR
02835             MesPrint("» Broadcast $-var: %s", AC.dollarnames->namebuffer + d->name);
02836 #endif
02837         }
02838 /*
02839         #[] Master :
02840 */
02841     }
02842     if ( PF_LongMultiBroadcast() ) return -1;
02843     if ( PF.me != MASTER ) {
02844 /*
02845         #[] Slave :
02846 */
02847         for ( i = 0; i < NumPotModdollars; i++ ) {
02848             WORD index = PotModdollars[i];
02849             DOLLARS d;
02850             if ( !dollar_to_be_broadcast(index) ) continue;
02851             d = Dollars + index;
02852             /* Clear the contents of the dollar variable. */
02853             if ( d->where && d->where != &AM.dollarzero )
02854                 M_free(d->where, "old content of dollar");
02855             d->where = &AM.dollarzero;
02856             d->size = 0;
02857             CleanDollarFactors(d);
02858             /* Unpack and store the contents. */
02859             PF_LongMultiUnpack(&d->type, 1, PF_WORD);
02860             if ( d->type != DOLZERO ) {
02861                 LONG size;
02862                 PF_LongMultiUnpack(&size, 1, PF_LONG);
02863                 d->size = size + 1;
02864                 d->where = (WORD *)Malloca(sizeof(WORD) * d->size, "dollar content");
02865                 PF_LongMultiUnpack(d->where, size, PF_WORD);
02866                 d->where[size] = 0; /* The null terminator is needed. */
02867                 /* ...and the factored stuff. */
02868                 PF_LongMultiUnpack(&d->nfactors, 1, PF_WORD);
02869                 if ( d->nfactors > 1 ) {
02870                     d->factors = (FACDOLLAR *)Malloca(sizeof(FACDOLLAR) * d->nfactors, "dollar
factored stuff");
02871                     for ( j = 0; j < d->nfactors; j++ ) {
02872                         FACDOLLAR *f = &d->factors[j];
02873                         PF_LongMultiUnpack(&f->type, 1, PF_WORD);
02874                         PF_LongMultiUnpack(&f->size, 1, PF_LONG);
02875                         if ( f->size > 0 ) {
02876                             f->where = (WORD *)Malloca(sizeof(WORD) * (f->size + 1), "dollar factor
content");
02877                             PF_LongMultiUnpack(f->where, f->size, PF_WORD);

```

```

02878             f->where[f->size] = 0; /* The null terminator is needed. */
02879             f->value = 0;
02880         }
02881         else {
02882             f->where = NULL;
02883             PF_LongMultiUnpack(&f->value, 1, PF_WORD);
02884         }
02885     }
02886 }
02887 }
02888 if ( d->type == DOLTERMS )
02889     cbuf[AM.dbufnum].CanCommu[index] = numcommute(d->where,
02890 &cbuf[AM.dbufnum].NumTerms[index]);
02891     cbuf[AM.dbufnum].rhs[index] = d->where;
02892 }
02893 /*
02894     #] Slave :
02895 */
02896 }
02897 return 0;
02898 }
02899 /*
02900     #] PF_BroadcastModifiedDollars :
02901     #] Synchronization of modified dollar variables :
02902     #] Synchronization of redefined preprocessor variables :
02903     #] Variables :
02904 */
02905 /* A buffer used in receivers. */
02906 static Vector(UBYTE, prevarbuf);
02907 /*
02908     #] Variables :
02909     #] PF_PackRedefinedPreVars :
02910 */
02911 static void PF_PackRedefinedPreVars(void)
02912 {
02913     int i;
02914     /* First, pack the number of redefined preprocessor variables. */
02915     int nredefs = 0;
02916     for ( i = 0; i < AC.numpfirstnum; i++ )
02917         if ( AC.inputnumbers[i] >= 0 ) nredefs++;
02918     PF_LongSinglePack(&nredefs, 1, PF_INT);
02919     /* Then, pack each variable. */
02920     for ( i = 0; i < AC.numpfirstnum; i++ )
02921         if ( AC.inputnumbers[i] >= 0 ) {
02922             WORD index = AC.pfirstnum[i];
02923             UBYTE *value = PreVar[index].value;
02924             int bytes = strlen((char *)value);
02925             PF_LongSinglePack(&index, 1, PF_WORD);
02926             PF_LongSinglePack(&bytes, 1, PF_INT);
02927             PF_LongSinglePack(value, bytes, PF_BYTE);
02928             PF_LongSinglePack(&AC.inputnumbers[i], 1, PF_LONG);
02929         }
02930 }
02931 /*
02932     #] PF_PackRedefinedPreVars :
02933     #] PF_UnpackRedefinedPreVars :
02934 */
02935 static void PF_UnpackRedefinedPreVars(void)
02936 {
02937     int i, j;
02938     /* Unpack the number of redefined preprocessor variables. */
02939     int nredefs;
02940     PF_LongSingleUnpack(&nredefs, 1, PF_INT);
02941     if ( nredefs > 0 ) {
02942         /* Then unpack each variable. */
02943         for ( i = 0; i < nredefs; i++ ) {
02944             WORD index;
02945             int bytes;
02946             UBYTE *value;
02947             LONG inputnumber;
02948             PF_LongSingleUnpack(&index, 1, PF_WORD);
02949             PF_LongSingleUnpack(&bytes, 1, PF_INT);
02950             VectorReserve(prevarbuf, bytes + 1);
02951             value = VectorPtr(prevarbuf);
02952             PF_LongSingleUnpack(value, bytes, PF_BYTE);
02953             value[bytes] = '\0'; /* The null terminator is needed. */
02954             PF_LongSingleUnpack(&inputnumber, 1, PF_LONG);
02955             /* Put this variable if it must be updated. */
02956             for ( j = 0; j < AC.numpfirstnum; j++ )
02957                 if ( AC.pfirstnum[j] == index ) break;
02958             if ( AC.inputnumbers[j] < inputnumber ) {

```

```

02981         AC.inputnumbers[j] = inputnumber;
02982         PutPreVar(PreVar[index].name, value, NULL, 1);
02983     }
02984 }
02985 }
02986 }
02987
02988 /*
02989     #] PF_UnpackRedefinedPreVars :
02990     #[ PF_BroadcastRedefinedPreVars :
02991 */
02992
03003 int PF_BroadcastRedefinedPreVars(void)
03004 {
03005     /*
03006      * NOTE: Because the compilation is performed on the all processes
03007      * independently on AC.mparallelfalg, we always have to broadcast redefined
03008      * preprocessor variables from the master to the all slaves.
03009      */
03010     if ( PF.me == MASTER ) {
03011         /*
03012          *#[ Master :
03013          */
03014         int i, nredefs;
03015         PF_PrepereLongMultiPack();
03016         /* First, pack the number of redefined preprocessor variables. */
03017         nredefs = 0;
03018         for ( i = 0; i < AC.numpfirstnum; i++ )
03019             if ( AC.inputnumbers[i] >= 0 ) nredefs++;
03020         PF_LongMultiPack(&nredefs, 1, PF_INT);
03021         /* Then, pack each variable. */
03022         for ( i = 0; i < AC.numpfirstnum; i++ )
03023             if ( AC.inputnumbers[i] >= 0 ) {
03024                 WORD index = AC.pfirstnum[i];
03025                 UBYTE *value = PreVar[index].value;
03026                 int bytes = strlen((char *)value);
03027                 PF_LongMultiPack(&index, 1, PF_WORD);
03028                 PF_LongMultiPack(&bytes, 1, PF_INT);
03029                 PF_LongMultiPack(value, bytes, PF_BYTE);
03030 #ifdef PF_DEBUG_BCAST_PREVAR
03031                 MesPrint("» Broadcast PreVar: %s = \"%s\"", PreVar[index].name, value);
03032 #endif
03033             }
03034         /*
03035          *#[ Master :
03036          */
03037     }
03038     if ( PF_LongMultiBroadcast() ) return -1;
03039     if ( PF.me != MASTER ) {
03040         /*
03041          *#[ Slave :
03042          */
03043         int i, nredefs;
03044         /* Unpack the number of redefined preprocessor variables. */
03045         PF_LongMultiUnpack(&nredefs, 1, PF_INT);
03046         if ( nredefs > 0 ) {
03047             /* Then unpack each variable and put it. */
03048             for ( i = 0; i < nredefs; i++ ) {
03049                 WORD index;
03050                 int bytes;
03051                 UBYTE *value;
03052                 PF_LongMultiUnpack(&index, 1, PF_WORD);
03053                 PF_LongMultiUnpack(&bytes, 1, PF_INT);
03054                 VectorReserve(prevarbuf, bytes + 1);
03055                 value = VectorPtr(prevarbuf);
03056                 PF_LongMultiUnpack(value, bytes, PF_BYTE);
03057                 value[bytes] = '\0'; /* The null terminator is needed. */
03058                 PutPreVar(PreVar[index].name, value, NULL, 1);
03059             }
03060         }
03061         /*
03062          *#[ Slave :
03063          */
03064     }
03065     return 0;
03066 }
03067
03068 /*
03069     #] PF_BroadcastRedefinedPreVars :
03070     #] Synchronization of redefined preprocessor variables :
03071     #[ Preprocessor Inside instruction :
03072     #[ Variables :
03073 */
03074
03075 /* Saved values of AC.RhsExprInModuleFlag, PotModdollars and AC.pfirstnum. */
03076 static WORD oldRhsExprInModuleFlag;
03077 static Vector(WORD, oldPotModdollars);

```



```

03078 static Vector(WORD, oldpfirstnum);
03079
03080 /*
03081      #] Variables :
03082      #[ PF_StoreInsideInfo :
03083 */
03084
03085 /*
03086  * Saves the current values of AC.RhsExprInModuleFlag, PotModdollars
03087  * and AC.pfirstnum.
03088  *
03089  * Called by DoInside().
03090  *
03091  * @return 0 if OK, nonzero on error.
03092  */
03093 int PF_StoreInsideInfo(void)
03094 {
03095     int i;
03096     oldRhsExprInModuleFlag = AC.RhsExprInModuleFlag;
03097     VectorClear(oldPotModdollars);
03098     for ( i = 0; i < NumPotModdollars; i++ )
03099         VectorPushBack(oldPotModdollars, PotModdollars[i]);
03100     VectorClear(oldpfirstnum);
03101     for ( i = 0; i < AC.numpfirstnum; i++ )
03102         VectorPushBack(oldpfirstnum, AC.pfirstnum[i]);
03103     return 0;
03104 }
03105
03106 /*
03107      #] PF_StoreInsideInfo :
03108      #[ PF_RestoreInsideInfo :
03109 */
03110
03111 /*
03112  * Restores the saved values of AC.RhsExprInModuleFlag, PotModdollars
03113  * and AC.pfirstnum.
03114  *
03115  * Called by DoEndInside().
03116  *
03117  * @return 0 if OK, nonzero on error.
03118  */
03119 int PF_RestoreInsideInfo(void)
03120 {
03121     int i;
03122     AC.RhsExprInModuleFlag = oldRhsExprInModuleFlag;
03123     NumPotModdollars = VectorSize(oldPotModdollars);
03124     for ( i = 0; i < NumPotModdollars; i++ )
03125         PotModdollars[i] = VectorPtr(oldPotModdollars)[i];
03126     AC.numpfirstnum = VectorSize(oldpfirstnum);
03127     for ( i = 0; i < AC.numpfirstnum; i++ )
03128         AC.pfirstnum[i] = VectorPtr(oldpfirstnum)[i];
03129     return 0;
03130 }
03131
03132 /*
03133      #] PF_RestoreInsideInfo :
03134      #] Preprocessor Inside instruction :
03135      #[ PF_BroadcastCBuf :
03136 */
03137
03145 int PF_BroadcastCBuf(int bufnum)
03146 {
03147     CBUF *C = cbuf + bufnum;
03148     int i;
03149     LONG l;
03150     if ( PF.me == MASTER ) {
03151 /*
03152      #] Master :
03153 */
03154         PF_PrepareLongMultiPack();
03155         /* Pack CBUF struct except pointers. */
03156         PF_LongMultiPack(&C->BufferSize, 1, PF_LONG);
03157         PF_LongMultiPack(&C->numlhs, 1, PF_INT);
03158         PF_LongMultiPack(&C->numrhs, 1, PF_INT);
03159         PF_LongMultiPack(&C->maxlhs, 1, PF_INT);
03160         PF_LongMultiPack(&C->maxrhs, 1, PF_INT);
03161         PF_LongMultiPack(&C->mnumlhs, 1, PF_INT);
03162         PF_LongMultiPack(&C->mnumrhs, 1, PF_INT);
03163         PF_LongMultiPack(&C->numtree, 1, PF_INT);
03164         PF_LongMultiPack(&C->rootnum, 1, PF_INT);
03165         PF_LongMultiPack(&C->MaxTreeSize, 1, PF_INT);
03166         /* Now pointers. Pointer, lhs and rhs are packed as offsets. We don't pack Top. */
03167         l = C->Pointer - C->Buffer;
03168         PF_LongMultiPack(&l, 1, PF_LONG);
03169         PF_LongMultiPack(C->Buffer, 1, PF_WORD);
03170         for ( i = 0; i < C->numlhs + 1; i++ ) {
03171             l = C->lhs[i] - C->Buffer;

```

```

03172         PF_LongMultiPack(&l, 1, PF_LONG);
03173     }
03174     for ( i = 0; i < C->numrhs + 1; i++ ) {
03175         l = C->rhs[i] - C->Buffer;
03176         PF_LongMultiPack(&l, 1, PF_LONG);
03177     }
03178     PF_LongMultiPack(C->CanCommu, C->numrhs + 1, PF_LONG);
03179     PF_LongMultiPack(C->NumTerms, C->numrhs + 1, PF_LONG);
03180     PF_LongMultiPack(C->numdum, C->numrhs + 1, PF_WORD);
03181     PF_LongMultiPack(C->dimension, C->numrhs + 1, PF_WORD);
03182     if ( C->MaxTreeSize > 0 )
03183         PF_LongMultiPack(C->boomlijst, (C->numtree + 1) * (sizeof(COMPTREE) / sizeof(int)),
PF_INT);
03184 #ifdef PF_DEBUG_BCAST_CBUF
03185     MesPrint("» Broadcast CBuf %d", bufnum);
03186 #endif
03187 /*
03188     #] Master :
03189 */
03190 }
03191 if ( PF_LongMultiBroadcast() ) return -1;
03192 if ( PF.me != MASTER ) {
03193 /*
03194     #[ Slave :
03195 */
03196     /* First, free already allocated buffers. */
03197     finishcbuf(bufnum);
03198     /* Unpack CBUF struct except pointers. */
03199     PF_LongMultiUnpack(&C->BufferSize, 1, PF_LONG);
03200     PF_LongMultiUnpack(&C->numlhs, 1, PF_INT);
03201     PF_LongMultiUnpack(&C->numrhs, 1, PF_INT);
03202     PF_LongMultiUnpack(&C->maxlhs, 1, PF_INT);
03203     PF_LongMultiUnpack(&C->maxrhs, 1, PF_INT);
03204     PF_LongMultiUnpack(&C->mnumlhs, 1, PF_INT);
03205     PF_LongMultiUnpack(&C->mnumrhs, 1, PF_INT);
03206     PF_LongMultiUnpack(&C->numtree, 1, PF_INT);
03207     PF_LongMultiUnpack(&C->rootnum, 1, PF_INT);
03208     PF_LongMultiUnpack(&C->MaxTreeSize, 1, PF_INT);
03209     /* Allocate new buffers. */
03210     C->Buffer = (WORD *)Malloca(C->BufferSize * sizeof(WORD), "compiler buffer");
03211     C->Top = C->Buffer + C->BufferSize;
03212     C->lhs = (WORD **)Malloca(C->maxlhs * sizeof(WORD *), "compiler buffer");
03213     C->rhs = (WORD **)Malloca(C->maxrhs * (sizeof(WORD *) + 2 * sizeof(LONG) + 2 * sizeof(WORD)),
"compiler buffer");
03214     C->CanCommu = (LONG *) (C->rhs + C->maxrhs);
03215     C->NumTerms = C->CanCommu + C->maxrhs;
03216     C->numdum = (WORD *) (C->NumTerms + C->maxrhs);
03217     C->dimension = C->numdum + C->maxrhs;
03218     if ( C->MaxTreeSize > 0 )
03219         C->boomlijst = (COMPTREE *)Malloca(C->MaxTreeSize * sizeof(COMPTREE), "compiler buffer");
03220     /* Unpack buffers. */
03221     PF_LongMultiUnpack(&l, 1, PF_LONG);
03222     PF_LongMultiUnpack(C->Buffer, 1, PF_WORD);
03223     C->Pointer = C->Buffer + 1;
03224     for ( i = 0; i < C->numlhs + 1; i++ ) {
03225         PF_LongMultiUnpack(&l, 1, PF_LONG);
03226         C->lhs[i] = C->Buffer + l;
03227     }
03228     for ( i = 0; i < C->numrhs + 1; i++ ) {
03229         PF_LongMultiUnpack(&l, 1, PF_LONG);
03230         C->rhs[i] = C->Buffer + l;
03231     }
03232     PF_LongMultiUnpack(C->CanCommu, C->numrhs + 1, PF_LONG);
03233     PF_LongMultiUnpack(C->NumTerms, C->numrhs + 1, PF_LONG);
03234     PF_LongMultiUnpack(C->numdum, C->numrhs + 1, PF_WORD);
03235     PF_LongMultiUnpack(C->dimension, C->numrhs + 1, PF_WORD);
03236     if ( C->MaxTreeSize > 0 )
03237         PF_LongMultiUnpack(C->boomlijst, (C->numtree + 1) * (sizeof(COMPTREE) / sizeof(int)),
PF_INT);
03238 /*
03239     #] Slave :
03240 */
03241 }
03242 return 0;
03243 }
03244
03245 /*
03246     #] PF_BroadcastCBuf :
03247     #[ PF_BroadcastExpFlags :
03248 */
03249
03250 int PF_BroadcastExpFlags(void)
03251 {
03252     WORD i;
03253     EXPRESSIONS e;
03254     if ( PF.me == MASTER ) {
03255

```

```

03262     #] Master :
03263  */
03264     PF_PrepareLongMultiPack();
03265     PF_LongMultiPack(&AR.expflags, 1, PF_WORD);
03266     for ( i = 0; i < NumExpressions; i++ ) {
03267         e = &Expressions[i];
03268         PF_LongMultiPack(&e->counter, 1, PF_WORD);
03269         PF_LongMultiPack(&e->vflags, 1, PF_WORD);
03270         PF_LongMultiPack(&e->uflags, 1, PF_WORD);
03271         PF_LongMultiPack(&e->numdummies, 1, PF_WORD);
03272         PF_LongMultiPack(&e->numfactors, 1, PF_WORD);
03273 #ifdef PF_DEBUG_BCAST_EXPRFLAGS
03274         MesPrint("> Broadcast ExprFlags: %s", AC.exprnames->namebuffer + e->name);
03275 #endif
03276     }
03277  */
03278     #] Master :
03279  */
03280 }
03281 if ( PF_LongMultiBroadcast() ) return -1;
03282 if ( PF.me != MASTER ) {
03283  */
03284     #] Slave :
03285  */
03286     PF_LongMultiUnpack(&AR.expflags, 1, PF_WORD);
03287     for ( i = 0; i < NumExpressions; i++ ) {
03288         e = &Expressions[i];
03289         PF_LongMultiUnpack(&e->counter, 1, PF_WORD);
03290         PF_LongMultiUnpack(&e->vflags, 1, PF_WORD);
03291         PF_LongMultiUnpack(&e->uflags, 1, PF_WORD);
03292         PF_LongMultiUnpack(&e->numdummies, 1, PF_WORD);
03293         PF_LongMultiUnpack(&e->numfactors, 1, PF_WORD);
03294     }
03295  */
03296     #] Slave :
03297  */
03298 }
03299 return 0;
03300 }
03301
03302  */
03303 #] PF_BroadcastExpFlags :
03304 #] PF_SetScratch :
03305  */
03306
03313 static void PF_SetScratch(FILEHANDLE *f, POSITION *position)
03314 {
03315     if(
03316         ( f->handle >= 0 ) && ISGEPOS(*position, f->POposition) &&
03317         ( ISGEPOSINC(*position, f->POposition, (f->POfull-f->PObuffer)*sizeof(WORD)) ==0 )
03318     ) /*position is inside the buffer! SetScratch() will do nothing.*/
03319         f->POfull=f->PObuffer; /*force SetScratch() to re-read the position from the beginning:*/
03320     SetScratch(f, position);
03321 }
03322
03323  */
03324 #] PF_SetScratch :
03325 #] PF_pushScratch :
03326  */
03327
03334 static int PF_pushScratch(FILEHANDLE *f)
03335 {
03336     LONG size, RetCode;
03337     if ( f->handle < 0 ){
03338         /*Create the file*/
03339         if ( ( RetCode = CreateFile(f->name) ) >= 0 ) {
03340             f->handle = (WORD) RetCode;
03341             PUTZERO(f->filesize);
03342             PUTZERO(f->POposition);
03343         }
03344         else{
03345             MesPrint("Cannot create scratch file %s", f->name);
03346             return(-1);
03347         }
03348     } /*if ( f->handle < 0)*/
03349     size = (f->POfill-f->PObuffer)*sizeof(WORD);
03350     if( size > 0 ){
03351         SeekFile(f->handle, &(f->POposition), SEEK_SET);
03352         if ( WriteFile(f->handle, (UBYTE *) (f->PObuffer), size) != size ){
03353             MesPrint("Error while writing to disk. Disk full?");
03354             return(-1);
03355         }
03356         ADDPOS(f->filesize, size);
03357         ADDPOS(f->POposition, size);
03358         f->POfill = f->POfull=f->PObuffer;
03359     } /*if( size > 0)*/
03360     return(0);

```

```

03361 }
03362
03363 /*
03364 #] PF_pushScratch :
03365 #[ Broadcasting RHS expressions :
03366 #[ PF_WalkThroughExprMaster :
03367 Returns <=0 if the expression is ready, or dl+1;
03368 */
03369
03370 static int PF_WalkThroughExprMaster(FILEHANDLE *curfile, int dl)
03371 {
03372     LONG l=0;
03373     for(;;){
03374         if(curfile->POfull-curfile->POfill < dl){
03375             POSITION pos;
03376             SeekScratch(curfile,&pos);
03377             PF_SetScratch(curfile,&pos);
03378             /*if(curfile->POfull-curfile->POfill < dl)*/
03379             curfile->POfill+=dl;
03380             l+=dl;
03381             if( l >= PF.exprbufsize){
03382                 if( l == PF.exprbufsize){
03383                     if( *(curfile->POfill) == 0)/*expression is ready*/
03384                         return(0);
03385                 }
03386                 l-=PF.exprbufsize;
03387                 curfile->POfill-=l;
03388                 return l+1;
03389             }
03390
03391             dl=*(curfile->POfill);
03392             if(dl == 0)
03393                 return l-PF.exprbufsize;
03394
03395             if(dl<0){/*compressed term*/
03396                 if(curfile->POfull-curfile->POfill < 1){
03397                     POSITION pos;
03398                     SeekScratch(curfile,&pos);
03399                     PF_SetScratch(curfile,&pos);
03400                     /*if(curfile->POfull-curfile->POfill < 1)*/
03401                     dl=*(curfile->POfill+1)+2;
03402                     /*if(*(curfile->POfill)<0)*/
03403                 }/*for(;;)*/
03404             }
03405
03406 /*
03407 #] PF_WalkThroughExprMaster :
03408 #[ PF_WalkThroughExprSlave :
03409 Returns <=0 if the expression is ready, or dl+1;
03410 */
03411
03412 static int PF_WalkThroughExprSlave(FILEHANDLE *curfile, LONG *counter, int dl)
03413 {
03414     LONG l=0;
03415     for(;;){
03416         if(curfile->POstop-curfile->POfill < dl){
03417             if(PF_pushScratch(curfile))
03418                 return(-PF.exprbufsize-1);
03419         }
03420         curfile->POfill+=dl;
03421         curfile->POfull=curfile->POfill;
03422         l+=dl;
03423         if( l >= PF.exprbufsize){
03424             if( l == PF.exprbufsize){
03425                 /*
03426                  * This access is valid because PF.exprbufsize+1 WORDs are
03427                  * broadcasted, this shortcut is not mandatory though. (TU 15 Sep 2011)
03428                  */
03429                 if( *(curfile->POfill) == 0)/*expression is ready*/
03430                     return(0);
03431             }
03432             l-=PF.exprbufsize;
03433             curfile->POfill-=l;
03434             curfile->POfull=curfile->POfill;
03435             return l+1;
03436         }
03437
03438         dl=*(curfile->POfill);
03439         if(dl == 0)
03440             return l-PF.exprbufsize;
03441         (*counter)++;
03442         if(dl<0){/*compressed term*/
03443             if(curfile->POstop-curfile->POfill < 1){
03444                 if(PF_pushScratch(curfile))
03445                     return(-PF.exprbufsize-1);
03446             }
03447             /*

```

```

03448         * This access is always valid because PF.exprbufsize+1 WORDs are
03449         * broadcasted. (TU 15 Sep 2011)
03450         */
03451         dl=(curfile->POfill+1)+2;
03452     }/*if(*(curfile->POfill)<0)*/
03453 }/*for(;;)*/
03454 }
03455
03456 /*
03457     #] PF_WalkThroughExprSlave :
03458     #[ PF_rhsBCastMaster :
03459 */
03460
03461 static int PF_rhsBCastMaster(FILEHANDLE *curfile, EXPRESSIONS e)
03462 {
03463     LONG l=1; /*PF_WalkThroughExpr returns length + 1*/
03464     SetScratch(curfile, &(e->onfile));
03465     do{
03466         /*
03467          * We need to broadcast PF.exprbufsize+1 WORDs because PF_WalkThroughExprSlave
03468          * may access to an additional 1 WORD. It is better to rewrite the routines
03469          * in such a way as to broadcast only PF.exprbufsize WORDs. (TU 15 Sep 2011)
03470          */
03471         if ( curfile->POfull - curfile->POfill < PF.exprbufsize + 1 ) {
03472             POSITION pos;
03473             SeekScratch(curfile, &pos);
03474             PF_SetScratch(curfile, &pos);
03475         }
03476         if ( PF_Bcast(curfile->POfill, (PF.exprbufsize + 1) * sizeof(WORD)) )
03477             return -1;
03478         l=PF_WalkThroughExprMaster(curfile, l-1);
03479     }while(l>0);
03480     if(l<0)/*The tail is extra, decrease POfill*/
03481         curfile->POfill-=l;
03482     return(0);
03483 }
03484
03485 /*
03486     #] PF_rhsBCastMaster :
03487     #[ PF_rhsBCastSlave :
03488 */
03489
03490 static int PF_rhsBCastSlave(FILEHANDLE *curfile, EXPRESSIONS e)
03491 {
03492     LONG l=1; /*PF_WalkThroughExpr returns length + 1*/
03493     LONG counter = 0;
03494     do{
03495         /*
03496          * We need to broadcast PF.exprbufsize+1 WORDs because PF_WalkThroughExprSlave
03497          * may access to an additional 1 WORD. It is better to rewrite the routines
03498          * in such a way as to broadcast only PF.exprbufsize WORDs. (TU 15 Sep 2011)
03499          */
03500         if ( curfile->POstop - curfile->POfill < PF.exprbufsize + 1 ) {
03501             if(PF_pushScratch(curfile))
03502                 return(-1);
03503         }
03504         if ( PF_Bcast(curfile->POfill, (PF.exprbufsize + 1) * sizeof(WORD)) )
03505             return(-1);
03506         l = PF_WalkThroughExprSlave(curfile, &counter, l - 1);
03507     }while(l>0);
03508     if(l<0)/*The tail is extra, decrease POfill*/
03509         if(l<-PF.exprbufsize)/*error due to a PF_pushScratch() failure */
03510             return(-1);
03511     curfile->POfill-=l;
03512 }
03513
03514 if ( curfile->handle >= 0 ) {
03515     if ( PF_pushScratch(curfile) ) return -1;
03516 }
03517 curfile->POfull=curfile->POfill;
03518 if ( curfile != AR.hidefile ) AR.InInBuf = curfile->POfull-curfile->PObuffer;
03519 else AR.InHiBuf = curfile->POfull-curfile->PObuffer;
03520 CHECK(counter == e->counter + 1); /* The first term is the prototype. */
03521 return(0);
03522 }
03523
03524 /*
03525     #] PF_rhsBCastSlave :
03526     #[ PF_BroadcastExpr :
03527 */
03528
03529 int PF_BroadcastExpr(EXPRESSIONS e, FILEHANDLE *file)
03530 {
03531     if ( PF.me == MASTER ) {
03532         if ( PF_rhsBCastMaster(file, e) ) return -1;
03533     }
03534     #ifdef PF_DEBUG_BCAST_RHSEXPR
03535     MesPrint("> Broadcast RhsExpr: %s", AC.exprnames->namebuffer + e->name);
03536     #endif
03537 }

```

```

03557     }
03558     else {
03559         POSITION pos;
03560         SetEndHScratch(file, &pos);
03561         e->onfile = pos;
03562         if ( PF_rhsBCastSlave(file, e) ) return -1;
03563     }
03564     return 0;
03565 }
03566
03567 /*
03568     #] PF_BroadcastExpr :
03569     #[ PF_BroadcastRHS :
03570 */
03571
03572 int PF_BroadcastRHS(void)
03573 {
03574     int i;
03575     for ( i = 0; i < NumExpressions; i++ ) {
03576         EXPRESSIONS e = &Expressions[i];
03577         if ( !(e->vflags & ISINRHS) ) continue;
03578         switch ( e->status ) {
03579             case LOCALEXPRESSION:
03580             case SKIPLEXPRESSION:
03581             case DROPLEXPRESSION:
03582             case GLOBALEXPRESSION:
03583             case SKIPGEXPRESSION:
03584             case DROPGEXPRESSION:
03585             case HIDELEXPRESSION:
03586             case HIDEGEXPRESSION:
03587             case INTOHIDELEXPRESSION:
03588             case INTOHIDEGEXPRESSION:
03589                 if ( PF_BroadcastExpr(e, AR.infile) ) return -1;
03590                 break;
03591             case HIDDENLEXPRESSION:
03592             case HIDDENGEXPRESSION:
03593             case DROPHLEXPRESSION:
03594             case DROPHGEXPRESSION:
03595             case UNHIDELEXPRESSION:
03596             case UNHIDEGEXPRESSION:
03597                 if ( PF_BroadcastExpr(e, AR.hidefile) ) return -1;
03598                 break;
03599         }
03600     }
03601     if ( PF.me != MASTER )
03602         UpdatePositions();
03603     return 0;
03604 }
03605
03606 /*
03607     #] PF_BroadcastRHS :
03608     #] Broadcasting RHS expressions :
03609     #[ InParallel mode :
03610     #[ PF_InParallelProcessor :
03611 */
03612
03613 int PF_InParallelProcessor(void)
03614 {
03615     GETIDENTITY
03616     int i, next, tag;
03617     EXPRESSIONS e;
03618     /*
03619     * Skip expressions with zero terms. All the master and slaves need to
03620     * change the "partodo" flag.
03621     */
03622     if ( PF.numtasks >= 3 ) {
03623         for ( i = 0; i < NumExpressions; i++ ) {
03624             e = Expressions + i;
03625             if ( e->partodo > 0 && e->counter == 0 ) {
03626                 e->partodo = 0;
03627             }
03628         }
03629     }
03630     if (PF.me == MASTER) {
03631         if ( PF.numtasks >= 3 ) {
03632             partodoexr = (WORD*)Malloca(sizeof(WORD)*(PF.numtasks+1), "PF_InParallelProcessor");
03633             for ( i = 0; i < NumExpressions; i++ ) {
03634                 e = Expressions+i;
03635                 if ( e->partodo <= 0 ) continue;
03636                 switch(e->status) {
03637                     case LOCALEXPRESSION:
03638                     case GLOBALEXPRESSION:
03639                     case UNHIDELEXPRESSION:
03640                     case UNHIDEGEXPRESSION:
03641                     case INTOHIDELEXPRESSION:
03642                     case INTOHIDEGEXPRESSION:
03643                         tag=PF_ANY_SOURCE;

```

```

03656         next=PF_Wait4SlaveIP(&tag);
03657         if(next<0)
03658             return(-1);
03659         if(tag == PF_DATA_MSGTAG){
03660             PF_Statistics(PF_stats,0);
03661             if(PF_Slave2MasterIP(next))
03662                 return(-1);
03663         }
03664         if(PF_Master2SlaveIP(next,e))
03665             return(-1);
03666         partodoexr[next]=i;
03667         break;
03668     default:
03669         e->partodo = 0;
03670         continue;
03671     }/*switch(e->status)*/
03672     }/*for ( i = 0; i < NumExpressions; i++ )*/
03673     /*Here some slaves are working, other are waiting on PF_Send.
03674     Wait all of them.*/
03675     /*At this point no new slaves may be launched so PF_WaitAllSlaves()
03676     does not modify partodoexr[] */
03677     if(PF_WaitAllSlaves())
03678         return(-1);
03679
03680     if ( AC.CollectFun ) AR.DeferFlag = 0;
03681     if(partodoexr){
03682         M_free(partodoexr,"PF_InParallelProcessor");
03683         partodoexr=NULL;
03684     }/*if(partodoexr)*/
03685     }/*if ( PF.numtasks >= 3 ) */
03686     else {
03687         for ( i = 0; i < NumExpressions; i++ ) {
03688             Expressions[i].partodo = 0;
03689         }
03690     }
03691     return(0);
03692 }/*if(PF.me == MASTER)*/
03693 /*Slave:*/
03694 if(PF_Wait4MasterIP(PF_EMPTY_MSGTAG))
03695     return(-1);
03696 /*master is ready to listen to me*/
03697 do{
03698     WORD *oldwork= AT.WorkPointer;
03699     tag=PF_ReadMaster();/*reads directly to its scratch!*/
03700     if(tag<0)
03701         return(-1);
03702     if(tag == PF_DATA_MSGTAG){
03703         oldwork = AT.WorkPointer;
03704
03705         /* For redefine statements. */
03706         if ( AC.numpfirstnum > 0 ) {
03707             int j;
03708             for ( j = 0; j < AC.numpfirstnum; j++ ) {
03709                 AC.inputnumbers[j] = -1;
03710             }
03711         }
03712
03713         if(PF_DoOneExpr())/*the processor*/
03714             return(-1);
03715         if(PF_Wait4MasterIP(PF_DATA_MSGTAG))
03716             return(-1);
03717         if(PF_Slave2MasterIP(PF.me))/*both master and slave*/
03718             return(-1);
03719         AT.WorkPointer=oldwork;
03720     }/*if(tag == PF_DATA_MSGTAG)*/
03721 }while(tag!=PF_EMPTY_MSGTAG);
03722 PF.exprtoto=-1;
03723 return(0);
03724 }/*PF_InParallelProcessor*/
03725
03726 /*
03727     #] PF_InParallelProcessor :
03728     #[ PF_Wait4MasterIP :
03729 */
03730
03731 static int PF_Wait4MasterIP(int tag)
03732 {
03733     int follow = 0;
03734     LONG cpu,space = 0;
03735
03736     if(PF.log){
03737         fprintf(stderr,"%[d] Starting to send to Master\n",PF.me);
03738         fflush(stderr);
03739     }
03740
03741     PF_PreparePack();
03742     cpu = TimeCPU(1);

```

```

03743     PF_Pack(&cpu                      ,1,PF_LONG);
03744     PF_Pack(&space                    ,1,PF_LONG);
03745     PF_Pack(&PF_linterms              ,1,PF_LONG);
03746     PF_Pack(&(AM.S0->GenTerms)        ,1,PF_LONG);
03747     PF_Pack(&(AM.S0->TermsLeft)       ,1,PF_LONG);
03748     PF_Pack(&follow                   ,1,PF_INT );
03749
03750     if(PF.log){
03751         fprintf(stderr,"%d] Now sending with tag = %d\n",PF.me,tag);
03752         fflush(stderr);
03753     }
03754
03755     PF_Send(MASTER, tag);
03756
03757     if(PF.log){
03758         fprintf(stderr,"%d] returning from send\n",PF.me);
03759         fflush(stderr);
03760     }
03761     return(0);
03762 }
03763 /*
03764     #] PF_Wait4MasterIP :
03765     #[ PF_DoOneExpr :
03766 */
03767
03775 static int PF_DoOneExpr(void)/*the processor*/
03776 {
03777     GETIDENTITY
03778     EXPRESSIONS e;
03779     int i;
03780     WORD *term;
03781     POSITION position, outposition;
03782     FILEHANDLE *fi, *fout;
03783     LONG dd = 0;
03784     WORD oldBracketOn = AR.BracketOn;
03785     WORD *oldBrackBuf = AT.BrackBuf;
03786     WORD oldbracketindexflag = AT.bracketindexflag;
03787     e = Expressions + PF.exprtoto;
03788     i = PF.exprtoto;
03789     AR.CurExpr = i;
03790     AR.SortType = AC.SortType;
03791     AR.expchanged = 0;
03792     if ( ( e->vflags & ISFACTORIZED ) != 0 ) {
03793         AR.BracketOn = 1;
03794         AT.BrackBuf = AM.BracketFactors;
03795         AT.bracketindexflag = 1;
03796     }
03797
03798     position = AS.OldOnFile[i];
03799     if ( e->status == HIDDENLEXPRESSION || e->status == HIDDENEXPRESSION ) {
03800         AR.GetFile = 2; fi = AR.hidefile;
03801     }
03802     else {
03803         AR.GetFile = 0; fi = AR.infile;
03804     }
03805 /*
03806     PUTZERO(fi->POposition);
03807     if ( fi->handle >= 0 ) {
03808         fi->POfill = fi->POfull = fi->PObuffer;
03809     }
03810 */
03811     SetScratch(fi,&position);
03812     term = AT.WorkPointer;
03813     AR.CompressPointer = AR.CompressBuffer;
03814     AR.CompressPointer[0] = 0;
03815     AR.KeptInHold = 0;
03816     if ( GetTerm(BHEAD term) <= 0 ) {
03817         MesPrint("Expression %d has problems in scratchfile",i);
03818         Terminate(-1);
03819     }
03820     if ( AT.bracketindexflag > 0 ) OpenBracketIndex(i);
03821     term[3] = i;
03822     PUTZERO(outposition);
03823     fout = AR.outfile;
03824     fout->POfill = fout->POfull = fout->PObuffer;
03825     fout->POposition = outposition;
03826     if ( fout->handle >= 0 ) {
03827         fout->POposition = outposition;
03828     }
03829 /*
03830     The next statement is needed because we need the system
03831     to believe that the expression is at position zero for
03832     the moment. In this worker, with no memory of other expressions,
03833     it is. This is needed for when a bracket index is made
03834     because there e->onfile is an offset. Afterwards, when the
03835     expression is written to its final location in the masters
03836     output e->onfile will get its real value.

```



```

03837 */
03838 PUTZERO(e->onfile);
03839 if ( PutOut(BHEAD term,&outposition,fout,0) < 0 ) return -1;
03840
03841 AR.DeferFlag = AC.ComDefer;
03842
03843 /* AR.sLevel = AB[0]->R.sLevel;*/
03844 term = AT.WorkPointer;
03845 NewSort(BHEAD0);
03846 AR.MaxDum = AM.IndDum;
03847 AN.ninterms = 0;
03848 while ( GetTerm(BHEAD term) ) {
03849     SeekScratch(fi,&position);
03850     AN.ninterms++; dd = AN.deferskipped;
03851     if ( ( e->vflags & ISFACTORIZED ) != 0 && term[1] == HAAKJE ) {
03852         StoreTerm(BHEAD term);
03853     }
03854     else {
03855         if ( AC.CollectFun && *term <= (AM.MaxTer/(2*(LONG)sizeof(WORD))) ) {
03856             if ( GetMoreTerms(term) < 0 ) {
03857                 LowerSortLevel(); return(-1);
03858             }
03859             SeekScratch(fi,&position);
03860         }
03861         AT.WorkPointer = term + *term;
03862         AN.RepPoint = AT.RepCount + 1;
03863         if ( AR.DeferFlag ) {
03864             AR.CurDum = AN.IndDum = Expressions[PF.exprtodo].numdummies;
03865         }
03866         else {
03867             AN.IndDum = AM.IndDum;
03868             AR.CurDum = ReNumber(BHEAD term);
03869         }
03870         if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
03871         if ( AN.ncmod ) {
03872             if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
03873             else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
03874         }
03875         else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
03876         if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
03877             && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
03878             PolyFunClean(BHEAD term);
03879         }
03880         if ( Generator(BHEAD term,0) ) {
03881             LowerSortLevel(); return(-1);
03882         }
03883         AN.ninterms += dd;
03884     }
03885     SetScratch(fi,&position);
03886     if ( fi == AR.hidefile ) {
03887         AR.InHiBuf = (fi->POfull-fi->PObuffer)
03888             -DIFBASE(position,fi->POposition)/sizeof(WORD);
03889     }
03890     else {
03891         AR.InInBuf = (fi->POfull-fi->PObuffer)
03892             -DIFBASE(position,fi->POposition)/sizeof(WORD);
03893     }
03894 }
03895 AN.ninterms += dd;
03896 if ( EndSort(BHEAD AM.S0->sBuffer,0) < 0 ) return(-1);
03897 e->numdummies = AR.MaxDum - AM.IndDum;
03898 AR.BracketOn = oldBracketOn;
03899 AT.BrackBuf = oldBrackBuf;
03900 if ( ( e->vflags & TOBEFACTORED ) != 0 )
03901     poly_factorize_expression(e);
03902 else if ( ( ( e->vflags & TOBEUNFACTORED ) != 0 )
03903     && ( ( e->vflags & ISFACTORIZED ) != 0 ) )
03904     poly_unfactorize_expression(e);
03905 if ( AM.S0->TermsLeft ) e->vflags &= ~ISZERO;
03906 else e->vflags |= ISZERO;
03907 if ( AR.expchanged == 0 ) e->vflags |= ISUNMODIFIED;
03908 /* if ( AM.S0->TermsLeft ) AR.expflags |= ISZERO;
03909 if ( AR.expchanged ) AR.expflags |= ISUNMODIFIED;*/
03910 AR.GetFile = 0;
03911 AT.bracketindexflag = oldbracketindexflag;
03912
03913 fout->POfull = fout->POfill;
03914 return(0);
03915 }
03916
03917 /*
03918 #] PF_DoOneExpr :
03919 #[ PF_Slave2MasterIP :
03920 */
03921
03922 typedef struct bufIPstruct {
03923     LONG i;

```

```

03924     struct ExprEsSiOn e;
03925 } bufIPstruct_t;
03926
03927 static int PF_Slave2MasterIP(int src) /*both master and slave*/
03928 {
03929     EXPRESSIONS e;
03930     bufIPstruct_t exprData;
03931     int i,l;
03932     FILEHANDLE *fout=AR.outfile;
03933     POSITION pos;
03934     /*Here we know the length of data to send in advance:
03935     slave has the only one expression in its scratch file, and it sends
03936     this information to the master.*/
03937     if (PF.me != MASTER) { /*slave*/
03938         e = Expressions + PF.exprtodo;
03939         /*Fill in the expression data:*/
03940         memcpy(&(exprData.e), e, sizeof(struct ExprEsSiOn));
03941         SeekScratch(fout,&pos);
03942         exprData.i=BASEPOSITION(pos);
03943         /*Send the metadata:*/
03944         if (PF_RawSend(MASTER,&exprData,sizeof(bufIPstruct_t),0))
03945             return(-1);
03946         i=exprData.i;
03947         SETBASEPOSITION(pos,0);
03948         do{
03949             int blen=PF.exprbufsize*sizeof(WORD);
03950             if(i<blen)
03951                 blen=i;
03952             l=PF_SendChunkIP(fout,&pos, MASTER, blen);
03953             /*Here always l == blen!*/
03954             if(l<0)
03955                 return(-1);
03956             ADDPOS(pos,l);
03957             i-=l;
03958         }while(i>0);
03959         if ( fout->handle >= 0 ) { /* Now get rid of the file */
03960             CloseFile(fout->handle);
03961             fout->handle = -1;
03962             remove(fout->name);
03963             PUTZERO(fout->POposition);
03964             PUTZERO(fout->filesize);
03965             fout->POfill = fout->PObuffer;
03966         }
03967         /* Now handle redefined preprocessor variables. */
03968         if ( AC.numpfirstnum > 0 ) {
03969             PF_PrepareLongSinglePack();
03970             PF_PackRedefinedPreVars();
03971             PF_LongSingleSend(MASTER, PF_MISC_MSGTAG);
03972         }
03973         return(0);
03974     } /*if (PF.me != MASTER)*/
03975     /*Master*/
03976     /*partodoexr[src] is the number of expression.*/
03977     e = Expressions +partodoexr[src];
03978     /*Get metadata:*/
03979     if (PF_RawRecv(&src, &exprData,sizeof(bufIPstruct_t),&i) != sizeof(bufIPstruct_t))
03980         return(-1);
03981     /*Fill in the expression data:*/
03982     /* memcpy(e, &(exprData.e), sizeof(struct ExprEsSiOn)); */
03983     e->counter = exprData.e.counter;
03984     e->vflags = exprData.e.vflags;
03985     e->uflags = exprData.e.uflags;
03986     e->numdummies = exprData.e.numdummies;
03987     e->numfactors = exprData.e.numfactors;
03988     if ( !(e->vflags & ISZERO) ) AR.expflags |= ISZERO;
03989     if ( !(e->vflags & ISUNMODIFIED) ) AR.expflags |= ISUNMODIFIED;
03990     SeekScratch(fout,&pos);
03991     e->onfile = pos;
03992     i=exprData.i;
03993     while(i>0){
03994         int blen=PF.exprbufsize*sizeof(WORD);
03995         if(i<blen)
03996             blen=i;
03997         l=PF_RecvChunkIP(fout,src,blen);
03998         /*Here always l == blen!*/
03999         if(l<0)
04000             return(-1);
04001         i-=l;
04002     }
04003     /* Now handle redefined preprocessor variables. */
04004     if ( AC.numpfirstnum > 0 ) {
04005         PF_LongSingleReceive(src, PF_MISC_MSGTAG, NULL, NULL);
04006         PF_UnpackRedefinedPreVars();
04007     }
04008     return(0);
04009 }
04010

```

```

04011 /*
04012      #] PF_Slave2MasterIP :
04013      #[ PF_Master2SlaveIP :
04014 */
04015
04016 static int PF_Master2SlaveIP(int dest, EXPRESSIONS e)
04017 {
04018     bufIPstruct_t exprData;
04019     FILEHANDLE *fi;
04020     POSITION pos;
04021     int l;
04022     LONG ll=0, count=0;
04023     WORD *t;
04024     if (e==NULL) { /*Say to the slave that no more job:*/
04025         if (PF_RawSend(dest, &exprData, sizeof(bufIPstruct_t), PF_EMPTY_MSGTAG))
04026             return(-1);
04027         return(0);
04028     }
04029     memcpy(&(exprData.e), e, sizeof(struct ExprEsSiOn));
04030     exprData.i=e-Expressions;
04031     if ( AC.StatsFlag && AC.OldParallelStats ) {
04032         MesPrint("");
04033         MesPrint(" Sending expression %s to slave %d", EXPRNAME(exprData.i), dest);
04034     }
04035     if (PF_RawSend(dest, &exprData, sizeof(bufIPstruct_t), PF_DATA_MSGTAG))
04036         return(-1);
04037     if ( e->status == HIDDENLEXPRESSION || e->status == HIDDENLEXPRESSION )
04038         fi = AR.hidefile;
04039     else
04040         fi = AR.infile;
04041     pos=e->onfile;
04042     SetScratch(fi, &pos);
04043     do{
04044         l=PF_SendChunkIP(fi, &pos, dest, PF.exprbufsize*sizeof(WORD));
04045         if (l<0)
04046             return(-1);
04047         t=fi->PObuffer+ (DIFBASE(pos, fi->POposition))/sizeof(WORD);
04048         ll=PF_WalkThrough(t, ll, l/sizeof(WORD), &count);
04049         ADDPOS(pos, l);
04050     }while (ll>-2);
04051     return(0);
04052 }
04053
04054 /*
04055      #] PF_Master2SlaveIP :
04056      #[ PF_ReadMaster :
04057 */
04058
04059 static int PF_ReadMaster(void) /*reads directly to its scratch!*/
04060 {
04061     bufIPstruct_t exprData;
04062     int tag, m=MASTER;
04063     EXPRESSIONS e;
04064     FILEHANDLE *fi;
04065     POSITION pos;
04066     LONG count=0;
04067     WORD *t;
04068     LONG ll=0;
04069     int l;
04070     /*Get metadata:*/
04071     if (PF_RawRecv(&m, &exprData, sizeof(bufIPstruct_t), &tag) != sizeof(bufIPstruct_t))
04072         return(-1);
04073
04074     if (tag == PF_EMPTY_MSGTAG) /*No data, no job*/
04075         return(tag);
04076
04077     /*data expected, tag must be == PF_DATA_MSTAG!*/
04078     PF.exprtodo=exprData.i;
04079     e=Expressions + PF.exprtodo;
04080     /*Fill in the expression data:*/
04081     memcpy(e, &(exprData.e), sizeof(struct ExprEsSiOn)); /*
04082     if ( e->status == HIDDENLEXPRESSION || e->status == HIDDENLEXPRESSION )
04083         fi = AR.hidefile;
04084     else
04085         fi = AR.infile;
04086     SetEndHScratch(fi, &pos);
04087     e->onfile=AS.OldOnFile[PF.exprtodo]=pos;
04088
04089     do{
04090         l=PF_RecvChunkIP(fi, MASTER, PF.exprbufsize*sizeof(WORD));
04091         if (l<0)
04092             return(-1);
04093         t=fi->POfull-l/sizeof(WORD);
04094         ll=PF_WalkThrough(t, ll, l/sizeof(WORD), &count);
04095     }while (ll>-2);
04096     /*Now -ll-2 is the number of "extra" elements transferred from the master.*/
04097     fi->POfull=-ll-2;

```

```

04098     fi->POfill=fi->POfull;
04099     return(PF_DATA_MSGTAG);
04100 }
04101
04102 /*
04103     #] PF_ReadMaster :
04104     #[ PF_SendChunkIP :
04105     thesize is in bytes. Returns the number of sent bytes or <0 on error:
04106 */
04107
04108 static int PF_SendChunkIP(FILEHANDLE *curfile, POSITION *position, int to, LONG thesize)
04109 {
04110     LONG l=thesize;
04111     if(
04112         ISLESSPOS(*position,curfile->POposition) ||
04113         ISGEPOSINC(*position,curfile->POposition,
04114             ((curfile->POfull-curfile->PObuffer)*sizeof(WORD)-thesize) )
04115     ){
04116         if(curfile->handle< 0)
04117             l=(curfile->POfull-curfile->PObuffer)*sizeof(WORD) - (LONG)(position->p1);
04118         else{
04119             PF_SetScratch(curfile,position);
04120             if(
04121                 ISGEPOSINC(*position,curfile->POposition,
04122                     ((curfile->POfull-curfile->PObuffer)*sizeof(WORD)-thesize) )
04123             )
04124                 l=(curfile->POfull-curfile->PObuffer)*sizeof(WORD) - (LONG)position->p1;
04125         }
04126     }
04127     /*Now we are able to sent l bytes from the
04128     curfile->PObuffer[position-curfile->POposition]*/
04129     if(PF_RawSend(to,curfile->PObuffer+ (DIFBASE(*position,curfile->POposition))/sizeof(WORD),l,0))
04130         return(-1);
04131     return(l);
04132 }
04133
04134 /*
04135     #] PF_SendChunkIP :
04136     #[ PF_RecvChunkIP :
04137     thesize is in bytes. Returns the number of sent bytes or <0 on error:
04138 */
04139
04140 static int PF_RecvChunkIP(FILEHANDLE *curfile, int from, LONG thesize)
04141 {
04142     LONG receivedBytes;
04143
04144     if( (LONG)((curfile->POstop - curfile->POfull)*sizeof(WORD)) < thesize )
04145         if(PF_pushScratch(curfile))
04146             return(-1);
04147     /*Now there is enough space from curfile->POfill to curfile->POstop*/
04148     /*Block:*/
04149     int tag=0;
04150     receivedBytes=PF_RawRecv(&from,curfile->POfull,thesize,&tag);
04151     /*Block*/
04152     if(receivedBytes >= 0 ){
04153         curfile->POfull+=receivedBytes/sizeof(WORD);
04154         curfile->POfill=curfile->POfull;
04155     }/*if(receivedBytes >= 0 )*/
04156     return(receivedBytes);
04157 }
04158
04159 /*
04160     #] PF_RecvChunkIP :
04161     #[ PF_WalkThrough :
04162     Returns:
04163     >= 0 -- initial offset,
04164     -1 -- the first element of t contains the length of the tail of compressed term,
04165     <= -2 -- -(d+2), where d is the number of extra transferred elements.
04166     Expects:
04167     l -- initial offset or -1,
04168     chunk -- number of transferred elements (not bytes!)
04169     *count -- incremented each time a new term is found
04170 */
04171
04172 static int PF_WalkThrough(WORD *t, LONG l, LONG chunk, LONG *count)
04173 {
04174     if(l<0) /*== -1!*/
04175         l=(*t)+1; /*the first element of t contains the length of
04176                 the tail of compressed term*/
04177     else{
04178         if(l>=chunk) /*next term is out of the chunk*/
04179             return(l-chunk);
04180         t+=l;
04181         chunk-=l; /*note, l was less than chunk so chunk >0!*/
04182         l=*t;
04183     }
04184     /*Main loop:*/

```

```

04185     while(l!=0){
04186         if(l>0){/*an offset to the next term*/
04187             if(l<chunk){
04188                 t+=l;
04189                 chunk-=l;/*note, l was less than chunk so chunk >0!*/
04190                 l=t;
04191                 (*count)++;
04192             }/*if(l<chunk)*/
04193             else
04194                 return(l-chunk);
04195         }/*if(l>0)*/
04196         else{ /* l<0 */
04197             if(chunk < 2)/*i.e., chunk == 1*/
04198                 return(-1);/*the first WORD in the next chunk is length of the tail of the compressed
term*/
04199             l=(t+1)+2;/**2 since
04200                 1. t points to the length field -1,
04201                 2. the size of a tail of compressed term is equal to the number of WORDs in this
tail*/
04202         }
04203     }/*while(l!=0)*/
04204     return(-l-chunk);/* -(2+(chunk-1)), chunk>0 ! */
04205 }
04206
04207 /*
04208     #] PF_WalkThrough :
04209     #] InParallel mode :
04210     #[ PF_SendFile :
04211 */
04212
04213 #define PF_SNDFILEBUFSIZE 4096
04214
04222 int PF_SendFile(int to, FILE *fd)
04223 {
04224     size_t len=0;
04225     if(fd == NULL){
04226         if(PF_RawSend(to,&to,sizeof(int),PF_EMPTY_MSGTAG)
04227             return(-1);
04228         return(0);
04229     }
04230     for(;;){
04231         char buf[PF_SNDFILEBUFSIZE];
04232         size_t l;
04233         l=fread(buf, 1, PF_SNDFILEBUFSIZE, fd);
04234         len+=l;
04235         if(l==PF_SNDFILEBUFSIZE){
04236             if(PF_RawSend(to,buf,PF_SNDFILEBUFSIZE,PF_BUFFER_MSGTAG)
04237                 return(-1);
04238         }
04239         else{
04240             if(PF_RawSend(to,buf,l,PF_ENDBUFFER_MSGTAG)
04241                 return(-1);
04242             break;
04243         }
04244     }/*for(;;)*/
04245     return(len);
04246 }
04247
04248 /*
04249     #] PF_SendFile :
04250     #[ PF_RecvFile :
04251 */
04252
04260 int PF_RecvFile(int from, FILE *fd)
04261 {
04262     size_t len=0;
04263     int tag;
04264     do{
04265         char buf[PF_SNDFILEBUFSIZE];
04266         int l;
04267         l=PF_RawRecv(&from,buf,PF_SNDFILEBUFSIZE,&tag);
04268         if(l<0)
04269             return(-1);
04270         if(tag == PF_EMPTY_MSGTAG)
04271             return(-1);
04272
04273         if( fwrite(buf,l,1,fd)!=1 )
04274             return(-1);
04275         len+=l;
04276     }while(tag!=PF_ENDBUFFER_MSGTAG);
04277     return(len);
04278 }
04279
04280 /*
04281     #] PF_RecvFile :
04282     #[ Synchronised output :
04283     #[ Explanations :

```

```

04284 */
04285
04286 /*
04287  * If the master and slaves output statistics or error messages to the same stream
04288  * or file (e.g., the standard output or the log file) simultaneously, then
04289  * a mixing of their outputs can occur. To avoid this, TFORM uses a lock of
04290  * ErrorMessageLock, but there is no locking functionality in the original MPI
04291  * specification. We need to synchronise the output from the master and slaves.
04292  *
04293  * The idea of the synchronised output (by, e.g., MesPrint()) implemented here is
04294  * Slaves:
04295  *     1. Save the output by WriteFile() (set to PF_WriteFileToFile())
04296  *        into some buffers between MLOCK(ErrorMessageLock) and
04297  *        MUNLOCK(ErrorMessageLock), which call PF_MLock() and PF_MUnlock(),
04298  *        respectively. The output for AM.Stdout and AC.LogHandle are saved to
04299  *        the buffers.
04300  *     2. At MUNLOCK(ErrorMessageLock), send the output in the buffer to the master,
04301  *        with PF_STDOUT_MSGTAG or PF_LOG_MSGTAG.
04302  * Master:
04303  *     1. Receive the buffered output from slaves, and write them by
04304  *        WriteFileToFile().
04305  * The main problem is how and where the master receives messages from
04306  * the slaves (PF_ReceiveErrorMessage()). For this purpose there are three
04307  * helper functions: PF_CatchErrorMessages() and PF_CatchErrorMessagesForAll()
04308  * which remove messages with PF_STDOUT_MSGTAG or PF_LOG_MSGTAG from the top
04309  * of the message queue, and PF_ProbeWithCatchingErrorMessages() which is same as
04310  * PF_Probe() except removing these messages.
04311 */
04312
04313 /*
04314     #] Explanations :
04315     #[ Variables :
04316 */
04317
04318 static int errorMessageLock = 0; /* (slaves) The lock count. See PF_MLock() and PF_MUnlock(). */
04319 static Vector(UBYTE, stdoutBuffer); /* (slaves) The buffer for AM.Stdout. */
04320 static Vector(UBYTE, logBuffer); /* (slaves) The buffer for AC.LogHandle. */
04321 #define recvBuffer logBuffer /* (master) The buffer for receiving messages. */
04322
04323 /*
04324  * If PF_ENABLE_STDOUT_BUFFERING is defined, the master performs the line buffering
04325  * (using stdoutBuffer) at PF_WriteFileToFile().
04326  */
04327 #ifndef PF_ENABLE_STDOUT_BUFFERING
04328 #ifdef UNIX
04329 #define PF_ENABLE_STDOUT_BUFFERING
04330 #endif
04331 #endif
04332
04333 /*
04334     #] Variables :
04335     #[ PF_MLock :
04336 */
04337
04341 void PF_MLock(void)
04342 {
04343     /* Only on slaves. */
04344     if ( errorMessageLock++ > 0 ) return;
04345     VectorClear(stdoutBuffer);
04346     VectorClear(logBuffer);
04347 }
04348
04349 /*
04350     #] PF_MLock :
04351     #[ PF_MUnlock :
04352 */
04353
04357 void PF_MUnlock(void)
04358 {
04359     /* Only on slaves. */
04360     if ( --errorMessageLock > 0 ) return;
04361     if ( !VectorEmpty(stdoutBuffer) ) {
04362         PF_RawSend(MASTER, VectorPtr(stdoutBuffer), VectorSize(stdoutBuffer), PF_STDOUT_MSGTAG);
04363     }
04364     if ( !VectorEmpty(logBuffer) ) {
04365         PF_RawSend(MASTER, VectorPtr(logBuffer), VectorSize(logBuffer), PF_LOG_MSGTAG);
04366     }
04367 }
04368
04369 /*
04370     #] PF_MUnlock :
04371     #[ PF_WriteFileToFile :
04372 */
04373
04386 LONG PF_WriteFileToFile(int handle, UBYTE *buffer, LONG size)
04387 {
04388     if ( PF.me != MASTER && errorMessageLock > 0 ) {

```

```

04389         if ( handle == AM.Stdout ) {
04390             VectorPushBacks(stdoutBuffer, buffer, size);
04391             return size;
04392         }
04393         else if ( handle == AC.LogHandle ) {
04394             VectorPushBacks(logBuffer, buffer, size);
04395             return size;
04396         }
04397     }
04398 #ifdef PF_ENABLE_STDOUT_BUFFERING
04399     /*
04400      * On my computer, sometimes a single linefeed "\n" sent to the standard
04401      * output is ignored on the execution of mpiexec. A typical example is:
04402      * $ cat foo.c
04403      * #include <unistd.h>
04404      * int main() {
04405      *     write(1, " ", 4);
04406      *     write(1, "\n", 1);
04407      *     write(1, " ", 4);
04408      *     write(1, "123\n", 4);
04409      *     return 0;
04410      * }
04411      * or even as a shell script:
04412      * $ cat foo.sh
04413      * #! bin/sh
04414      * printf " "
04415      * printf "\n"
04416      * printf " "
04417      * printf "123\n"
04418      * When I ran it on mpiexec
04419      * $ while :; do mpiexec -np 1 ./foo.sh; done
04420      * I observed the single linefeed (printf "\n") was sometimes ignored. Even
04421      * though this phenomenon might be specific to my environment, I added this
04422      * code because someone may encounter a similar phenomenon and feel it
04423      * frustrating. (TU 16 Jun 2011)
04424      *
04425      * Phenomenon:
04426      *   A single linefeed sent to the standard output occasionally ignored
04427      *   on mpiexec.
04428      *
04429      * Environment:
04430      *   openSUSE 11.4 (x86_64)
04431      *   kernel: 2.6.37.6-0.5-desktop
04432      *   gcc: 4.5.1 20101208
04433      *   mpich2-1.3.2p1 configured with '--enable-shared --with-pm=smpd'
04434      *
04435      * Solution:
04436      *   In Unix (in which Uwrite() calls write() system call without any buffering),
04437      *   we perform the line buffering here. A single linefeed is also buffered.
04438      *
04439      * XXX:
04440      *   At the end of the program the buffered output (text without LF) will not be flushed,
04441      *   i.e., will not be written to the standard output. This is not problematic at a normal run.
04442      *   The buffer can be explicitly flushed by PF_FlushStdOutBuffer().
04443      */
04444     if ( PF.me == MASTER && handle == AM.Stdout ) {
04445         size_t oldsize;
04446         /* Assume the newline character is LF (when UNIX is defined). */
04447         if ( (size > 0 && buffer[size - 1] != LINEFEED) || (size == 1 && buffer[0] == LINEFEED) ) {
04448             VectorPushBacks(stdoutBuffer, buffer, size);
04449             return size;
04450         }
04451         if ( (oldsize = VectorSize(stdoutBuffer)) > 0 ) {
04452             LONG ret;
04453             VectorPushBacks(stdoutBuffer, buffer, size);
04454             ret = WriteFileToFile(handle, VectorPtr(stdoutBuffer), VectorSize(stdoutBuffer));
04455             VectorClear(stdoutBuffer);
04456             if ( ret < 0 ) {
04457                 return ret;
04458             }
04459             else if ( ret < (LONG)oldsize ) {
04460                 return 0; /* This means the buffered output in previous calls is lost. */
04461             }
04462             else {
04463                 return ret - (LONG)oldsize;
04464             }
04465         }
04466     }
04467 #endif
04468     return WriteFileToFile(handle, buffer, size);
04469 }
04470
04471 /*
04472      #] PF_WriteFileToFile :
04473      #[ PF_FlushStdOutBuffer :
04474 */
04475

```

```

04480 void PF_FlushStdOutBuffer(void)
04481 {
04482     #ifndef PF_ENABLE_STDOUT_BUFFERING
04483         if ( PF.me == MASTER && VectorSize(stdoutBuffer) > 0 ) {
04484             WriteFileToFile(AM.StdOut, VectorPtr(stdoutBuffer), VectorSize(stdoutBuffer));
04485             VectorClear(stdoutBuffer);
04486         }
04487     #endif
04488 }
04489
04490 /*
04491     #] PF_FlushStdOutBuffer :
04492     #[ PF_ReceiveErrorMessage :
04493 */
04494
04503 static void PF_ReceiveErrorMessage(int src, int tag)
04504 {
04505     /* Only on the master. */
04506     int size;
04507     int ret = PF_RawProbe(&src, &tag, &size);
04508     CHECK(ret == 0);
04509     switch ( tag ) {
04510         case PF_STDOUT_MSGTAG:
04511             case PF_LOG_MSGTAG:
04512                 VectorReserve(recvBuffer, size);
04513                 ret = PF_RawRecv(&src, VectorPtr(recvBuffer), size, &tag);
04514                 CHECK(ret == size);
04515                 if ( size > 0 ) {
04516                     int handle = (tag == PF_STDOUT_MSGTAG) ? AM.StdOut : AC.LogHandle;
04517                     #ifndef PF_ENABLE_STDOUT_BUFFERING
04518                         if ( handle == AM.StdOut ) PF_WriteFileToFile(handle, VectorPtr(recvBuffer), size);
04519                     else
04520                         WriteFileToFile(handle, VectorPtr(recvBuffer), size);
04521                     }
04522                 }
04523                 break;
04524     }
04525 }
04526
04527 /*
04528     #] PF_ReceiveErrorMessage :
04529     #[ PF_CatchErrorMessages :
04530 */
04531
04540 static void PF_CatchErrorMessages(int *src, int *tag)
04541 {
04542     /* Only on the master. */
04543     for (;;) {
04544         int asrc = *src;
04545         int atag = *tag;
04546         int ret = PF_RawProbe(&asrc, &atag, NULL);
04547         CHECK(ret == 0);
04548         if ( atag == PF_STDOUT_MSGTAG || atag == PF_LOG_MSGTAG ) {
04549             PF_ReceiveErrorMessage(asrc, atag);
04550             continue;
04551         }
04552         *src = asrc;
04553         *tag = atag;
04554         break;
04555     }
04556 }
04557
04558 /*
04559     #] PF_CatchErrorMessages :
04560     #[ PF_CatchErrorMessagesForAll :
04561 */
04562
04567 static void PF_CatchErrorMessagesForAll(void)
04568 {
04569     /* Only on the master. */
04570     int i;
04571     for ( i = 1; i < PF.numtasks; i++ ) {
04572         int src = i;
04573         int tag = PF_ANY_MSGTAG;
04574         PF_CatchErrorMessages(&src, &tag);
04575     }
04576 }
04577
04578 /*
04579     #] PF_CatchErrorMessagesForAll :
04580     #[ PF_ProbeWithCatchingErrorMessages :
04581 */
04582
04592 static int PF_ProbeWithCatchingErrorMessages(int *src)
04593 {
04594     for (;;) {
04595         int newsrc = *src;

```



```

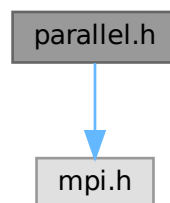
04596     int tag = PF_Probe(&newsrc);
04597     if ( tag == PF_STDOUT_MSGTAG || tag == PF_LOG_MSGTAG ) {
04598         PF_ReceiveErrorMessage(newsrc, tag);
04599         continue;
04600     }
04601     if ( tag > 0 ) *src = newsrc;
04602     return tag;
04603 }
04604 }
04605
04606 /*
04607     #] PF_ProbeWithCatchingErrorMessages :
04608     #[ PF_FreeErrorMessageBuffers :
04609 */
04610
04616 void PF_FreeErrorMessageBuffers(void)
04617 {
04618     VectorFree(stdoutBuffer);
04619     VectorFree(logBuffer);
04620 }
04621
04622 /*
04623     #] PF_FreeErrorMessageBuffers :
04624     #] Synchronised output :
04625 */

```

4.97 parallel.h File Reference

#include <mpi.h>

Include dependency graph for parallel.h:



Data Structures

- struct [PF_BUFFER](#)
- struct [ParallelVars](#)

Macros

- #define [MASTER](#) 0
- #define [PF_RESET](#) 0
- #define [PF_TIME](#) 1
- #define [PF_TERM_MSGTAG](#) 10 /* master -> slave: sending terms */
- #define [PF_ENDSORT_MSGTAG](#) 11 /* master -> slave: no more terms to be distributed, slave -> master: after [EndSort\(\)](#) */
- #define [PF_DOLLAR_MSGTAG](#) 12 /* slave -> master: sending \$-variables */

- `#define PF_BUFFER_MSGTAG 20 /* slave -> master: sending sorted terms, or in PF_SendFile()/PF_RecvFile() */`
- `#define PF_ENDBUFFER_MSGTAG 21 /* same as PF_BUFFER_MSGTAG, but indicates the end of operation */`
- `#define PF_READY_MSGTAG 30 /* slave -> master: slave is idle and can accept terms */`
- `#define PF_DATA_MSGTAG 50 /* InParallel, DoCheckpoint() */`
- `#define PF_EMPTY_MSGTAG 52 /* InParallel, DoCheckpoint(), PF_SendFile(), PF_RecvFile() */`
- `#define PF_STDOUT_MSGTAG 60 /* slave -> master: sending text to the stdout */`
- `#define PF_LOG_MSGTAG 61 /* slave -> master: sending text to the log file */`
- `#define PF_OPT_MCTS_MSGTAG 70 /* master <-> slave: optimization */`
- `#define PF_OPT_HORNER_MSGTAG 71 /* master <-> slave: optimization */`
- `#define PF_OPT_COLLECT_MSGTAG 72 /* slave -> master: optimization */`
- `#define PF_MISC_MSGTAG 100`
- `#define GNUMC_PREREQ(major, minor, patchlevel) 0`
- `#define OMPI_SKIP_MPICXX 1`
- `#define indices ((INDICES)(AC.IndexList.lijst))`
- `#define PF_ANY_SOURCE MPI_ANY_SOURCE`
- `#define PF_ANY_MSGTAG MPI_ANY_TAG`
- `#define PF_COMM MPI_COMM_WORLD`
- `#define PF_BYTE MPI_BYTE`
- `#define PF_INT MPI_INT`
- `#define PF_LongMultiPack(buffer, count, type) PF_LongMultiPackImpl(buffer, count, sizeof_datatype(type), type)`
- `#define PF_LongMultiUnpack(buffer, count, type) PF_LongMultiUnpackImpl(buffer, count, sizeof_datatype(type), type)`

Typedefs

- `typedef struct ParallelVars PARALLELVARS`

Functions

- `int PF_ISendSbuf (int, int)`
- `int PF_Bcast (void *buffer, int count)`
- `int PF_RawSend (int, void *, LONG, int)`
- `LONG PF_RawRecv (int *, void *, LONG, int *)`
- `int PF_PreparePack (void)`
- `int PF_Pack (const void *buffer, size_t count, MPI_Datatype type)`
- `int PF_Unpack (void *buffer, size_t count, MPI_Datatype type)`
- `int PF_PackString (const UBYTE *str)`
- `int PF_UnpackString (UBYTE *str)`
- `int PF_Send (int to, int tag)`
- `int PF_Receive (int src, int tag, int *psrc, int *ptag)`
- `int PF_Broadcast (void)`
- `int PF_PrepareLongSinglePack (void)`
- `int PF_LongSinglePack (const void *buffer, size_t count, MPI_Datatype type)`
- `int PF_LongSingleUnpack (void *buffer, size_t count, MPI_Datatype type)`
- `int PF_LongSingleSend (int to, int tag)`
- `int PF_LongSingleReceive (int src, int tag, int *psrc, int *ptag)`
- `int PF_PrepareLongMultiPack (void)`
- `int PF_LongMultiPackImpl (const void *buffer, size_t count, size_t eSize, MPI_Datatype type)`
- `int PF_LongMultiUnpackImpl (void *buffer, size_t count, size_t eSize, MPI_Datatype type)`
- `int PF_LongMultiBroadcast (void)`

- int [PF_EndSort](#) (void)
- WORD [PF_Deferred](#) (WORD *, WORD)
- int [PF_Processor](#) (EXPRESSIONS, WORD, WORD)
- int [PF_Init](#) (int *, char ***)
- int [PF_Terminate](#) (int)
- LONG [PF_GetSlaveTimes](#) (void)
- LONG [PF_BroadcastNumber](#) (LONG)
- void [PF_BroadcastBuffer](#) (WORD **buffer, LONG *length)
- int [PF_BroadcastString](#) (UBYTE *)
- int [PF_BroadcastPreDollar](#) (WORD **, LONG *, int *)
- int [PF_CollectModifiedDollars](#) (void)
- int [PF_BroadcastModifiedDollars](#) (void)
- int [PF_BroadcastRedefinedPreVars](#) (void)
- int [PF_BroadcastCBuf](#) (int bufnum)
- int [PF_BroadcastExpFlags](#) (void)
- int [PF_StoreInsideInfo](#) (void)
- int [PF_RestoreInsideInfo](#) (void)
- int [PF_BroadcastExpr](#) (EXPRESSIONS e, FILEHANDLE *file)
- int [PF_BroadcastRHS](#) (void)
- int [PF_InParallelProcessor](#) (void)
- int [PF_SendFile](#) (int to, FILE *fd)
- int [PF_RecvFile](#) (int from, FILE *fd)
- void [PF_MLock](#) (void)
- void [PF_MUnlock](#) (void)
- LONG [PF_WriteFileToFile](#) (int, UBYTE *, LONG)
- void [PF_FlushStdOutBuffer](#) (void)

Variables

- [PARALLELVARS](#) PF
- LONG [PF_maxDollarChunkSize](#)

4.97.1 Detailed Description

Header file with things relevant to ParForm.

Definition in file [parallel.h](#).

4.97.2 Macro Definition Documentation

4.97.2.1 MASTER

```
#define MASTER 0
```

Definition at line 39 of file [parallel.h](#).

4.97.2.2 PF_RESET

```
#define PF_RESET 0
```

Definition at line 41 of file [parallel.h](#).

4.97.2.3 PF_TIME

```
#define PF_TIME 1
```

Definition at line 42 of file [parallel.h](#).

4.97.2.4 PF_TERM_MSGTAG

```
#define PF_TERM_MSGTAG 10 /* master -> slave:  sending terms */
```

Definition at line 44 of file [parallel.h](#).

4.97.2.5 PF_ENDSORT_MSGTAG

```
#define PF_ENDSORT_MSGTAG 11 /* master -> slave:  no more terms to be distributed, slave ->
master:  after EndSort() */
```

Definition at line 45 of file [parallel.h](#).

4.97.2.6 PF_DOLLAR_MSGTAG

```
#define PF_DOLLAR_MSGTAG 12 /* slave -> master:  sending $-variables */
```

Definition at line 46 of file [parallel.h](#).

4.97.2.7 PF_BUFFER_MSGTAG

```
#define PF_BUFFER_MSGTAG 20 /* slave -> master:  sending sorted terms, or in PF_SendFile()/PF_RecvFile()
*/
```

Definition at line 47 of file [parallel.h](#).

4.97.2.8 PF_ENDBUFFER_MSGTAG

```
#define PF_ENDBUFFER_MSGTAG 21 /* same as PF_BUFFER_MSGTAG, but indicates the end of operation
*/
```

Definition at line 48 of file [parallel.h](#).

4.97.2.9 PF_READY_MSGTAG

```
#define PF_READY_MSGTAG 30 /* slave -> master:  slave is idle and can accept terms */
```

Definition at line 49 of file [parallel.h](#).

4.97.2.10 PF_DATA_MSGTAG

```
#define PF_DATA_MSGTAG 50 /* InParallel, DoCheckpoint() */
```

Definition at line 50 of file [parallel.h](#).

4.97.2.11 PF_EMPTY_MSGTAG

```
#define PF_EMPTY_MSGTAG 52 /* InParallel, DoCheckpoint(), PF_SendFile(), PF_RecvFile() */
```

Definition at line 51 of file [parallel.h](#).

4.97.2.12 PF_STDOUT_MSGTAG

```
#define PF_STDOUT_MSGTAG 60 /* slave -> master: sending text to the stdout */
```

Definition at line 52 of file [parallel.h](#).

4.97.2.13 PF_LOG_MSGTAG

```
#define PF_LOG_MSGTAG 61 /* slave -> master: sending text to the log file */
```

Definition at line 53 of file [parallel.h](#).

4.97.2.14 PF_OPT_MCTS_MSGTAG

```
#define PF_OPT_MCTS_MSGTAG 70 /* master <-> slave: optimization */
```

Definition at line 54 of file [parallel.h](#).

4.97.2.15 PF_OPT_HORNER_MSGTAG

```
#define PF_OPT_HORNER_MSGTAG 71 /* master <-> slave: optimization */
```

Definition at line 55 of file [parallel.h](#).

4.97.2.16 PF_OPT_COLLECT_MSGTAG

```
#define PF_OPT_COLLECT_MSGTAG 72 /* slave -> master: optimization */
```

Definition at line 56 of file [parallel.h](#).

4.97.2.17 PF_MISC_MSGTAG

```
#define PF_MISC_MSGTAG 100
```

Definition at line 57 of file [parallel.h](#).

4.97.2.18 GNUC_PREREQ

```
#define GNUC_PREREQ(  
    major,  
    minor,  
    patchlevel ) 0
```

Definition at line 67 of file [parallel.h](#).

4.97.2.19 OMPI_SKIP_MPICXX

```
#define OMPI_SKIP_MPICXX 1
```

Definition at line 108 of file [parallel.h](#).

4.97.2.20 indices

```
#define indices ((INDICES)(AC.IndexList.lijst))
```

Definition at line 113 of file [parallel.h](#).

4.97.2.21 PF_ANY_SOURCE

```
#define PF_ANY_SOURCE MPI_ANY_SOURCE
```

Definition at line 123 of file [parallel.h](#).

4.97.2.22 PF_ANY_MSGTAG

```
#define PF_ANY_MSGTAG MPI_ANY_TAG
```

Definition at line 124 of file [parallel.h](#).

4.97.2.23 PF_COMM

```
#define PF_COMM MPI_COMM_WORLD
```

Definition at line 125 of file [parallel.h](#).

4.97.2.24 PF_BYTE

```
#define PF_BYTE MPI_BYTE
```

Definition at line 126 of file [parallel.h](#).

4.97.2.25 PF_INT

```
#define PF_INT MPI_INT
```

Definition at line 127 of file [parallel.h](#).

4.97.2.26 PF_LongMultiPack

```
#define PF_LongMultiPack(  
    buffer,  
    count,  
    type ) PF_LongMultiPackImpl(buffer, count, sizeof_datatype(type), type)
```

Definition at line 232 of file [parallel.h](#).

4.97.2.27 PF_LongMultiUnpack

```
#define PF_LongMultiUnpack(  
    buffer,  
    count,  
    type ) PF_LongMultiUnpackImpl(buffer, count, sizeof_datatype(type), type)
```

Definition at line 233 of file [parallel.h](#).

4.97.3 Function Documentation

4.97.3.1 PF_IsendSbuf()

```
int PF_IsendSbuf (  
    int to,  
    int tag ) [extern]
```

Nonblocking send operation. It sends everything from *buff* to *fill* of the active buffer. Depending on *tag* it also can do waiting for other sends to finish or set the active buffer to the next one.

Parameters

<i>to</i>	the destination process number.
<i>tag</i>	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 261 of file [mpi.c](#).

Referenced by [FlushOut\(\)](#), [PF_Processor\(\)](#), and [PutOut\(\)](#).

4.97.3.2 PF_Bcast()

```
int PF_Bcast (
    void * buffer,
    int count ) [extern]
```

Broadcasts a message from the master to slaves.

Parameters

<i>in, out</i>	<i>buffer</i>	the starting address of buffer. The contents in this buffer on the master will be transferred to those on the slaves.
	<i>count</i>	the length of the buffer in bytes.

Returns

0 if OK, nonzero on error.

Definition at line [440](#) of file [mpi.c](#).

Referenced by [PF_BroadcastBuffer\(\)](#), and [PF_BroadcastNumber\(\)](#).

4.97.3.3 PF_RawSend()

```
int PF_RawSend (
    int dest,
    void * buf,
    LONG l,
    int tag ) [extern]
```

Sends *l* bytes from *buf* to *dest*. Returns 0 on success, or -1.

Parameters

<i>dest</i>	the destination process number.
<i>buf</i>	the send buffer.
<i>l</i>	the size of data in the send buffer in bytes.
<i>tag</i>	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line [463](#) of file [mpi.c](#).

4.97.3.4 PF_RawRecv()

```
LONG PF_RawRecv (
    int * src,
```



```
void * buf,
LONG thesize,
int * tag ) [extern]
```

Receives not more than *thesize* bytes from *src*, returns the actual number of received bytes, or -1 on failure.

Parameters

in, out	<i>src</i>	the source process number. In output, that of the actual received message.
out	<i>buf</i>	the receive buffer.
	<i>thesize</i>	the size of the receive buffer in bytes.
out	<i>tag</i>	the message tag of the actual received message.

Returns

the actual sizeof received data in bytes, or -1 on failure.

Definition at line 484 of file [mpi.c](#).

4.97.3.5 PF_PreparePack()

```
int PF_PreparePack (
    void ) [extern]
```

Prepares for the next pack operations on the sender.

Returns

0 if OK, nonzero on error.

Definition at line 624 of file [mpi.c](#).

Referenced by [Optimize\(\)](#), [PF_BroadcastPreDollar\(\)](#), [PF_BroadcastString\(\)](#), and [PF_Init\(\)](#).

4.97.3.6 PF_Pack()

```
int PF_Pack (
    const void * buffer,
    size_t count,
    MPI_Datatype type ) [extern]
```

Adds data into the pack buffer.

Parameters

<i>buffer</i>	the pointer to the buffer storing the data to be packed.
<i>count</i>	the number of elements in the buffer.
<i>type</i>	the data type of elements in the buffer.

Returns

0 if OK, nonzero on error.

Definition at line 642 of file [mpi.c](#).

References [MPI_ERRCODE_CHECK](#).

Referenced by [Optimize\(\)](#), [PF_BroadcastPreDollar\(\)](#), and [PF_Init\(\)](#).

4.97.3.7 PF_Unpack()

```
int PF_Unpack (
    void * buffer,
    size_t count,
    MPI_Datatype type ) [extern]
```

Retrieves the next data in the pack buffer.

Parameters

out	<i>buffer</i>	the pointer to the buffer to store the unpacked data.
	<i>count</i>	the number of elements of data to be received.
	<i>type</i>	the data type of elements of data to be received.

Returns

0 if OK, nonzero on error.

Definition at line 671 of file [mpi.c](#).

References [MPI_ERRCODE_CHECK](#).

Referenced by [Optimize\(\)](#), [PF_BroadcastPreDollar\(\)](#), and [PF_Init\(\)](#).

4.97.3.8 PF_PackString()

```
int PF_PackString (
    const UBYTE * str ) [extern]
```

Packs a string *str* into the packed buffer `PF_packbuf`, including the trailing zero.

The first element (`PF_INT`) is the length of the packed portion of the string. If the string does not fit to the buffer `PF_packbuf`, the function packs only the initial portion. It returns the number of packed bytes, so if `(str[length-1]=='\0')` then the whole string fits to the buffer, if not, then the rest (`str+length`) must be packed and send again. On error, the function returns the negative error code.

One exception: the string `"\0\0"` is used as an image of the NULL, so all 3 characters will be packed.

Parameters

<i>str</i>	a string to be packed.

Returns

the number of packed bytes, or a negative value on failure.

Definition at line 706 of file [mpi.c](#).

Referenced by [PF_BroadcastString\(\)](#).

4.97.3.9 PF_UnpackString()

```
int PF_UnpackString (
    UBYTE * str ) [extern]
```

Unpacks a string to *str* from the packed buffer PF_packbuf, including the trailing zero.

It returns the number of unpacked bytes, so if (str[length-1]=='\0') then the whole string was unpacked, if not, then the rest must be appended to (str+length). On error, the function returns the negative error code.

Parameters

<i>out</i>	<i>str</i>	the buffer to store the unpacked string

Returns

the number of unpacked bytes, or a negative value on failure.

Definition at line 774 of file [mpi.c](#).

Referenced by [PF_BroadcastString\(\)](#).

4.97.3.10 PF_Send()

```
int PF_Send (
    int to,
    int tag ) [extern]
```

Sends the contents in the pack buffer to the process specified by *to*.

Example:

```
if ( PF.me == SRC ) {
    PF_PreparePack();
    // Packing operations here...
    PF_Send(DEST, TAG);
}
else if ( PF.me == DEST ) {
    PF_Receive(SRC, TAG, &actual_src, &actual_tag);
    // Unpacking operations here...
}
```

Parameters

<i>to</i>	the destination process number.
<i>tag</i>	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 822 of file [mpi.c](#).

References [MPI_ERRCODE_CHECK](#).

4.97.3.11 PF_Receive()

```
int PF_Receive (
    int src,
    int tag,
    int * psrc,
    int * ptag ) [extern]
```

Receives data into the pack buffer from the process specified by *src*. This function allows &*src* == *psrc* or &*tag* == *ptag*. Either *psrc* or *ptag* can be NULL.

See the example of [PF_Send\(\)](#).

Parameters

	<i>src</i>	the source process number (can be PF_ANY_SOURCE).
	<i>tag</i>	the source message tag (can be PF_ANY_TAG).
out	<i>psrc</i>	the actual source process number of received message.
out	<i>ptag</i>	the received message tag.

Returns

0 if OK, nonzero on error.

Definition at line 848 of file [mpi.c](#).

References [MPI_ERRCODE_CHECK](#).

4.97.3.12 PF_Broadcast()

```
int PF_Broadcast (
    void ) [extern]
```

Broadcasts the contents in the pack buffer on the master to those on the slaves.

Example:

```
if ( PF.me == MASTER ) {
    PF_PreparePack();
    // Packing operations here...
}
PF_Broadcast();
if ( PF.me != MASTER ) {
    // Unpacking operations here...
}
```

Returns

0 if OK, nonzero on error.

Definition at line 883 of file [mpi.c](#).

References [MPI_ERRCODE_CHECK](#).

Referenced by [Optimize\(\)](#), [PF_BroadcastPreDollar\(\)](#), [PF_BroadcastString\(\)](#), and [PF_Init\(\)](#).

4.97.3.13 PF_PrepareLongSinglePack()

```
int PF_PrepareLongSinglePack (
    void ) [extern]
```

Prepares for the next long-single-pack operations on the sender.

Returns

0 if OK, nonzero on error.

Definition at line 1451 of file [mpi.c](#).

Referenced by [PF_CollectModifiedDollars\(\)](#), and [PF_Processor\(\)](#).

4.97.3.14 PF_LongSinglePack()

```
int PF_LongSinglePack (
    const void * buffer,
    size_t count,
    MPI_Datatype type ) [extern]
```

Adds data into the "long single" pack buffer.

Parameters

<i>buffer</i>	the pointer to the buffer storing the data to be packed.
<i>count</i>	the number of elements in the buffer.
<i>type</i>	the data type of elements in the buffer.

Returns

0 if OK, nonzero on error.

Definition at line 1469 of file [mpi.c](#).

Referenced by [PF_CollectModifiedDollars\(\)](#), and [PF_Processor\(\)](#).

4.97.3.15 PF_LongSingleUnpack()

```
int PF_LongSingleUnpack (
    void * buffer,
    size_t count,
    MPI_Datatype type ) [extern]
```

Retrieves the next data in the "long single" pack buffer.

Parameters

out	<i>buffer</i>	the pointer to the buffer to store the unpacked data.
	<i>count</i>	the number of elements of data to be received.
	<i>type</i>	the data type of elements of data to be received.

Returns

0 if OK, nonzero on error.

Definition at line 1503 of file [mpi.c](#).

Referenced by [PF_CollectModifiedDollars\(\)](#), and [PF_Processor\(\)](#).

4.97.3.16 PF_LongSingleSend()

```
int PF_LongSingleSend (
    int to,
    int tag ) [extern]
```

Sends the contents in the "long single" pack buffer to the process specified by *to*.

Example:

```
if ( PF.me == SRC ) {
    PF_PrepareLongSinglePack();
    // Packing operations here...
    PF_LongSingleSend(DEST, TAG);
}
else if ( PF.me == DEST ) {
    PF_LongSingleReceive(SRC, TAG, &actual_src, &actual_tag);
    // Unpacking operations here...
}
```

Parameters

<i>to</i>	the destination process number.
<i>tag</i>	the message tag.

Returns

0 if OK, nonzero on error.

Definition at line 1540 of file [mpi.c](#).

Referenced by [PF_CollectModifiedDollars\(\)](#), and [PF_Processor\(\)](#).

4.97.3.17 PF_LongSingleReceive()

```
int PF_LongSingleReceive (
    int src,
    int tag,
    int * psrc,
    int * ptag ) [extern]
```

Receives data into the "long single" pack buffer from the process specified by *src*. This function allows &src == psrc or &tag == ptag. Either *psrc* or *ptag* can be NULL.

See the example of [PF_LongSingleSend\(\)](#).

Parameters

	<i>src</i>	the source process number (can be PF_ANY_SOURCE).
	<i>tag</i>	the source message tag (can be PF_ANY_TAG).
out	<i>psrc</i>	the actual source process number of received message.
out	<i>ptag</i>	the received message tag.

Returns

0 if OK, nonzero on error.

Definition at line 1583 of file [mpi.c](#).

Referenced by [PF_CollectModifiedDollars\(\)](#), and [PF_Processor\(\)](#).

4.97.3.18 PF_PrepareLongMultiPack()

```
int PF_PrepareLongMultiPack (
    void ) [extern]
```

Prepares for the next long-multi-pack operations on the sender.

Returns

0 if OK, nonzero on error.

Definition at line 1643 of file [mpi.c](#).

Referenced by [Optimize\(\)](#), [PF_BroadcastCBuf\(\)](#), [PF_BroadcastExpFlags\(\)](#), [PF_BroadcastModifiedDollars\(\)](#), and [PF_BroadcastRedefinedPreVars\(\)](#).

4.97.3.19 PF_LongMultiPackImpl()

```
int PF_LongMultiPackImpl (
    const void * buffer,
    size_t count,
    size_t eSize,
    MPI_Datatype type ) [extern]
```

Adds data into the "long multi" pack buffer.

Parameters

<i>buffer</i>	the pointer to the buffer storing the data to be packed.
<i>count</i>	the number of elements in the buffer.
<i>eSize</i>	the byte size of each element of data.
<i>type</i>	the data type of elements in the buffer.

Returns

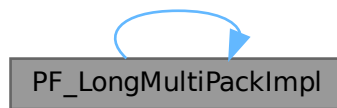
0 if OK, nonzero on error.

Definition at line 1662 of file [mpi.c](#).

References [PF_LongMultiPackImpl\(\)](#).

Referenced by [PF_LongMultiPackImpl\(\)](#).

Here is the call graph for this function:

**4.97.3.20 PF_LongMultiUnpackImpl()**

```

int PF_LongMultiUnpackImpl (
    void * buffer,
    size_t count,
    size_t eSize,
    MPI_Datatype type ) [extern]
  
```

Retrieves the next data in the "long multi" pack buffer.

Parameters

out	<i>buffer</i>	the pointer to the buffer to store the unpacked data.
	<i>count</i>	the number of elements of data to be received.
	<i>eSize</i>	the byte size of each element of data.
	<i>type</i>	the data type of elements of data to be received.

Returns

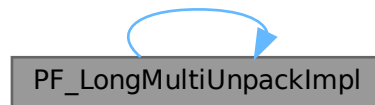
0 if OK, nonzero on error.

Definition at line 1721 of file [mpi.c](#).

References [PF_LongMultiUnpackImpl\(\)](#).

Referenced by [PF_LongMultiUnpackImpl\(\)](#).

Here is the call graph for this function:



4.97.3.21 PF_LongMultiBroadcast()

```
int PF_LongMultiBroadcast (
    void ) [extern]
```

Broadcasts the contents in the "long multi" pack buffer on the master to those on the slaves.

Example:

```
if ( PF.me == MASTER ) {
    PF_PrepareLongMultiPack();
    // Packing operations here...
}
PF_LongMultiBroadcast();
if ( PF.me != MASTER ) {
    // Unpacking operations here...
}
```

Returns

0 if OK, nonzero on error.

Definition at line 1807 of file [mpi.c](#).

Referenced by [Optimize\(\)](#), [PF_BroadcastCBuf\(\)](#), [PF_BroadcastExpFlags\(\)](#), [PF_BroadcastModifiedDollars\(\)](#), and [PF_BroadcastRedefinedPreVars\(\)](#).

4.97.3.22 PF_EndSort()

```
int PF_EndSort (
    void ) [extern]
```

Finishes a master sorting with collecting terms from slaves. Called by [EndSort\(\)](#).

If this is not the masterprocess, just initialize the sendbuffers and return 0, else [PF_EndSort\(\)](#) sends the rest of the terms in the sendbuffer to the next slave and a dummy message to all slaves with tag PF_ENDSORT_MSGTAG. Then it receives the sorted terms, sorts them using a recursive 'tree of losers' ([PF_GetLoser\(\)](#)) and writes them to the outputfile.

Returns

1 if the sorting on the master was done. 0 if [EndSort\(\)](#) still must perform a regular sorting because it is not at the ground level or not on the master or in the sequential mode or in the InParallel mode. -1 if an error occurred.

Remarks

The slaves will send the sorted terms back to the master in the regular sorting (after the initialization of the send buffer in [PF_EndSort\(\)](#)). See [PutOut\(\)](#) and [FlushOut\(\)](#).

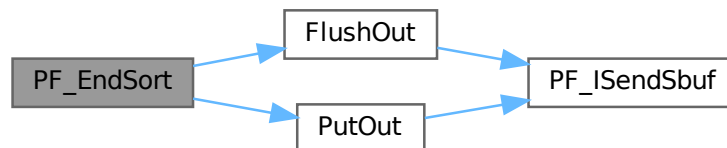
This function has been changed such that when it returns 1, AM.S0->TermsLeft is set correctly. But AM.S0->GenTerms is not set: it will be set after collecting the statistics from the slaves at the end of [PF_Processor\(\)](#). (TU 30 Jun 2011)

Definition at line 864 of file [parallel.c](#).

References [FlushOut\(\)](#), and [PutOut\(\)](#).

Referenced by [EndSort\(\)](#).

Here is the call graph for this function:

**4.97.3.23 PF_Deferred()**

```
WORD PF_Deferred (
    WORD * term,
    WORD level ) [extern]
```

Replaces [Deferred\(\)](#) on the slaves.

Parameters

<i>term</i>	the term that must be multiplied by the contents of the current bracket.
<i>level</i>	the compiler level.

Returns

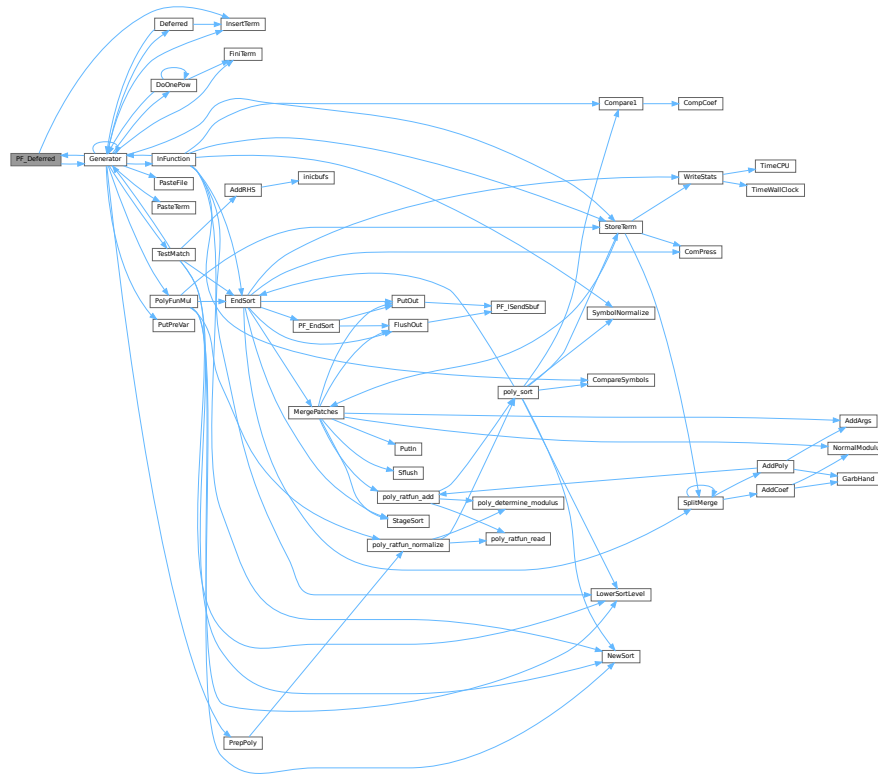
0 if OK, nonzero on error.

Definition at line 1208 of file [parallel.c](#).

References [Generator\(\)](#), and [InsertTerm\(\)](#).

Referenced by [Generator\(\)](#).

Here is the call graph for this function:



4.97.3.24 PF_Processor()

```
int PF_Processor (
    EXPRESSIONS e,
    WORD i,
    WORD LastExpression ) [extern]
```

Replaces parts of [Processor\(\)](#) on the masters and slaves. On the master [PF_Processor\(\)](#) is responsible for proper distribution of terms from the input file to the slaves. On the slaves it calls [Generator\(\)](#) for all the terms that this process gets, but [PF_GetTerm\(\)](#) gets terms from the master (not directly from infile).

Parameters

<i>e</i>	The pointer to the current expression.
<i>i</i>	The index for the current expression.
<i>LastExpression</i>	The flag indicating whether it is the last expression.

Returns

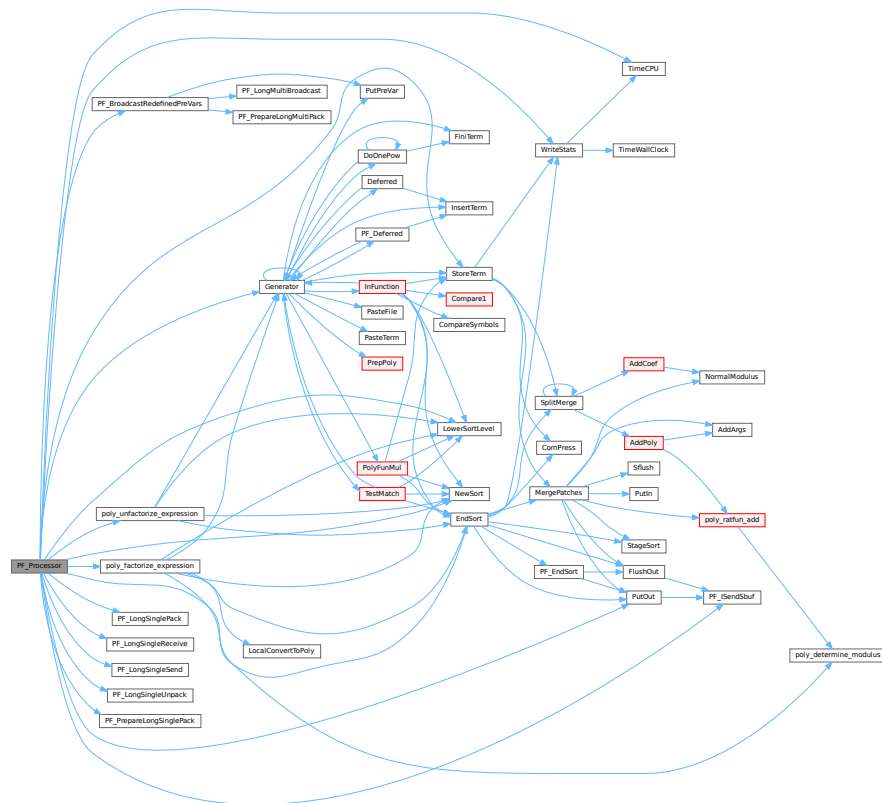
0 if OK, nonzero on error.

Definition at line 1540 of file [parallel.c](#).

References [EndSort\(\)](#), [Generator\(\)](#), [LowerSortLevel\(\)](#), [NewSort\(\)](#), [PACK_LONG](#), [PF_BroadcastRedefinePreVars\(\)](#), [PF_ISendSbuf\(\)](#), [PF_LongSinglePack\(\)](#), [PF_LongSingleReceive\(\)](#), [PF_LongSingleSend\(\)](#), [PF_LongSingleUnpack\(\)](#), [PF_PrepareLongSinglePack\(\)](#), [poly_factorize_expression\(\)](#), [poly_unfactorize_expression\(\)](#), [PutOut\(\)](#), [StoreTerm\(\)](#), [SWAP](#), [TimeCPU\(\)](#), and [WriteStats\(\)](#).

Referenced by [Processor\(\)](#).

Here is the call graph for this function:



4.97.3.25 PF_Init()

```
int PF_Init (
    int * argc,
    char *** argv ) [extern]
```

All the library independent stuff. [PF_LibInit\(\)](#) should do all library dependent initializations.

Parameters

<i>argc</i>	pointer to the number of arguments.
<i>argv</i>	pointer to the arguments.

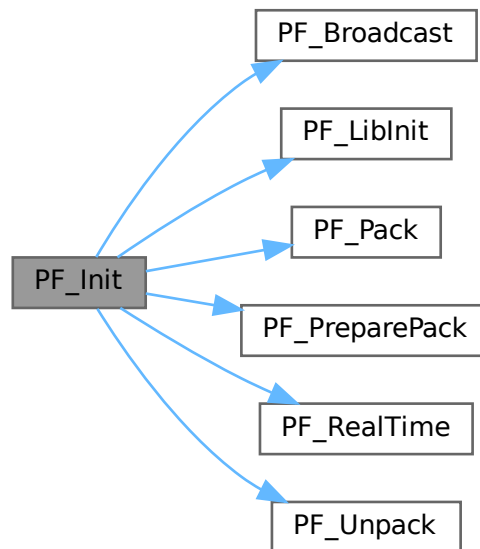
Returns

0 if OK, nonzero on error.

Definition at line 1965 of file [parallel.c](#).

References [PF_Broadcast\(\)](#), [PF_LibInit\(\)](#), [PF_Pack\(\)](#), [PF_PreparePack\(\)](#), [PF_RealTime\(\)](#), and [PF_Unpack\(\)](#).

Here is the call graph for this function:



4.97.3.26 PF_Terminate()

```
int PF_Terminate (
    int errorcode ) [extern]
```

Performs the finalization of ParFORM. To be called by `Terminate()`.

Parameters

<i>error</i>	an error code.

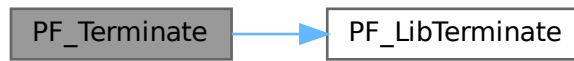
Returns

0 if OK, nonzero on error.

Definition at line 2060 of file [parallel.c](#).

References [PF_LibTerminate\(\)](#).

Here is the call graph for this function:



4.97.3.27 PF_GetSlaveTimes()

```
LONG PF_GetSlaveTimes (
    void ) [extern]
```

Returns the total CPU time of all slaves together. This function must be called on the master and all slaves.

Returns

on the master, the sum of CPU times on all slaves.

Definition at line 2076 of file [parallel.c](#).

References [TimeCPU\(\)](#).

Here is the call graph for this function:



4.97.3.28 PF_BroadcastNumber()

```
LONG PF_BroadcastNumber (
    LONG x ) [extern]
```

Broadcasts a LONG value from the master to the all slaves.

Parameters

x	the number to be broadcast (set on the master).

Returns

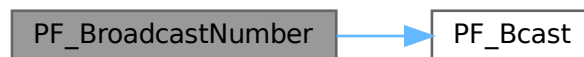
the synchronised result.

Definition at line 2096 of file [parallel.c](#).

References [PF_Bcast\(\)](#).

Referenced by [DoCheckpoint\(\)](#), [PF_BroadcastBuffer\(\)](#), and [StartVariables\(\)](#).

Here is the call graph for this function:

**4.97.3.29 PF_BroadcastBuffer()**

```
void PF_BroadcastBuffer (
    WORD ** buffer,
    LONG * length ) [extern]
```

Broadcasts a buffer from the master to all the slaves.

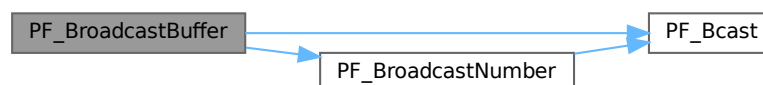
Parameters

in, out	<i>buffer</i>	on the master, the buffer to be broadcast. On the slaves, the buffer will be allocated if the length is greater than 0. The caller must free it.
in, out	<i>length</i>	on the master, the length of the buffer to be broadcast. On the slaves, it receives the length of transferred buffer. The actual transfer occurs only if the length is greater than 0.

Definition at line 2123 of file [parallel.c](#).

References [PF_Bcast\(\)](#), and [PF_BroadcastNumber\(\)](#).

Here is the call graph for this function:



4.97.3.30 PF_BroadcastString()

```
int PF_BroadcastString (
    UBYTE * str ) [extern]
```

Broadcasts a string from the master to all slaves.

Parameters

<i>in, out</i>	<i>str</i>	The pointer to a null-terminated string.

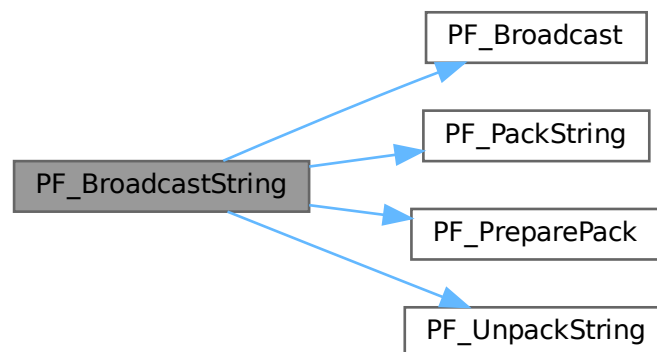
Returns

0 if OK, nonzero on error.

Definition at line 2165 of file [parallel.c](#).

References [PF_Broadcast\(\)](#), [PF_PackString\(\)](#), [PF_PreparePack\(\)](#), and [PF_UnpackString\(\)](#).

Here is the call graph for this function:



4.97.3.31 PF_BroadcastPreDollar()

```
int PF_BroadcastPreDollar (
    WORD ** dbuffer,
    LONG * newsize,
    int * numterms ) [extern]
```

Broadcasts dollar variables set as a preprocessor variables. Only the master is able to make an assignment like `#$a=g;` where `g` is an expression: only the master has an access to the expression. So, the master broadcasts the result to slaves.

The result is in `*dbuffer` of the size is `*newsize` (in number of WORDs), +1 for trailing zero. For slave `newsize` and `numterms` are output parameters.

Parameters

<i>in, out</i>	<i>dbuffer</i>	the buffer for a dollar variable.
<i>in, out</i>	<i>newsize</i>	the size of the dollar variable in WORDs.
<i>in, out</i>	<i>numterms</i>	the number of terms in the dollar variable.

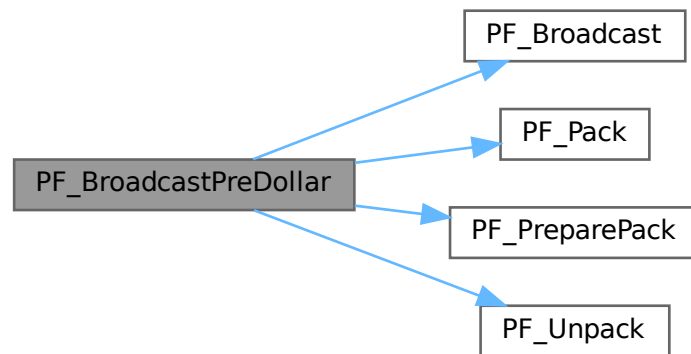
Returns

0 if OK, nonzero on error.

Definition at line 2220 of file [parallel.c](#).

References [PF_Broadcast\(\)](#), [PF_Pack\(\)](#), [PF_PreparePack\(\)](#), and [PF_Unpack\(\)](#).

Here is the call graph for this function:



4.97.3.32 PF_CollectModifiedDollars()

```
int PF_CollectModifiedDollars (
    void ) [extern]
```

Combines modified dollar variables on the all slaves, and store them into those on the master.

The potentially modified dollar variables are given in `PotModdollars`, and the number of them is given by `NumPotModdollars`.

The current module could be executed in parallel only if all potentially modified variables are listed in `ModOptdollars`, otherwise the module was switched to the sequential mode.

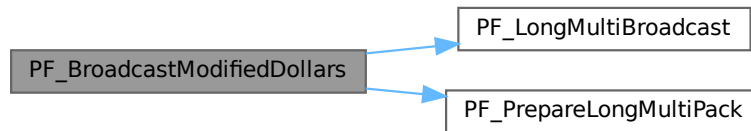
Returns

0 if OK, nonzero on error.

Definition at line 2786 of file [parallel.c](#).

References [PF_LongMultiBroadcast\(\)](#), and [PF_PrepareLongMultiPack\(\)](#).

Here is the call graph for this function:

**4.97.3.34 PF_BroadcastRedefinedPreVars()**

```
int PF_BroadcastRedefinedPreVars (
    void ) [extern]
```

Broadcasts preprocessor variables, which were changed by the Redefine statements in the current module, from the master to the all slaves.

The potentially redefined preprocessor variables are given in `AC.pfirstnum`, and the number of them is given by `AC.numpfirstnum`. For an actually redefined variable, the corresponding value in `AC.inputnumbers` is non-negative.

Returns

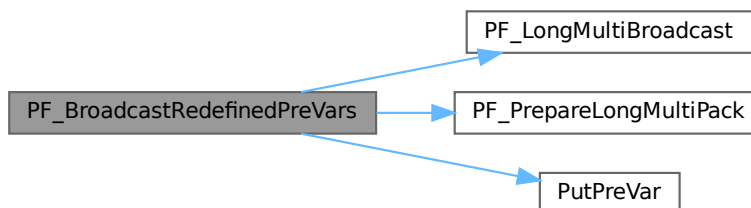
0 if OK, nonzero on error.

Definition at line 3003 of file [parallel.c](#).

References [PF_LongMultiBroadcast\(\)](#), [PF_PrepareLongMultiPack\(\)](#), [PutPreVar\(\)](#), [VectorPtr](#), and [VectorReserve](#).

Referenced by [PF_Processor\(\)](#).

Here is the call graph for this function:



4.97.3.35 PF_BroadcastCBuf()

```
int PF_BroadcastCBuf (
    int bufnum ) [extern]
```

Broadcasts a compiler buffer specified by *bufnum* from the master to the all slaves.

Parameters

<i>bufnum</i>	The index of the compiler buffer to be broadcast.

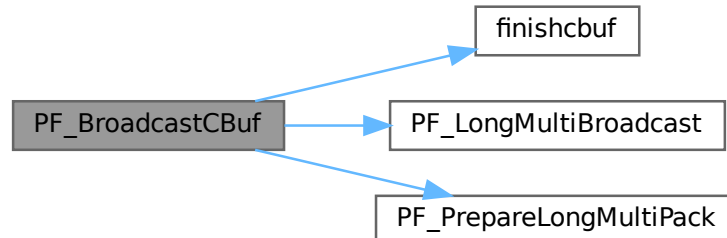
Returns

0 if OK, nonzero on error.

Definition at line 3145 of file [parallel.c](#).

References [CbUf::boomlijst](#), [CbUf::Buffer](#), [CbUf::BufferSize](#), [CbUf::CanCommu](#), [CbUf::dimension](#), [finishcbuf\(\)](#), [CbUf::lhs](#), [CbUf::numdum](#), [CbUf::NumTerms](#), [PF_LongMultiBroadcast\(\)](#), [PF_PrepareLongMultiPack\(\)](#), [CbUf::Pointer](#), [CbUf::rhs](#), and [CbUf::Top](#).

Here is the call graph for this function:



4.97.3.36 PF_BroadcastExpFlags()

```
int PF_BroadcastExpFlags (
    void ) [extern]
```

Broadcasts `AR.expflags` and several properties of each expression, e.g., `e->vflags`, from the master to all slaves.

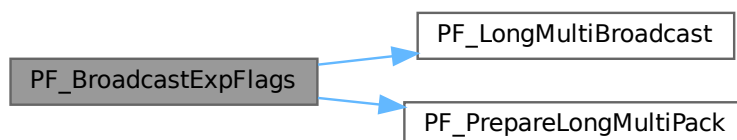
Returns

0 if OK, nonzero on error.

Definition at line 3256 of file [parallel.c](#).

References [PF_LongMultiBroadcast\(\)](#), and [PF_PrepareLongMultiPack\(\)](#).

Here is the call graph for this function:

**4.97.3.37 PF_StoreInsideInfo()**

```
int PF_StoreInsideInfo (
    void ) [extern]
```

Definition at line 3093 of file [parallel.c](#).

4.97.3.38 PF_RestoreInsideInfo()

```
int PF_RestoreInsideInfo (
    void ) [extern]
```

Definition at line 3119 of file [parallel.c](#).

4.97.3.39 PF_BroadcastExpr()

```
int PF_BroadcastExpr (
    EXPRESSIONS e,
    FILEHANDLE * file ) [extern]
```

Broadcasts an expression from the master to the all slaves.

Parameters

<i>e</i>	The expression to be broadcast.
<i>file</i>	The file in which the expression is sitting.

Returns

0 if OK, nonzero on error.

Definition at line 3550 of file [parallel.c](#).

Referenced by [PF_BroadcastRHS\(\)](#).

4.97.3.40 PF_BroadcastRHS()

```
int PF_BroadcastRHS (
    void ) [extern]
```

Broadcasts expressions appearing in the right-hand side from the master to the all slaves.

Returns

0 if OK, nonzero on error.

Definition at line 3578 of file [parallel.c](#).

References [PF_BroadcastExpr\(\)](#).

Referenced by [Processor\(\)](#).

Here is the call graph for this function:

**4.97.3.41 PF_InParallelProcessor()**

```
int PF_InParallelProcessor (
    void ) [extern]
```

Processes expressions in the InParallel mode, i.e., dividing expressions marked by `partodo` over the slaves.

Returns

0 if OK, nonzero on error.

Definition at line 3625 of file [parallel.c](#).

Referenced by [Processor\(\)](#).

4.97.3.42 PF_SendFile()

```
int PF_SendFile (
    int to,
    FILE * fd ) [extern]
```

Sends a file to the process specified by `to`.

Parameters

<i>to</i>	the destination process number.
<i>fd</i>	the file to be sent.

Returns

the size of sent data in bytes, or -1 on error.

Definition at line 4222 of file [parallel.c](#).

References [PF_RawSend\(\)](#).

Referenced by [DoCheckpoint\(\)](#).

Here is the call graph for this function:



4.97.3.43 PF_RecvFile()

```
int PF_RecvFile (
    int from,
    FILE * fd ) [extern]
```

Receives a file from the process specified by *from*.

Parameters

<i>from</i>	the source process number.
<i>fd</i>	the file to save the received data.

Returns

the size of received data in bytes, or -1 on error.

Definition at line 4260 of file [parallel.c](#).

References [PF_RawRecv\(\)](#).

Referenced by [DoCheckpoint\(\)](#).

Here is the call graph for this function:



4.97.3.44 PF_MLock()

```
void PF_MLock (
    void ) [extern]
```

A function called by MLOCK(ErrorMessageLock) for slaves.

Definition at line [4341](#) of file [parallel.c](#).

References [VectorClear](#).

4.97.3.45 PF_MUnlock()

```
void PF_MUnlock (
    void ) [extern]
```

A function called by MUNLOCK(ErrorMessageLock) for slaves.

Definition at line [4357](#) of file [parallel.c](#).

References [PF_RawSend\(\)](#), [VectorEmpty](#), [VectorPtr](#), and [VectorSize](#).

Here is the call graph for this function:



4.97.3.46 PF_WriteFileToFile()

```
LONG PF_WriteFileToFile (
    int handle,
    UBYTE * buffer,
    LONG size ) [extern]
```

Replaces WriteFileToFile() on the master and slaves.

It copies the given buffer into internal buffers if called between MLOCK(ErrorMessageLock) and MUNLOCK(ErrorMessageLock) for slaves and handle is StdOut or LogHandle, otherwise calls WriteFileToFile().

Parameters

<i>handle</i>	a file handle that specifies the output.
<i>buffer</i>	a pointer to the source buffer containing the data to be written.
<i>size</i>	the size of data to be written in bytes.

Returns

the actual size of data written to the output in bytes.

Definition at line 4386 of file [parallel.c](#).

References [VectorClear](#), [VectorPtr](#), [VectorPushBacks](#), and [VectorSize](#).

4.97.3.47 PF_FlushStdOutBuffer()

```
void PF_FlushStdOutBuffer (
    void ) [extern]
```

Explicitly Flushes the buffer for the standard output on the master, which is used if PF_ENABLE_STDOUT_↵
BUFFERING is defined.

Definition at line 4480 of file [parallel.c](#).

References [VectorClear](#), [VectorPtr](#), and [VectorSize](#).

4.97.4 Variable Documentation

4.97.4.1 PF

```
PARALLELVARS PF [extern]
```

Definition at line 90 of file [parallel.c](#).

4.97.4.2 PF_maxDollarChunkSize

```
LONG PF_maxDollarChunkSize [extern]
```

Definition at line 69 of file [mpi.c](#).

4.98 parallel.h

[Go to the documentation of this file.](#)

```

00001 #ifndef __PARALLEL__
00002 #define __PARALLEL__
00003
00009 /* #[ License : */
00010 /*
00011 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00012 * When using this file you are requested to refer to the publication
00013 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014 * This is considered a matter of courtesy as the development was paid
00015 * for by FOM the Dutch physics granting agency and we would like to
00016 * be able to track its scientific use to convince FOM of its value
00017 * for the community.
00018 *
00019 * This file is part of FORM.
00020 *
00021 * FORM is free software: you can redistribute it and/or modify it under the
00022 * terms of the GNU General Public License as published by the Free Software
00023 * Foundation, either version 3 of the License, or (at your option) any later
00024 * version.
00025 *
00026 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00029 * details.
00030 *
00031 * You should have received a copy of the GNU General Public License along
00032 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00033 */
00034 /* #[ License : */
00035
00036 /*
00037 #[ macros & definitions :
00038 */
00039 #define MASTER 0
00040
00041 #define PF_RESET 0
00042 #define PF_TIME 1
00043
00044 #define PF_TERM_MSGTAG 10 /* master -> slave: sending terms */
00045 #define PF_ENDSORT_MSGTAG 11 /* master -> slave: no more terms to be distributed, slave ->
master: after EndSort() */
00046 #define PF_DOLLAR_MSGTAG 12 /* slave -> master: sending $-variables */
00047 #define PF_BUFFER_MSGTAG 20 /* slave -> master: sending sorted terms, or in
PF_SendFile()/PF_RecvFile() */
00048 #define PF_ENDBUFFER_MSGTAG 21 /* same as PF_BUFFER_MSGTAG, but indicates the end of operation */
00049 #define PF_READY_MSGTAG 30 /* slave -> master: slave is idle and can accept terms */
00050 #define PF_DATA_MSGTAG 50 /* InParallel, DoCheckpoint() */
00051 #define PF_EMPTY_MSGTAG 52 /* InParallel, DoCheckpoint(), PF_SendFile(), PF_RecvFile() */
00052 #define PF_STDOUT_MSGTAG 60 /* slave -> master: sending text to the stdout */
00053 #define PF_LOG_MSGTAG 61 /* slave -> master: sending text to the log file */
00054 #define PF_OPT_MCTS_MSGTAG 70 /* master <-> slave: optimization */
00055 #define PF_OPT_HORNER_MSGTAG 71 /* master <-> slave: optimization */
00056 #define PF_OPT_COLLECT_MSGTAG 72 /* slave -> master: optimization */
00057 #define PF_MISC_MSGTAG 100
00058
00059 /*
00060 * A macro for checking the version of gcc.
00061 */
00062 #if defined(__GNUC__) && defined(__GNUC_MINOR__) && defined(__GNUC_PATCHLEVEL__)
00063 # define GNUC_PREREQ(major, minor, patchlevel) \
00064 ((__GNUC__ < 16) + (__GNUC_MINOR__ < 8) + __GNUC_PATCHLEVEL__ >= \
00065 ((major) < 16) + ((minor) < 8) + (patchlevel))
00066 #else
00067 # define GNUC_PREREQ(major, minor, patchlevel) 0
00068 #endif
00069
00070 /*
00071 * The macro "indices" defined in variable.h collides with some function
00072 * argument names in the MPI-3.0 standard.
00073 */
00074 #undef indices
00075
00076 /* Avoid messy padding warnings which may appear in mpi.h. */
00077 #if GNUC_PREREQ(4, 6, 0)
00078 # pragma GCC diagnostic push
00079 # pragma GCC diagnostic ignored "-Wpadded"
00080 # pragma GCC diagnostic ignored "-Wunused-parameter"
00081 #endif
00082 #if defined(__clang__) && defined(__has_warning)
00083 # pragma clang diagnostic push
00084 # if __has_warning("-Wpadded")
00085 # pragma clang diagnostic ignored "-Wpadded"

```

```

00086 # endif
00087 # if __has_warning("-Wunused-parameter")
00088 #   pragma clang diagnostic ignored "-Wunused-parameter"
00089 # endif
00090 #endif
00091
00092 # ifdef __cplusplus
00093     /*
00094      * form3.h (which includes parallel.h) is included from newpoly.h as
00095      * extern "C" {
00096      *   #include "form3.h"
00097      * }
00098      * On the other hand, C++ interfaces to MPI are defined in mpi.h if it is
00099      * included from C++ sources. We first leave from the C-linkage, include
00100      * mpi.h, and then go back to the C-linkage.
00101      * (TU 7 Jun 2011)
00102      */
00103 }
00104 #   define OMPI_SKIP_MPICXX 1
00105 #   include <mpi.h>
00106 extern "C" {
00107 #   else
00108 #   define OMPI_SKIP_MPICXX 1
00109 #   include <mpi.h>
00110 #   endif
00111
00112 /* Now redefine "indices" in the same way as in variable.h. */
00113 #define indices ((INDICES)(AC.IndexList.lijst))
00114
00115 /* Restore the warning settings. */
00116 #if GNUC_PREREQ(4, 6, 0)
00117 #   pragma GCC diagnostic pop
00118 #endif
00119 #if defined(__clang__) && defined(__has_warning)
00120 #   pragma clang diagnostic pop
00121 #endif
00122
00123 #   define PF_ANY_SOURCE MPI_ANY_SOURCE
00124 #   define PF_ANY_MSGTAG MPI_ANY_TAG
00125 #   define PF_COMM MPI_COMM_WORLD
00126 #   define PF_BYTE MPI_BYTE
00127 #   define PF_INT MPI_INT
00128 #if BITSINWORD == 32
00129 #   define PF_WORD MPI_INT32_T
00130 #   define PF_LONG MPI_INT64_T
00131 #elif BITSINWORD == 16
00132 #   define PF_WORD MPI_INT16_T
00133 #   define PF_LONG MPI_INT32_T
00134 #else
00135 #   error Can not detect if this is a 32-bit or 64-bit platform.
00136 #endif
00137
00138 /*
00139      #] macros & definitions :
00140      #[ s/r-bufs :
00141      */
00142
00147 typedef struct {
00148     WORD **buff;
00149     WORD **fill;
00150     WORD **full;
00151     WORD **stop;
00152     MPI_Status *status;
00153     MPI_Status *retstat;
00154     MPI_Request *request;
00155     MPI_Datatype *type; /* this is needed in PF_Wait for Get_count */
00156     int *index; /* dummies for returnvalues */
00157     int *tag; /* for the version with blocking send/receives */
00158     int *from;
00159     int numbufs; /* number of cyclic buffers */
00160     int active; /* flag telling which buffer is active */
00161 } PF_BUFFER;
00162
00163 /*
00164      #] s/r-bufs :
00165      #[ global variables used by the PF_functions : need to be known everywhere
00166      */
00167
00168 typedef struct ParallelVars {
00169     FILEHANDLE slavebuf; /* (slave) allocated if there are RHS expressions */
00170     /* special buffers for nonblocking, unbuffered send/receives */
00171     PF_BUFFER *sbuf; /* set of cyclic send buffers for master_and_ slave */
00172     PF_BUFFER **rbufs; /* array of sets of cyclic receive buffers for master */
00173     int me; /* Internal number of task: master is 0 */
00174     int numtasks; /* total number of tasks */
00175     int parallel; /* flags telling the master and slaves to do the sorting parallel */
00176     /* [05nov2003 mt] This flag must be set to 0 in iniModule! */

```

```

00177     int          rhsInParallel; /* flag for parallel executing even if there are RHS expressions */
00178     int          mkSlaveInfile; /* flag tells that slavebuf is used on the slaves */
00179     int          exprbufsize; /* buffer size in WORDs to be used for transferring expressions */
00180     int          exprtodo; /* >= 0: the expression to do in InParallel, -1: otherwise */
00181     int          log; /* flag for logging mode */
00182     WORD         numsbufs; /* number of cyclic send buffers (PF.sbuf->numsbufs) */
00183     WORD         numrbufs; /* number of cyclic receive buffers (PF.rbufs[i]->numsbufs,
                                i=1,...,numtasks-1) */
00184 } PARALLELVARS;
00185
00186 extern PARALLELVARS PF;
00187 /*[04oct2005 mt]:*/
00188 /*for broadcasting dollarvars, see parallel.c:PF_BroadcastPreDollar():*/
00189 extern LONG PF_maxDollarChunkSize;
00190 /*:[04oct2005 mt]*/
00191
00192 /*
00193  #] global variables used by the PF_functions :
00194  #[ Function prototypes :
00195  */
00196
00197 /* mpi.c */
00198 extern int PF_ISendSbuf(int,int);
00199 extern int PF_Bcast(void *buffer, int count);
00200 extern int PF_RawSend(int,void *,LONG,int);
00201 extern LONG PF_RawRecv(int *,void *,LONG,int *);
00202
00203 extern int PF_PreparePack(void);
00204 extern int PF_Pack(const void *buffer, size_t count, MPI_Datatype type);
00205 extern int PF_Unpack(void *buffer, size_t count, MPI_Datatype type);
00206 extern int PF_PackString(const UBYTE *str);
00207 extern int PF_UnpackString(UBYTE *str);
00208 extern int PF_Send(int to, int tag);
00209 extern int PF_Receive(int src, int tag, int *psrc, int *ptag);
00210 extern int PF_Broadcast(void);
00211
00212 extern int PF_PrepareLongSinglePack(void);
00213 extern int PF_LongSinglePack(const void *buffer, size_t count, MPI_Datatype type);
00214 extern int PF_LongSingleUnpack(void *buffer, size_t count, MPI_Datatype type);
00215 extern int PF_LongSingleSend(int to, int tag);
00216 extern int PF_LongSingleReceive(int src, int tag, int *psrc, int *ptag);
00217
00218 extern int PF_PrepareLongMultiPack(void);
00219 extern int PF_LongMultiPackImpl(const void *buffer, size_t count, size_t eSize, MPI_Datatype type);
00220 extern int PF_LongMultiUnpackImpl(void *buffer, size_t count, size_t eSize, MPI_Datatype type);
00221 extern int PF_LongMultiBroadcast(void);
00222
00223 static inline size_t sizeof_datatype(MPI_Datatype type)
00224 {
00225     if ( type == PF_BYTE ) return sizeof(char);
00226     if ( type == PF_INT ) return sizeof(int);
00227     if ( type == PF_WORD ) return sizeof(WORD);
00228     if ( type == PF_LONG ) return sizeof(LONG);
00229     return(0);
00230 }
00231
00232 #define PF_LongMultiPack(buffer, count, type) PF_LongMultiPackImpl(buffer, count,
                                sizeof_datatype(type), type)
00233 #define PF_LongMultiUnpack(buffer, count, type) PF_LongMultiUnpackImpl(buffer, count,
                                sizeof_datatype(type), type)
00234
00235 /* parallel.c */
00236 extern int PF_EndSort(void);
00237 extern WORD PF_Deferred(WORD *,WORD);
00238 extern int PF_Processor(EXPRESSIONS,WORD,WORD);
00239 extern int PF_Init(int*,char ***);
00240 extern int PF_Terminate(int);
00241 extern LONG PF_GetSlaveTimes(void);
00242 extern LONG PF_BroadcastNumber(LONG);
00243 extern void PF_BroadcastBuffer(WORD **buffer, LONG *length);
00244 extern int PF_BroadcastString(UBYTE *);
00245 extern int PF_BroadcastPreDollar(WORD **, LONG *,int *);
00246 extern int PF_CollectModifiedDollars(void);
00247 extern int PF_BroadcastModifiedDollars(void);
00248 extern int PF_BroadcastRedefinedPreVars(void);
00249 extern int PF_BroadcastCBuf(int bufnum);
00250 extern int PF_BroadcastExpFlags(void);
00251 extern int PF_StoreInsideInfo(void);
00252 extern int PF_RestoreInsideInfo(void);
00253 extern int PF_BroadcastExpr(EXPRESSIONS e, FILEHANDLE *file);
00254 extern int PF_BroadcastRHS(void);
00255 extern int PF_InParallelProcessor(void);
00256 extern int PF_SendFile(int to, FILE *fd);
00257 extern int PF_RecvFile(int from, FILE *fd);
00258 extern void PF_MLock(void);
00259 extern void PF_MUnLock(void);
00260 extern LONG PF_WriteFileToFile(int,UBYTE *,LONG);

```

```

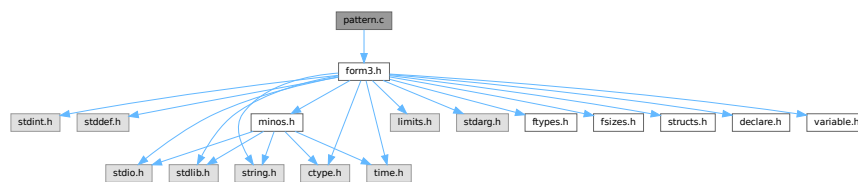
00261 extern void    PF_FlushStdOutBuffer(void);
00262
00263 /*
00264    #] Function prototypes :
00265 */
00266
00267 #endif

```

4.99 pattern.c File Reference

```
#include "form3.h"
```

Include dependency graph for pattern.c:



Macros

- `#define` [PutInBuffers](#)(pow)

Functions

- `int` [TestMatch](#) (PHEAD WORD *term, WORD *level)
- `void` [Substitute](#) (PHEAD WORD *term, WORD *pattern, WORD power)
- `int` [FindAll](#) (PHEAD WORD *term, WORD *pattern, WORD level, WORD *par)
- `int` [TestSelect](#) (WORD *term, WORD *setp)
- `void` [SubsInAll](#) (PHEAD0)
- `void` [TransferBuffer](#) (int from, int to, int spectator)
- `int` [TakeIDfunction](#) (PHEAD WORD *term)

4.99.1 Detailed Description

Top level pattern matching routines. More pattern matching is found in [findpat.c](#), [function.c](#), [symmetr.c](#) and [smart.c](#). The last three files contain the matching inside functions. The file [pattern.c](#) contains also the very important routine `Substitute`. All regular pattern matching is just the finding of the pattern and indicating what are the wildcards etc. The routine `Substitute` does the actual removal of the pattern and replaces it by a subterm of the type SUBEXPRESSION.

Definition in file [pattern.c](#).

4.99.2 Macro Definition Documentation

4.99.2.1 PutInBuffers

```
#define PutInBuffers(
    pow )
```

Value:

```
AddRHS(AT.ebufnum,1); \
*out++ = SUBEXPRESSION; \
*out++ = SUBEXPSIZE; \
*out++ = C->numrhs; \
*out++ = pow; \
*out++ = AT.ebufnum; \
FILLSUB(out) \
r = AT.pWorkSpace[rhs+i]; \
if ( *r > 0 ) { \
    oldinr = r[*r]; r[*r] = 0; \
    AddNtoC(AT.ebufnum, (*r+1-ARGHEAD), (r+ARGHEAD), 14); \
    r[*r] = oldinr; \
} \
else { \
    ToGeneral(r,buffer,1); \
    buffer[buffer[0]] = 0; \
    AddNtoC(AT.ebufnum,buffer[0]+1,buffer,15); \
}
```

Definition at line 2193 of file [pattern.c](#).

4.99.3 Function Documentation

4.99.3.1 TestMatch()

```
int TestMatch (
    PHEAD WORD * term,
    WORD * level )
```

This routine governs the pattern matching. If it decides that a substitution should be made, this can be either the insertion of a right hand side (C->rhs) or the automatic generation of terms as a result of an operation (like trace). The object to be replaced is removed from term and a subexpression pointer is inserted. If the substitution is made more than once there can be more subexpression pointers. Its number is positive as it corresponds to the level at which the C->rhs can be found in the compiler output. The subexpression pointer contains the wildcard substitution information. The power is found in *AT.TMout. For operations the subexpression pointer is negative and corresponds to an address in the array AT.TMout. In this array are then the instructions for the routine to be called and its number in the array 'Operations' The format is here: length,functionnumber,length-2 parameters

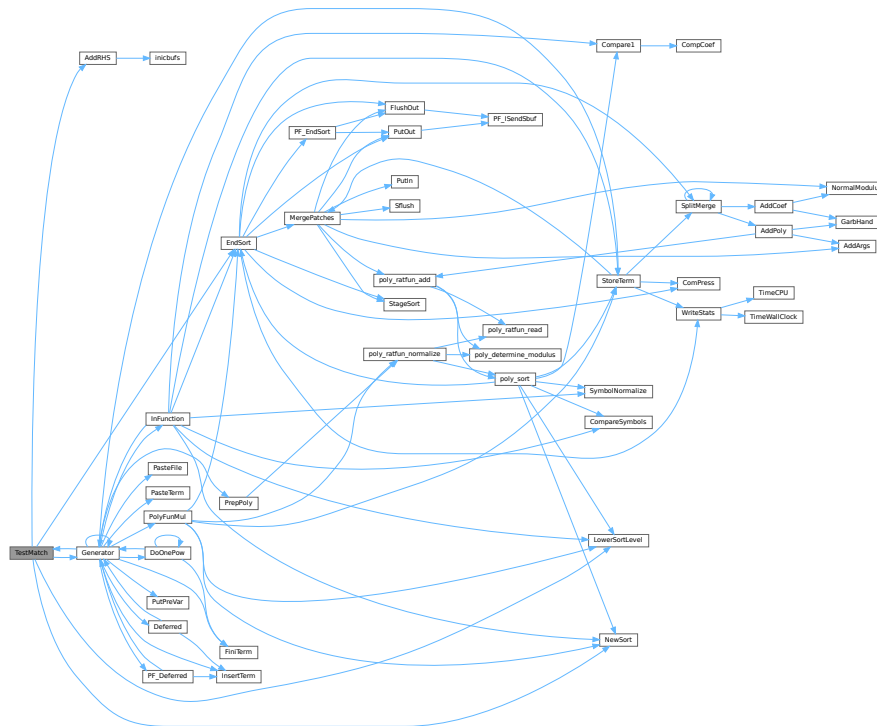
There is a certain complexity wrt repeat levels. Another complication is the poking of the wildcard values in the subexpression prototype in the compiler buffer. This was how things were done in the past with sequential FORM, but with the advent of TFORM this cannot be maintained. Now, for TFORM we make a copy of it. 7-may-2008 (JV): We cannot yet guarantee that this has been done 100% correctly. There are errors that occur in TFORM only and that may indicate problems.

Definition at line 97 of file [pattern.c](#).

References [AddRHS\(\)](#), [EndSort\(\)](#), [Generator\(\)](#), [CbUf::lhs](#), [LowerSortLevel\(\)](#), and [NewSort\(\)](#).

Referenced by [Generator\(\)](#).

Here is the call graph for this function:



4.99.3.2 Substitute()

```
void Substitute (
    PHEAD WORD * term,
    WORD * pattern,
    WORD power )
```

Definition at line 641 of file [pattern.c](#).

4.99.3.3 FindAll()

```
int FindAll (
    PHEAD WORD * term,
    WORD * pattern,
    WORD level,
    WORD * par )
```

Definition at line 1197 of file [pattern.c](#).

4.99.3.4 TestSelect()

```
int TestSelect (
    WORD * term,
    WORD * setp )
```

Definition at line 1748 of file [pattern.c](#).

4.99.3.5 SubsInAll()

```
void SubsInAll (
    PHEAD0 )
```

Definition at line 1982 of file [pattern.c](#).

4.99.3.6 TransferBuffer()

```
void TransferBuffer (
    int from,
    int to,
    int spectator )
```

Definition at line 2143 of file [pattern.c](#).

4.99.3.7 TakeIDfunction()

```
int TakeIDfunction (
    PHEAD WORD * term )
```

Definition at line 2213 of file [pattern.c](#).

4.100 pattern.c

[Go to the documentation of this file.](#)

```
00001
00012 /* #[] License : */
00013 /*
00014 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00015 * When using this file you are requested to refer to the publication
00016 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00017 * This is considered a matter of courtesy as the development was paid
00018 * for by FOM the Dutch physics granting agency and we would like to
00019 * be able to track its scientific use to convince FOM of its value
00020 * for the community.
00021 *
00022 * This file is part of FORM.
00023 *
00024 * FORM is free software: you can redistribute it and/or modify it under the
00025 * terms of the GNU General Public License as published by the Free Software
00026 * Foundation, either version 3 of the License, or (at your option) any later
00027 * version.
00028 *
00029 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00030 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00031 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00032 * details.
00033 *
00034 * You should have received a copy of the GNU General Public License along
00035 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00036 */
00037 /* #[] License : */
00038 /*
00039 !!! Notice the change in OnePV in FindAll (7-may-2008 JV).
00040
00041 #[] Includes : pattern.c
00042 */
00043
00044 #include "form3.h"
00045
00046 /*
00047 #[] Includes :
00048 #[] Patterns :
00049 #[] Rules :
```



```

00050
00051     There are several rules governing the allowable replacements.
00052     1: Multi with anything but symbols or dotproducts reverts
00053         to many.
00054     2: Each symbol can have only one (wildcard) power, so
00055         x^2*x^n? is illegal.
00056     3: when a single vector is used it replaces all occurrences
00057         of the vector. Therefore q*q(mu) or q*q(mu) cannot occur.
00058         Also q*q cannot be done.
00059     4: Loose vector elements are replaced with p(mu), dotproducts
00060         with p?.q.
00061     5: p?.q? is allowed.
00062     6: x^n? can revert to n = 0 if there is no power of x.
00063     7: x?^n? must match some x. There could be an ambiguity otherwise.
00064
00065     #] Rules :
00066     #[ TestMatch :          WORD TestMatch(term,level)
00067 */
00068
00097 int TestMatch(PHEAD WORD *term, WORD *level)
00098 {
00099     GETBIDENTITY
00100     WORD *ll, *m, *w, *llf, *OldWork, *StartWork, *ww, *mm, *t, *OldTermBuffer = 0;
00101     WORD power = 0, i, msign = 0, ll2;
00102     int match = 0;
00103     int numdollars = 0, protosize, oldallnumrhs;
00104     CBUF *C = cbuf+AM.rbufnum, *CC;
00105     AT.idallflag = 0;
00106     do {
00107 /*
00108         #[ Preliminaries :
00109 */
00110         ll = C->lhs[*level];
00111         if ( *ll == TYPEEXPRESSION ) {
00112 /*
00113             Expressions are not subject to anything.
00114 */
00115             return(0);
00116         }
00117         else if ( *ll == TYPEREPEAT ) {
00118             ***AN.RepPoint = 0;
00119             return(0); /* Will force the next level */
00120         }
00121         else if ( *ll == TYPEENDREPEAT ) {
00122             if ( *AN.RepPoint ) {
00123                 AN.RepPoint[-1] = 1; /* Mark the higher level as dirty */
00124                 *AN.RepPoint = 0;
00125                 *level = ll[2]; /* Level to jump back to */
00126             }
00127             else {
00128                 AN.RepPoint--;
00129                 if ( AN.RepPoint < AT.RepCount ) {
00130                     MLOCK(ErrorMessageLock);
00131                     MesPrint("Internal problems with REPEAT count");
00132                     MUNLOCK(ErrorMessageLock);
00133                     Terminate(-1);
00134                 }
00135             }
00136             return(0); /* Force the next level */
00137         }
00138         else if ( *ll == TYPEOPERATION ) {
00139 /*
00140             Operations have always their own level.
00141 */
00142             if ( (*(FG.OperaFind[ll[2]])) (BHEAD term,ll) ) return(-1);
00143             else return(0);
00144         }
00145 /*
00146         #] Preliminaries :
00147 */
00148         OldWork = AT.WorkPointer;
00149         if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
00150         ww = AT.WorkPointer;
00151 /*
00152         Here we need to make a copy of the subexpression object because we
00153         will be writing the values of the wildcards in it.
00154         Originally we copied it into the private version of the compiler buffer
00155         that is used for scratch space (ebufnum). This caused errors in the
00156         routines like ScanFunctions when the ebufnum Buffer was expanded
00157         and inpat was still pointing at the old Buffer. This expansion
00158         could be done in AddWild and hence cannot be fixed at > 100 places.
00159         The solution is to use AN.patternbuffer (JV 16-mar-2009).
00160 */
00161         {
00162             WORD *ta = ll, *ma;
00163             int ja = ta[1];
00164 /*

```

```

00165         New code (16-mar-2009) JV
00166     */
00167     if ( ( ja + 2 ) > AN.patternbuffersize ) {
00168         if ( AN.patternbuffer ) M_free(AN.patternbuffer,"AN.patternbuffer");
00169         AN.patternbuffersize = 2 * ja + 2;
00170         AN.patternbuffer = (WORD *)Malloc(AN.patternbuffersize * sizeof(WORD),
00171             "AN.patternbuffer");
00172     }
00173     ma = AN.patternbuffer;
00174     m = ma + IDHEAD;
00175     NCOPY(ma,ta,ja);
00176     *ma = 0;
00177 }
00178 AN.FullProto = m;
00179 AN.WildValue = w = m + SUBEXPSIZE;
00180 protosize = IDHEAD + m[1];
00181 m += m[1];
00182 AN.WildStop = m;
00183 StartWork = ww;
00184 ll2 = ll[2];
00185 /*
00186     #! Expand dollars :
00187 */
00188 if ( ( ll[4] & DOLLARFLAG ) != 0 ) { /* We have at least one dollar in the pattern */
00189     WORD oldRepPoint = *AN.RepPoint, olddefer = AR.DeferFlag;
00190     AR.Eside = LHSIDEX;
00191 /*
00192     Copy into WorkSpace. This means that AN.patternbuffer will be free.
00193 */
00194     ww = AT.WorkPointer; i = m[0]; mm = m;
00195     NCOPY(ww,mm,i);
00196     *StartWork += 3;
00197     *ww++ = 1; *ww++ = 1; *ww++ = 3;
00198     AT.WorkPointer = ww;
00199     AR.DeferFlag = 0;
00200     NewSort(BHEAD0);
00201     if ( Generator(BHEAD StartWork,AR.Cnumlhs) ) {
00202         LowerSortLevel();
00203         AT.WorkPointer = OldWork;
00204         AR.DeferFlag = olddefer;
00205         return(-1);
00206     }
00207     AT.WorkPointer = ww;
00208     if ( EndSort(BHEAD ww,0) < 0 ) {}
00209     AR.DeferFlag = olddefer;
00210     if ( *ww == 0 || *(ww+ww) != 0 ) {
00211         if ( AP.lhdollarerror == 0 ) {
00212 /*
00213             If race condition we just get more error messages
00214 */
00215             MLOCK(ErrorMessageLock);
00216             MesPrint("&LHS must be one term");
00217             MUNLOCK(ErrorMessageLock);
00218             AP.lhdollarerror = 1;
00219         }
00220         AT.WorkPointer = OldWork;
00221         return(-1);
00222     }
00223     m = ww; ww = m + *m;
00224     if ( m[*m-1] < 0 ) { msign = 1; m[*m-1] = -m[*m-1]; }
00225     if ( *ww || m[*m-1] != 3 || m[*m-2] != 1 || m[*m-3] != 1 ) {
00226         MLOCK(ErrorMessageLock);
00227         MesPrint("Dollar variable develops into an illegal pattern in id-statement");
00228         MUNLOCK(ErrorMessageLock);
00229         return(-1);
00230     }
00231     *m -= m[*m-1];
00232     if ( ( *m + 1 + protosize ) > AN.patternbuffersize ) {
00233         if ( AN.patternbuffer ) M_free(AN.patternbuffer,"AN.patternbuffer");
00234         AN.patternbuffersize = 2 * (*m) + 2 + protosize;
00235         AN.patternbuffer = (WORD *)Malloc(AN.patternbuffersize * sizeof(WORD),
00236             "AN.patternbuffer");
00237         mm = ll; ww = AN.patternbuffer; i = protosize;
00238         NCOPY(ww,mm,i);
00239         AN.FullProto = AN.patternbuffer + IDHEAD;
00240         AN.WildValue = w = AN.FullProto + SUBEXPSIZE;
00241         AN.WildStop = AN.patternbuffer + protosize;
00242     }
00243     mm = AN.patternbuffer + protosize;
00244     i = *m;
00245     NCOPY(mm,m,i);
00246     m = AN.patternbuffer + protosize;
00247     AR.Eside = RHSIDE;
00248     *mm = 0;
00249 /*
00250     Test the pattern. If only wildcard powers -> SUBONCE
00251 */

```

```

00252     {
00253         WORD *mmm = m + *m, *m1 = m+1, jm, noveto = 0;
00254         while ( m1 < mmm ) {
00255             if ( *m1 == SYMBOL ) {
00256                 for ( jm = 2; jm < m1[1]; jm+=2 ) {
00257                     if ( m1[jm+1] < MAXPOWER && m1[jm+1] > -MAXPOWER ) break;
00258                 }
00259                 if ( jm < m1[1] ) { noveto = 1; break; }
00260             }
00261             else if ( *m1 == DOTPRODUCT ) {
00262                 for ( jm = 2; jm < m1[1]; jm+=3 ) {
00263                     if ( m1[jm+2] < MAXPOWER && m1[jm+2] > -MAXPOWER ) break;
00264                 }
00265                 if ( jm < m1[1] ) { noveto = 1; break; }
00266             }
00267             else { noveto = 1; break; }
00268             m1 += m1[1];
00269         }
00270         if ( noveto == 0 ) {
00271             ll2 = ll2 & ~SUBMASK;
00272             ll2 |= SUBONCE;
00273         }
00274     }
00275     AT.WorkPointer = ww = StartWork;
00276     *AN.RepPoint = oldRepPoint;
00277 }
00278 /*
00279     #] Expand dollars :
00280
00281     In case of id,all we have to check at this point that there are only
00282     functions in the pattern.
00283 */
00284 if ( ( ll2 & SUBMASK ) == SUBALL ) {
00285     WORD *t = AN.patternbuffer+IDHEAD, *tt;
00286     WORD *tstop, *ttstop, ii;
00287     t += t[1]; tstop = t + *t; t++;
00288     while ( t < tstop ) {
00289         if ( *t < FUNCTION ) break;
00290         t += t[1];
00291     }
00292     if ( t < tstop ) {
00293         MLOCK(ErrorMessageLock);
00294         MesPrint("Error: id,all can only be used with (products of) functions and/or tensors.");
00295         MUNLOCK(ErrorMessageLock);
00296         return(-1);
00297     }
00298     OldTermBuffer = AN.termbuffer;
00299     AN.termbuffer = TermMalloc("id,all");
00300 /*
00301     Now make sure that only regular functions and tensors can take part.
00302 */
00303     tt = term; ttstop = tt+*tt; ttstop -= ABS(ttstop[-1]); tt++;
00304     t = AN.termbuffer+1;
00305     while ( tt < ttstop ) {
00306         if ( *tt >= FUNCTION && *tt != AR.PolyFun && *tt != AR.PolyFunInv ) {
00307             ii = tt[1]; NCOPY(t,tt,ii);
00308         }
00309         else tt += tt[1];
00310     }
00311     *t++ = 1; *t++ = 1; *t++ = 3; AN.termbuffer[0] = t-AN.termbuffer;
00312 }
00313 /*
00314     To be puristic, we need to check that all wildcards in the prototype
00315     are actually present. If the LHS contained a replace_ this may not be
00316     the case.
00317 */
00318 ClearWild(BHEAD0);
00319 while ( w < AN.WildStop ) {
00320     if ( *w == LOADDOLLAR ) numdollars++;
00321     w += w[1];
00322 }
00323 AN.RepFunNum = 0;
00324 /* rep = */ AN.RepFunList = AT.WorkPointer;
00325 AT.WorkPointer = (WORD *) ((UBYTE *) (AT.WorkPointer)) + AM.MaxTer/2;
00326 if ( AT.WorkPointer >= AT.WorkTop ) {
00327     MLOCK(ErrorMessageLock);
00328     MesWork();
00329     MUNLOCK(ErrorMessageLock);
00330     return(-1);
00331 }
00332 AN.DisOrderFlag = ll2 & SUBDISORDER;
00333 AN.nogroundlevel = 0;
00334 switch ( ll2 & SUBMASK ) {
00335     case SUBONLY :
00336         /* Must be an exact match */
00337         AN.UseFindOnly = 1; AN.ForFindOnly = 0;
00338         if ( FindRest(BHEAD term,m) && ( AN.UsedOtherFind ||

```

```

00339         FindOnly(BHEAD term,m) ) ) {
00340             power = 1;
00341             if ( msign ) term[term[0]-1] = -term[term[0]-1];
00342         }
00343         else power = 0;
00344         break;
00345     case SUBMANY :
00346         AN.UseFindOnly = -1;
00347         if ( ( power = FindRest(BHEAD term,m) ) > 0 ) {
00348             if ( ( power = FindOnce(BHEAD term,m) ) > 0 ) {
00349                 AN.UseFindOnly = 0;
00350                 do {
00351                     if ( msign ) term[term[0]-1] = -term[term[0]-1];
00352                     Substitute(BHEAD term,m,1);
00353                     if ( numdollars ) {
00354                         WildDollars(BHEAD (WORD *)0);
00355                         numdollars = 0;
00356                     }
00357                     if ( ww < term+term[0] ) ww = term+term[0];
00358                     ClearWild(BHEAD0);
00359                     AT.WorkPointer = ww;
00360                     /* if ( rep < ww ) { */
00361                         AN.RepFunNum = 0;
00362                         /* rep = */ AN.RepFunList = ww;
00363                         AT.WorkPointer = (WORD *)(((UBYTE *) (AT.WorkPointer)) + AM.MaxTer/2);
00364                         if ( AT.WorkPointer >= AT.WorkTop ) {
00365                             MLOCK(ErrorMessageLock);
00366                             MesWork();
00367                             MUNLOCK(ErrorMessageLock);
00368                             return(-1);
00369                         }
00370                     /* }
00371                 }
00372                 else {
00373                     AN.RepFunList = rep;
00374                     AN.RepFunNum = 0;
00375                 }
00376             */
00377                 AN.nogroundlevel = 0;
00378             } while ( FindRest(BHEAD term,m) && ( AN.UsedOtherFind ||
00379                 FindOnce(BHEAD term,m) ) );
00380             match = 1;
00381         }
00382         else if ( power < 0 ) {
00383             do {
00384                 if ( msign ) term[term[0]-1] = -term[term[0]-1];
00385                 Substitute(BHEAD term,m,1);
00386                 if ( numdollars ) {
00387                     WildDollars(BHEAD (WORD *)0);
00388                     numdollars = 0;
00389                 }
00390                 if ( ww < term+term[0] ) ww = term+term[0];
00391                 ClearWild(BHEAD0);
00392                 AT.WorkPointer = ww;
00393                 /* if ( rep < ww ) { */
00394                     AN.RepFunNum = 0;
00395                     /* rep = */ AN.RepFunList = ww;
00396                     AT.WorkPointer = (WORD *)(((UBYTE *) (AT.WorkPointer)) + AM.MaxTer/2);
00397                     if ( AT.WorkPointer >= AT.WorkTop ) {
00398                         MLOCK(ErrorMessageLock);
00399                         MesWork();
00400                         MUNLOCK(ErrorMessageLock);
00401                         return(-1);
00402                     }
00403                 /* }
00404             }
00405             else {
00406                 AN.RepFunList = rep;
00407                 AN.RepFunNum = 0;
00408             }
00409         */
00410             } while ( FindRest(BHEAD term,m) );
00411             match = 1;
00412         }
00413     }
00414     else if ( power < 0 ) {
00415         if ( FindOnce(BHEAD term,m) ) {
00416             do {
00417                 if ( msign ) term[term[0]-1] = -term[term[0]-1];
00418                 Substitute(BHEAD term,m,1);
00419                 if ( numdollars ) {
00420                     WildDollars(BHEAD (WORD *)0);
00421                     numdollars = 0;
00422                 }
00423                 if ( ww < term+term[0] ) ww = term+term[0];
00424                 ClearWild(BHEAD0);
00425                 AT.WorkPointer = ww;

```

```

00426 /*                      if ( rep < ww ) { */
00427                          AN.RepFunNum = 0;
00428                          /* rep = */ AN.RepFunList = ww;
00429                          AT.WorkPointer = (WORD *) ((UBYTE *) (AT.WorkPointer)) + AM.MaxTer/2;
00430                          if ( AT.WorkPointer >= AT.WorkTop ) {
00431                              MLOCK(ErrorMessageLock);
00432                              MesWork();
00433                              MUNLOCK(ErrorMessageLock);
00434                              return(-1);
00435                          }
00436 /*                      }
00437                      }
00438                      else {
00439                          AN.RepFunList = rep;
00440                          AN.RepFunNum = 0;
00441                      }
00442 */
00443                      } while ( FindOnce(BHEAD term,m) );
00444                      match = 1;
00445                      }
00446                  }
00447                  if ( match ) {
00448                      if ( ( ll2 & SUBAFTER ) != 0 ) *level = AC.Labels[ll[3]];
00449                  }
00450                  else {
00451                      if ( ( ll2 & SUBAFTERNOT ) != 0 ) *level = AC.Labels[ll[3]];
00452                  }
00453                  goto nextlevel;
00454              case SUBONCE :
00455                  AN.UseFindOnly = 0;
00456                  if ( FindRest(BHEAD term,m) && ( AN.UsedOtherFind || FindOnce(BHEAD term,m) ) ) {
00457                      power = 1;
00458                      if ( msign ) term[term[0]-1] = -term[term[0]-1];
00459                  }
00460                  else power = 0;
00461                  break;
00462              case SUBMULTI :
00463                  power = FindMulti(BHEAD term,m);
00464                  if ( ( power & 1 ) != 0 && msign ) term[term[0]-1] = -term[term[0]-1];
00465                  break;
00466              case SUBVECTOR :
00467                  while ( ( power = FindAll(BHEAD term,m,*level,(WORD *)0) ) != 0 ) {
00468                      if ( ( power & 1 ) != 0 && msign ) term[term[0]-1] = -term[term[0]-1];
00469                      match = 1;
00470                  }
00471                  break;
00472              case SUBSELECT :
00473                  llf = ll + IDHEAD; llf += llf[1]; llf += *llf;
00474                  AN.UseFindOnly = 1; AN.ForFindOnly = llf;
00475                  if ( FindRest(BHEAD term,m) && ( AN.UsedOtherFind || FindOnly(BHEAD term,m) ) ) {
00476                      if ( msign ) term[term[0]-1] = -term[term[0]-1];
00477 /*
00478                      The following code needs to be hacked a bit to allow for
00479                      all types of sets and for occurrence anywhere in the term
00480                      The code at the end of FindOnly is a bit mysterious.
00481 */
00482                      if ( llf[1] > 2 ) {
00483                          WORD *t1, *t2;
00484                          if ( *term > AN.sizeselecttermundo ) {
00485                              if ( AN.selecttermundo ) M_free(AN.selecttermundo, "AN.selecttermundo");
00486                              AN.sizeselecttermundo = *term + 10;
00487                              AN.selecttermundo = (WORD *) Malloc1(
00488                                  AN.sizeselecttermundo*sizeof(WORD), "AN.selecttermundo");
00489                          }
00490                          t1 = term; t2 = AN.selecttermundo; i = *term;
00491                          NCOPY(t2,t1,i);
00492                      }
00493                      power = 1;
00494                      Substitute(BHEAD term,m,power);
00495                      if ( llf[1] > 2 ) {
00496                          if ( TestSelect(term,llf) ) {
00497                              WORD *t1, *t2;
00498                              power = 0;
00499                              t1 = term; t2 = AN.selecttermundo; i = *t2;
00500                              NCOPY(t1,t2,i);
00501 #if IDHEAD > 3
00502                              if ( ( ll2 & SUBAFTERNOT ) != 0 ) {
00503                                  *level = AC.Labels[ll[3]];
00504                              }
00505 #endif
00506                              goto nextlevel;
00507                          }
00508                      }
00509                      if ( numdollars ) {
00510                          WildDollars(BHEAD (WORD *)0);
00511                          numdollars = 0;
00512                      }

```

```

00513         match = 1;
00514         if ( ( ll2 & SUBAFTER ) != 0 ) {
00515             *level = AC.Labels[ll[3]];
00516         }
00517     }
00518     else {
00519         if ( ( ll2 & SUBAFTERNOT ) != 0 ) {
00520             *level = AC.Labels[ll[3]];
00521         }
00522         power = 0;
00523     }
00524     goto nextlevel;
00525 case SUBALL:
00526     AN.UseFindOnly = 0;
00527     CC = cbuf+AT.allbufnum;
00528     oldallnumrhs = CC->numrhs;
00529     t = AddRHS(AT.allbufnum,1);
00530     *t = 0;
00531     AT.idallflag = 1;
00532     AT.idallmaxnum = ll[5];
00533     AT.idallnum = 0;
00534     if ( FindRest(BHEAD AN.termbuffer,m) || AT.idallflag > 1 ) {
00535         WORD *t, *tstop, *tt, first = 1, ii;
00536         power = 1;
00537         *CC->Pointer++ = 0;
00538         if ( msign ) term[term[0]-1] = -term[term[0]-1];
00539     /*
00540
00541         If we come here the matches are all already in the
00542         compiler buffer. All we need to do is take out all
00543         functions and replace them by a SUBEXPRESSION that
00544         points to this buffer.
00545         Note: the PolyFun/PolyRatFun should be excluded from this.
00546         This works because each match writes incrementally to
00547         the buffer using the routine SubsInAll.
00548
00549         The call to WildDollars should be made in Generator.....
00550     */
00551     t = term; tstop = t + *t; ii = ABS(tstop[-1]); tstop -= ii;
00552     tt = AT.WorkPointer+1;
00553     t++;
00554     while ( t < tstop ) {
00555         if ( *t >= FUNCTION && *t != AR.PolyFun && *t != AR.PolyFunInv ) {
00556             if ( first ) { /* SUBEXPRESSION */
00557                 *tt++ = SUBEXPRESSION;
00558                 *tt++ = SUBEXPSIZE;
00559                 *tt++ = CC->numrhs;
00560                 *tt++ = 1;
00561                 *tt++ = AT.allbufnum;
00562                 FILLSUB(tt)
00563                 first = 0;
00564             }
00565             t += t[1];
00566         }
00567         else {
00568             i = t[1]; NCOPY(tt,t,i);
00569         }
00570     }
00571     if ( ( ll[4] & NORMALIZEFLAG ) != 0 ) {
00572     /*
00573
00574         In case of the normalization option, we have to divide
00575         by AT.idallnum;
00576
00577         WORD na = t[ii-1];
00578         na = REDLENG(na);
00579         for ( i = 0; i < ii; i++ ) tt[i] = t[i];
00580         Divvy(BHEAD (UWORD *)tt,&na,(UWORD *)&(AT.idallnum),1);
00581         na = INCLENG(na);
00582         ii = ABS(na);
00583         tt[ii-1] = na;
00584         tt += ii;
00585     */
00586     }
00587     else {
00588         NCOPY(tt,t,ii);
00589     }
00590     ii = tt-AT.WorkPointer;
00591     *(AT.WorkPointer) = ii;
00592     tt = AT.WorkPointer; t = term;
00593     NCOPY(t,tt,ii);
00594
00595     if ( ( ll2 & SUBAFTER ) != 0 ) { /* ifmatch -> */
00596         *level = AC.Labels[ll[3]];
00597     }
00598     TermFree(AN.termbuffer,"id,all");
00599     AN.termbuffer = OldTermBuffer;
00600     AT.WorkPointer = AN.RepFunList;
00601     AT.idallflag = 0;
00602     CC->Pointer[0] = 0;

```

```

00600         TransferBuffer(AT.aebufnum,AT.ebufnum,AT.allbufnum);
00601         return(1);
00602     }
00603     AT.idallflag = 0;
00604     power = 0;
00605     CC->numrhs = oldallnumrhs;
00606     TermFree(AN.termbuffer,"id,all");
00607     AN.termbuffer = OldTermBuffer;
00608     break;
00609     default :
00610         break;
00611 }
00612 if ( power ) {
00613     Substitute(BHEAD term,m,power);
00614     if ( numdollars ) {
00615         WildDollars(BHEAD (WORD *)0);
00616         numdollars = 0;
00617     }
00618     match = 1;
00619     if ( ( ll2 & SUBAFTER ) != 0 ) { /* ifmatch -> */
00620         *level = AC.Labels[ll[3]];
00621     }
00622 }
00623 else {
00624     AT.WorkPointer = AN.RepFunList;
00625     if ( ( ll2 & SUBAFTERNOT ) != 0 ) { /* ifnomatch -> */
00626         *level = AC.Labels[ll[3]];
00627     }
00628 }
00629 nextlevel:;
00630 } while ( (*level)++ < AR.Cnumlhs && C->lhs[*level][0] == TYPEIDOLD );
00631 (*level)--;
00632 AT.WorkPointer = AN.RepFunList;
00633 return(match);
00634 }
00635
00636 /*
00637     #] TestMatch :
00638     #[ Substitute :          void Substitute(term,pattern,power)
00639 */
00640
00641 void Substitute(PHEAD WORD *term, WORD *pattern, WORD power)
00642 {
00643     GETBIDENTITY
00644     WORD *TemTerm;
00645     WORD *t, *m;
00646     WORD *tstop, *mstop;
00647     WORD *xstop, *ystop;
00648     WORD nt, *fill, nq, mt;
00649     WORD *q, *subterm, *tcoef, oldval1 = 0, newval3, i = 0;
00650     WORD PutExpr = 0, sign = 0;
00651     TemTerm = AT.WorkPointer;
00652     if ( ( (WORD *)(((UBYTE *) (AT.WorkPointer)) + AM.MaxTer*2) ) > AT.WorkTop ) {
00653         MLOCK(ErrorMessageLock);
00654         MesWork();
00655         MUNLOCK(ErrorMessageLock);
00656         Terminate(-1);
00657     }
00658     m = pattern;
00659     mstop = m + *m;
00660     m++;
00661     t = term;
00662     t += *term - 1;
00663     tcoef = t;
00664     tstop = t - ABS(*t) + 1;
00665     t = term;
00666     t++;
00667     fill = TemTerm;
00668     fill++;
00669     if ( m < mstop ) { do {
00670 /*
00671         #[ SYMBOLS :
00672 */
00673         if ( *m == SYMBOL ) {
00674             ystop = m + m[1];
00675             m += 2;
00676             while ( *t != SYMBOL && t < tstop ) {
00677                 nq = t[1];
00678                 NCOPY(fill,t,nq);
00679             }
00680             if ( t >= tstop ) goto SubCoeef;
00681             *fill++ = SYMBOL;
00682             fill++;
00683             subterm = fill;
00684             xstop = t + t[1];
00685             t += 2;
00686             do {

```

```

00687         if ( *m == *t && t < xstop ) {
00688             nt = t[1];
00689             mt = m[1];
00690             if ( mt >= 2*MAXPOWER ) {
00691                 if ( CheckWild(BHEAD mt-2*MAXPOWER, SYMTONUM, -MAXPOWER, &newval3) ) {
00692                     nt -= AN.oldvalue;
00693                     goto SubsL1;
00694                 }
00695             }
00696             else if ( mt <= -2*MAXPOWER ) {
00697                 if ( CheckWild(BHEAD -mt-2*MAXPOWER, SYMTONUM, -MAXPOWER, &newval3) ) {
00698                     nt += AN.oldvalue;
00699                     goto SubsL1;
00700                 }
00701             }
00702             else {
00703                 nt -= mt * power;
00704             SubsL1:
00705                 if ( nt ) {
00706                     *fill++ = *t;
00707                     *fill++ = nt;
00708                 }
00709                 m += 2; t += 2;
00710             }
00711             else if ( *m >= 2*MAXPOWER ) {
00712                 while ( t < xstop ) { *fill++ = *t++; *fill++ = *t++; }
00713                 nq = WORDDIF(fill, subterm);
00714                 fill = subterm;
00715                 while ( nq > 0 ) {
00716                     if ( !CheckWild(BHEAD *m-2*MAXPOWER, SYMTOSYM, *fill, &newval3) ) {
00717                         mt = m[1];
00718                         if ( mt >= 2*MAXPOWER ) {
00719                             if ( CheckWild(BHEAD mt-2*MAXPOWER, SYMTONUM, -MAXPOWER, &newval3) ) {
00720                                 if ( fill[1] -= AN.oldvalue ) goto SubsL2;
00721                             }
00722                         }
00723                         else if ( mt <= -2*MAXPOWER ) {
00724                             if ( CheckWild(BHEAD -mt-2*MAXPOWER, SYMTONUM, -MAXPOWER, &newval3) ) {
00725                                 if ( fill[1] += AN.oldvalue ) goto SubsL2;
00726                             }
00727                         }
00728                         else {
00729                             if ( fill[1] -= mt * power ) {
00730             SubsL2:
00731                                 fill += nq;
00732                                 nq = 0;
00733                             }
00734                             break;
00735                         }
00736                     }
00737                     nq -= 2;
00738                     fill += 2;
00739                 }
00740                 if ( nq ) {
00741                     nq -= 2;
00742                     q = fill + 2;
00743                     while ( --nq >= 0 ) *fill++ = *q++;
00744                 }
00745                 m += 2;
00746             }
00747             else if ( *m < *t || t >= xstop ) { m += 2; }
00748             else { *fill++ = *t++; *fill++ = *t++; }
00749         } while ( m < ystop );
00750         while ( t < xstop ) *fill++ = *t++;
00751         nq = WORDDIF(fill, subterm);
00752         if ( nq > 0 ) {
00753             nq += 2;
00754             subterm[-1] = nq;
00755         }
00756         else { fill = subterm; fill -= 2; }
00757     }
00758     /*
00759     #] SYMBOLS :
00760     #[ DOTPRODUCTS :
00761     */
00762     else if ( *m == DOTPRODUCT ) {
00763         ystop = m + m[1];
00764         m += 2;
00765         while ( *t > DOTPRODUCT && t < tstop ) {
00766             nq = t[1];
00767             NCOPY(fill, t, nq);
00768         }
00769         if ( t >= tstop ) goto SubCoef;
00770         if ( *t != DOTPRODUCT ) {
00771             m = ystop;
00772             goto EndLoop;
00773         }
00774         *fill++ = DOTPRODUCT;

```



```

00774         fill++;
00775         subterm = fill;
00776         xstop = t + t[1];
00777         t += 2;
00778         do {
00779             if ( *m == *t && m[1] == t[1] && t < xstop ) {
00780                 nt = t[2];
00781                 mt = m[2];
00782                 if ( mt >= 2*MAXPOWER ) {
00783                     if ( CheckWild(BHEAD mt-2*MAXPOWER, SYMTONUM, -MAXPOWER, &newval3) ) {
00784                         nt -= AN.oldvalue;
00785                         goto SubsL3;
00786                     }
00787                 }
00788                 else if ( mt <= -2*MAXPOWER ) {
00789                     if ( CheckWild(BHEAD -mt-2*MAXPOWER, SYMTONUM, -MAXPOWER, &newval3) ) {
00790                         nt += AN.oldvalue;
00791                         goto SubsL3;
00792                     }
00793                 }
00794                 else {
00795                     nt -= mt * power;
00796 SubsL3:
00797                     if ( nt ) {
00798                         *fill++ = *t++;
00799                         *fill++ = *t;
00800                         *fill++ = nt;
00801                         t += 2;
00802                     }
00803                     else t += 3;
00804                 }
00805                 m += 3;
00806             }
00807             else if ( *m >= (AM.OffsetVector+WILDOFFSET) ) {
00808                 while ( t < xstop ) {
00809                     *fill++ = *t++; *fill++ = *t++; *fill++ = *t++;
00810                 }
00811                 oldvall = 1;
00812                 goto SubsL4;
00813             }
00814             else if ( m[1] >= (AM.OffsetVector+WILDOFFSET) ) {
00815                 while ( *m >= *t && t < xstop ) {
00816                     *fill++ = *t++; *fill++ = *t++; *fill++ = *t++;
00817                 }
00818                 oldvall = 0;
00819 SubsL4:
00820                 nq = WORDDIF(fill, subterm);
00821                 fill = subterm;
00822                 while ( nq > 0 ) {
00823                     if ( ( oldvall && ( (
00824                         !CheckWild(BHEAD *m-WILDOFFSET, VECTOVEC, *fill, &newval3)
00825                         && !CheckWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, fill[1], &newval3)
00826                     ) || (
00827                         !CheckWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, *fill, &newval3)
00828                         && !CheckWild(BHEAD *m-WILDOFFSET, VECTOVEC, fill[1], &newval3)
00829                     ) ) || ( !oldvall && ( (
00830                         *m == *fill
00831                         && !CheckWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, fill[1], &newval3)
00832                     ) || (
00833                         !CheckWild(BHEAD m[1]-WILDOFFSET, VECTOVEC, *fill, &newval3)
00834                         && *m == fill[1] ) ) ) ) {
00835                         mt = m[2];
00836                         if ( mt >= 2*MAXPOWER ) {
00837                             if ( CheckWild(BHEAD mt-2*MAXPOWER, SYMTONUM, -MAXPOWER, &newval3) ) {
00838                                 if ( fill[2] -= AN.oldvalue )
00839                                     goto SubsL5;
00840                             }
00841                         }
00842                         else if ( mt <= -2*MAXPOWER ) {
00843                             if ( CheckWild(BHEAD -mt-2*MAXPOWER, SYMTONUM, -MAXPOWER, &newval3) ) {
00844                                 if ( fill[2] += AN.oldvalue )
00845                                     goto SubsL5;
00846                             }
00847                         }
00848                     }
00849                     else {
00850                         if ( fill[2] -= mt * power ) {
00851                             fill += nq;
00852                             nq = 0;
00853                         }
00854                     }
00855                     m += 3;
00856                     break;
00857                 }
00858                 fill += 3; nq -= 3;
00859             }
00860             if ( nq ) {
00861                 nq -= 3;
00862                 q = fill + 3;
00863                 while ( --nq >= 0 ) *fill++ = *q++;

```

```

00861         }
00862     }
00863     else if ( t >= xstop || *m < *t || ( *m == *t && m[1] < t[1] ) )
00864     { m += 3; }
00865     else {
00866         *fill++ = *t++; *fill++ = *t++; *fill++ = *t++;
00867     }
00868     while ( m < ystop );
00869     while ( t < xstop ) *fill++ = *t++;
00870     nq = WORDDIF(fill,subterm);
00871     if ( nq > 0 ) {
00872         nq += 2;
00873         subterm[-1] = nq;
00874     }
00875     else { fill = subterm; fill -= 2; }
00876 }
00877 /*
00878     #] DOTPRODUCTS :
00879     #[ FUNCTIONS :
00880 */
00881 else if ( *m >= FUNCTION ) {
00882     while ( *t >= FUNCTION || *t == SUBEXPRESSION ) {
00883         nt = WORDDIF(t,term);
00884         for ( mt = 0; mt < AN.RepFunNum; mt += 2 ) {
00885             if ( nt == AN.RepFunList[mt] ) break;
00886         }
00887         if ( mt >= AN.RepFunNum ) {
00888             nq = t[1];
00889             NCOPY(fill,t,nq);
00890         }
00891         else {
00892             WORD *oldt = 0;
00893             if ( *m == GAMMA && m[1] != FUNHEAD+1 ) {
00894                 oldt = t;
00895                 if ( ( i = AN.RepFunList[mt+1] ) > 0 ) {
00896                     *fill++ = GAMMA;
00897                     *fill++ = i + FUNHEAD+1;
00898                     FILLFUN(fill)
00899                     nq = i + 1;
00900                     t += FUNHEAD;
00901                     NCOPY(fill,t,nq);
00902                 }
00903                 t = oldt;
00904             }
00905             else if ( ( *t == LEVICIVITA ) || ( *t >= FUNCTION
00906                 && (functions[*t-FUNCTION].symmetric & ~REVERSEORDER) == ANTISYMMETRIC )
00907                 ) sign += AN.RepFunList[mt+1];
00908             else if ( *m >= FUNCTION+WILDOFFSET
00909                 && (functions[*m-FUNCTION-WILDOFFSET].symmetric & ~REVERSEORDER) == ANTISYMMETRIC
00910                 ) sign += AN.RepFunList[mt+1];
00911             if ( !PutExpr ) {
00912                 xstop = t + t[1];
00913                 t = AN.FullProto;
00914                 nq = t[1];
00915                 t[3] = power;
00916                 NCOPY(fill,t,nq);
00917                 t = xstop;
00918                 PutExpr = 1;
00919             }
00920             else t += t[1];
00921             if ( *m == GAMMA && m[1] != FUNHEAD+1 ) {
00922                 i = oldt[1] - m[1] - i;
00923                 if ( i > 0 ) {
00924                     *fill++ = GAMMA;
00925                     *fill++ = i + FUNHEAD+1;
00926                     FILLFUN(fill)
00927                     *fill++ = oldt[FUNHEAD];
00928                     t = t - i;
00929                     NCOPY(fill,t,i);
00930                 }
00931             }
00932             break;
00933         }
00934     }
00935     m += m[1];
00936 }
00937 /*
00938     #] FUNCTIONS :
00939     #[ VECTORS :
00940 */
00941 else if ( *m == VECTOR ) {
00942     while ( *t > VECTOR ) {
00943         nq = t[1];
00944         NCOPY(fill,t,nq);
00945     }
00946     xstop = t + t[1];
00947     ystop = m + m[1];

```

```

00948         t += 2;
00949         m += 2;
00950         *fill++ = VECTOR;
00951         fill++;
00952         subterm = fill;
00953         do {
00954             if ( *m == *t && m[1] == t[1] ) {
00955                 m += 2; t += 2;
00956             }
00957             else if ( *m >= (AM.OffsetVector+WILDOFFSET) ) {
00958                 while ( t < xstop ) *fill++ = *t++;
00959                 nq = WORDDIF(fill,subterm);
00960                 fill = subterm;
00961                 if ( m[1] < (AM.OffsetIndex+WILDOFFSET) ) {
00962                     do {
00963                         if ( m[1] == fill[1] &&
00964                             !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*fill,&newval3) )
00965                             break;
00966                         fill += 2;
00967                         nq -= 2;
00968                     } while ( nq > 0 );
00969                 }
00970                 else { /* Double wildcard */
00971                     do {
00972                         if ( !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,fill[1],&newval3)
00973                             && !CheckWild(BHEAD *m-WILDOFFSET,VECTOVEC,*fill,&newval3) )
00974                             break;
00975                         if ( *fill == oldvall && fill[1] == AN.oldvalue ) break;
00976                         fill += 2;
00977                         nq -= 2;
00978                     } while ( nq > 0 );
00979                 }
00980                 nq -= 2;
00981                 q = fill + 2;
00982                 if ( nq > 0 ) { NCOPY(fill,q,nq); }
00983                 m += 2;
00984             }
00985             else if ( *m <= *t &&
00986                 m[1] >= (AM.OffsetIndex + WILDOFFSET) ) {
00987                 while ( *m == *t && t < xstop )
00988                     { *fill++ = *t++; *fill++ = *t++; }
00989                 nq = WORDDIF(fill,subterm);
00990                 fill = subterm;
00991                 do {
00992                     if ( *m == *fill &&
00993                         !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,fill[1],&newval3) )
00994                         break;
00995                     nq -= 2;
00996                     fill += 2;
00997                 } while ( nq > 0 );
00998                 nq -= 2;
00999                 q = fill + 2;
01000                 if ( nq > 0 ) { NCOPY(fill,q,nq); }
01001                 m += 2;
01002             }
01003             else { *fill++ = *t++; *fill++ = *t++; }
01004         } while ( m < ystop );
01005         while ( t < xstop ) *fill++ = *t++;
01006         nq = WORDDIF(fill,subterm);
01007         if ( nq > 0 ) {
01008             nq += 2;
01009             subterm[-1] = nq;
01010         }
01011         else { fill = subterm; fill -= 2; }
01012     }
01013     /*
01014     #] VECTORS :
01015     #[ INDICES :
01016
01017     Currently without wildcards
01018     */
01019     else if ( *m == INDEX ) {
01020         while ( *t > INDEX ) {
01021             nq = t[1];
01022             NCOPY(fill,t,nq);
01023         }
01024         xstop = t + t[1];
01025         ystop = m + m[1];
01026         t += 2;
01027         m += 2;
01028         *fill++ = INDEX;
01029         fill++;
01030         subterm = fill;
01031         do {
01032             if ( *m == *t ) {
01033                 m += 1; t += 1;
01034             }

```

```

01035         else if ( *m >= (AM.OffsetIndex+WILDOFFSET) ) {
01036             while ( t < xstop ) *fill++ = *t++;
01037             nq = WORDDIF(fill, subterm);
01038             fill = subterm;
01039             do {
01040                 if ( !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,*fill,&newval3) ) {
01041                     break;
01042                 }
01043                 fill += 1;
01044                 nq -= 1;
01045             } while ( nq > 0 );
01046             nq -= 1;
01047             if ( nq > 0 ) {
01048                 q = fill + 1;
01049                 NCOPY(fill,q,nq);
01050             }
01051             m += 1;
01052         }
01053         else {
01054             *fill++ = *t++;
01055         }
01056     } while ( m < ystop );
01057     while ( t < xstop ) *fill++ = *t++;
01058     nq = WORDDIF(fill,subterm);
01059     if ( nq > 0 ) {
01060         nq += 2;
01061         subterm[-1] = nq;
01062     }
01063     else { fill = subterm; fill -= 2; }
01064 }
01065 /*
01066     #] INDICES :
01067     #[ DELTAS :
01068 */
01069     else if ( *m == DELTA ) {
01070         while ( *t > DELTA ) {
01071             nq = t[1];
01072             NCOPY(fill,t,nq);
01073         }
01074         xstop = t + t[1];
01075         ystop = m + m[1];
01076         t += 2;
01077         m += 2;
01078         *fill++ = DELTA;
01079         fill++;
01080         subterm = fill;
01081         do {
01082             if ( *t == *m && t[1] == m[1] ) { m += 2; t += 2; }
01083             else if ( *m >= (AM.OffsetIndex+WILDOFFSET) ) { /* Two dummies */
01084                 while ( t < xstop ) *fill++ = *t++;
01085                 fill = subterm; /*
01086                 oldvall = 1;
01087                 goto SubsL6;
01088             }
01089             else if ( m[1] >= (AM.OffsetIndex+WILDOFFSET) ) {
01090                 while ( (*m == *t || *m == t[1]) && ( t < xstop ) ) {
01091                     *fill++ = *t++; *fill++ = *t++;
01092                 }
01093                 oldvall = 0;
01094                 nq = WORDDIF(fill,subterm);
01095                 fill = subterm;
01096                 do {
01097                     if ( ( oldvall && ( (
01098                         !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,*fill,&newval3)
01099                         && !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,fill[1],&newval3)
01100                     ) || (
01101                         !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,*fill,&newval3)
01102                         && !CheckWild(BHEAD *m-WILDOFFSET,INDTOIND,fill[1],&newval3)
01103                     ) ) ) || ( !oldvall && ( (
01104                         *m == *fill
01105                         && !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,fill[1],&newval3)
01106                     ) || (
01107                         *m == fill[1]
01108                         && !CheckWild(BHEAD m[1]-WILDOFFSET,INDTOIND,*fill,&newval3)
01109                     ) ) ) ) break;
01110                     fill += 2;
01111                     nq -= 2;
01112                 } while ( nq > 0 );
01113                 nq -= 2;
01114                 if ( nq > 0 ) {
01115                     q = fill + 2;
01116                     NCOPY(fill,q,nq);
01117                 }
01118                 m += 2;
01119             }
01120             else {
01121                 *fill++ = *t++; *fill++ = *t++;

```

```

01122     }
01123     } while ( m < ystop );
01124     while ( t < xstop ) *fill++ = *t++;
01125     nq = WORDDIF(fill,subterm);
01126     if ( nq > 0 ) {
01127         nq += 2;
01128         subterm[-1] = nq;
01129     }
01130     else { fill = subterm; fill -= 2; }
01131 }
01132 /*
01133     #] DELTAS :
01134 */
01135 EndLoop;;
01136     } while ( m < mstop ); }
01137     while ( t < tstop ) *fill++ = *t++;
01138 SubCoef:
01139     if ( !PutExpr ) {
01140         t = AN.FullProto;
01141         nq = t[1];
01142         t[3] = power;
01143         NCOPY(fill,t,nq);
01144     }
01145     t = tcoef;
01146     nq = ABS(*t);
01147     t = tstop;
01148     NCOPY(fill,t,nq);
01149     nq = WORDDIF(fill,TemTerm);
01150     fill = term;
01151     t = TemTerm;
01152     *fill++ = nq--;
01153     t++;
01154     NCOPY(fill,t,nq);
01155     if ( sign ) {
01156         if ( ( sign & 1 ) != 0 ) fill[-1] = -fill[-1];
01157     }
01158     if ( AT.WorkPointer < fill ) AT.WorkPointer = fill;
01159     AN.RepFunNum = 0;
01160 }
01161
01162 /*
01163     #] Substitute :
01164     #[ FindSpecial :          WORD FindSpecial(term)
01165
01166     Routine to detect simplifications regarding the special functions
01167     exponent, denominator.
01168
01169 void FindSpecial(WORD *term)
01170 {
01171     WORD *t;
01172     WORD *tstop;
01173     t = term; t += *t - 1; tstop = t - ABS(*t) + 1; t = term;
01174     t++;
01175     if ( t < tstop ) { do {
01176         if ( *t == EXPONENT ) {
01177             Exponents can become simpler when:
01178             a: the exponent of an expression becomes an integer.
01179             b: The expression becomes zero.
01180         }
01181         else if ( *t == DENOMINATOR ) {
01182             Denominators can become simpler when:
01183             a: The denominator is a single term without functions.
01184             b: An overall coefficient can be removed.
01185             c: An overall object can be removed.
01186             The task is here to bring the denominator in an unique form.
01187         }
01188         t += *t;
01189     } while ( t < tstop ); }
01190 }
01191
01192     #] FindSpecial :
01193     #[ FindAll :              WORD FindAll(term,pattern,level,par)
01194 */
01195
01196 int FindAll(PHEAD WORD *term, WORD *pattern, WORD level, WORD *par)
01197 {
01198     GETBIDENTITY
01199     WORD *t, *m, *r, *mm, rnum;
01200     WORD *tstop, *mstop, *TwoProto, *vwhere = 0, oldv, oldvv, vv, level2;
01201     WORD v, nq, OffNum = AM.OffsetVector + WILD OFFSET, i, ii = 0, jj;
01202     WORD fromindex, *intens, notflag1 = 0, notflag2 = 0;
01203     CBUF *C;
01204     C = cbuf+AM.rbufnum;
01205     v = pattern[3]; /* The vector to be found */
01206     m = t = term;
01207     m += *m;

```

```

01209     m -= ABS(m[-1]);
01210     t++;
01211     if ( t < m ) do {
01212         tstop = t + t[1];
01213         fromindex = 2;
01214     /*
01215         #[ VECTOR :
01216     */
01217     if ( *t == VECTOR ) {
01218         r = t;
01219         r += 2;
01220 InVect:
01221         while ( r < tstop ) {
01222             oldv = *r;
01223             if ( v >= OffNum ) {
01224                 vwhere = AN.FullProto + 3 + SUBEXPSIZE;
01225                 if ( vwhere[1] == FROMSET || vwhere[1] == SETTONUM ) {
01226                     WORD *afirst, *alast, j;
01227                     j = vwhere[3];
01228                     if ( j > WILDOFFSET ) { j -= 2*WILDOFFSET; notflag1 = 1; }
01229                     else { notflag1 = 0; }
01230                     afirst = SetElements + Sets[j].first;
01231                     alast = SetElements + Sets[j].last;
01232                     ii = 1;
01233                     if ( notflag1 == 0 ) {
01234                         do {
01235                             if ( *afirst == *r ) {
01236                                 if ( vwhere[1] == SETTONUM ) {
01237                                     AN.FullProto[8+SUBEXPSIZE] = SYMTONUM;
01238                                     AN.FullProto[11+SUBEXPSIZE] = ii;
01239                                 }
01240                                 else if ( vwhere[4] >= 0 ) {
01241                                     oldv = *(afirst - Sets[j].first
01242                                         + Sets[vwhere[4]].first);
01243                                 }
01244                                 goto DoVect;
01245                             }
01246                             ii++;
01247                         } while ( ++afirst < alast );
01248                     }
01249                     else {
01250                         do {
01251                             if ( *afirst == *r ) break;
01252                         } while ( ++afirst < alast );
01253                         if ( afirst >= alast ) goto DoVect;
01254                     }
01255                 }
01256                 else goto DoVect;
01257             }
01258             else if ( v == *r ) {
01259 DoVect:
01260                 m = AT.WorkPointer;
01261                 tstop = t;
01262                 t = term;
01263                 mstop = t + *t;
01264                 do { *m++ = *t++; } while ( t < tstop );
01265                 vwhere = m;
01266                 t = AN.FullProto;
01267                 nq = t[1];
01268                 t[3] = 1;
01269                 NCOPY(m,t,nq);
01270                 t = tstop;
01271                 if ( fromindex == 1 ) m[-1] = FUNNYVEC;
01272                 else m[-1] = r[1]; /* The index is always here! */
01273                 if ( v >= OffNum ) vwhere[3+SUBEXPSIZE] = oldv;
01274                 if ( vwhere[1] > 12+SUBEXPSIZE ) {
01275                     vwhere[11+SUBEXPSIZE] = ii;
01276                     vwhere[8+SUBEXPSIZE] = SYMTONUM;
01277                 }
01278                 if ( t[1] > fromindex+2 ) {
01279                     *m++ = *t++;
01280                     *m++ = *t++ - fromindex;
01281                     while ( t < r ) *m++ = *t++;
01282                     t += fromindex;
01283                 }
01284                 else t += t[1];
01285                 do { *m++ = *t++; } while ( t < mstop );
01286                 *AT.WorkPointer = nq = WORDDIF(m,AT.WorkPointer);
01287                 m = AT.WorkPointer;
01288                 t = term;
01289                 NCOPY(t,m,nq);
01290                 AT.WorkPointer = t;
01291                 return(1);
01292             }
01293             r += fromindex;
01294         }
01295     /*

```

```

01296         #] VECTOR :
01297         #[ DOTPRODUCT :
01298     */
01299     else if ( *t == DOTPRODUCT ) {
01300         r = t;
01301         r += 2;
01302         do {
01303             if ( ( i = r[2] ) < 0 ) goto NextDot;
01304             if ( *r == r[1] ) { /* p.p */
01305                 oldv = *r;
01306                 if ( v == *r ) { /* v.v */
01307                     TwoVec:
01308                         m = AT.WorkPointer;
01309                         tstop = t;
01310                         t = term;
01311                         mstop = t + *t;
01312                         do { *m++ = *t++; } while ( t < tstop );
01313                         do {
01314                             vwhere = m;
01315                             t = AN.FullProto;
01316                             nq = t[1];
01317                             t[3] = 2;
01318                             NCOPY(m,t,nq);
01319                             m[-1] = ++AR.CurDum;
01320                             if ( v >= OffNum ) vwhere[3+SUBEXPSIZE] = oldv;
01321                         } while ( --i > 0 );
01322                         t = tstop;
01323                         if ( t[1] > 5 ) {
01324                             *m++ = *t++;
01325                             *m++ = *t++ - 3;
01326                             while ( t < r ) *m++ = *t++;
01327                             t += 3;
01328                         }
01329                         else t += t[1];
01330                         do { *m++ = *t++; } while ( t < mstop );
01331                         *AT.WorkPointer = nq = WORDDIF(m,AT.WorkPointer);
01332                         m = AT.WorkPointer;
01333                         t = term;
01334                         NCOPY(t,m,nq);
01335                         AT.WorkPointer = t;
01336                         return(1);
01337                     }
01338                 else if ( v >= OffNum ) { /* v?.v? */
01339                     vwhere = AN.FullProto + 3+SUBEXPSIZE;
01340                     if ( vwhere[1] == FROMSET || vwhere[1] == SETTONUM ) {
01341                         WORD *afirst, *alast, j;
01342                         j = vwhere[3];
01343                         if ( j > WILDOFFSET ) { j -= 2*WILDOFFSET; notflag1 = 1; }
01344                         else { notflag1 = 0; }
01345                         afirst = SetElements + Sets[j].first;
01346                         alast = SetElements + Sets[j].last;
01347                         ii = 1;
01348                         if ( notflag1 == 0 ) {
01349                             do {
01350                                 if ( *afirst == *r ) {
01351                                     if ( vwhere[1] == SETTONUM ) {
01352                                         AN.FullProto[8+SUBEXPSIZE] = SYMTONUM;
01353                                         AN.FullProto[11+SUBEXPSIZE] = ii;
01354                                     }
01355                                     else if ( vwhere[4] >= 0 ) {
01356                                         oldv = *(afirst - Sets[j].first
01357                                             + Sets[vwhere[4]].first);
01358                                         goto TwoVec;
01359                                     }
01360                                 }
01361                                 ii++;
01362                             } while ( ++afirst < alast );
01363                         }
01364                         else {
01365                             do {
01366                                 if ( *afirst == *r ) break;
01367                             } while ( ++afirst < alast );
01368                             if ( afirst >= alast ) goto TwoVec;
01369                         }
01370                     }
01371                     else goto TwoVec;
01372                 }
01373             }
01374         }
01375         else {
01376             if ( v == r[1] ) { r[1] = *r; *r = v; }
01377             oldv = *r;
01378             oldvv = r[1];
01379             if ( v == *r ) {
01380                 if ( !par ) { while ( ++level <= AR.Cnumlhs
01381                     && C->lhs[level][0] == TYPEIDOLD ) {
01382                     m = C->lhs[level];
01383                     m += IDHEAD;
01384                     if ( m[-IDHEAD+2] == SUBVECTOR ) {

```

```

01383         if ( ( vv = m[m[1]+3] ) == r[1] ) {
01384 OnePV:         TwoProto = AN.FullProto;
01385 TwoPV:         m = AT.WorkPointer;
01386                 tstop = t;
01387                 t = term;
01388                 mstop = t + *t;
01389                 do { *m++ = *t++; } while ( t < tstop );
01390                 do {
01391                     t = AN.FullProto;
01392                     vwhere = m + 3 + SUBEXPSIZE;
01393                     nq = t[1];
01394                     t[3] = 1;
01395                     NCOPY(m,t,nq);
01396                     m[-1] = ++AR.CurDum;
01397                     if ( v >= OffNum ) *vwhere = oldv;
01398                     if ( vwhere[-2-SUBEXPSIZE] > 12+SUBEXPSIZE ) {
01399                         vwhere[8] = ii;
01400                         vwhere[5] = SYMTONUM;
01401                     }
01402                     t = TwoProto;
01403                     vwhere = m + 3+SUBEXPSIZE;
01404                     mm = m;
01405                     nq = t[1];
01406                     t[3] = 1;
01407                     NCOPY(m,t,nq);
01408 /*
01409 The next two lines repair a bug. without them it takes twice
01410 the rhs of the first vector.
01411 */
01412                     mm[2] = C->lhs[level][IDHEAD+2];
01413                     mm[4] = C->lhs[level][IDHEAD+4];
01414                     m[-1] = AR.CurDum;
01415                     if ( vv >= OffNum ) *vwhere = oldvv;
01416                 } while ( --i > 0 );
01417                 goto CopRest;
01418             }
01419             else if ( vv > OffNum ) {
01420                 vwhere = AN.FullProto + 3+SUBEXPSIZE;
01421                 if ( vwhere[1] == FROMSET || vwhere[1] == SETTONUM ) {
01422                     WORD *afirst, *alast, j;
01423                     j = vwhere[3];
01424                     if ( j > WILDOFFSET ) { j -= 2*WILDOFFSET; notflag1 = 1; }
01425                     else { notflag1 = 0; }
01426                     afirst = SetElements + Sets[j].first;
01427                     alast = SetElements + Sets[j].last;
01428                     if ( notflag1 == 0 ) {
01429                         ii = 1;
01430                         do {
01431                             if ( *afirst == r[1] ) {
01432                                 if ( vwhere[1] == SETTONUM ) {
01433                                     AN.FullProto[8+SUBEXPSIZE] = SYMTONUM;
01434                                     AN.FullProto[11+SUBEXPSIZE] = ii;
01435                                 }
01436                                 else if ( vwhere[4] >= 0 ) {
01437                                     oldvv = *(afirst - Sets[j].first
01438                                         + Sets[vwhere[4]].first);
01439                                 }
01440                                 goto OnePV;
01441                             }
01442                             ii++;
01443                             } while ( ++afirst < alast );
01444                         }
01445                     else {
01446                         do {
01447                             if ( *afirst == *r ) break;
01448                             } while ( ++afirst < alast );
01449                             if ( afirst >= alast ) goto OnePV;
01450                         }
01451                     }
01452                 else goto OnePV;
01453             }
01454         }
01455     }}
01456 /*
01457 v.q with v matching and no match for the q, also
01458 not in following idold statements.
01459 Notice that a following q.p? cannot match.
01460 */
01461     rnum = r[1];
01462 OneOnly:     m = AT.WorkPointer;
01463             tstop = t;
01464             t = term;
01465             mstop = t + *t;
01466             do { *m++ = *t++; } while ( t < tstop );
01467             vwhere = m;
01468             t = AN.FullProto;
01469             nq = t[1];

```



```

01470         t[3] = i;
01471         NCOPY(m,t,nq);
01472         m[-4] = INDTOIND;
01473         m[-1] = rnum;
01474         if ( v >= OffNum ) vwhere[3+SUBEXPSIZE] = oldv;
01475         goto CopRest;
01476     }
01477     else if ( v >= OffNum ) {
01478         vwhere = AN.FullProto + 3+SUBEXPSIZE;
01479         if ( vwhere[1] == FROMSET || vwhere[1] == SETTONUM ) {
01480             WORD *afirst, *alast, *bfirst, *blast, j;
01481             j = vwhere[3];
01482             if ( j > WILDOFFSET ) { j -= 2*WILDOFFSET; notflag1 = 1; }
01483             else { notflag1 = 0; }
01484             afirst = SetElements + Sets[j].first;
01485             alast = SetElements + Sets[j].last;
01486             ii = 1;
01487             if ( notflag1 == 0 ) {
01488                 do {
01489                     if ( *afirst == *r ) {
01490                         if ( vwhere[1] == SETTONUM ) {
01491                             AN.FullProto[8+SUBEXPSIZE] = SYMTONUM;
01492                             AN.FullProto[11+SUBEXPSIZE] = ii;
01493                         }
01494                         else if ( vwhere[4] >= 0 ) {
01495                             oldv = *(afirst - Sets[j].first
01496                                 + Sets[vwhere[4]].first);
01497                         }
01498                         level2 = level;
01499                         do {
01500                             if ( !par ) m = C->lhs[level2];
01501                             else m = par;
01502                             m += IDHEAD;
01503                             if ( m[-IDHEAD+2] == SUBVECTOR ) {
01504                                 if ( ( vv = m[m[1]+3] ) == r[1] )
01505                                     goto OnePV;
01506                                 else if ( vv >= OffNum ) {
01507                                     if ( m[SUBEXPSIZE+4] != FROMSET &&
01508                                         m[SUBEXPSIZE+4] != SETTONUM ) goto OnePV;
01509                                     j = m[SUBEXPSIZE+6];
01510                                     if ( j > WILDOFFSET ) { j -= 2*WILDOFFSET; notflag2 = 1; }
01511                                     else { notflag2 = 0; }
01512                                     bfirst = SetElements + Sets[j].first;
01513                                     blast = SetElements + Sets[j].last;
01514                                     jj = 1;
01515                                     if ( notflag2 == 0 ) {
01516                                         do {
01517                                             if ( *bfirst == r[1] ) {
01518                                                 if ( m[SUBEXPSIZE+4] == SETTONUM ) {
01519                                                     m[SUBEXPSIZE+8] = SYMTONUM;
01520                                                     m[SUBEXPSIZE+11] = jj;
01521                                                 }
01522                                                 else if ( m[SUBEXPSIZE+7] >= 0 ) {
01523                                                     oldvv = *(bfirst - Sets[j].first
01524                                                         + Sets[m[SUBEXPSIZE+7]].first);
01525                                                 }
01526                                                 goto OnePV;
01527                                             }
01528                                             jj++;
01529                                         } while ( ++bfirst < blast );
01530                                     }
01531                                     else {
01532                                         do {
01533                                             if ( *bfirst == r[1] ) break;
01534                                         } while ( ++bfirst < blast );
01535                                         if ( bfirst >= blast ) goto OnePV;
01536                                     }
01537                                 }
01538                             } while ( ++level2 < AR.Cnumlhs &&
01539                                 C->lhs[level2][0] == TYPEIDOLD );
01540                             rnum = r[1];
01541                             goto OneOnly;
01542                         }
01543                     }
01544                     else if ( *afirst == r[1] ) {
01545                         if ( vwhere[1] == SETTONUM ) {
01546                             AN.FullProto[8+SUBEXPSIZE] = SYMTONUM;
01547                             AN.FullProto[11+SUBEXPSIZE] = ii;
01548                         }
01549                         else if ( vwhere[4] >= 0 ) {
01550                             oldv = *(afirst - Sets[j].first
01551                                 + Sets[vwhere[4]].first);
01552                         }
01553                         level2 = level;
01554                         while ( ++level2 < AR.Cnumlhs &&
01555                             C->lhs[level2][0] == TYPEIDOLD ) {
01556                             if ( !par ) m = C->lhs[level2];

```

```

01557         else m = par;
01558         m += IDHEAD;
01559         if ( m[-IDHEAD+2] == SUBVECTOR ) {
01560             if ( ( vv = m[6] ) == *r )
01561                 goto OnePV;
01562             else if ( vv >= OffNum ) {
01563                 if ( m[SUBEXPSIZE+4] != FROMSET && m[SUBEXPSIZE+4]
01564                     != SETTONUM ) {
01565                     j = *r;
01566                     *r = r[1];
01567                     r[1] = j;
01568                     goto OnePV;
01569                 }
01570                 j = m[SUBEXPSIZE+6];
01571                 bfirst = SetElements + Sets[j].first;
01572                 blast = SetElements + Sets[j].last;
01573                 jj = 1;
01574                 do {
01575                     if ( *bfirst == *r ) {
01576                         if ( m[SUBEXPSIZE+4] == SETTONUM ) {
01577                             m[SUBEXPSIZE+8] = SYMTONUM;
01578                             m[SUBEXPSIZE+11] = jj;
01579                         }
01580                         else if ( m[SUBEXPSIZE+7] >= 0 ) {
01581                             oldvv = *(bfirst - Sets[j].first
01582                                 + Sets[m[SUBEXPSIZE+7]].first);
01583                         }
01584                         j = *r;
01585                         *r = r[1];
01586                         r[1] = j;
01587                         j = oldv; oldv = oldvv; oldvv = j;
01588                         goto OnePV;
01589                     }
01590                     jj++;
01591                 } while ( ++bfirst < blast );
01592             }
01593         }
01594     }
01595     jj = *r; *r = r[1]; r[1] = jj;
01596     jj = oldv; oldv = oldvv; oldvv = j;
01597     rnum = r[1];
01598     goto OneOnly;
01599 }
01600 ii++;
01601 } while ( ++afirst < alast );
01602 }
01603 else {
01604     do {
01605         if ( *afirst == *r ) break;
01606     } while ( ++afirst < alast );
01607     if ( afirst >= alast ) goto Hitlevel1;
01608     do {
01609         if ( *afirst == r[1] ) break;
01610     } while ( ++afirst < alast );
01611     if ( afirst >= alast ) goto Hitlevel2;
01612 }
01613 }
01614 else { /* Matches twice */
01615     vv = v;
01616     TwoProto = AN.FullProto;
01617     goto TwoPV;
01618 }
01619 }
01620 }
01621 NextDot:    r += 3;
01622 } while ( r < tstop );
01623 }
01624 /*
01625     #] DOTPRODUCT :
01626     #[ LEVICIVITA :
01627 */
01628 else if ( *t == LEVICIVITA ) {
01629     intens = 0;
01630     r = t;
01631     r += FUNHEAD;
01632 OneVect:;
01633     while ( r < tstop ) {
01634         oldv = *r;
01635         if ( v >= OffNum && *r < -10 ) {
01636             vwhere = AN.FullProto + 3+SUBEXPSIZE;
01637             if ( vwhere[1] == FROMSET || vwhere[1] == SETTONUM ) {
01638                 WORD *afirst, *alast, j;
01639                 j = vwhere[3];
01640                 if ( j > WILDOFFSET ) { j -= 2*WILDOFFSET; notflag1 = 1; }
01641                 else { notflag1 = 0; }
01642                 afirst = SetElements + Sets[j].first;
01643                 alast = SetElements + Sets[j].last;

```

```

01644             ii = 1;
01645             if ( notflag1 == 0 ) {
01646                 do {
01647                     if ( *afirst == *r ) {
01648                         if ( vwhere[1] == SETTONUM ) {
01649                             AN.FullProto[8+SUBEXPSIZE] = SYMTONUM;
01650                             AN.FullProto[11+SUBEXPSIZE] = ii;
01651                         }
01652                         else if ( vwhere[4] >= 0 ) {
01653                             oldv = *(afirst - Sets[j].first
01654                                 + Sets[vwhere[4]].first);
01655                         }
01656                         goto DoVect;
01657                     }
01658                     ii++;
01659                 } while ( ++afirst < alast );
01660             }
01661             else {
01662                 do {
01663                     if ( *afirst == *r ) break;
01664                 } while ( ++afirst < alast );
01665                 if ( afirst >= alast ) goto DoVect;
01666             }
01667         }
01668         else goto LeVect;
01669     }
01670     else if ( v == *r ) {
01671 LeVect:
01672         m = AT.WorkPointer;
01673         mstop = term + *term;
01674         t = term;
01675         *r = ++AR.CurDum;
01676         if ( intens ) *intens = DIRTYSYMFLAG;
01677         do { *m++ = *t++; } while ( t < tstop );
01678         t = AN.FullProto;
01679         nq = t[1];
01680         t[3] = 1;
01681         if ( v >= OffNum ) *vwhere = oldv;
01682         NCOPY(m,t,nq);
01683         m[-1] = AR.CurDum;
01684         t = tstop;
01685         do { *m++ = *t++; } while ( t < mstop );
01686         *AT.WorkPointer = nq = WORDDIF(m,AT.WorkPointer);
01687         m = AT.WorkPointer;
01688         t = term;
01689         NCOPY(t,m,nq);
01690         AT.WorkPointer = t;
01691         return(1);
01692     }
01693     r++;
01694 }
01695 /*
01696     #] LEVICIVITA :
01697     #[ GAMMA :
01698 */
01699     else if ( *t == GAMMA ) {
01700         intens = 0;
01701         r = t;
01702         r += FUNHEAD+1;
01703         if ( r < tstop ) goto OneVect;
01704     }
01705 /*
01706     #] GAMMA :
01707     #[ INDEX :
01708 */
01709     else if ( *t == INDEX ) { /* The 'forgotten' part */
01710         r = t;
01711         r += 2;
01712         fromindex = 1;
01713         goto InVect;
01714     }
01715 /*
01716     #] INDEX :
01717     #[ FUNCTION :
01718 */
01719     else if ( *t >= FUNCTION ) {
01720         if ( *t >= FUNCTION
01721             && functions[*t-FUNCTION].spec >= TENSORFUNCTION
01722             && t[1] > FUNHEAD ) {
01723 /*
01724             Tensors are linear in their vectors!
01725 */
01726             r = t;
01727             r += FUNHEAD;
01728             intens = t+2;
01729             goto OneVect;
01730         }

```

```

01731     }
01732  /*
01733     #] FUNCTION :
01734  */
01735     t += t[1];
01736   } while ( t < m );
01737   return(0);
01738 }
01739
01740 /*
01741     #] FindAll :
01742     #[ TestSelect :
01743
01744     Returns 1 if any of the objects in any of the sets in setp
01745     occur anywhere in the term
01746  */
01747
01748 int TestSelect(WORD *term, WORD *setp)
01749 {
01750     WORD *tstop, *t, *s, *el, *elstop, *termstop, *tt, n, ns;
01751     GETSTOP(term,tstop);
01752     term += 1;
01753     while ( term < tstop ) {
01754         switch ( *term ) {
01755             case SYMBOL:
01756                 n = term[1] - 2;
01757                 t = term + 2;
01758                 while ( n > 0 ) {
01759                     ns = setp[1] - 2;
01760                     s = setp + 2;
01761                     while ( --ns >= 0 ) {
01762                         if ( Sets[*s].type != CSYMBOL ) { s++; continue; }
01763                         el = SetElements + Sets[*s].first;
01764                         elstop = SetElements + Sets[*s].last;
01765                         while ( el < elstop ) {
01766                             if ( *el++ == *t ) return(1);
01767                         }
01768                         s++;
01769                     }
01770                     n -= 2;
01771                     t += 2;
01772                 }
01773                 break;
01774             case VECTOR:
01775                 n = term[1] - 2;
01776                 t = term + 2;
01777                 while ( n > 0 ) {
01778                     ns = setp[1] - 2;
01779                     s = setp + 2;
01780                     while ( --ns >= 0 ) {
01781                         if ( Sets[*s].type != CVECTOR ) { s++; continue; }
01782                         el = SetElements + Sets[*s].first;
01783                         elstop = SetElements + Sets[*s].last;
01784                         while ( el < elstop ) {
01785                             if ( *el++ == *t ) return(1);
01786                         }
01787                         s++;
01788                     }
01789                     t++;
01790                     ns = setp[1] - 2;
01791                     s = setp + 2;
01792                     while ( --ns >= 0 ) {
01793                         if ( Sets[*s].type != CINDEX
01794                             && Sets[*s].type != CNUMBER ) { s++; continue; }
01795                         el = SetElements + Sets[*s].first;
01796                         elstop = SetElements + Sets[*s].last;
01797                         while ( el < elstop ) {
01798                             if ( *el++ == *t ) return(1);
01799                         }
01800                         s++;
01801                     }
01802                     n -= 2;
01803                     t++;
01804                 }
01805                 break;
01806             case INDEX:
01807                 n = term[1] - 2;
01808                 t = term + 2;
01809                 goto dotensor;
01810             case DOTPRODUCT:
01811                 n = term[1] - 2;
01812                 t = term + 2;
01813                 while ( n > 0 ) {
01814                     ns = setp[1] - 2;
01815                     s = setp + 2;
01816                     while ( --ns >= 0 ) {
01817                         if ( Sets[*s].type != CVECTOR ) { s++; continue; }

```

```

01818         el = SetElements + Sets[*s].first;
01819         elstop = SetElements + Sets[*s].last;
01820         while ( el < elstop ) {
01821             if ( *el++ == *t ) return(1);
01822         }
01823         s++;
01824     }
01825     t++;
01826     ns = setp[1] - 2;
01827     s = setp + 2;
01828     while ( --ns >= 0 ) {
01829         if ( Sets[*s].type != CVECTOR ) { s++; continue; }
01830         el = SetElements + Sets[*s].first;
01831         elstop = SetElements + Sets[*s].last;
01832         while ( el < elstop ) {
01833             if ( *el++ == *t ) return(1);
01834         }
01835         s++;
01836     }
01837     n -= 3;
01838     t += 2;
01839 }
01840 break;
01841 case DELTA:
01842     n = term[1] - 2;
01843     t = term + 2;
01844     goto dotensor;
01845 default:
01846     if ( *term < FUNCTION ) break;
01847     ns = setp[1] - 2;
01848     s = setp + 2;
01849     while ( --ns >= 0 ) {
01850         if ( Sets[*s].type != CFUNCTION ) { s++; continue; }
01851         el = SetElements + Sets[*s].first;
01852         elstop = SetElements + Sets[*s].last;
01853         while ( el < elstop ) {
01854             if ( *el++ == *term ) return(1);
01855         }
01856         s++;
01857     }
01858     if ( functions[*term-FUNCTION].spec > 0 ) {
01859         n = term[1] - FUNHEAD;
01860         t = term + FUNHEAD;
01861 dotensor:
01862         while ( n > 0 ) {
01863             ns = setp[1] - 2;
01864             s = setp + 2;
01865             while ( --ns >= 0 ) {
01866                 if ( *t < MINSPEC ) {
01867                     if ( Sets[*s].type != CVECTOR ) { s++; continue; }
01868                 }
01869                 else if ( *t >= 0 ) {
01870                     if ( Sets[*s].type != CINDEX
01871                         && Sets[*s].type != CNUMBER ) { s++; continue; }
01872                 }
01873                 else { s++; continue; }
01874                 el = SetElements + Sets[*s].first;
01875                 elstop = SetElements + Sets[*s].last;
01876                 while ( el < elstop ) {
01877                     if ( *el++ == *t ) return(1);
01878                 }
01879                 s++;
01880             }
01881             t++;
01882             n--;
01883         }
01884     }
01885     else {
01886         termstop = term + term[1];
01887         tt = term + FUNHEAD;
01888         while ( tt < termstop ) {
01889             if ( *tt < 0 ) {
01890                 if ( *tt == -SYMBOL ) {
01891                     ns = setp[1] - 2;
01892                     s = setp + 2;
01893                     while ( --ns >= 0 ) {
01894                         if ( Sets[*s].type != CSYMBOL ) { s++; continue; }
01895                         el = SetElements + Sets[*s].first;
01896                         elstop = SetElements + Sets[*s].last;
01897                         while ( el < elstop ) {
01898                             if ( *el++ == tt[1] ) return(1);
01899                         }
01900                         s++;
01901                     }
01902                     tt += 2;
01903                 }
01904                 else if ( *tt == -VECTOR || *tt == -MINVECTOR ) {

```

```

01905         ns = setp[1] - 2;
01906         s = setp + 2;
01907         while ( --ns >= 0 ) {
01908             if ( Sets[*s].type != CVECTOR ) { s++; continue; }
01909             el = SetElements + Sets[*s].first;
01910             elstop = SetElements + Sets[*s].last;
01911             while ( el < elstop ) {
01912                 if ( *el++ == tt[1] ) return(1);
01913             }
01914             s++;
01915         }
01916         tt += 2;
01917     }
01918     else if ( *tt == -INDEX ) {
01919         ns = setp[1] - 2;
01920         s = setp + 2;
01921         while ( --ns >= 0 ) {
01922             if ( Sets[*s].type != CINDEX
01923                 && Sets[*s].type != CNUMBER ) { s++; continue; }
01924             el = SetElements + Sets[*s].first;
01925             elstop = SetElements + Sets[*s].last;
01926             while ( el < elstop ) {
01927                 if ( *el++ == tt[1] ) return(1);
01928             }
01929             s++;
01930         }
01931         tt += 2;
01932     }
01933     else if ( *tt <= -FUNCTION ) {
01934         ns = setp[1] - 2;
01935         s = setp + 2;
01936         while ( --ns >= 0 ) {
01937             if ( Sets[*s].type != CFUNCTION ) { s++; continue; }
01938             el = SetElements + Sets[*s].first;
01939             elstop = SetElements + Sets[*s].last;
01940             while ( el < elstop ) {
01941                 if ( *el++ == -( *tt ) ) return(1);
01942             }
01943             s++;
01944         }
01945         tt++;
01946     }
01947     else tt += 2;
01948 }
01949 else {
01950     t = tt + ARGHEAD;
01951     tt += *tt;
01952     while ( t < tt ) {
01953         if ( TestSelect(t,setp) ) return(1);
01954         t += *t;
01955     }
01956 }
01957 }
01958 }
01959 break;
01960 }
01961 term += term[1];
01962 }
01963 return(0);
01964 }
01965
01966 /*
01967     #] TestSelect :
01968     #[ SubsInAll :      void SubsInAll()
01969
01970     This routine takes a match in id,all and stores it away in
01971     the AT.allbufnum 'compiler' buffer, after taking out the pattern.
01972     The main problem here is that id,all usually has (lots of) wildcards
01973     and their assignments are on stack and the difficult ones are in
01974     AT.ebufnum. Popping the stack while looking for more matches would
01975     loose those. Hence we have to copy them into yet another compiler
01976     buffer: AT.aebufnum. Because this may involve many matches and
01977     because the original term has only a limited number of arguments,
01978     it will pay to look for already existing ones in this buffer.
01979     (to be done later).
01980 */
01981
01982 void SubsInAll(PHEAD0)
01983 {
01984     GETBIDENTITY
01985     WORD *TemTerm;
01986     WORD *t, *m, *term;
01987     WORD *tstop, *mstop, *xstop;
01988     WORD nt, *fill, nq, mt;
01989     WORD *tcoef, i = 0;
01990     WORD PutExpr = 0, sign = 0;
01991 }

```

```

01992     We start with building the term in the Workspace.
01993     Afterwards we will transfer it to AT.allbufnum.
01994     We have to make sure there is room in the Workspace.
01995 */
01996     AT.idallflag = 2;
01997     TemTerm = AT.WorkPointer;
01998     if ( ( (WORD *)(((UBYTE *) (AT.WorkPointer)) + AM.MaxTer*2) ) > AT.WorkTop ) {
01999         MLOCK(ErrorMessageLock);
02000         MesWork();
02001         MUNLOCK(ErrorMessageLock);
02002         Terminate(-1);
02003     }
02004     m = AN.patternbuffer + IDHEAD; m += m[1];
02005     mstop = m + *m;
02006     m++;
02007     term = AN.termbuffer;
02008     tstop = term + *term; tcoef = tstop-1; tstop -= ABS(tstop[-1]);
02009     t = term;
02010     t++;
02011     fill = TemTerm;
02012     fill++;
02013     while ( m < mstop ) {
02014         while ( t < tstop ) {
02015             nt = WORDDIF(t,term);
02016             for ( mt = 0; mt < AN.RepFunNum; mt += 2 ) {
02017                 if ( nt == AN.RepFunList[mt] ) break;
02018             }
02019             if ( mt >= AN.RepFunNum ) {
02020                 nq = t[1];
02021                 NCOPY(fill,t,nq);
02022             }
02023             else {
02024                 WORD *oldt = 0;
02025                 if ( *m == GAMMA && m[1] != FUNHEAD+1 ) {
02026                     oldt = t;
02027                     if ( ( i = AN.RepFunList[mt+1] ) > 0 ) {
02028                         *fill++ = GAMMA;
02029                         *fill++ = i + FUNHEAD+1;
02030                         FILLFUN(fill)
02031                         nq = i + 1;
02032                         t += FUNHEAD;
02033                         NCOPY(fill,t,nq);
02034                     }
02035                     t = oldt;
02036                 }
02037                 else if ( ( *t == LEVICIVITA ) || ( *t >= FUNCTION
02038                     && (functions[*t-FUNCTION].symmetric & ~REVERSEORDER) == ANTISYMMETRIC )
02039                     ) sign += AN.RepFunList[mt+1];
02040                 else if ( *m >= FUNCTION+WILDOFFSET
02041                     && (functions[*m-FUNCTION-WILDOFFSET].symmetric & ~REVERSEORDER) == ANTISYMMETRIC
02042                     ) sign += AN.RepFunList[mt+1];
02043                 if ( !PutExpr ) {
02044                     WORD *pstart = fill, *p, *w, *ww;
02045                     xstop = t + t[1];
02046                     t = AN.FullProto;
02047                     nq = t[1];
02048                     t[3] = 1;
02049                     NCOPY(fill,t,nq);
02050                     t = xstop;
02051                     PutExpr = 1;
02052 */
02053
02054     Here we need provisions for keeping wildcard matches
02055     that reside in AT.ebufnum. We will move them to
02056     AT.aebufnum.
02057     Problem: the SUBEXPRESSION assumes automatically
02058     that the compiler buffer is AT.ebufnum. We have to
02059     correct that in TransferBuffer.
02059 */
02060     p = pstart + SUBEXPSIZE;
02061     while ( p < fill ) {
02062         switch ( *p ) {
02063             case SYMTOSUB:
02064             case VECTOSUB:
02065             case INDTOSUB:
02066             case ARGTOARG:
02067             case ARLTOARL:
02068                 w = cbuf[AT.ebufnum].rhs[p[3]];
02069                 ww = cbuf[AT.ebufnum].rhs[p[3]+1];
02070 */
02071
02072     Here we could search for whether this
02073     object sits in the buffer already.
02074     To be done later.
02075     By the way: ww-w fits inside a WORD.
02075 */
02076     AddRHS(AT.aebufnum,1);
02077     AddNtoC(AT.aebufnum,ww-w,w,11);
02078     p[3] = cbuf[AT.aebufnum].numrhs;

```

```

02079             cbuf[AT.aebufnum].rhs[p[3]+1] = cbuf[AT.aebufnum].Pointer;
02080             p += p[1];
02081             break;
02082             case FROMSET:
02083             case SETTONUM:
02084             case LOADDOLLAR:
02085                 p += p[1];
02086                 break;
02087             default:
02088                 p += p[1];
02089                 break;
02090         }
02091     }
02092 }
02093 }
02094 else t += t[1];
02095 if ( *m == GAMMA && m[1] != FUNHEAD+1 ) {
02096     i = oldt[1] - m[1] - i;
02097     if ( i > 0 ) {
02098         *fill++ = GAMMA;
02099         *fill++ = i + FUNHEAD+1;
02100         FILLFUN(fill)
02101         *fill++ = oldt[FUNHEAD];
02102         t = t - i;
02103         NCOPY(fill,t,i);
02104     }
02105 }
02106 break;
02107 }
02108 }
02109 m += m[1];
02110 }
02111 while ( t < tstop ) *fill++ = *t++;
02112 if ( !PutExpr ) {
02113     t = AN.FullProto;
02114     nq = t[1];
02115     t[3] = 1;
02116     NCOPY(fill,t,nq);
02117 }
02118 t = tcoef;
02119 nq = ABS(*t);
02120 t = tstop;
02121 NCOPY(fill,t,nq);
02122 if ( sign ) {
02123     if ( ( sign & 1 ) != 0 ) fill[-1] = -fill[-1];
02124 }
02125 *TemTerm = fill-TemTerm;
02126 /*
02127 And now we copy this to AT.allbufnum
02128 */
02129 AddNtoC(AT.allbufnum,TemTerm[0],TemTerm,12);
02130 cbuf[AT.allbufnum].Pointer[0] = 0;
02131 AN.RepFunNum = 0;
02132 }
02133
02134 /*
02135 #] SubsInAll :
02136 #[ TransferBuffer :
02137
02138 Adds the whole content of a (compiler)buffer to another buffer.
02139 In spectator we have an expression in the RHS that needs the
02140 wildcard resolutions adapted by an offset.
02141 */
02142
02143 void TransferBuffer(int from,int to,int spectator)
02144 {
02145     CBUF *C = cbuf + spectator;
02146     CBUF *Cf = cbuf + from;
02147     CBUF *Ct = cbuf + to;
02148     int offset = Ct->numrhs;
02149     LONG i;
02150     WORD *t, *tt, *ttt, *tstop, size;
02151     for ( i = 1; i <= Cf->numrhs; i++ ) {
02152         size = Cf->rhs[i+1]-Cf->rhs[i];
02153         AddRHS(to,1);
02154         AddNtoC(to,size,Cf->rhs[i],13);
02155     }
02156     Ct->rhs[Ct->numrhs+1] = Ct->Pointer;
02157     Cf->numrhs = 0;
02158 /*
02159 Now we have to update the 'pointers' in the spectator.
02160 */
02161 t = C->rhs[C->numrhs];
02162 while ( *t ) {
02163     tt = t+1; t += *t;
02164     tstop = t-ABS(t[-1]);
02165     while ( tt < tstop ) {

```



```

02166         if ( *tt == SUBEXPRESSION ) {
02167             ttt = tt+SUBEXPSIZE; tt += tt[1];
02168             while ( ttt < tt ) {
02169                 switch ( *ttt ) {
02170                     case SYMTOSUB:
02171                     case VECTOSUB:
02172                     case INDIOSUB:
02173                     case ARGTOARG:
02174                     case ARLTOARL:
02175                         ttt[3] += offset;
02176                         break;
02177                     default:
02178                         break;
02179                 }
02180                 ttt += 4;
02181             }
02182         }
02183         else tt += tt[1];
02184     }
02185 }
02186 }
02187
02188 /*
02189     #] TransferBuffer :
02190     #[ TakeIDfunction :
02191 */
02192
02193 #define PutInBuffers(pow) \
02194     AddrRHS(AT.ebufnum,1); \
02195     *out++ = SUBEXPRESSION; \
02196     *out++ = SUBEXPSIZE; \
02197     *out++ = C->numrhs; \
02198     *out++ = pow; \
02199     *out++ = AT.ebufnum; \
02200     FILLSUB(out) \
02201     r = AT.pWorkspace[rhs+i]; \
02202     if ( *r > 0 ) { \
02203         oldinr = r[*r]; r[*r] = 0; \
02204         AddNtoC(AT.ebufnum, (*r+1-ARGHEAD), (r+ARGHEAD), 14); \
02205         r[*r] = oldinr; \
02206     } \
02207     else { \
02208         ToGeneral(r,buffer,1); \
02209         buffer[buffer[0]] = 0; \
02210         AddNtoC(AT.ebufnum,buffer[0]+1,buffer,15); \
02211     }
02212
02213 int TakeIDfunction(PHEAD WORD *term)
02214 {
02215     WORD *tstop, *t, *r, *m, *f, *nextf, *funstop, *left, *l, *newterm;
02216     WORD *out, oldinr, pow;
02217     WORD buffer[20];
02218     int i, ii, j, numsub, numfound = 0, first;
02219     LONG lhs,rhs;
02220     CBUF *C;
02221     GETSTOP(term,tstop);
02222     for ( t = term+1; t < tstop; t += t[1] ) { if ( *t == IDFUNCTION ) break; }
02223     if ( t >= tstop ) return(0);
02224 /*
02225     Step 1: test validity
02226 */
02227     funstop = t + t[1]; f = t + FUNHEAD;
02228     left = term + *term;
02229     l = left+1; numsub = 0;
02230     while ( f < funstop ) {
02231         nextf = f; NEXTARG(nextf)
02232         if ( nextf >= funstop ) { return(0); } /* odd number of arguments */
02233         if ( *f == -SYMBOL ) { *l++ = SYMBOL; *l++ = 4; *l++ = f[1]; *l++ = 1; }
02234         else if ( *f < -FUNCTION ) { *l++ = *f; *l++ = FUNHEAD; FILLFUN(1) }
02235         else if ( *f > 0 ) {
02236             if ( *f != f[ARGHEAD]+ARGHEAD ) goto noaction;
02237             if ( nextf[-1] != 3 || nextf[-2] != 1 || nextf[-3] != 1 ) goto noaction;
02238             if ( f[ARGHEAD] <= 4 ) goto noaction;
02239             if ( f[ARGHEAD] != f[ARGHEAD+2]+4 ) goto noaction;
02240             if ( f[ARGHEAD] == 8 && f[ARGHEAD+1] == SYMBOL ) {
02241                 for ( i = 0; i < 4; i++ ) *l++ = f[ARGHEAD+1+i];
02242             }
02243             else if ( f[ARGHEAD] == 9 && f[ARGHEAD+1] == DOTPRODUCT ) {
02244                 for ( i = 0; i < 5; i++ ) *l++ = f[ARGHEAD+1+i];
02245             }
02246             else if ( f[ARGHEAD+1] >= FUNCTION ) {
02247                 for ( i = 0; i < f[ARGHEAD+1]-4; i++ ) *l++ = f[ARGHEAD+1+i];
02248             }
02249             else goto noaction;
02250         }
02251         else goto noaction;
02252         numsub++;

```

```

02253         f = nextf;
02254         NEXTARG(f)
02255     }
02256     C = cbuf+AT.ebufnum;
02257     AT.WorkPointer = l;
02258     *left = l-left;
02259 /*
02260 Put the pointers to the lhs and the rhs in the pointer workspace
02261 */
02262 WantAddPointers(2*numsub);
02263 lhs = AT.pWorkPointer;
02264 rhs = lhs+numsub;
02265 AT.pWorkPointer = rhs+numsub;
02266 f = t + FUNHEAD; l = left+1;
02267 for ( i = 0; i < numsub; i++ ) {
02268     AT.pWorkSpace[lhs+i] = l; l += l[1];
02269     NEXTARG(f);
02270     AT.pWorkSpace[rhs+i] = f;
02271     NEXTARG(f);
02272 }
02273 /*
02274 Take out the patterns and replace them by SUBEXPRESSIONs pointing at
02275 the e buffer. We put the resulting term above the left sides.
02276 Note that we take out only the first id_ if there is more than one!
02277 */
02278 first = 1;
02279 t = term+1; newterm = AT.WorkPointer; out = newterm+1;
02280 while ( t < tstop ) {
02281     if ( *t == IDFUNCTION && first ) { first = 0; t += t[1]; continue; }
02282     if ( *t >= FUNCTION ) {
02283         for ( i = 0; i < numsub; i++ ) {
02284             m = AT.pWorkSpace[lhs+i];
02285             if ( *m != *t ) continue;
02286             for ( j = 1; j < t[1]; j++ ) {
02287                 if ( m[j] != t[j] ) break;
02288             }
02289             if ( j != t[1] ) continue;
02290             numfound++;
02291 /*
02292 We have a match! Set up a SUBEXPRESSION subterm and put the
02293 corresponding rhs in the eBuffer.
02294 */
02295             PutInBuffers(1)
02296             t += t[1];
02297         }
02298         if ( i == numsub ) { /* no match. Just copy to output. */
02299             j = t[1]; NCOPY(out,t,j)
02300         }
02301     }
02302     else if ( *t == SYMBOL ) {
02303         for ( i = 0; i < numsub; i++ ) {
02304             m = AT.pWorkSpace[lhs+i];
02305             if ( *m != SYMBOL ) continue;
02306             for ( ii = 2; ii < t[1]; ii += 2 ) {
02307                 if ( m[2] != t[ii] ) continue;
02308                 pow = t[ii+1]/m[3];
02309                 if ( pow <= 0 ) continue;
02310                 t[ii+1] = t[ii+1]*m[3];
02311                 numfound++;
02312 /*
02313 Create the proper rhs in the eBuffer and set up a
02314 SUBEXPRESSION subterm.
02315 */
02316                 PutInBuffers(pow)
02317             }
02318         }
02319 /*
02320 Now we copy whatever remains of the SYMBOL subterm to the output
02321 */
02322         m = out; *out++ = t[0]; *out++ = t[1];
02323         for ( ii = 2; ii < t[1]; ii += 2 ) {
02324             if ( t[ii+1] ) { *out++ = t[ii]; *out++ = t[ii+1]; }
02325         }
02326         m[1] = out-m;
02327         if ( m[1] == 2 ) out = m;
02328         t += t[1];
02329     }
02330     else if ( *t == DOTPRODUCT ) {
02331         for ( i = 0; i < numsub; i++ ) {
02332             m = AT.pWorkSpace[lhs+i];
02333             if ( *m != DOTPRODUCT ) continue;
02334             for ( ii = 2; ii < t[1]; ii += 3 ) {
02335                 if ( m[2] != t[ii] || m[3] != t[ii+1] ) continue;
02336                 pow = t[ii+2]/m[4];
02337                 if ( pow <= 0 ) continue;
02338                 t[ii+2] = t[ii+2]*m[4];
02339                 numfound++;

```

```

02340 /*
02341             Create the proper rhs in the eBuffer and set up a
02342             SUBEXPRESSION subterm.
02343 */
02344             PutInBuffers(pow)
02345         }
02346     }
02347 /*
02348             Now we copy whatever remains of the DOTPRODUCT subterm to the output
02349 */
02350     m = out; *out++ = t[0]; *out++ = t[1];
02351     for ( ii = 2; ii < t[1]; ii += 3 ) {
02352         if ( t[ii+2] ) { *out++ = t[ii]; *out++ = t[ii+1]; *out++ = t[ii+2]; }
02353     }
02354     m[1] = out-m;
02355     if ( m[1] == 2 ) out = m;
02356     t += t[1];
02357 }
02358 else {
02359     j = t[1]; NCOPY(out,t,j)
02360 }
02361 }
02362 /*
02363             Copy the coefficient and set the size.
02364 */
02365     t = tstop; r = term*term; while ( t < r ) *out++ = *t++;
02366     *newterm = out-newterm;
02367 /*
02368             Finally we move the new term over the original term.
02369 */
02370     i = *newterm;
02371     t = term; r = newterm; NCOPY(t,r,i)
02372 /*
02373             At this point we can return and if the calling Generator jumps back to
02374             its start, TestSub can take care of the expansions of SUBEXPRESSIONs.
02375 */
02376     AT.pWorkPointer = lhs;
02377     AT.WorkPointer = t;
02378     return(numfound);
02379 noaction:
02380     return(0);
02381 }
02382
02383 /*
02384             #] TakeIDfunction :
02385             #] Patterns :
02386 */
02387

```

4.101 poly.cc

```

00001 /* @file poly.cc
00002 *
00003 * Contains the class for representing sparse multivariate
00004 * polynomials with integer coefficients
00005 */
00006 /* #] License : */
00007 /*
00008 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00009 * When using this file you are requested to refer to the publication
00010 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011 * This is considered a matter of courtesy as the development was paid
00012 * for by FOM the Dutch physics granting agency and we would like to
00013 * be able to track its scientific use to convince FOM of its value
00014 * for the community.
00015 *
00016 * This file is part of FORM.
00017 *
00018 * FORM is free software: you can redistribute it and/or modify it under the
00019 * terms of the GNU General Public License as published by the Free Software
00020 * Foundation, either version 3 of the License, or (at your option) any later
00021 * version.
00022 *
00023 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00026 * details.
00027 *
00028 * You should have received a copy of the GNU General Public License along
00029 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00030 */
00031 /* #] License : */
00032 /*

```

```

00033     #[ includes :
00034 */
00035
00036 #include "poly.h"
00037 #include "polygcd.h"
00038
00039 #include <cassert>
00040 #include <cctype>
00041 #include <cmath>
00042 #include <algorithm>
00043 #include <map>
00044 #include <iostream>
00045
00046 using namespace std;
00047
00048 /*
00049     #] includes :
00050     #[ constructor (small constant polynomial) :
00051 */
00052
00053 // constructor for a small constant polynomial
00054 poly::poly (PHEAD int a, WORD modp, WORD modn):
00055     size_of_terms(AM.MaxTer/(LONG) sizeof(WORD)),
00056     modp(0),
00057     modn(1)
00058 {
00059     POLY_STOREIDENTITY;
00060
00061     terms = TermMalloc("poly::poly(int)");
00062
00063     if (a == 0) {
00064         terms[0] = 1; // length
00065     }
00066     else {
00067         terms[0] = 4 + AN.poly_num_vars; // length
00068         terms[1] = 3 + AN.poly_num_vars; // length
00069         for (int i=0; i<AN.poly_num_vars; i++) terms[2+i] = 0; // powers
00070         terms[2+AN.poly_num_vars] = ABS(a); // coefficient
00071         terms[3+AN.poly_num_vars] = a>0 ? 1 : -1; // length coefficient
00072     }
00073
00074     if (modp!=-1) setmod(modp,modn);
00075 }
00076
00077 /*
00078     #] constructor (small constant polynomial) :
00079     #[ constructor (large constant polynomial) :
00080 */
00081
00082 // constructor for a large constant polynomial
00083 poly::poly (PHEAD const UWORD *a, WORD na, WORD modp, WORD modn):
00084     size_of_terms(AM.MaxTer/(LONG) sizeof(WORD)),
00085     modp(0),
00086     modn(1)
00087 {
00088     POLY_STOREIDENTITY;
00089
00090     terms = TermMalloc("poly::poly(UWORD*,WORD)");
00091
00092     // remove leading zeros
00093     while (*(a+ABS(na)-1)==0)
00094         na -= SGN(na);
00095
00096     terms[0] = 3 + AN.poly_num_vars + ABS(na); // length
00097     terms[1] = terms[0] - 1; // length
00098     for (int i=0; i<AN.poly_num_vars; i++) terms[2+i] = 0; // powers
00099     termscopy((WORD *)a, 2+AN.poly_num_vars, ABS(na)); // coefficient
00100     terms[2+AN.poly_num_vars+ABS(na)] = na; // length coefficient
00101
00102     if (modp!=-1) setmod(modp,modn);
00103 }
00104
00105 /*
00106     #] constructor (large constant polynomial) :
00107     #[ constructor (deep copy polynomial) :
00108 */
00109
00110 // copy constructor: makes a deep copy of a polynomial
00111 poly::poly (const poly &a, WORD modp, WORD modn):
00112     size_of_terms(AM.MaxTer/(LONG) sizeof(WORD))
00113 {
00114     #ifdef POLY_MOVE_DEBUG
00115         std::cout << "poly copy ctor" << std::endl;
00116     #endif
00117     POLY_GETIDENTITY(a);
00118     POLY_STOREIDENTITY;
00119 }

```

```

00120     terms = TermMalloc("poly::poly(poly&)");
00121
00122     *this = a;
00123
00124     if (modp!= -1) setmod(modp,modn);
00125 }
00126
00127 /*
00128  #] constructor (deep copy polynomial) :
00129  #[ destructor :
00130  */
00131
00132 // destructor
00133 poly::~poly () {
00134
00135     POLY_GETIDENTITY(*this);
00136
00137     if (size_of_terms == AM.MaxTer/(LONG)sizeof(WORD))
00138         TermFree(terms, "poly::~poly");
00139     else
00140         M_free(terms, "poly::~poly");
00141 }
00142
00143 /*
00144  #] destructor :
00145  #[ expand_memory :
00146  */
00147
00148 // expands the memory allocated for terms to at least twice its size
00149 void poly::expand_memory (int i) {
00150
00151     POLY_GETIDENTITY(*this);
00152
00153     LONG new_size_of_terms = MaX(2*size_of_terms, i);
00154     if (new_size_of_terms > MAXPOSITIVE) {
00155         MLOCK(ErrorMessageLock);
00156         MesPrint ((char*)"ERROR: polynomials too large (> WORDSIZE)");
00157         MUNLOCK(ErrorMessageLock);
00158         Terminate(1);
00159     }
00160
00161     WORD *newterms = (WORD *)Malloc1(new_size_of_terms * sizeof(WORD), "poly::expand_memory");
00162     WCOPY(newterms, terms, size_of_terms);
00163
00164     if (size_of_terms == AM.MaxTer/(LONG)sizeof(WORD))
00165         TermFree(terms, "poly::expand_memory");
00166     else
00167         M_free(terms, "poly::expand_memory");
00168
00169     terms = newterms;
00170     size_of_terms = new_size_of_terms;
00171 }
00172
00173 /*
00174  #] expand_memory :
00175  #[ setmod :
00176  */
00177
00178 // sets the coefficient space to ZZ/p^n
00179 void poly::setmod(WORD _modp, WORD _modn) {
00180
00181     POLY_GETIDENTITY(*this);
00182
00183     if (_modp>0 && (_modp!=modp || _modn<modn)) {
00184         modp = _modp;
00185         modn = _modn;
00186
00187         WORD nmodq=0;
00188         UWORD *modq=NULL;
00189         bool smallq;
00190
00191         if (modn == 1) {
00192             modq = (UWORD *)&modp;
00193             nmodq = 1;
00194             smallq = true;
00195         }
00196         else {
00197             RaisPowCached(BHEAD modp,modn,&modq,&nmodq);
00198             smallq = false;
00199         }
00200
00201         coefficients_modulo(modq,nmodq,smallq);
00202     }
00203     else {
00204         modp = _modp;
00205         modn = _modn;
00206     }

```

```

00207 }
00208
00209 /*
00210     #] setmod :
00211     #[ coefficients_modulo :
00212 */
00213
00214 // reduces all coefficients of the polynomial modulo a
00215 void poly::coefficients_modulo (UWORD *a, WORD na, bool small) {
00216     POLY_GETIDENTITY(*this);
00217
00218     int j=1;
00219     for (int i=1, di; i<terms[0]; i+=di) {
00220         di = terms[i];
00221
00222         if (i!=j)
00223             for (int k=0; k<di; k++)
00224                 terms[j+k] = terms[i+k];
00225
00226         WORD n = terms[j+terms[j]-1];
00227
00228         if (ABS(n)==1 && small) {
00229             // small number modulo small number
00230             terms[j+AN.poly_num_vars] = n*((UWORD)terms[j+1+AN.poly_num_vars] % a[0]);
00231             if (terms[j+1+AN.poly_num_vars] == 0)
00232                 n=0;
00233             else {
00234                 if (terms[j+1+AN.poly_num_vars] > +(WORD)a[0]/2) terms[j+1+AN.poly_num_vars]-=a[0];
00235                 if (terms[j+1+AN.poly_num_vars] < -(WORD)a[0]/2) terms[j+1+AN.poly_num_vars]+=a[0];
00236                 n = SGN(terms[j+1+AN.poly_num_vars]);
00237                 terms[j+1+AN.poly_num_vars] = ABS(terms[j+1+AN.poly_num_vars]);
00238             }
00239         }
00240         else
00241             TakeNormalModulus((UWORD *)&terms[j+1+AN.poly_num_vars], &n, a, na, NOUNPACK);
00242
00243         if (n!=0) {
00244             terms[j] = 2+AN.poly_num_vars+ABS(n);
00245             terms[j+terms[j]-1] = n;
00246             j += terms[j];
00247         }
00248     }
00249     terms[0] = j;
00250 }
00251
00252 }
00253
00254 /*
00255     #] coefficients_modulo :
00256     #[ to_string :
00257 */
00258
00259 // converts an integer to a string
00260 const string int_to_string (WORD x) {
00261     char res[41];
00262     snprintf (res, 41, "%i",x);
00263     return res;
00264 }
00265
00266 // converts a polynomial to a string
00267 const string poly::to_string() const {
00268     POLY_GETIDENTITY(*this);
00269
00270     string res;
00271
00272     int printtimes;
00273     UBYTE *scoeff = (UBYTE *)NumberMalloc("poly::to_string");
00274
00275     if (terms[0]==1)
00276         // zero
00277         res = "0";
00278     else {
00279         for (int i=1; i<terms[0]; i+=terms[i]) {
00280             // sign
00281             WORD ncoeff = terms[i+terms[i]-1];
00282             if (ncoeff < 0) {
00283                 ncoeff*=-1;
00284                 res += "-";
00285             }
00286             else {
00287                 if (i>1) res += "+";
00288             }
00289
00290             if (ncoeff==1 && terms[i+terms[i]-1-ncoeff]==1) {
00291                 // coeff=1, so don't print coefficient and '*'

```

```

00294         printtimes = 0;
00295     }
00296     else {
00297         // print coefficient
00298         PRTLong((UWORD*)&terms[i+terms[i]-1-ncoeff], ncoeff, scoeff);
00299         res += string((char *)scoeff);
00300         printtimes=1;
00301     }
00302
00303     // print variables
00304     for (int j=0; j<AN.poly_num_vars; j++) {
00305         if (terms[i+1+j] > 0) {
00306             if (printtimes) res += "*";
00307             res += string(1,'a'+j);
00308             if (terms[i+1+j] > 1) res += "^" + int_to_string(terms[i+1+j]);
00309             printtimes = 1;
00310         }
00311     }
00312
00313     // iff coeff=1 and all power=0, print '1' after all
00314     if (!printtimes) res += "1";
00315 }
00316 }
00317
00318 // eventual modulo
00319 if (modp>0) {
00320     res += " (mod ";
00321     res += int_to_string(modp);
00322     if (modn>1) {
00323         res += "^";
00324         res += int_to_string(modn);
00325     }
00326     res += ")";
00327 }
00328
00329 NumberFree(scoeff,"poly::to_string");
00330
00331 return res;
00332 }
00333
00334 /*
00335 #] to_string :
00336 #[ ostream operator :
00337 */
00338
00339 // output stream operator
00340 ostream& operator<< (ostream &out, const poly &a) {
00341     return out << a.to_string();
00342 }
00343
00344 /*
00345 #] ostream operator :
00346 #[ monomial_compare :
00347 */
00348
00349 // compares two monomials (0:equal, <0:a smaller, >0:b smaller)
00350 int poly::monomial_compare (PHEAD const WORD *a, const WORD *b) {
00351     for (int i=0; i<AN.poly_num_vars; i++)
00352         if (a[i+1]!=b[i+1]) return a[i+1]-b[i+1];
00353     return 0;
00354 }
00355
00356 /*
00357 #] monomial_compare :
00358 #[ normalize :
00359 */
00360
00361 // brings a polynomial to normal form
00362 const poly & poly::normalize() {
00363
00364     POLY_GETIDENTITY(*this);
00365
00366     WORD nmodq=0;
00367     UWORD *modq=NULL;
00368
00369     if (modp!=0) {
00370         if (modn == 1) {
00371             modq = (UWORD *)&modp;
00372             nmodq = 1;
00373         }
00374         else {
00375             RaisPowCached(BHEAD modp, modn, &modq, &nmodq);
00376         }
00377     }
00378
00379     // find and sort all monomials
00380     // terms[0]/num_vars+3 is an upper bound for number of terms in a

```

```

00381
00382     LONG poffset = AT.pWorkPointer;
00383     WantAddPointers((terms[0]/(AN.poly_num_vars+3)));
00384     AT.pWorkPointer += terms[0]/(AN.poly_num_vars+3);
00385     #define p (&(AT.pWorkspace[poffset]))
00386
00387 // WORD **p = (WORD **)Malloc1(terms[0]/(AN.poly_num_vars+3) * sizeof(WORD*), "poly::normalize");
00388
00389     int nterms = 0;
00390     for (int i=1; i<terms[0]; i+=terms[i])
00391         p[nterms++] = terms + i;
00392
00393     sort(p, p + nterms, monomial_larger(BHEAD0));
00394
00395     WORD *tmp;
00396     if (size_of_terms == AM.MaxTer/(LONG)sizeof(WORD)) {
00397         tmp = (WORD *)TermMalloc("poly::normalize");
00398     }
00399     else {
00400         tmp = (WORD *)Malloc1(size_of_terms * sizeof(WORD), "poly::normalize");
00401     }
00402     int j=1;
00403     int prevj=0;
00404     tmp[0]=0;
00405     tmp[1]=0;
00406
00407     for (int i=0; i<nterms; i++) {
00408         if (i>0 && monomial_compare(BHEAD &tmp[j], p[i])==0) {
00409             // duplicate term, so add coefficients
00410             WORD ncoeff = tmp[j+tmp[j]-1];
00411             AddLong((UWORD *)&tmp[j+1+AN.poly_num_vars], ncoeff,
00412                 (UWORD *)&p[i][1+AN.poly_num_vars], p[i][p[i][0]-1],
00413                 (UWORD *)&tmp[j+1+AN.poly_num_vars], &ncoeff);
00414
00415             tmp[j+1+AN.poly_num_vars+ABS(ncoeff)] = ncoeff;
00416             tmp[j] = 2+AN.poly_num_vars+ABS(ncoeff);
00417         }
00418         else {
00419             // new term
00420             prevj = j;
00421             j += tmp[j];
00422             WCOPY(&tmp[j],p[i],p[i][0]);
00423         }
00424
00425         if (modp!=0) {
00426             // bring coefficient to normal form mod p^n
00427             WORD ntmp = tmp[j+tmp[j]-1];
00428             TakeNormalModulus((UWORD *)&tmp[j+1+AN.poly_num_vars], &ntmp,
00429                 modq,nmodq, NOUNPACK);
00430             tmp[j] = 2+AN.poly_num_vars+ABS(ntmp);
00431             tmp[j+tmp[j]-1] = ntmp;
00432         }
00433
00434         // add terms to polynomial
00435         if (tmp[j+tmp[j]-1]==0) {
00436             tmp[j]=0;
00437             j=prevj;
00438         }
00439     }
00440
00441     j+=tmp[j];
00442
00443     tmp[0] = j;
00444     WCOPY(terms,tmp,tmp[0]);
00445
00446 // M_free(p, "poly::normalize");
00447
00448     if (size_of_terms == AM.MaxTer/(LONG)sizeof(WORD))
00449         TermFree(tmp, "poly::normalize");
00450     else
00451         M_free(tmp, "poly::normalize");
00452
00453     AT.pWorkPointer = poffset;
00454     #undef p
00455
00456     return *this;
00457 }
00458
00459 /*
00460 #] normalize :
00461 #[ last_monomial_index :
00462 */
00463
00464 // index of the last monomial, i.e., the constant term
00465 WORD poly::last_monomial_index () const {
00466     POLY_GETIDENTITY(*this);
00467     return terms[0] - ABS(terms[terms[0]-1]) - AN.poly_num_vars - 2;

```



```

00468 }
00469
00470
00471 // pointer to the last monomial (the constant term)
00472 WORD * poly::last_monomial () const {
00473     return &terms[last_monomial_index()];
00474 }
00475
00476 /*
00477     #] last_monomial_index :
00478     #[ compare_degree_vector :
00479 */
00480
00481 int poly::compare_degree_vector(const poly & b) const {
00482     POLY_GETIDENTITY(*this);
00483
00484     // special cases if one or both are 0
00485     if (terms[0] == 1 && b[0] == 1) return 0;
00486     if (terms[0] == 1) return -1;
00487     if (b[0] == 1) return 1;
00488
00489     for (int i = 0; i < AN.poly_num_vars; i++) {
00490         int d1 = degree(i);
00491         int d2 = b.degree(i);
00492         if (d1 != d2) return d1 - d2;
00493     }
00494
00495     return 0;
00496 }
00497
00498 /*
00499     #] compare_degree_vector :
00500     #[ degree_vector :
00501 */
00502
00503 std::vector<int> poly::degree_vector() const {
00504     POLY_GETIDENTITY(*this);
00505
00506     std::vector<int> degrees(AN.poly_num_vars);
00507     for (int i = 0; i < AN.poly_num_vars; i++) {
00508         degrees[i] = degree(i);
00509     }
00510
00511     return degrees;
00512 }
00513
00514 /*
00515     #] degree_vector :
00516     #[ compare_degree_vector :
00517 */
00518
00519 int poly::compare_degree_vector(const vector<int> & b) const {
00520     POLY_GETIDENTITY(*this);
00521
00522     if (terms[0] == 1) return -1;
00523
00524     for (int i = 0; i < AN.poly_num_vars; i++) {
00525         int d1 = degree(i);
00526         if (d1 != b[i]) return d1 - b[i];
00527     }
00528
00529     return 0;
00530 }
00531
00532 /*
00533     #] compare_degree_vector :
00534     #[ add :
00535 */
00536
00537 // addition of polynomials by merging
00538 void poly::add (const poly &a, const poly &b, poly &c) {
00539     POLY_GETIDENTITY(a);
00540
00541     c.modp = a.modp;
00542     c.modn = a.modn;
00543
00544     WORD nmodq=0;
00545     UWORD *modq=NULL;
00546
00547     bool both_mod_small=false;
00548
00549     if (c.modp!=0) {
00550         if (c.modn == 1) {
00551             modq = (UWORD *)&c.modp;
00552             nmodq = 1;
00553             if (a.modp>0 && b.modp>0 && a.modn==1 && b.modn==1)

```

```

00555         both_mod_small=true;
00556     }
00557     else {
00558         RaisPowCached(BHEAD c.modp, c.modn, &modq, &nmodq);
00559     }
00560 }
00561
00562 int ai=1, bi=1, ci=1;
00563
00564 while (ai<a[0] || bi<b[0]) {
00565
00566     c.check_memory(ci);
00567
00568     int cmp = ai<a[0] && bi<b[0] ? monomial_compare(BHEAD &a[ai], &b[bi]) : 0;
00569
00570     if (bi==b[0] || cmp>0) {
00571         // insert term from a
00572         c.termscopy(&a[ai], ci, a[ai]);
00573         ci+=a[ai];
00574         ai+=a[ai];
00575     }
00576     else if (ai==a[0] || cmp<0) {
00577         // insert term from b
00578         c.termscopy(&b[bi], ci, b[bi]);
00579         ci+=b[bi];
00580         bi+=b[bi];
00581     }
00582     else {
00583         // insert term from a+b
00584         c.termscopy(&a[ai], ci, MaX(a[ai], b[bi]));
00585         WORD nc=0;
00586         if (both_mod_small) {
00587             c[ci+1+AN.poly_num_vars] = ((LONG)a[ai+1+AN.poly_num_vars]*a[ai+a[ai]-1]+
00588 (LONG)b[bi+1+AN.poly_num_vars]*b[bi+b[bi]-1]) % c.modp;
00589             if ((WORD)c[ci+1+AN.poly_num_vars] > +c.modp/2) c[ci+1+AN.poly_num_vars] -= c.modp;
00590             if ((WORD)c[ci+1+AN.poly_num_vars] < -c.modp/2) c[ci+1+AN.poly_num_vars] += c.modp;
00591             nc = (c[ci+1+AN.poly_num_vars]==0 ? 0 : SGN((WORD)c[ci+1+AN.poly_num_vars]));
00592             c[ci+1+AN.poly_num_vars] = ABS((WORD)c[ci+1+AN.poly_num_vars]);
00593         }
00594         else {
00595             AddLong((UWORD *)&a[ai+1+AN.poly_num_vars], a[ai+a[ai]-1],
00596 (UWORD *)&b[bi+1+AN.poly_num_vars], b[bi+b[bi]-1],
00597 (UWORD *)&c[ci+1+AN.poly_num_vars], &nc);
00598             if (c.modp!=0) TakeNormalModulus((UWORD *)&c[ci+1+AN.poly_num_vars], &nc,
00599 modq, nmodq,
00600 NOUNPACK);
00601         }
00602
00603         if (nc!=0) {
00604             c[ci] = 2+AN.poly_num_vars+ABS(nc);
00605             c[ci+c[ci]-1] = nc;
00606             ci += c[ci];
00607         }
00608
00609         ai+=a[ai];
00610         bi+=b[bi];
00611     }
00612 }
00613 c[0]=ci;
00614 }
00615
00616 /*
00617 #] add :
00618 #[ sub :
00619 */
00620
00621 // subtraction of polynomials by merging
00622 void poly::sub (const poly &a, const poly &b, poly &c) {
00623
00624     POLY_GETIDENTITY(a);
00625
00626     c.modp = a.modp;
00627     c.modn = a.modn;
00628
00629     WORD nmodq=0;
00630     UWORD *modq=NULL;
00631
00632     bool both_mod_small=false;
00633
00634     if (c.modp!=0) {
00635         if (c.modn == 1) {
00636             modq = (UWORD *)&c.modp;
00637             nmodq = 1;
00638             if (a.modp>0 && b.modp>0 && a.modn==1 && b.modn==1)

```

```

00639         both_mod_small=true;
00640     }
00641     else {
00642         RaisPowCached(BHEAD c.modp, c.modn, &modq, &nmodq);
00643     }
00644 }
00645
00646 int ai=1, bi=1, ci=1;
00647
00648 while (ai<a[0] || bi<b[0]) {
00649     c.check_memory(ci);
00650
00651     int cmp = ai<a[0] && bi<b[0] ? monomial_compare(BHEAD &a[ai], &b[bi]) : 0;
00652
00653     if (bi==b[0] || cmp>0) {
00654         // insert term from a
00655         c.termscopy(&a[ai], ci, a[ai]);
00656         ci+=a[ai];
00657         ai+=a[ai];
00658     }
00659     else if (ai==a[0] || cmp<0) {
00660         // insert term from b
00661         c.termscopy(&b[bi], ci, b[bi]);
00662         ci+=b[bi];
00663         bi+=b[bi];
00664         c[ci-1]*=-1;
00665     }
00666     else {
00667         // insert term from a+b
00668         c.termscopy(&a[ai], ci, MaX(a[ai], b[bi]));
00669         WORD nc=0;
00670         if (both_mod_small) {
00671             c[ci+1+AN.poly_num_vars] = ((LONG)a[ai+1+AN.poly_num_vars]*a[ai+a[ai]-1]-
00672 (LONG)b[bi+1+AN.poly_num_vars]*b[bi+b[bi]-1]) % c.modp;
00673             if ((WORD)c[ci+1+AN.poly_num_vars] > +c.modp/2) c[ci+1+AN.poly_num_vars] -= c.modp;
00674             if ((WORD)c[ci+1+AN.poly_num_vars] < -c.modp/2) c[ci+1+AN.poly_num_vars] += c.modp;
00675             nc = (c[ci+1+AN.poly_num_vars]==0 ? 0 : SGN((WORD)c[ci+1+AN.poly_num_vars]));
00676             c[ci+1+AN.poly_num_vars] = ABS((WORD)c[ci+1+AN.poly_num_vars]);
00677         }
00678         else {
00679             AddLong((UWORD *)&a[ai+1+AN.poly_num_vars], a[ai+a[ai]-1],
00680 (UWORD *)&b[bi+1+AN.poly_num_vars], -b[bi+b[bi]-1], // -b[...] causes
00681 subtraction
00682 (UWORD *)&c[ci+1+AN.poly_num_vars], &nc);
00683             if (c.modp!=0) TakeNormalModulus((UWORD *)&c[ci+1+AN.poly_num_vars], &nc,
00684 modq, nmodq,
00685 NOUNPACK);
00686         }
00687         if (nc!=0) {
00688             c[ci] = 2+AN.poly_num_vars+ABS(nc);
00689             c[ci+c[ci]-1] = nc;
00690             ci += c[ci];
00691         }
00692         ai+=a[ai];
00693         bi+=b[bi];
00694     }
00695 }
00696 }
00697
00698 c[0]=ci;
00699 }
00700
00701 /*
00702 #] sub :
00703 #[ pop_heap :
00704 */
00705
00706 // pops the largest monomial from the heap and stores it in heap[n]
00707 void poly::pop_heap (PHEAD WORD **heap, int n) {
00708     WORD *old = heap[0];
00709
00710     heap[0] = heap[--n];
00711
00712     int i=0;
00713     while (2*i+2<n && (monomial_compare(BHEAD heap[2*i+1]+3, heap[i]+3)>0 ||
00714 monomial_compare(BHEAD heap[2*i+2]+3, heap[i]+3)>0)) {
00715         if (monomial_compare(BHEAD heap[2*i+1]+3, heap[2*i+2]+3)>0) {
00716             swap(heap[i], heap[2*i+1]);
00717             i=2*i+1;
00718         }
00719         else {
00720

```

```

00722         swap(heap[i], heap[2*i+2]);
00723         i=2*i+2;
00724     }
00725 }
00726
00727 if (2*i+1<n && monomial_compare(BHEAD heap[2*i+1]+3, heap[i]+3)>0)
00728     swap(heap[i], heap[2*i+1]);
00729
00730 heap[n] = old;
00731 }
00732
00733 /*
00734  #] pop_heap :
00735  #[ push_heap :
00736 */
00737
00738 // pushes the monomial in heap[n] onto the heap
00739 void poly::push_heap (PHEAD WORD **heap, int n) {
00740
00741     int i=n-1;
00742
00743     while (i>0 && monomial_compare(BHEAD heap[i]+3, heap[(i-1)/2]+3) > 0) {
00744         swap(heap[(i-1)/2], heap[i]);
00745         i=(i-1)/2;
00746     }
00747 }
00748
00749 /*
00750  #] push_heap :
00751  #[ mul_one_term :
00752 */
00753
00754 // multiplies a polynomial with a monomial
00755 void poly::mul_one_term (const poly &a, const poly &b, poly &c) {
00756
00757     POLY_GETIDENTITY(a);
00758
00759     int ci=1;
00760
00761     WORD nmodq=0;
00762     UWORD *modq=NULL;
00763
00764     bool both_mod_small=false;
00765
00766     if (c.modp!=0) {
00767         if (c.modn == 1) {
00768             modq = (UWORD *)&c.modp;
00769             nmodq = 1;
00770             if (a.modp>0 && b.modp>0 && a.modn==1 && b.modn==1)
00771                 both_mod_small=true;
00772         }
00773         else {
00774             RaisPowCached(BHEAD c.modp, c.modn, &modq, &nmodq);
00775         }
00776     }
00777
00778     for (int ai=1; ai<a[0]; ai+=a[ai])
00779         for (int bi=1; bi<b[0]; bi+=b[bi]) {
00780
00781             c.check_memory(ci);
00782
00783             for (int i=0; i<AN.poly_num_vars; i++)
00784                 c[ci+1+i] = a[ai+1+i] + b[bi+1+i];
00785             WORD nc;
00786
00787             // if both polynomials are modulo p^1, use integer calculus
00788             if (both_mod_small) {
00789                 c[ci+1+AN.poly_num_vars] =
00790                     ((LONG)a[ai+1+AN.poly_num_vars] * a[ai+2+AN.poly_num_vars] *
00791                      b[bi+1+AN.poly_num_vars] * b[bi+2+AN.poly_num_vars]) % c.modp;
00792                 nc = (c[ci+1+AN.poly_num_vars]==0 ? 0 : 1);
00793             }
00794             else {
00795                 // otherwise, use form long calculus
00796                 MulLong((UWORD *)&a[ai+1+AN.poly_num_vars], a[ai+a[ai]-1],
00797                        (UWORD *)&b[bi+1+AN.poly_num_vars], b[bi+b[bi]-1],
00798                        (UWORD *)&c[ci+1+AN.poly_num_vars], &nc);
00799                 if (c.modp!=0) TakeNormalModulus((UWORD *)&c[ci+1+AN.poly_num_vars], &nc,
00800                                                  modq, nmodq,
00801
00802 NOUNPACK);
00803             }
00804
00805             if (nc!=0) {
00806                 c[ci] = 2+AN.poly_num_vars+ABS(nc);
00807                 ci += c[ci];
00808
00809                 if (both_mod_small) {

```

```

00808             if (c[ci-2] > +c.modp/2) c[ci-2] -= c.modp;
00809             if (c[ci-2] < -c.modp/2) c[ci-2] += c.modp;
00810             c[ci-1] = SGN(c[ci-2]);
00811             c[ci-2] = ABS(c[ci-2]);
00812         }
00813         else {
00814             c[ci-1] = nc;
00815         }
00816     }
00817 }
00818
00819     c[0]=ci;
00820 }
00821
00822 /*
00823     #] mul_one_term :
00824     #[ mul_univar :
00825 */
00826
00827 // dense univariate multiplication, i.e., for each power find all
00828 // pairs of monomials that result in that power
00829 void poly::mul_univar (const poly &a, const poly &b, poly &c, int var) {
00830
00831     POLY_GETIDENTITY(a);
00832
00833     WORD nmodq=0;
00834     UWORD *modq=NULL;
00835
00836     bool both_mod_small=false;
00837
00838     if (c.modp!=0) {
00839         if (c.modn == 1) {
00840             modq = (UWORD *)&c.modp;
00841             nmodq = 1;
00842             if (a.modp>0 && b.modp>0 && a.modn==1 && b.modn==1)
00843                 both_mod_small=true;
00844         }
00845         else {
00846             RaisPowCached(BHEAD c.modp,c.modn,&modq,&nmodq);
00847         }
00848     }
00849
00850     poly t(BHEAD 0);
00851     WORD nt;
00852
00853     int ci=1;
00854
00855     // bounds on the powers in a*b
00856     int minpow = AN.poly_num_vars==0 ? 0 : a.last_monomial()[1+var] + b.last_monomial()[1+var];
00857     int maxpow = AN.poly_num_vars==0 ? 0 : a[2+var]+b[2+var];
00858     int afirst=1, blast=1;
00859
00860     for (int pow=maxpow; pow>=minpow; pow--) {
00861
00862         c.check_memory(ci);
00863
00864         WORD nc=0;
00865
00866         // adjust range in a or b
00867         if (a[afirst+1+var] + b[blast+1+var] > pow) {
00868             if (blast+b[blast] < b[0])
00869                 blast+=b[blast];
00870             else
00871                 afirst+=a[afirst];
00872         }
00873
00874         // find terms that result in the correct power
00875         for (int ai=afirst, bi=blast; ai<a[0] && bi>=1;) {
00876
00877             int thispow = AN.poly_num_vars==0 ? 0 : a[ai+1+var] + b[bi+1+var];
00878
00879             if (thispow == pow) {
00880
00881                 // if both polynomials are modulo p^1, use integer calculus
00882                 if (both_mod_small) {
00883                     c[ci+1+AN.poly_num_vars] =
00884                         ((nc==0 ? 0 : (LONG)c[ci+1+AN.poly_num_vars] * nc) +
00885                         (LONG)a[ai+1+AN.poly_num_vars] * a[ai+2+AN.poly_num_vars] *
00886                         b[bi+1+AN.poly_num_vars] * b[bi+2+AN.poly_num_vars]) % c.modp;
00887                     nc = (c[ci+1+AN.poly_num_vars]==0 ? 0 : 1);
00888                 }
00889                 else {
00890                     // otherwise, use form long calculus
00891
00892                     MulLong((UWORD *)&a[ai+1+AN.poly_num_vars], a[ai+a[ai]-1],
00893                             (UWORD *)&b[bi+1+AN.poly_num_vars], b[bi+b[bi]-1],
00894                             (UWORD *)&t[0], &nt);

```

```

00895
00896         AddLong ((UWORD *)&t[0], nt,
00897                 (UWORD *)&c[ci+1+AN.poly_num_vars], nc,
00898                 (UWORD *)&c[ci+1+AN.poly_num_vars], &nc);
00899
00900         if (c.modp!=0) TakeNormalModulus((UWORD *)&c[ci+1+AN.poly_num_vars], &nc,
00901                                         modq, nmodq,
NOUNPACK);
00902     }
00903
00904     ai += a[ai];
00905     bi -= ABS(b[bi-1]) + 2 + AN.poly_num_vars;
00906 }
00907 else if (thispow > pow)
00908     ai += a[ai];
00909 else
00910     bi -= ABS(b[bi-1]) + 2 + AN.poly_num_vars;
00911 }
00912
00913 // add term to result
00914 if (nc != 0) {
00915     for (int j=0; j<AN.poly_num_vars; j++)
00916         c[ci+1+j] = 0;
00917     if (AN.poly_num_vars > 0)
00918         c[ci+1+var] = pow;
00919
00920     c[ci] = 2+AN.poly_num_vars+ABS(nc);
00921     ci += c[ci];
00922
00923     // if necessary, adjust to range [-p/2,p/2]
00924     if (both_mod_small) {
00925         if (c[ci-2] > +c.modp/2) c[ci-2] -= c.modp;
00926         if (c[ci-2] < -c.modp/2) c[ci-2] += c.modp;
00927         c[ci-1] = SGN(c[ci-2]);
00928         c[ci-2] = ABS(c[ci-2]);
00929     }
00930     else {
00931         c[ci-1] = nc;
00932     }
00933 }
00934 }
00935
00936 c[0] = ci;
00937 }
00938
00939 /*
00940 #] mul_univar :
00941 #[ mul_heap :
00942 */
00943
00944 void poly::mul_heap (const poly &a, const poly &b, poly &c) {
00945
00946     POLY_GETIDENTITY(a);
00947
00948     WORD nmodq=0;
00949     UWORD *modq=NULL;
00950
00951     bool both_mod_small=false;
00952
00953     if (c.modp!=0) {
00954         if (c.modn == 1) {
00955             modq = (UWORD *)&c.modp;
00956             nmodq = 1;
00957             if (a.modp>0 && b.modp>0 && a.modn==1 && b.modn==1)
00958                 both_mod_small=true;
00959         }
00960         else {
00961             RaisPowCached(BHEAD c.modp, c.modn, &modq, &nmodq);
00962         }
00963     }
00964
00965     // find maximum powers in different variables
00966     WORD *maxpower = AT.WorkPointer;
00967     AT.WorkPointer += AN.poly_num_vars;
00968     WORD *maxpowera = AT.WorkPointer;
00969     AT.WorkPointer += AN.poly_num_vars;
00970     WORD *maxpowerb = AT.WorkPointer;
00971     AT.WorkPointer += AN.poly_num_vars;
00972
00973     for (int i=0; i<AN.poly_num_vars; i++)
00974         maxpowera[i] = maxpowerb[i] = 0;
00975
00976     for (int ai=1; ai<a[0]; ai+=a[ai])
00977         for (int j=0; j<AN.poly_num_vars; j++)
00978             maxpowera[j] = MaX(maxpowera[j], a[ai+1+j]);
00979
00980     for (int bi=1; bi<b[0]; bi+=b[bi])

```

```

00998         for (int j=0; j<AN.poly_num_vars; j++)
00999             maxpowerb[j] = MaX(maxpowerb[j], b[bi+1+j]);
01000
01001     for (int i=0; i<AN.poly_num_vars; i++)
01002         maxpower[i] = maxpowera[i] + maxpowerb[i];
01003
01004     // if PROD(maxpower) small, use hash array
01005     bool use_hash = true;
01006     int nhash = 1;
01007
01008     for (int i=0; i<AN.poly_num_vars; i++) {
01009         if (nhash > POLY_MAX_HASH_SIZE / (maxpower[i]+1)) {
01010             nhash = 1;
01011             use_hash = false;
01012             break;
01013         }
01014         nhash *= maxpower[i]+1;
01015     }
01016
01017     // allocate heap and hash
01018     int nheap=a.number_of_terms();
01019
01020     WantAddPointers(nheap+nhash);
01021     WORD **heap = AT.pWorkSpace + AT.pWorkPointer;
01022
01023     for (int ai=1, i=0; ai<a[0]; ai+=a[ai], i++) {
01024         heap[i] = (WORD *) NumberMalloc("poly::mul_heap");
01025         heap[i][0] = ai;
01026         heap[i][1] = 1;
01027         heap[i][2] = -1;
01028         heap[i][3] = 0;
01029         for (int j=0; j<AN.poly_num_vars; j++)
01030             heap[i][4+j] = MAXPOSITIVE;
01031     }
01032
01033     WORD **hash = AT.pWorkSpace + AT.pWorkPointer + nheap;
01034     for (int i=0; i<nhash; i++)
01035         hash[i] = NULL;
01036
01037     int ci = 1;
01038
01039     // multiply
01040     while (nheap > 0) {
01041
01042         c.check_memory(ci);
01043
01044         pop_heap(BHEAD heap, nheap--);
01045         WORD *p = heap[nheap];
01046
01047         // if non-zero
01048         if (p[3] != 0) {
01049             if (use_hash) hash[p[2]] = NULL;
01050
01051             c[0] = ci;
01052
01053             // append this term to the result
01054             if (use_hash || ci==1 || monomial_compare(BHEAD p+3, c.last_monomial())!=0) {
01055                 p[4 + AN.poly_num_vars + ABS(p[3])] = p[3];
01056                 p[3] = 2 + AN.poly_num_vars + ABS(p[3]);
01057                 c.termscopy(&p[3],ci,p[3]);
01058                 ci += c[ci];
01059             }
01060             else {
01061                 // add this term to the last term of the result
01062                 ci = c.last_monomial_index();
01063                 WORD nc = c[ci+c[ci]-1];
01064
01065                 // if both polynomials are modulo p^1, use integer calculus
01066                 if (both_mod_small) {
01067                     c[ci+AN.poly_num_vars+1] = ((LONG)c[ci+AN.poly_num_vars+1]*nc +
01068 p[4+AN.poly_num_vars]*p[3]) % c.modp;
01069                     if (c[ci+1+AN.poly_num_vars]==0)
01070                         nc = 0;
01071                     else {
01072                         if (c[ci+1+AN.poly_num_vars] > +c.modp/2) c[ci+1+AN.poly_num_vars] -= c.modp;
01073                         if (c[ci+1+AN.poly_num_vars] < -c.modp/2) c[ci+1+AN.poly_num_vars] += c.modp;
01074                         nc = SGN(c[ci+1+AN.poly_num_vars]);
01075                         c[ci+1+AN.poly_num_vars] = ABS(c[ci+1+AN.poly_num_vars]);
01076                     }
01077                 }
01078                 else {
01079                     // otherwise, use form long calculus
01080                     AddLong ((UWORD *)&p[4+AN.poly_num_vars], p[3],
01081                             (UWORD *)&c[ci+AN.poly_num_vars+1], nc,
01082                             (UWORD *)&c[ci+AN.poly_num_vars+1],&nc);
01083
01084                     if (c.modp!=0) TakeNormalModulus((UWORD *)&c[ci+1+AN.poly_num_vars], &nc,

```

```

01084                                     modq, nmodq,
NOUNPACK);
01085     }
01086
01087     if (nc!=0) {
01088         c[ci] = 2 + AN.poly_num_vars + ABS(nc);
01089         ci += c[ci];
01090         c[ci-1] = nc;
01091     }
01092 }
01093 }
01094
01095 // add new term to the heap (ai, bi+1)
01096 while (p[1] < b[0]) {
01097
01098     for (int j=0; j<AN.poly_num_vars; j++)
01099         p[4+j] = a[p[0]+1+j] + b[p[1]+1+j];
01100
01101     // if both polynomials are modulo p^1, use integer calculus
01102     if (both_mod_small) {
01103         p[4+AN.poly_num_vars] = ((LONG)a[p[0]+1+AN.poly_num_vars]*a[p[0]+a[p[0]]-1]*
01104 b[p[1]+1+AN.poly_num_vars]*b[p[1]+b[p[1]]-1]) % c.modp;
01105         if (p[4+AN.poly_num_vars]==0)
01106             p[3]=0;
01107         else {
01108             if (p[4+AN.poly_num_vars] > +c.modp/2) p[4+AN.poly_num_vars] -= c.modp;
01109             if (p[4+AN.poly_num_vars] < -c.modp/2) p[4+AN.poly_num_vars] += c.modp;
01110             p[3] = SGN(p[4+AN.poly_num_vars]);
01111             p[4+AN.poly_num_vars] = ABS(p[4+AN.poly_num_vars]);
01112         }
01113     }
01114     else {
01115         // otherwise, use form long calculus
01116         MulLong ((UWORD *)&a[p[0]+1+AN.poly_num_vars], a[p[0]+a[p[0]]-1],
01117 (UWORD *)&b[p[1]+1+AN.poly_num_vars], b[p[1]+b[p[1]]-1],
01118 (UWORD *)&p[4+AN.poly_num_vars], &p[3]);
01119
01120         if (c.modp!=0) TakeNormalModulus((UWORD *)&p[4+AN.poly_num_vars], &p[3],
01121                                     modq, nmodq,
NOUNPACK);
01122     }
01123
01124     p[1] += b[p[1]];
01125
01126     if (use_hash) {
01127         int ID = 0;
01128         for (int i=0; i<AN.poly_num_vars; i++)
01129             ID = (maxpower[i]+1)*ID + p[4+i];
01130
01131         // if hash and unused, push onto heap
01132         if (hash[ID] == NULL) {
01133             p[2] = ID;
01134             hash[ID] = p;
01135             push_heap(BHEAD heap, ++nheap);
01136             break;
01137         }
01138         else {
01139             // if hash and used, add to heap element
01140             WORD *h = hash[ID];
01141             // if both polynomials are modulo p^1, use integer calculus
01142             if (both_mod_small) {
01143                 h[4+AN.poly_num_vars] = ((LONG)p[4+AN.poly_num_vars]*p[3] +
h[4+AN.poly_num_vars]*h[3]) % c.modp;
01144                 if (h[4+AN.poly_num_vars]==0)
01145                     h[3]=0;
01146                 else {
01147                     if (h[4+AN.poly_num_vars] > +c.modp/2) h[4+AN.poly_num_vars] -= c.modp;
01148                     if (h[4+AN.poly_num_vars] < -c.modp/2) h[4+AN.poly_num_vars] += c.modp;
01149                     h[3] = SGN(h[4+AN.poly_num_vars]);
01150                     h[4+AN.poly_num_vars] = ABS(h[4+AN.poly_num_vars]);
01151                 }
01152             }
01153             else {
01154                 // otherwise, use form long calculus
01155                 AddLong ((UWORD *)&p[4+AN.poly_num_vars], p[3],
01156 (UWORD *)&h[4+AN.poly_num_vars], h[3],
01157 (UWORD *)&h[4+AN.poly_num_vars], &h[3]);
01158
01159                 if (c.modp!=0) TakeNormalModulus((UWORD *)&h[4+AN.poly_num_vars], &h[3],
01160                                     modq, nmodq,
NOUNPACK);
01161             }
01162         }
01163     }
01164     else {
01165         // if no hash, push onto heap

```



```

01166         p[2] = -1;
01167         push_heap(BHEAD heap, ++nheap);
01168         break;
01169     }
01170 }
01171 }
01172
01173 c[0] = ci;
01174
01175 for (int ai=1, i=0; ai<a[0]; ai+=a[ai], i++)
01176     NumberFree(heap[i], "poly::mul_heap");
01177 AT.WorkPointer -= 3 * AN.poly_num_vars;
01178 }
01179
01180 /*
01181  #] mul_heap :
01182  #[ mul :
01183  */
01184
01195 void poly::mul (const poly &a, const poly &b, poly &c) {
01196
01197     c.modp = a.modp;
01198     c.modn = a.modn;
01199
01200     if (a.is_zero() || b.is_zero()) { c[0]=1; return; }
01201     if (a.is_one()) {
01202         c.check_memory(b[0]);
01203         c.termscopy(b.terms,0,b[0]);
01204         // normalize if necessary
01205         if (a.modp!=b.modp || a.modn!=b.modn) {
01206             c.modp=0;
01207             c.setmod(a.modp,a.modn);
01208         }
01209         return;
01210     }
01211     if (b.is_one()) {
01212         c.check_memory(a[0]);
01213         c.termscopy(a.terms,0,a[0]);
01214         return;
01215     }
01216
01217     int na=a.number_of_terms();
01218     int nb=b.number_of_terms();
01219
01220     if (na==1 || nb==1) {
01221         mul_one_term(a,b,c);
01222         return;
01223     }
01224
01225     int vara = a.is_dense_univariate();
01226     int varb = b.is_dense_univariate();
01227
01228     if (vara!=-2 && varb!=-2 && vara==varb) {
01229         mul_univar(a,b,c,vara);
01230         return;
01231     }
01232
01233     if (na < nb)
01234         mul_heap(a,b,c);
01235     else
01236         mul_heap(b,a,c);
01237 }
01238
01239 /*
01240  #] mul :
01241  #[ divmod_one_term : :
01242  */
01243
01244 // divides a polynomial by a monomial
01245 void poly::divmod_one_term (const poly &a, const poly &b, poly &q, poly &r, bool only_divides) {
01246
01247     POLY_GETIDENTITY(a);
01248
01249     int qi=1, ri=1;
01250
01251     WORD nmodq=0;
01252     UWORD *modq=NULL;
01253
01254     WORD nltbinv=0;
01255     UWORD *ltbinv=NULL;
01256
01257     bool both_mod_small=false;
01258
01259     if (q.modp!=0) {
01260         if (q.modn == 1) {
01261             modq = (UWORD *) &q.modp;
01262             nmodq = 1;

```

```

01263         if (a.modp>0 && b.modp>0 && a.modn==1 && b.modn==1)
01264             both_mod_small=true;
01265     }
01266     else {
01267         RaisPowCached(BHEAD q.modp,q.modn,&modq,&nmodq);
01268     }
01269     ltbinv = NumberMalloc("poly::div_one_term");
01270
01271     if (both_mod_small) {
01272         WORD ltb = b[b[1]]*b[2+AN.poly_num_vars];
01273         GetModInverses(ltb + (ltb<0?q.modp:0), q.modp, (WORD*)ltbinv, NULL);
01274         nltbinv = 1;
01275     }
01276     else
01277         GetLongModInverses(BHEAD (UWORD *)&b[2+AN.poly_num_vars], b[b[1]], modq, nmodq, ltbinv,
01278 &nltbinv, NULL, NULL);
01279 }
01280
01281 for (int ai=1; ai<a[0]; ai+=a[ai]) {
01282
01283     q.check_memory(qi);
01284     r.check_memory(ri);
01285
01286     // check divisibility of powers
01287     bool div=true;
01288     for (int j=0; j<AN.poly_num_vars; j++) {
01289         q[qi+1+j] = a[ai+1+j]-b[2+j];
01290         r[ri+1+j] = a[ai+1+j];
01291         if (q[qi+1+j] < 0) div=false;
01292     }
01293
01294     WORD nq,nr;
01295
01296     if (div) {
01297         // if variables are divisible, divide coefficient
01298         if (q.modp==0) {
01299             DivLong((UWORD *)&a[ai+1+AN.poly_num_vars], a[ai+a[ai]-1],
01300 (UWORD *)&b[2+AN.poly_num_vars], b[b[1]],
01301 (UWORD *)&q[qi+1+AN.poly_num_vars], &nq,
01302 (UWORD *)&r[ri+1+AN.poly_num_vars], &nr);
01303         }
01304         else {
01305             // if both polynomials are modulo p^1, use integer calculus
01306             if (both_mod_small) {
01307                 q[qi+1+AN.poly_num_vars] =
01308 (LONG)a[ai+1+AN.poly_num_vars] * a[ai+a[ai]-1] * ltbinv[0] * nltbinv) %
01309 q.modp;
01310                 nq = (q[qi+1+AN.poly_num_vars]==0 ? 0 : 1);
01311             }
01312             else {
01313                 // otherwise, use form long calculus
01314                 MulLong((UWORD *)&a[ai+1+AN.poly_num_vars], a[ai+a[ai]-1],
01315 ltbinv, nltbinv,
01316 (UWORD *)&q[qi+1+AN.poly_num_vars], &nq);
01317                 TakeNormalModulus((UWORD *)&q[qi+1+AN.poly_num_vars], &nq,
01318 modq,nmodq, NOUNPACK);
01319             }
01320
01321             nr=0;
01322         }
01323     }
01324     else {
01325         // if not, term becomes part of the remainder
01326         nq=0;
01327         nr=a[ai+a[ai]-1];
01328         r.termscopy(&a[ai+1+AN.poly_num_vars], ri+1+AN.poly_num_vars, ABS(nr));
01329     }
01330
01331     // add terms to quotient/remainder
01332     if (nq!=0) {
01333         q[qi] = 2+AN.poly_num_vars+ABS(nq);
01334         qi += q[qi];
01335
01336         // if necessary, adjust to range [-p/2,p/2]
01337         if (both_mod_small) {
01338             if (q[qi-2] > +q.modp/2) q[qi-2] -= q.modp;
01339             if (q[qi-2] < -q.modp/2) q[qi-2] += q.modp;
01340             q[qi-1] = SGN(q[qi-2]);
01341             q[qi-2] = ABS(q[qi-2]);
01342         }
01343         else {
01344             q[qi-1] = nq;
01345         }
01346     }
01347     if (nr != 0) {

```

```

01348         if (only_divides) { r = poly(BHEAD 1); ri=r[0]; break; }
01349         r[ri] = 2+AN.poly_num_vars+ABS(nr);
01350         ri += r[ri];
01351         r[ri-1] = nr;
01352     }
01353 }
01354
01355 q[0]=qi;
01356 r[0]=ri;
01357
01358 if (q.modp!=0 || ltbinv != NULL) NumberFree(ltbinv,"poly::div_one_term");
01359 }
01360
01361 /*
01362 #] divmod_one_term :
01363 #[ divmod_univar : :
01364 */
01365
01376 void poly::divmod_univar (const poly &a, const poly &b, poly &q, poly &r, int var, bool only_divides)
01377 {
01378     POLY_GETIDENTITY(a);
01379
01380     WORD nmodq=0;
01381     UWORD *modq=NULL;
01382
01383     WORD nltbinv=0;
01384     UWORD *ltbinv=NULL;
01385
01386     bool both_mod_small=false;
01387
01388     if (q.modp!=0) {
01389         if (q.modn == 1) {
01390             modq = (UWORD *) &q.modp;
01391             nmodq = 1;
01392             if (a.modp>0 && b.modp>0 && a.modn==1 && b.modn==1)
01393                 both_mod_small=true;
01394         }
01395         else {
01396             RaisPowCached(BHEAD q.modp,q.modn,&modq,&nmodq);
01397         }
01398         ltbinv = NumberMalloc("poly::div_univar");
01399
01400         if (both_mod_small) {
01401             WORD ltb = b[b[1]]*b[2+AN.poly_num_vars];
01402             GetModInverses(ltb + (ltb<0?q.modp:0), q.modp, (WORD*)ltbinv, NULL);
01403             nltbinv = 1;
01404         }
01405         else
01406             GetLongModInverses(BHEAD (UWORD *) &b[2+AN.poly_num_vars], b[b[1]], modq, nmodq, ltbinv,
01407 &nltbinv, NULL, NULL);
01408     }
01409
01410     WORD ns=0;
01411     WORD nt;
01412     UWORD *s = NumberMalloc("poly::div_univar");
01413     UWORD *t = NumberMalloc("poly::div_univar");
01414     s[0]=0;
01415
01416     int bpow = b[2+var];
01417
01418     int ai=1, qi=1, ri=1;
01419
01420     for (int pow=a[2+var]; pow>=0; pow--) {
01421         q.check_memory(qi);
01422         r.check_memory(ri);
01423
01424         // look for the correct power in a
01425         while (ai<a[0] && a[ai+1+var] > pow)
01426             ai+=a[ai];
01427
01428         // first term of the r.h.s. of the above equation
01429         if (ai<a[0] && a[ai+1+var] == pow) {
01430             ns = a[ai+a[ai]-1];
01431             WCOPY(s, &a[ai+1+AN.poly_num_vars], ABS(ns));
01432         }
01433         else {
01434             ns = 0;
01435         }
01436
01437         int bi=1, qj=qi;
01438
01439         // second term(s) of the r.h.s. of the above equation
01440         while (qj>1 && bi<b[0]) {
01441
01442             qj -= 2 + AN.poly_num_vars + ABS(q[qj-1]);

```

```

01443
01444 while (bi<b[0] && b[bi+1+var]+q[qj+1+var] > pow)
01445     bi += b[bi];
01446
01447 if (bi<b[0] && b[bi+1+var]+q[qj+1+var] == pow) {
01448     // if both polynomials are modulo p^1, use integer calculus
01449
01450     if (both_mod_small) {
01451         s[0] = ((WORD)s[0]*ns - (LONG)b[bi+1+AN.poly_num_vars] * b[bi+b[bi]-1] *
01452             q[qj+1+AN.poly_num_vars] * q[qj+q[qj]-1]) % q.modp;
01453         ns = (s[0]==0 ? 0 : 1);
01454     }
01455     else {
01456         // otherwise, use form long calculus
01457         MulLong((UWORD *)&b[bi+1+AN.poly_num_vars], b[bi+b[bi]-1],
01458             (UWORD *)&q[qj+1+AN.poly_num_vars], q[qj+q[qj]-1],
01459             t, &nt);
01460         nt *= -1;
01461         AddLong(t,nt,s,ns,s,&ns);
01462         if (q.modp!=0) TakeNormalModulus((UWORD *)s,&ns,modq, nmodq, NOUNPACK);
01463     }
01464 }
01465 }
01466
01467 if (ns != 0) {
01468     // if necessary, adjust to range [-p/2,p/2]
01469     if (both_mod_small) {
01470         s[0]*=ns;
01471         if ((WORD)s[0] > +q.modp/2) s[0] -= q.modp;
01472         if ((WORD)s[0] < -q.modp/2) s[0] += q.modp;
01473         ns = SGN((WORD)s[0]);
01474         s[0] = ABS((WORD)s[0]);
01475     }
01476
01477     if (pow >= bpow) {
01478         // large power, so divide by b
01479         if (q.modp == 0) {
01480             DivLong(s, ns,
01481                 (UWORD *)&b[2+AN.poly_num_vars], b[b[1]],
01482                 (UWORD *)&q[qi+1+AN.poly_num_vars], &ns, t, &nt);
01483         }
01484         else {
01485             if (both_mod_small) {
01486                 q[qi+1+AN.poly_num_vars] = ((LONG)s[0]*ns*ltbinv[0]*nltbinv) % q.modp;
01487                 if ((WORD)q[qi+1+AN.poly_num_vars] > +q.modp/2) q[qi+1+AN.poly_num_vars] -=
q.modp;
01488                 if ((WORD)q[qi+1+AN.poly_num_vars] < -q.modp/2) q[qi+1+AN.poly_num_vars] +=
q.modp;
01489                 ns = (q[qi+1+AN.poly_num_vars]==0 ? 0 : SGN((WORD)q[qi+1+AN.poly_num_vars]));
01490                 q[qi+1+AN.poly_num_vars] = ABS((WORD)q[qi+1+AN.poly_num_vars]);
01491             }
01492             else {
01493                 MulLong(s, ns, ltbinv, nltbinv, (UWORD *)&q[qi+1+AN.poly_num_vars], &ns);
01494                 TakeNormalModulus((UWORD *)&q[qi+1+AN.poly_num_vars], &ns,
01495                     modq,nmodq, NOUNPACK);
01496             }
01497             nt=0;
01498         }
01499     }
01500     else {
01501         // small power, so remainder
01502         WCOPY(t,s,ABS(ns));
01503         nt = ns;
01504         ns = 0;
01505     }
01506
01507     // add terms to quotient/remainder
01508     if (ns!=0) {
01509         for (int i=0; i<AN.poly_num_vars; i++)
01510             q[qi+1+i] = 0;
01511         q[qi+1+var] = pow-bpow;
01512
01513         q[qi] = 2+AN.poly_num_vars+ABS(ns);
01514         qi += q[qi];
01515         q[qi-1] = ns;
01516     }
01517
01518     if (nt != 0) {
01519         if (only_divides) { r=poly(BHEAD 1); ri=r[0]; break; }
01520         for (int i=0; i<AN.poly_num_vars; i++)
01521             r[ri+1+i] = 0;
01522         r[ri+1+var] = pow;
01523
01524         for (int i=0; i<ABS(nt); i++)
01525             r[ri+1+AN.poly_num_vars+i] = t[i];
01526

```

```

01527             r[ri] = 2+AN.poly_num_vars+ABS(nt);
01528             ri += r[ri];
01529             r[ri-1] = nt;
01530         }
01531     }
01532 }
01533
01534 q[0] = qi;
01535 r[0] = ri;
01536
01537 NumberFree(s,"poly::div_univar");
01538 NumberFree(t,"poly::div_univar");
01539
01540 if (q.modp!=0 || ltbinv!=NULL) NumberFree(ltbinv,"poly::div_univar");
01541 }
01542
01543 /*
01544 #] divmod_univar :
01545 #[ divmod_heap :
01546 */
01547
01579 void poly::divmod_heap (const poly &a, const poly &b, poly &q, poly &r, bool only_divides, bool
check_div, bool &div_fail) {
01580
01581     POLY_GETIDENTITY(a);
01582
01583     div_fail = false;
01584     q[0] = r[0] = 1;
01585
01586     WORD nmodq=0;
01587     UWORD *modq=NULL;
01588
01589     WORD nltbinv=0;
01590     UWORD *ltbinv=NULL;
01591     LONG oldpWorkPointer = AT.pWorkPointer;
01592
01593     bool both_mod_small=false;
01594
01595     if (q.modp!=0) {
01596         if (q.modn == 1) {
01597             modq = (UWORD *) &q.modp;
01598             nmodq = 1;
01599             if (a.modp>0 && b.modp>0 && a.modn==1 && b.modn==1)
01600                 both_mod_small=true;
01601         }
01602         else {
01603             RaisPowCached(BHEAD q.modp,q.modn,&modq,&nmodq);
01604         }
01605         ltbinv = NumberMalloc("poly::div_heap-a");
01606
01607         if (both_mod_small) {
01608             WORD ltb = b[b[1]]*b[2+AN.poly_num_vars];
01609             GetModInverses(ltb + (ltb<0?q.modp:0), q.modp, (WORD*)ltbinv, NULL);
01610             nltbinv = 1;
01611         }
01612         else
01613             GetLongModInverses(BHEAD (UWORD *) &b[2+AN.poly_num_vars], b[b[1]], modq, nmodq, ltbinv,
&nltbinv, NULL, NULL);
01614     }
01615
01616     // allocate heap
01617     int nb=b.number_of_terms();
01618     WantAddPointers(nb);
01619     WORD **heap = AT.pWorkSpace + AT.pWorkPointer;
01620     AT.pWorkPointer += nb;
01621
01622     for (int i=0; i<nb; i++)
01623         heap[i] = (WORD *) NumberMalloc("poly::div_heap-b");
01624     int nheap = 1;
01625     heap[0][0] = 1;
01626     heap[0][1] = 0;
01627     heap[0][2] = -1;
01628     WCOPY(&heap[0][3], &a[1], a[1]);
01629     heap[0][3] = a[a[1]];
01630
01631     int qi=1, ri=1;
01632
01633     int s = nb;
01634     WORD *t = (WORD *) NumberMalloc("poly::div_heap-c");
01635
01636     // insert contains element that still have to be inserted to the heap
01637     // (exists to avoid code duplication).
01638     vector<pair<int,int> > insert;
01639
01640     while (insert.size()>0 || nheap>0) {
01641
01642         q.check_memory(qi);

```

```

01643     r.check_memory(ri);
01644
01645     // collect a term t for the quotient/remainder
01646     t[0] = -1;
01647
01648     do {
01649
01650         WORD *p = heap[nheap];
01651         bool this_insert;
01652
01653         if (insert.empty()) {
01654             // extract element from the heap and prepare adding new ones
01655             this_insert = false;
01656
01657             pop_heap(BHEAD heap, nheap--);
01658             p = heap[nheap];
01659
01660             if (t[0] == -1) {
01661                 WCOPY(t, p, (5*ABS(p[3])+AN.poly_num_vars));
01662             }
01663             else {
01664                 // if both polynomials are modulo p^1, use integer calculus
01665                 if (both_mod_small) {
01666                     t[4+AN.poly_num_vars] = ((LONG)t[4+AN.poly_num_vars]*t[3] +
01667 p[4+AN.poly_num_vars]*p[3]) % q.modp;
01668                     if (t[4+AN.poly_num_vars]==0)
01669                         t[3]=0;
01670                     else {
01671                         if (t[4+AN.poly_num_vars] > +q.modp/2) t[4+AN.poly_num_vars] -= q.modp;
01672                         if (t[4+AN.poly_num_vars] < -q.modp/2) t[4+AN.poly_num_vars] += q.modp;
01673                         t[3] = SGN(t[4+AN.poly_num_vars]);
01674                         t[4+AN.poly_num_vars] = ABS(t[4+AN.poly_num_vars]);
01675                     }
01676                 }
01677                 else {
01678                     // otherwise, use form long calculus
01679                     AddLong ((UWORD *)&p[4+AN.poly_num_vars], p[3],
01680                             (UWORD *)&t[4+AN.poly_num_vars], t[3],
01681                             (UWORD *)&t[4+AN.poly_num_vars], &t[3]);
01682                     if (q.modp!=0) TakeNormalModulus((UWORD *)&t[4+AN.poly_num_vars], &t[3],
01683                                                     modq, nmodq,
01684 NOUNPACK);
01685                 }
01686             }
01687             }
01688         }
01689         else {
01690             // prepare adding an element of insert to the heap
01691             this_insert = true;
01692
01693             p[0] = insert.back().first;
01694             p[1] = insert.back().second;
01695             insert.pop_back();
01696
01697             // add elements to the heap
01698             while (true) {
01699                 // prepare the element
01700                 if (p[1]==0) {
01701                     p[0] += a[p[0]];
01702                     if (p[0]==a[0]) break;
01703                     WCOPY(&p[3], &a[p[0]], a[p[0]]);
01704                     p[3] = p[2+p[3]];
01705                 }
01706                 else {
01707                     if (!this_insert)
01708                         p[1] += q[p[1]];
01709                     this_insert = false;
01710
01711                     if (p[1]==qi) { s++; break; }
01712
01713                     for (int i=0; i<AN.poly_num_vars; i++)
01714                         p[4+i] = b[p[0]+1+i] + q[p[1]+1+i];
01715
01716                     // if both polynomials are modulo p^1, use integer calculus
01717                     if (both_mod_small) {
01718                         p[4+AN.poly_num_vars] = ((LONG)b[p[0]+1+AN.poly_num_vars]*b[p[0]+b[p[0]]-1]*
01719 q[p[1]+1+AN.poly_num_vars]*q[p[1]+q[p[1]]-1]) % q.modp;
01720                         if (p[4+AN.poly_num_vars]==0)
01721                             p[3]=0;
01722                         else {
01723                             if (p[4+AN.poly_num_vars] > +q.modp/2) p[4+AN.poly_num_vars] -= q.modp;
01724                             if (p[4+AN.poly_num_vars] < -q.modp/2) p[4+AN.poly_num_vars] += q.modp;
01725                             p[3] = SGN(p[4+AN.poly_num_vars]);
01726                             p[4+AN.poly_num_vars] = ABS(p[4+AN.poly_num_vars]);
01727                         }
01728                     }
01729                 }
01730             }
01731         }
01732     }

```

```

01727         else {
01728             // otherwise, use form long calculus
01729             MulLong((UWORD *) &b[p[0]+1+AN.poly_num_vars], b[p[0]+b[p[0]]-1],
01730                     (UWORD *) &q[p[1]+1+AN.poly_num_vars], q[p[1]+q[p[1]]-1],
01731                     (UWORD *) &p[4+AN.poly_num_vars], &p[3]);
01732             if (q.modp!=0) TakeNormalModulus((UWORD *) &p[4+AN.poly_num_vars], &p[3],
01733                                             modq, nmodq,
NOUNPACK);
01734         }
01735
01736         p[3] *= -1;
01737     }
01738
01739     // no hashing
01740     p[2] = -1;
01741
01742     // add it to a heap element
01743     swap (heap[nheap],p);
01744     push_heap(BHEAD heap, ++nheap);
01745     break;
01746 }
01747 }
01748 while (t[0]==-1 || (nheap>0 && monomial_compare(BHEAD heap[0]+3, t+3)==0));
01749
01750 if (t[3] == 0) continue;
01751
01752 // check divisibility
01753 bool div = true;
01754 for (int i=0; i<AN.poly_num_vars; i++)
01755     if (t[4+i] < b[2+i]) div=false;
01756
01757 if (!div) {
01758     // not divisible, so append it to the remainder
01759     if (only_divides) { r=poly(BHEAD 1); ri=r[0]; break;}
01760     t[4 + AN.poly_num_vars + ABS(t[3])] = t[3];
01761     t[3] = 2 + AN.poly_num_vars + ABS(t[3]);
01762     r.termscopy(&t[3], ri, t[3]);
01763     ri += t[3];
01764 }
01765 else {
01766     // divisible, so divide coefficient as well
01767     WORD nq, nr;
01768
01769     if (q.modp==0) {
01770         DivLong((UWORD *) &t[4+AN.poly_num_vars], t[3],
01771                 (UWORD *) &b[2+AN.poly_num_vars], b[b[1]],
01772                 (UWORD *) &q[qi+1+AN.poly_num_vars], &nq,
01773                 (UWORD *) &r[ri+1+AN.poly_num_vars], &nr);
01774         if(check_div && nr != 0)
01775         {
01776             // non-zero remainder from coefficient division => result not expressible in
integers
01777             div_fail = true;
01778             break;
01779         }
01780     }
01781     else {
01782         // if both polynomials are modulo p^1, use integer calculus
01783         if (both_mod_small) {
01784             q[qi+1+AN.poly_num_vars] = ((LONG)t[4+AN.poly_num_vars]*t[3]*ltbinv[0]*nltbinv) %
q.modp;
01785             if (q[qi+1+AN.poly_num_vars]==0)
01786                 nq=0;
01787             else {
01788                 if (q[qi+1+AN.poly_num_vars] > +q.modp/2) q[qi+1+AN.poly_num_vars] -= q.modp;
01789                 if (q[qi+1+AN.poly_num_vars] < -q.modp/2) q[qi+1+AN.poly_num_vars] += q.modp;
01790                 nq = SGN(q[qi+1+AN.poly_num_vars]);
01791                 q[qi+1+AN.poly_num_vars] = ABS(q[qi+1+AN.poly_num_vars]);
01792             }
01793         }
01794         else {
01795             // otherwise, use form long calculus
01796             MulLong((UWORD *) &t[4+AN.poly_num_vars], t[3], ltbinv, nltbinv, (UWORD
*) &q[qi+1+AN.poly_num_vars], &nq);
01797             TakeNormalModulus((UWORD *) &q[qi+1+AN.poly_num_vars], &nq, modq, nmodq, NOUNPACK);
01798         }
01799
01800         nr=0;
01801     }
01802
01803     // add terms to quotient and remainder
01804     if (nq != 0) {
01805         int bi = 1;
01806         for (int j=1; j<s; j++) {
01807             bi += b[bi];
01808             insert.push_back(make_pair(bi,qi));
01809         }

```

```

01810             s=1;
01811
01812             q[qi] = 2+AN.poly_num_vars+ABS(nq);
01813             for (int i=0; i<AN.poly_num_vars; i++)
01814                 q[qi+1+i] = t[4+i] - b[2+i];
01815             qi += q[qi];
01816             q[qi-1] = nq;
01817         }
01818
01819         if (nr != 0) {
01820             r[ri] = 2+AN.poly_num_vars+ABS(nr);
01821             for (int i=0; i<AN.poly_num_vars; i++)
01822                 r[ri+1+i] = t[4+i];
01823             ri += r[ri];
01824             r[ri-1] = nr;
01825         }
01826     }
01827 }
01828
01829 q[0] = qi;
01830 r[0] = ri;
01831
01832 for (int i=0; i<nb; i++)
01833     NumberFree(heap[i], "poly::div_heap-b");
01834
01835 NumberFree(t, "poly::div_heap-c");
01836
01837 if (q.modp!=0 || ltbinv!=NULL) NumberFree(ltbinv, "poly::div_heap-a");
01838 AT.pWorkPointer = oldpWorkPointer;
01839 }
01840
01841 /*
01842 #] divmod_heap :
01843 #[ divmod :
01844 */
01845
01856 void poly::divmod (const poly &a, const poly &b, poly &q, poly &r, bool only_divides) {
01857     q.modp = r.modp = a.modp;
01858     q.modn = r.modn = a.modn;
01859
01860     if (a.is_zero()) {
01861         q[0]=1;
01862         r[0]=1;
01863         return;
01864     }
01865
01866     if (b.is_zero()) {
01867         MLOCK(ErrorMessageLock);
01868         MesPrint ((char*)"ERROR: division by zero polynomial");
01869         MUNLOCK(ErrorMessageLock);
01870         Terminate(1);
01871     }
01872
01873     if (b.is_one()) {
01874         q.check_memory(a[0]);
01875         q.termscopy(a.terms,0,a[0]);
01876         r[0]=1;
01877         return;
01878     }
01879
01880     if (b.is_monomial()) {
01881         divmod_one_term(a,b,q,r,only_divides);
01882         return;
01883     }
01884
01885     int vara = a.is_dense_univariate();
01886     int varb = b.is_dense_univariate();
01887
01888     if (vara!=-2 && varb!=-2 && (vara==--1 || varb==--1 || vara==varb)) {
01889         divmod_univar(a,b,q,r,Max(vara,varb),only_divides);
01890     }
01891     else {
01892         bool div_fail;
01893         divmod_heap(a,b,q,r,only_divides,false,div_fail);
01894     }
01895 }
01896
01897 /*
01898 #] divmod :
01899 #[ divides :
01900 */
01901 // returns whether a exactly divides b
01902 bool poly::divides (const poly &a, const poly &b) {
01903     POLY_GETIDENTITY(a);
01904     poly q(BHEAD 0), r(BHEAD 0);
01905     divmod(b,a,q,r,true);
01906     return r.is_zero();

```



```

01907 }
01908
01909 /*
01910     #] divides :
01911     #[ div :
01912 */
01913
01914 // the quotient of two polynomials
01915 void poly::div (const poly &a, const poly &b, poly &c) {
01916     POLY_GETIDENTITY(a);
01917     poly d(BHEAD 0);
01918     divmod(a,b,c,d,false);
01919 }
01920
01921 /*
01922     #] div :
01923     #[ mod :
01924 */
01925
01926 // the remainder of division of two polynomials
01927 void poly::mod (const poly &a, const poly &b, poly &c) {
01928     POLY_GETIDENTITY(a);
01929     poly d(BHEAD 0);
01930     divmod(a,b,c,d,false);
01931 }
01932
01933 /*
01934     #] mod :
01935     #[ copy operator :
01936 */
01937
01938 // copy operator
01939 poly & poly::operator= (const poly &a) {
01940     #ifdef POLY_MOVE_DEBUG
01941         std::cout << "poly copy assign" << std::endl;
01942     #endif
01943     if (&a != this) {
01944         modp = a.modp;
01945         modn = a.modn;
01946         check_memory(a[0]);
01947         termscopy(a.terms,0,a[0]);
01948     }
01949     return *this;
01950 }
01951
01952
01953 /*
01954     #] copy operator :
01955     #[ operator overloads :
01956 */
01957
01958 // binary operators for polynomial arithmetic
01959 const poly poly::operator+ (const poly &a) const { POLY_GETIDENTITY(a); poly b(BHEAD 0);
    add(*this,a,b); return b; }
01960 const poly poly::operator- (const poly &a) const { POLY_GETIDENTITY(a); poly b(BHEAD 0);
    sub(*this,a,b); return b; }
01961 const poly poly::operator* (const poly &a) const { POLY_GETIDENTITY(a); poly b(BHEAD 0);
    mul(*this,a,b); return b; }
01962 const poly poly::operator/ (const poly &a) const { POLY_GETIDENTITY(a); poly b(BHEAD 0);
    div(*this,a,b); return b; }
01963 const poly poly::operator% (const poly &a) const { POLY_GETIDENTITY(a); poly b(BHEAD 0);
    mod(*this,a,b); return b; }
01964
01965 // assignment operators for polynomial arithmetic
01966 poly& poly::operator+= (const poly &a) { return *this = *this + a; }
01967 poly& poly::operator-= (const poly &a) { return *this = *this - a; }
01968 poly& poly::operator*= (const poly &a) { return *this = *this * a; }
01969 poly& poly::operator/= (const poly &a) { return *this = *this / a; }
01970 poly& poly::operator%= (const poly &a) { return *this = *this % a; }
01971
01972 // comparison operators
01973 bool poly::operator== (const poly &a) const {
01974     for (int i=0; i<terms[0]; i++)
01975         if (terms[i] != a[i]) return 0;
01976     return 1;
01977 }
01978
01979 bool poly::operator!= (const poly &a) const { return !(*this == a); }
01980
01981 /*
01982     #] operator overloads :
01983     #[ num_terms :
01984 */
01985
01986 int poly::number_of_terms () const {
01987
01988     int res=0;

```

```

01989     for (int i=1; i<terms[0]; i+=terms[i])
01990         res++;
01991     return res;
01992 }
01993
01994 /*
01995     #] num_terms :
01996     #[ first_variable :
01997 */
01998
01999 // returns the lexicographically first variable of a polynomial
02000 int poly::first_variable () const {
02001
02002     POLY_GETIDENTITY(*this);
02003
02004     int var = AN.poly_num_vars;
02005     for (int j=0; j<var; j++)
02006         if (terms[2+j]>0) return j;
02007     return var;
02008 }
02009
02010 /*
02011     #] first_variable :
02012     #[ all_variables :
02013 */
02014
02015 // returns a list of all variables of a polynomial
02016 const vector<int> poly::all_variables () const {
02017
02018     POLY_GETIDENTITY(*this);
02019
02020     vector<bool> used(AN.poly_num_vars, false);
02021
02022     for (int i=1; i<terms[0]; i+=terms[i])
02023         for (int j=0; j<AN.poly_num_vars; j++)
02024             if (terms[i+1+j]>0) used[j] = true;
02025
02026     vector<int> vars;
02027     for (int i=0; i<AN.poly_num_vars; i++)
02028         if (used[i]) vars.push_back(i);
02029
02030     return vars;
02031 }
02032
02033 /*
02034     #] all_variables :
02035     #[ sign :
02036 */
02037
02038 // returns the sign of the leading coefficient
02039 int poly::sign () const {
02040     if (terms[0]==1) return 0;
02041     return terms[terms[1]] > 0 ? 1 : -1;
02042 }
02043
02044 /*
02045     #] sign :
02046     #[ degree :
02047 */
02048
02049 // returns the degree of x of a polynomial (deg=-1 iff a=0)
02050 int poly::degree (int x) const {
02051     int deg = -1;
02052     for (int i=1; i<terms[0]; i+=terms[i])
02053         deg = MaX(deg, terms[i+1+x]);
02054     return deg;
02055 }
02056
02057 /*
02058     #] degree :
02059     #[ total_degree :
02060 */
02061
02062 // returns the total degree of a polynomial (deg=-1 iff a=0)
02063 int poly::total_degree () const {
02064
02065     POLY_GETIDENTITY(*this);
02066
02067     int tot_deg = -1;
02068     for (int i=1; i<terms[0]; i+=terms[i]) {
02069         int deg=0;
02070         for (int j=0; j<AN.poly_num_vars; j++)
02071             deg += terms[i+1+j];
02072         tot_deg = MaX(deg, tot_deg);
02073     }
02074     return tot_deg;
02075 }

```

```

02076
02077 /*
02078     #] total_degree :
02079     #[ integer_lcoeff :
02080 */
02081
02082 // returns the integer coefficient of the leading monomial
02083 const poly poly::integer_lcoeff () const {
02084
02085     POLY_GETIDENTITY(*this);
02086
02087     poly res(BHEAD 0);
02088     res.modp = modp;
02089     res.modn = modn;
02090
02091     res.termscopy(&terms[1],1,terms[1]);
02092     res[0] = res[1] + 1; // length
02093     for (int i=0; i<AN.poly_num_vars; i++)
02094         res[2+i] = 0; // powers
02095
02096     return res;
02097 }
02098
02099 /*
02100     #] integer_lcoeff :
02101     #[ coefficient :
02102 */
02103
02104 // returns the polynomial coefficient of x^n
02105 const poly poly::coefficient (int x, int n) const {
02106
02107     POLY_GETIDENTITY(*this);
02108
02109     poly res(BHEAD 0);
02110     res.modp = modp;
02111     res.modn = modn;
02112     res[0] = 1;
02113
02114     for (int i=1; i<terms[0]; i+=terms[i])
02115         if (terms[i+1+x] == n) {
02116             res.check_memory(res[0]+terms[i]);
02117             res.termscopy(&terms[i], res[0], terms[i]);
02118             res[res[0]+1+x] -= n; // power of x
02119             res[0] += res[res[0]]; // length
02120         }
02121
02122     return res;
02123 }
02124
02125 /*
02126     #] coefficient :
02127     #[ lcoeff_multivar :
02128 */
02129
02130 // returns the leading coefficient with the polynomial viewed as a
02131 // polynomial in x, so that the result is a polynomial in the
02132 // remaining variables
02133 const poly poly::lcoeff_univar (int x) const {
02134     return coefficient(x, degree(x));
02135 }
02136
02137 // returns the leading coefficient with the polynomial viewed as a
02138 // polynomial with coefficients in ZZ[x], so that the result
02139 // polynomial is in ZZ[x] too
02140 const poly poly::lcoeff_multivar (int x) const {
02141
02142     POLY_GETIDENTITY(*this);
02143
02144     poly res(BHEAD 0,modp,modn);
02145
02146     for (int i=1; i<terms[0]; i+=terms[i]) {
02147         bool same_powers = true;
02148         for (int j=0; j<AN.poly_num_vars; j++)
02149             if (j!=x && terms[i+1+j]!=terms[2+j]) {
02150                 same_powers = false;
02151                 break;
02152             }
02153         if (!same_powers) break;
02154
02155         res.check_memory(res[0]+terms[i]);
02156         res.termscopy(&terms[i],res[0],terms[i]);
02157         for (int k=0; k<AN.poly_num_vars; k++)
02158             if (k!=x) res[res[0]+1+k]=0;
02159
02160         res[0] += terms[i];
02161     }
02162

```

```

02163     return res;
02164 }
02165
02166 /*
02167  #] lcoeff_multivar :
02168  #[ derivative :
02169 */
02170
02171 // returns the derivative with respect to x
02172 const poly poly::derivative (int x) const {
02173
02174     POLY_GETIDENTITY(*this);
02175
02176     poly b(BHEAD 0);
02177     int bi=1;
02178
02179     for (int i=1; i<terms[0]; i+=terms[i]) {
02180
02181         int power = terms[i+1+x];
02182
02183         if (power > 0) {
02184             b.check_memory(bi);
02185             b.termscopy(&terms[i], bi, terms[i]);
02186             b[bi+1+x]--;
02187
02188             WORD nb = b[bi+b[bi]-1];
02189             Product((UWORD *)&b[bi+1+AN.poly_num_vars], &nb, power);
02190
02191             b[bi] = 2 + AN.poly_num_vars + ABS(nb);
02192             b[bi+b[bi]-1] = nb;
02193
02194             bi += b[bi];
02195         }
02196     }
02197
02198     b[0] = bi;
02199     b.setmod(modp, modn);
02200     return b;
02201 }
02202
02203 /*
02204  #] derivative :
02205  #[ is_zero :
02206 */
02207
02208 // returns whether the polynomial is zero
02209 bool poly::is_zero () const {
02210     return terms[0] == 1;
02211 }
02212
02213 /*
02214  #] is_zero :
02215  #[ is_one :
02216 */
02217
02218 // returns whether the polynomial is one
02219 bool poly::is_one () const {
02220
02221     POLY_GETIDENTITY(*this);
02222
02223     if (terms[0] != 4+AN.poly_num_vars) return false;
02224     if (terms[1] != 3+AN.poly_num_vars) return false;
02225     for (int i=0; i<AN.poly_num_vars; i++)
02226         if (terms[2+i] != 0) return false;
02227     if (terms[2+AN.poly_num_vars] != 1) return false;
02228     if (terms[3+AN.poly_num_vars] != 1) return false;
02229
02230     return true;
02231 }
02232
02233 /*
02234  #] is_one :
02235  #[ is_integer :
02236 */
02237
02238 // returns whether the polynomial is an integer
02239 bool poly::is_integer () const {
02240
02241     POLY_GETIDENTITY(*this);
02242
02243     if (terms[0] == 1) return true;
02244     if (terms[0] != terms[1]+1) return false;
02245
02246     for (int j=0; j<AN.poly_num_vars; j++)
02247         if (terms[2+j] != 0)
02248             return false;
02249

```

```

02250     return true;
02251 }
02252
02253 /*
02254  #] is_integer :
02255  #[ is_monomial :
02256 */
02257
02258 // returns whether the polynomial consist of one term
02259 bool poly::is_monomial () const {
02260     return terms[0]>1 && terms[0]==terms[1]+1;
02261 }
02262
02263 /*
02264  #] is_monomial :
02265  #[ is_dense_univariate :
02266 */
02267
02284 int poly::is_dense_univariate () const {
02285     POLY_GETIDENTITY(*this);
02286
02287     int num_terms=0, res=-1;
02288
02289     // test univariate
02290     for (int i=1; i<terms[0]; i+=terms[i]) {
02291         for (int j=0; j<AN.poly_num_vars; j++)
02292             if (terms[i+1+j] > 0) {
02293                 if (res == -1) res = j;
02294                 if (res != j) return -2;
02295             }
02296         num_terms++;
02297     }
02298
02299     // constant polynomial
02300     if (res == -1) return -1;
02301
02302     // test density
02303     int deg = terms[2+res];
02304     if (2*num_terms < deg+1) return -2;
02305
02306     return res;
02307 }
02308
02309 /*
02310  #] is_dense_univariate :
02311  #[ simple_poly (small) :
02312 */
02313
02314 // returns the polynomial (x-a)^b mod p^n with a small
02315 const poly poly::simple_poly (PHEAD int x, int a, int b, int p, int n) {
02316     poly tmp(BHEAD 0,p,n);
02317
02318     int idx=1;
02319     tmp[idx++] = 3 + AN.poly_num_vars; // length
02320     for (int i=0; i<AN.poly_num_vars; i++)
02321         tmp[idx++] = i==x ? 1 : 0; // powers
02322     tmp[idx++] = 1; // coefficient
02323     tmp[idx++] = 1; // length coefficient
02324
02325     if (a != 0) {
02326         tmp[idx++] = 3 + AN.poly_num_vars; // length
02327         for (int i=0; i<AN.poly_num_vars; i++) tmp[idx++] = 0; // powers
02328         tmp[idx++] = ABS(a); // coefficient
02329         tmp[idx++] = -SGN(a); // length coefficient
02330     }
02331
02332     tmp[0] = idx; // length
02333
02334     if (b == 1) return tmp;
02335
02336     poly res(BHEAD 1,p,n);
02337
02338     while (b!=0) {
02339         if (b&1) res*=tmp;
02340         tmp*=tmp;
02341         b>>=1;
02342     }
02343
02344     return res;
02345 }
02346
02347 /*
02348  #] simple_poly (small) :
02349  #[ simple_poly (large) :

```

```

02353 */
02354
02355 // returns the polynomial (x-a)^b mod p^n with a large
02356 const poly poly::simple_poly (PHEAD int x, const poly &a, int b, int p, int n) {
02357
02358     poly res(BHEAD 1,p,n);
02359     poly tmp(BHEAD 0,p,n);
02360
02361     int idx=1;
02362
02363     tmp[idx++] = 3 + AN.poly_num_vars; // length
02364     for (int i=0; i<AN.poly_num_vars; i++)
02365         tmp[idx++] = i==x ? 1 : 0; // powers
02366     tmp[idx++] = 1; // coefficient
02367     tmp[idx++] = 1; // length coefficient
02368
02369     if (!a.is_zero()) {
02370         tmp[idx++] = 2 + AN.poly_num_vars + ABS(a[a[0]-1]); // length
02371         for (int i=0; i<AN.poly_num_vars; i++) tmp[idx++] = 0; // powers
02372         for (int i=0; i<ABS(a[a[0]-1]); i++)
02373             tmp[idx++] = a[2 + AN.poly_num_vars + i]; // coefficient
02374         tmp[idx++] = -a[a[0]-1]; // length coefficient
02375     }
02376
02377     tmp[0] = idx; // length
02378
02379     while (b!=0) {
02380         if (b&1) res*=tmp;
02381         tmp*=tmp;
02382         b>>=1;
02383     }
02384
02385     return res;
02386 }
02387
02388 /*
02389 #] simple_poly (large) :
02390 #[ get_variables :
02391 */
02392
02393 // gets all variables in the expressions and stores them in AN.poly_vars
02394 // (it is assumed that AN.poly_vars=NULL)
02395 void poly::get_variables (PHEAD const vector<WORD *> &es, bool with_arghead, bool sort_vars) {
02396     assert(AN.poly_vars == NULL);
02397
02398     AN.poly_num_vars = 0;
02399
02400     vector<int> vars;
02401     vector<int> degrees;
02402     map<int,int> var_to_idx;
02403
02404     // extract all variables
02405     for (int ei=0; ei<(int)es.size(); ei++) {
02406         const WORD *e = es[ei];
02407
02408         // fast notation
02409         if (*e == -SNUMBER) {
02410         }
02411         else if (*e == -SYMBOL) {
02412             if (!var_to_idx.count(e[1])) {
02413                 vars.push_back(e[1]);
02414                 var_to_idx[e[1]] = AN.poly_num_vars++;
02415                 degrees.push_back(1);
02416             }
02417         }
02418         // JD: Here we need to check for non-symbol/number terms in fast notation.
02419         else if (*e < 0) {
02420             MLOCK(ErrorMessageLock);
02421             MesPrint ((char*)"ERROR: polynomials and polyratfuns must contain symbols only");
02422             MUNLOCK(ErrorMessageLock);
02423             Terminate(1);
02424         }
02425         else {
02426             for (int i=with_arghead ? ARGHEAD : 0; with_arghead ? i<e[0] : e[i]!=0; i+=e[i]) {
02427                 if (i+1<i+e[i]-ABS(e[i+e[i]-1]) && e[i+1]!=SYMBOL) {
02428                     MLOCK(ErrorMessageLock);
02429                     MesPrint ((char*)"ERROR: polynomials and polyratfuns must contain symbols only");
02430                     MUNLOCK(ErrorMessageLock);
02431                     Terminate(1);
02432                 }
02433
02434                 for (int j=i+3; j<i+e[i]-ABS(e[i+e[i]-1]); j+=2) {
02435                     if (!var_to_idx.count(e[j])) {
02436                         vars.push_back(e[j]);
02437                         var_to_idx[e[j]] = AN.poly_num_vars++;
02438                         degrees.push_back(e[j+1]);
02439                     }

```

```

02440         else {
02441             degrees[var_to_idx[e[j]]] = MaX(degrees[var_to_idx[e[j]]], e[j+1]);
02442         }
02443     }
02444 }
02445 }
02446 }
02447
02448 // make sure an eventual FACTORSYMBOL appear as last
02449 if (var_to_idx.count(FACTORSYMBOL)) {
02450     int i = var_to_idx[FACTORSYMBOL];
02451     while (i+1<AN.poly_num_vars) {
02452         swap(vars[i], vars[i+1]);
02453         swap(degrees[i], degrees[i+1]);
02454         i++;
02455     }
02456     degrees[i] = -1; // makes sure it stays last
02457 }
02458
02459 // AN.poly_vars will be deleted in calling functions from polywrap.c
02460 // For that we use the function poly_free_poly_vars
02461 // We add one to the allocation to avoid copying in poly_divmod
02462
02463 if ( (AN.poly_num_vars+1)*sizeof(WORD) > (size_t)(AM.MaxTer) ) {
02464     // This can happen only in expressions with excessively many variables.
02465     AN.poly_vars = (WORD *)Malloc1((AN.poly_num_vars+1)*sizeof(WORD), "AN.poly_vars");
02466     AN.poly_vars_type = 1;
02467 }
02468 else {
02469     AN.poly_vars = TermMalloc("AN.poly_vars");
02470     AN.poly_vars_type = 0;
02471 }
02472
02473 for (int i=0; i<AN.poly_num_vars; i++)
02474     AN.poly_vars[i] = vars[i];
02475
02476 if (sort_vars) {
02477     // bubble sort variables in decreasing order of degree
02478     // (this seems better for factorization)
02479     for (int i=0; i<AN.poly_num_vars; i++)
02480         for (int j=0; j+1<AN.poly_num_vars; j++)
02481             if (degrees[j] < degrees[j+1]) {
02482                 swap(degrees[j], degrees[j+1]);
02483                 swap(AN.poly_vars[j], AN.poly_vars[j+1]);
02484             }
02485 }
02486 else {
02487     // sort lexicographically
02488     sort(AN.poly_vars, AN.poly_vars+AN.poly_num_vars - var_to_idx.count(FACTORSYMBOL));
02489 }
02490 }
02491
02492 /*
02493 #] get_variables :
02494 #[ argument_to_poly :
02495 */
02496
02497 // converts a form expression to a polynomial class "poly"
02498 const poly poly::argument_to_poly (PHEAD WORD *e, bool with_arghead, bool sort_univar, poly *denpoly)
02499 {
02500     map<int,int> var_to_idx;
02501     for (int i=0; i<AN.poly_num_vars; i++)
02502         var_to_idx[AN.poly_vars[i]] = i;
02503
02504     poly res(BHEAD 0);
02505
02506     // fast notation
02507     if (*e == -SNUMBER) {
02508         if (denpoly!=NULL) *denpoly = poly(BHEAD 1);
02509
02510         if (e[1] == 0) {
02511             res[0] = 1;
02512             return res;
02513         }
02514         else {
02515             res[0] = 4 + AN.poly_num_vars;
02516             res[1] = 3 + AN.poly_num_vars;
02517             for (int i=0; i<AN.poly_num_vars; i++)
02518                 res[2+i] = 0;
02519             res[2+AN.poly_num_vars] = ABS(e[1]);
02520             res[3+AN.poly_num_vars] = SGN(e[1]);
02521
02522             return res;
02523         }
02524     }
02525 }

```

```

02526     if (*e == -SYMBOL) {
02527         if (denpoly!=NULL) *denpoly = poly(BHEAD 1);
02528
02529         res[0] = 4 + AN.poly_num_vars;
02530         res[1] = 3 + AN.poly_num_vars;
02531         for (int i=0; i<AN.poly_num_vars; i++)
02532             res[2+i] = 0;
02533         res[2+var_to_idx.find(e[1])>second] = 1;
02534         res[2+AN.poly_num_vars] = 1;
02535         res[3+AN.poly_num_vars] = 1;
02536         return res;
02537     }
02538
02539     // find LCM of denominators of all terms
02540     WORD nden=1, npro=0, ngcd=0, ndum=0;
02541     UWORD *den = NumberMalloc("poly::argument_to_poly");
02542     UWORD *pro = NumberMalloc("poly::argument_to_poly");
02543     UWORD *gcd = NumberMalloc("poly::argument_to_poly");
02544     UWORD *dum = NumberMalloc("poly::argument_to_poly");
02545     den[0]=1;
02546
02547     for (int i=with_arghead ? ARGHEAD : 0; with_arghead ? i<e[0] : e[i]!=0; i+=e[i]) {
02548         int ncoe = ABS(e[i+e[i]-1]/2);
02549         UWORD *coe = (UWORD *)&e[i+e[i]-ncoe-1];
02550         while (ncoe>0 && coe[ncoe-1]==0) ncoe--;
02551         MullLong(den, nden, coe, ncoe, pro, &npro);
02552         GcdLong(BHEAD den, nden, coe, ncoe, gcd, &ngcd);
02553         DivLong(pro, npro, gcd, ngcd, den, &nden, dum, &ndum);
02554     }
02555
02556     if (denpoly!=NULL) *denpoly = poly(BHEAD den, nden);
02557
02558     int ri=1;
02559
02560     // ordinary notation
02561     for (int i=with_arghead ? ARGHEAD : 0; with_arghead ? i<e[0] : e[i]!=0; i+=e[i]) {
02562         res.check_memory(ri);
02563         WORD nc = e[i+e[i]-1]; // length coefficient
02564         for (int j=0; j<AN.poly_num_vars; j++) // powers=0
02565             res[ri+1+j]=0; // coefficient to dummy
02566         WCOPY(dum, &e[i+e[i]-ABS(nc)], ABS(nc)); // remove denominator
02567         nc /= 2; // multiply with
02568         Mully(BHEAD dum, &nc, den, nden);
02569         overall den
02570             res.termscopy((WORD *)dum, ri+1+AN.poly_num_vars, ABS(nc)); // coefficient to res
02571         res[ri] = ABS(nc) + AN.poly_num_vars + 2; // length
02572         res[ri+res[ri]-1] = nc; // length coefficient
02573         for (int j=i+3; j<i+e[i]-ABS(e[i+e[i]-1]); j+=2)
02574             res[ri+1+var_to_idx.find(e[j])>second] = e[j+1]; // powers
02575         ri += res[ri]; // length
02576     }
02577
02578     res[0] = ri;
02579
02580     // JD: For univariate cases, check whether the ordering is correct or not.
02581     // There are various ways to arrive here, but with an incorrect ordering, such
02582     // as interactions with collect, moving a polyratfun out of another function
02583     // argument, etc.
02584     // If the ordering is not correct, make sure we normalize. In multivariate cases,
02585     // always normalize as before.
02586     if (sort_univar == false && AN.poly_num_vars == 1) {
02587         if (res.number_of_terms() >= 2) {
02588             if (res[2] < res.last_monomial()[2]) {
02589                 sort_univar = true;
02590             }
02591         }
02592     }
02593
02594     if (sort_univar || AN.poly_num_vars>1) {
02595         res.normalize();
02596     }
02597
02598     NumberFree(den, "poly::argument_to_poly");
02599     NumberFree(pro, "poly::argument_to_poly");
02600     NumberFree(gcd, "poly::argument_to_poly");
02601     NumberFree(dum, "poly::argument_to_poly");
02602
02603     return res;
02604 }
02605
02606 /*
02607 #] argument_to_poly :
02608 #[ poly_to_argument :
02609 */
02610 // converts a polynomial class "poly" to a form expression
02611 void poly::poly_to_argument (const poly &a, WORD *res, bool with_arghead) {

```



```

02612
02613     POLY_GETIDENTITY(a);
02614
02615     // special case: a=0
02616     if (a[0]==1) {
02617         if (with_arghead) {
02618             res[0] = -SNUMBER;
02619             res[1] = 0;
02620         }
02621         else {
02622             res[0] = 0;
02623         }
02624         return;
02625     }
02626
02627     if (with_arghead) {
02628         res[1] = AN.poly_num_vars>1 ? DIRTYFLAG : 0; // dirty flag
02629         for (int i=2; i<ARGHEAD; i++)
02630             res[i] = 0; // remainder of arghead
02631     }
02632
02633     int L = with_arghead ? ARGHEAD : 0;
02634
02635     for (int i=1; i!=a[0]; i+=a[i]) {
02636
02637         res[L]=1; // length
02638
02639         bool first=true;
02640
02641         for (int j=0; j<AN.poly_num_vars; j++)
02642             if (a[i+1+j] > 0) {
02643                 if (first) {
02644                     first=false;
02645                     res[L+1] = 1; // symbols
02646                     res[L+2] = 2; // length
02647                 }
02648                 res[L+1+res[L+2]++] = AN.poly_vars[j]; // symbol
02649                 res[L+1+res[L+2]++] = a[i+1+j]; // power
02650             }
02651
02652         if (!first) res[L] += res[L+2]; // fix length
02653
02654         WORD nc = a[i+a[i]-1];
02655         WCOPY(&res[L+res[L]], &a[i+a[i]-1-ABS(nc)], ABS(nc)); // numerator
02656
02657         res[L] += ABS(nc); // fix length
02658         memset(&res[L+res[L]], 0, ABS(nc)*sizeof(WORD)); // denominator one
02659         res[L+res[L]] = 1; // denominator one
02660         res[L] += ABS(nc); // fix length
02661         res[L+res[L]] = SGN(nc) * (2*ABS(nc)+1); // length of coefficient
02662         res[L]++; // fix length
02663         L += res[L]; // fix length
02664     }
02665
02666     if (with_arghead) {
02667         res[0] = L;
02668         // convert to fast notation if possible
02669         ToFast(res,res);
02670     }
02671     else {
02672         res[L] = 0;
02673     }
02674 }
02675
02676 /*
02677 #] poly_to_argument :
02678 #[ poly_to_argument_with_den :
02679 */
02680
02681 // converts a polynomial class "poly" divided by a number (nden, den) to a form expression
02682 // cf. poly::poly_to_argument()
02683 void poly::poly_to_argument_with_den (const poly &a, WORD nden, const UWORD *den, WORD *res, bool
    with_arghead) {
02684
02685     POLY_GETIDENTITY(a);
02686
02687     // special case: a=0
02688     if (a[0]==1) {
02689         if (with_arghead) {
02690             res[0] = -SNUMBER;
02691             res[1] = 0;
02692         }
02693         else {
02694             res[0] = 0;
02695         }
02696         return;
02697     }

```

```

02698
02699     if (with_arghead) {
02700         res[1] = AN.poly_num_vars>1 ? DIRTYFLAG : 0; // dirty flag
02701         for (int i=2; i<ARGHEAD; i++)
02702             res[i] = 0; // remainder of arghead
02703     }
02704
02705     WORD nden1;
02706     UWORD *den1 = (UWORD *)NumberMalloc("poly_to_argument_with_den");
02707
02708     int L = with_arghead ? ARGHEAD : 0;
02709
02710     for (int i=1; i!=a[0]; i+=a[i]) {
02711
02712         res[L]=1; // length
02713
02714         bool first=true;
02715
02716         for (int j=0; j<AN.poly_num_vars; j++)
02717             if (a[i+1+j] > 0) {
02718                 if (first) {
02719                     first=false;
02720                     res[L+1] = 1; // symbols
02721                     res[L+2] = 2; // length
02722                 }
02723                 res[L+1+res[L+2]++] = AN.poly_vars[j]; // symbol
02724                 res[L+1+res[L+2]++] = a[i+1+j]; // power
02725             }
02726
02727         if (!first) res[L] += res[L+2]; // fix length
02728
02729         // numerator
02730         WORD nc = a[i+a[i]-1];
02731         WCOPY(&res[L+res[L]], &a[i+a[i]-1-ABS(nc)], ABS(nc));
02732
02733         // denominator
02734         nden1 = nden;
02735         WCOPY(den1, den, ABS(nden));
02736
02737         if (nden != 1 || den[0] != 1) {
02738             // remove gcd(num,den)
02739             Simplify(BHEAD (UWORD *)&res[L+res[L]], &nc, den1, &nden1);
02740         }
02741
02742         Pack((UWORD *)&res[L+res[L]], &nc, den1, nden1); // format
02743         res[L] += 2*ABS(nc)+1; // fix length
02744         res[L+res[L]-1] = SGN(nc)*(2*ABS(nc)+1); // length of coefficient
02745         L += res[L]; // fix length
02746     }
02747
02748     NumberFree(den1, "poly_to_argument_with_den");
02749
02750     if (with_arghead) {
02751         res[0] = L;
02752         // convert to fast notation if possible
02753         ToFast(res,res);
02754     }
02755     else {
02756         res[L] = 0;
02757     }
02758 }
02759
02760 /*
02761 #] poly_to_argument_with_den :
02762 #[ size_of_form_notation :
02763 */
02764
02765 // the size of the polynomial in form notation (without argheads and fast notation)
02766 int poly::size_of_form_notation () const {
02767
02768     POLY_GETIDENTITY(*this);
02769
02770     // special case: a=0
02771     if (terms[0]==1) return 0;
02772
02773     int len = 0;
02774
02775     for (int i=1; i!=terms[0]; i+=terms[i]) {
02776         len++;
02777         int npow = 0;
02778         for (int j=0; j<AN.poly_num_vars; j++)
02779             if (terms[i+1+j] > 0) npow++;
02780         if (npow > 0) len += 2*npow + 2;
02781         len += 2 * ABS(terms[i+terms[i]-1]) + 1;
02782     }
02783
02784     return len;

```

```

02785 }
02786
02787 /*
02788 #] size_of_form_notation :
02789 #[ size_of_form_notation_with_den :
02790 */
02791
02792 // the size of the polynomial divided by a number (its size is given by nden)
02793 // in form notation (without argheads and fast notation)
02794 // cf. poly::size_of_form_notation()
02795 int poly::size_of_form_notation_with_den (WORD nden) const {
02796
02797     POLY_GETIDENTITY(*this);
02798
02799     // special case: a=0
02800     if (terms[0]==1) return 0;
02801
02802     nden = ABS(nden);
02803     int len = 0;
02804
02805     for (int i=1; i!=terms[0]; i+=terms[i]) {
02806         len++;
02807         int npow = 0;
02808         for (int j=0; j<AN.poly_num_vars; j++)
02809             if (terms[i+1+j] > 0) npow++;
02810         if (npow > 0) len += 2*npow + 2;
02811         len += 2 * MaX(ABS(terms[i+terms[i]-1]), nden) + 1;
02812     }
02813
02814     return len;
02815 }
02816
02817 /*
02818 #] size_of_form_notation_with_den :
02819 #[ to_coefficient_list :
02820 */
02821
02822 // returns the coefficient list of a univariate polynomial
02823 const vector<WORD> poly::to_coefficient_list (const poly &a) {
02824
02825     POLY_GETIDENTITY(a);
02826
02827     if (a.is_zero()) return vector<WORD>();
02828
02829     int x = a.first_variable();
02830     if (x == AN.poly_num_vars) x=0;
02831
02832     vector<WORD> res(1+a[2+x],0);
02833
02834     for (int i=1; i<a[0]; i+=a[i])
02835         res[a[i+1+x]] = (a[i+a[i]-1] * a[i+a[i]-2]) % a.modp;
02836
02837     return res;
02838 }
02839
02840 /*
02841 #] to_coefficient_list :
02842 #[ coefficient_list_divmod :
02843 */
02844
02845 // divides two polynomials represented by coefficient lists
02846 const vector<WORD> poly::coefficient_list_divmod (const vector<WORD> &a, const vector<WORD> &b, WORD
p, int divmod) {
02847
02848     int bsize = (int)b.size();
02849     while (b[bsize-1]==0) bsize--;
02850
02851     WORD inv;
02852     GetModInverses(b[bsize-1] + (b[bsize-1]<0?p:0), p, &inv, NULL);
02853
02854     vector<WORD> q(a.size(),0);
02855     vector<WORD> r(a);
02856
02857     while ((int)r.size() >= bsize) {
02858         LONG mul = ((LONG)inv * r.back()) % p;
02859         int off = r.size()-bsize;
02860         q[off] = mul;
02861         for (int i=0; i<bsize; i++)
02862             r[off+i] = (r[off+i] - mul*b[i]) % p;
02863         while (r.size()>0 && r.back()==0)
02864             r.pop_back();
02865     }
02866
02867     if (divmod==0) {
02868         while (q.size()>0 && q.back()==0)
02869             q.pop_back();
02870         return q;

```

```

02871     }
02872     else {
02873         while (r.size() > 0 && r.back() == 0)
02874             r.pop_back();
02875         return r;
02876     }
02877 }
02878
02879 /*
02880  #] coefficient_list_divmod :
02881  #[ from_coefficient_list :
02882 */
02883
02884 // converts a coefficient list to a "poly"
02885 const poly poly::from_coefficient_list (PHEAD const vector<WORD> &a, int x, WORD p) {
02886     poly res(BHEAD 0);
02887     int ri=1;
02888
02889     for (int i=(int)a.size()-1; i>=0; i--)
02890         if (a[i] != 0) {
02891             res.check_memory(ri);
02892             res[ri] = AN.poly_num_vars+3; // length
02893             for (int j=0; j<AN.poly_num_vars; j++)
02894                 res[ri+1+j]=0; // powers
02895             res[ri+1+x] = i; // power of x
02896             res[ri+1+AN.poly_num_vars] = ABS(a[i]); // coefficient
02897             res[ri+1+AN.poly_num_vars+1] = SGN(a[i]); // sign
02898             ri += res[ri];
02899         }
02900
02901
02902     res[0]=ri; // total length
02903     res.setmod(p,1);
02904
02905     return res;
02906 }
02907
02908 /*
02909  #] from_coefficient_list :
02910 */

```

4.102 poly.h

```

00001 #pragma once
00002 /* #] License : */
00003 /*
00004  * Copyright (C) 1984-2026 J.A.M. Vermaseren
00005  * When using this file you are requested to refer to the publication
00006  * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00007  * This is considered a matter of courtesy as the development was paid
00008  * for by FOM the Dutch physics granting agency and we would like to
00009  * be able to track its scientific use to convince FOM of its value
00010  * for the community.
00011  *
00012  * This file is part of FORM.
00013  *
00014  * FORM is free software: you can redistribute it and/or modify it under the
00015  * terms of the GNU General Public License as published by the Free Software
00016  * Foundation, either version 3 of the License, or (at your option) any later
00017  * version.
00018  *
00019  * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00020  * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00021  * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00022  * details.
00023  *
00024  * You should have received a copy of the GNU General Public License along
00025  * with FORM. If not, see <http://www.gnu.org/licenses/>.
00026 */
00027 /* #] License : */
00028
00029 extern "C" {
00030 #include "form3.h"
00031 }
00032
00033 #include <iostream>
00034 #include <string>
00035 #include <vector>
00036
00037 // for debugging
00038 // #define POLY_MOVE_DEBUG
00039
00040 // macros for tform

```

```

00041 #ifndef WITHPTHREADS
00042 #define POLY_GETIDENTITY(X)
00043 #define POLY_STOREIDENTITY
00044 #else
00045 #define POLY_GETIDENTITY(X) ALLPRIVATES *B = (X).Bpointer
00046 #define POLY_STOREIDENTITY Bpointer = B
00047 #endif
00048
00049 // maximum size of the hash table used for multiplication and division
00050
00051 const int POLY_MAX_HASH_SIZE = Min(1<<20, MAXPOSITIVE);
00052
00053 class poly {
00054 public:
00055     // variables
00056     #ifdef WITHPTHREADS
00057         ALLPRIVATES *Bpointer;
00058     #endif
00059     WORD *terms;
00060     LONG size_of_terms;
00061     WORD modp, modn;
00062
00063     // constructors/destructor
00064     poly (PHEAD int, WORD=-1, WORD=1);
00065     poly (PHEAD const UWORD *, WORD, WORD=-1, WORD=1);
00066     poly (const poly &, WORD=-1, WORD=1);
00067     ~poly ();
00068
00069     // operators
00070     poly& operator+= (const poly&);
00071     poly& operator-= (const poly&);
00072     poly& operator*= (const poly&);
00073     poly& operator/= (const poly&);
00074     poly& operator%= (const poly&);
00075
00076     const poly operator+ (const poly&) const;
00077     const poly operator- (const poly&) const;
00078     const poly operator* (const poly&) const;
00079     const poly operator/ (const poly&) const;
00080     const poly operator% (const poly&) const;
00081
00082     bool operator== (const poly&) const;
00083     bool operator!= (const poly&) const;
00084     poly& operator= (const poly &);
00085     WORD& operator[] (int);
00086     const WORD& operator[] (int) const;
00087
00088     #if __cplusplus >= 201103L
00089         // support move semantics
00090         poly (poly &&, WORD=-1, WORD=1) noexcept;
00091         poly& operator= (poly &&) noexcept;
00092     #endif
00093
00094     // memory management
00095     void termscopy (const WORD *, int, int);
00096     void check_memory(int);
00097     void expand_memory(int);
00098
00099     // check type of polynomial
00100     bool is_zero () const;
00101     bool is_one () const;
00102     bool is_integer () const;
00103     bool is_monomial () const;
00104     int is_dense_univariate () const;
00105
00106     // properties
00107     int sign () const;
00108     int degree (int) const;
00109     int total_degree () const;
00110     int first_variable () const;
00111     int number_of_terms () const;
00112     const std::vector<int> all_variables () const;
00113     const poly integer_lcoeff () const;
00114     const poly lcoeff_univar (int) const;
00115     const poly lcoeff_multivar (int) const;
00116     const poly coefficient (int, int) const;
00117     const poly derivative (int) const;
00118
00119     // modulo calculus
00120     void setmod(WORD, WORD=1);
00121     void coefficients_modulo (UWORD *, WORD, bool);
00122
00123     // simple polynomials
00124     static const poly simple_poly (PHEAD int, int=0, int=1, int=0, int=1);

```

```

00128     static const poly simple_poly (PHEAD int, const poly&, int=1, int=0, int=1);
00129
00130     // conversion from/to form notation
00131     static void get_variables (PHEAD const std::vector<WORD *> &, bool, bool);
00132     static const poly argument_to_poly (PHEAD WORD *, bool, bool, poly *den=NULL);
00133     static void poly_to_argument (const poly &, WORD *, bool);
00134     static void poly_to_argument_with_den (const poly &, WORD, const UWORD *, WORD *, bool);
00135     int size_of_form_notation () const;
00136     int size_of_form_notation_with_den (WORD) const;
00137     const poly & normalize ();
00138
00139     // operations for coefficient lists
00140     static const poly from_coefficient_list (PHEAD const std::vector<WORD> &, int, WORD);
00141     static const std::vector<WORD> to_coefficient_list (const poly &);
00142     static const std::vector<WORD> coefficient_list_divmod (const std::vector<WORD> &, const
std::vector<WORD> &, WORD, int);
00143
00144     // string output for debugging
00145     const std::string to_string() const;
00146
00147     // monomials
00148     static int monomial_compare (PHEAD const WORD *, const WORD *);
00149     WORD last_monomial_index () const;
00150     WORD* last_monomial () const;
00151     int compare_degree_vector(const poly &) const;
00152     std::vector<int> degree_vector() const;
00153     int compare_degree_vector(const std::vector<int> &) const;
00154
00155     // (internal) mathematical operations
00156     static void add (const poly &, const poly &, poly &);
00157     static void sub (const poly &, const poly &, poly &);
00158     static void mul (const poly &, const poly &, poly &);
00159     static void div (const poly &, const poly &, poly &);
00160     static void mod (const poly &, const poly &, poly &);
00161     static void divmod (const poly &, const poly &, poly &, poly &, bool only_divides);
00162     static bool divides (const poly &, const poly &);
00163
00164     static void mul_one_term (const poly &, const poly &, poly &);
00165     static void mul_univar (const poly &, const poly &, poly &, int);
00166     static void mul_heap (const poly &, const poly &, poly &);
00167
00168     static void divmod_one_term (const poly &, const poly &, poly &, poly &, bool);
00169     static void divmod_univar (const poly &, const poly &, poly &, poly &, int, bool);
00170     static void divmod_heap (const poly &, const poly &, poly &, poly &, bool, bool, bool&);
00171
00172     static void push_heap (PHEAD WORD **, int);
00173     static void pop_heap (PHEAD WORD **, int);
00174 };
00175
00176 // comparison class for monomials (for std::sort)
00177 class monomial_larger {
00178 public:
00179
00180     #ifndef WITHPTHREADS
00181     monomial_larger() {}
00182     #else
00183     ALLPRIVATES *B;
00184     monomial_larger(ALLPRIVATES *b): B(b) {}
00185     #endif
00186
00187     bool operator()(const WORD *a, const WORD *b) {
00188         return poly::monomial_compare(BHEAD a, b) > 0;
00189     }
00190 };
00191
00192 // stream operator
00193 std::ostream& operator<< (std::ostream &, const poly &);
00194
00195 // inline function definitions
00196
00197 #if __cplusplus >= 201103L
00198 inline poly::poly (poly &a, WORD modp, WORD modn) noexcept {
00199     #ifdef POLY_MOVE_DEBUG
00200     std::cout << "poly move ctor" << std::endl;
00201     #endif
00202     POLY_GETIDENTITY(a);
00203     POLY_STOREIDENTITY;
00204     terms = a.terms;
00205     size_of_terms = a.size_of_terms;
00206     this->modp = a.modp;
00207     this->modn = a.modn;
00208     if (modp != -1) {
00209         setmod(modp,modn);
00210     }
00211     a.terms = nullptr;
00212 }

```

```

00214     a.size_of_terms = -1;
00215 }
00216
00217 inline poly& poly::operator= (poly &&a) noexcept {
00218 #ifdef POLY_MOVE_DEBUG
00219     std::cout << "poly move assign" << std::endl;
00220 #endif
00221     if (this != &a) {
00222         WORD *old_terms = terms;
00223         LONG old_size_of_terms = size_of_terms;
00224         terms = a.terms;
00225         size_of_terms = a.size_of_terms;
00226         modp = a.modp;
00227         modn = a.modn;
00228         a.terms = old_terms;
00229         a.size_of_terms = old_size_of_terms;
00230     }
00231     return *this;
00232 }
00233
00234 #endif
00235
00236 /* Checks whether the terms array is large enough to add another
00237 * term (of size AM.MaxTal) to the polynomials. In case not, it is
00238 * expanded.
00239 */
00240 inline void poly::check_memory (int i) {
00241     POLY_GETIDENTITY(*this);
00242     if (i + 3 + AN.poly_num_vars + AM.MaxTal >= size_of_terms) expand_memory(i + AM.MaxTal);
00243 // Used to be i+2 but there should also be space for a trailing zero
00244 }
00245
00246 // indexing operators
00247 inline WORD& poly::operator[] (int i) {
00248     return terms[i];
00249 }
00250
00251 inline const WORD& poly::operator[] (int i) const {
00252     return terms[i];
00253 }
00254
00255 /* Copies "num" WORD-sized terms from the pointer "source" to the
00256 * current polynomial at index "dest"
00257 */
00258 inline void poly::termscopy (const WORD *source, int dest, int num) {
00259 #ifdef POLY_MOVE_DEBUG
00260     std::cout << "poly termscopy: num=" << num << std::endl;
00261 #endif
00262     memcpy (terms+dest, source, num*sizeof(WORD));
00263 }

```

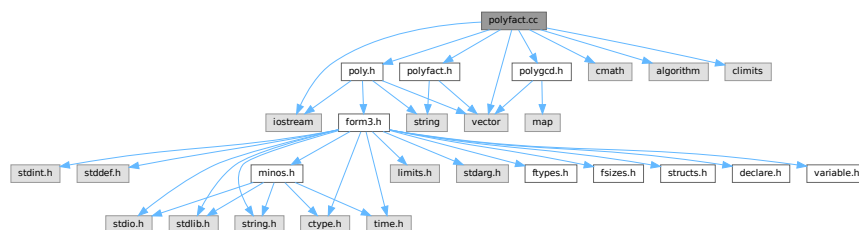
4.103 polyfact.cc File Reference

```

#include "poly.h"
#include "polygcd.h"
#include "polyfact.h"
#include <cmath>
#include <vector>
#include <iostream>
#include <algorithm>
#include <climits>

```

Include dependency graph for polyfact.cc:



Functions

- ostream & [operator<<](#) (ostream &out, const [factorized_poly](#) &a)
- template<class T >
ostream & [operator<<](#) (ostream &out, const vector< T > &v)

4.103.1 Detailed Description

Contains the routines for factorizing multivariate polynomials

Definition in file [polyfact.cc](#).

4.103.2 Function Documentation

4.103.2.1 [operator<<\(\)](#) [1/2]

```
ostream & operator<< (
    ostream & out,
    const factorized\_poly & a )
```

Definition at line 104 of file [polyfact.cc](#).

4.103.2.2 [operator<<\(\)](#) [2/2]

```
template<class T >
ostream & operator<< (
    ostream & out,
    const vector< T > & v )
```

Definition at line 109 of file [polyfact.cc](#).

4.104 polyfact.cc

[Go to the documentation of this file.](#)

```
00001
00005 /* # License : */
00006 /*
00007 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00008 * When using this file you are requested to refer to the publication
00009 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010 * This is considered a matter of courtesy as the development was paid
00011 * for by FOM the Dutch physics granting agency and we would like to
00012 * be able to track its scientific use to convince FOM of its value
00013 * for the community.
00014 *
00015 * This file is part of FORM.
00016 *
00017 * FORM is free software: you can redistribute it and/or modify it under the
00018 * terms of the GNU General Public License as published by the Free Software
00019 * Foundation, either version 3 of the License, or (at your option) any later
00020 * version.
00021 *
00022 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00025 * details.
00026 *
00027 * You should have received a copy of the GNU General Public License along
```



```

00028  *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00029  */
00030  /* #] License : */
00031  /*
00032     #[ include :
00033  */
00034
00035  #include "poly.h"
00036  #include "polygcd.h"
00037  #include "polyfact.h"
00038
00039  #include <cmath>
00040  #include <vector>
00041  #include <iostream>
00042  #include <algorithm>
00043  #include <climits>
00044
00045  // #define DEBUG
00046
00047  #ifdef DEBUG
00048  #include "mytime.h"
00049  #endif
00050
00051  using namespace std;
00052
00053  /*
00054     #] include :
00055     #[ tostring :
00056  */
00057
00058  // Turns a factorized_poly into a readable string
00059  const string factorized_poly::tostring () const {
00060
00061     // empty
00062     if (factor.size()==0)
00063         return "no_factors";
00064
00065     string res;
00066
00067     // polynomial
00068     for (int i=0; i<(int)factor.size(); i++) {
00069         if (i>0) res += "*";
00070         res += "(";
00071         res += poly(factor[i],0,1).to_string();
00072         res += ")";
00073         if (power[i]>1) {
00074             res += "^";
00075             char tmp[100];
00076             snprintf (tmp,100,"%i",power[i]);
00077             res += tmp;
00078         }
00079     }
00080
00081     // modulo p^n
00082     if (factor[0].modp>0) {
00083         res += "(mod ";
00084         char tmp[12];
00085         snprintf (tmp,12,"%i",factor[0].modp);
00086         res += tmp;
00087         if (factor[0].modn > 1) {
00088             snprintf (tmp,12,"%i",factor[0].modn);
00089             res += "^";
00090             res += tmp;
00091         }
00092         res += ")";
00093     }
00094
00095     return res;
00096 }
00097
00098 /*
00099     #] tostring :
00100     #[ ostream operator :
00101  */
00102
00103  // ostream operator for outputting a factorized_poly
00104  ostream& operator<< (ostream &out, const factorized_poly &a) {
00105     return out << a.tostring();
00106 }
00107
00108  // ostream operator for outputting a vector<T>
00109  template<class T> ostream& operator<< (ostream &out, const vector<T> &v) {
00110     out<<"{";
00111     for (int i=0; i<(int)v.size(); i++) {
00112         if (i>0) out<<",";
00113         out<<v[i];
00114     }

```

```

00115     out<<"}";
00116     return out;
00117 }
00118
00119 /*
00120 #] ostream operator :
00121 #[ add_factor :
00122 */
00123
00124 // adds a factor f^p to a factorization
00125 void factorized_poly::add_factor(const poly &f, int p) {
00126     factor.push_back(f);
00127     power.push_back(p);
00128 }
00129
00130 /*
00131 #] add_factor :
00132 #[ extended_gcd_Euclidean_lifted :
00133 */
00134
00152 const vector<poly> polyfact::extended_gcd_Euclidean_lifted (const poly &a, const poly &b) {
00153
00154 #ifdef DEBUGALL
00155     cout << "*** [" << thetime() << "]" CALL: extended_Euclidean_lifted("«a«","«b«")"«endl;
00156 #endif
00157
00158     POLY_GETIDENTITY(a);
00159
00160     // Calculate s,t,gcd (mod p) with the Extended Euclidean Algorithm.
00161     poly s(a,a.modp,1);
00162     poly t(b,b.modp,1);
00163     poly sa(BHEAD 1,a.modp,1);
00164     poly sb(BHEAD 0,b.modp,1);
00165     poly ta(BHEAD 0,a.modp,1);
00166     poly tb(BHEAD 1,b.modp,1);
00167
00168     while (!t.is_zero()) {
00169         poly x(s/t);
00170         swap(s -=x*t , t);
00171         swap(sa-=x*ta, ta);
00172         swap(sb-=x*tb, tb);
00173     }
00174
00175     // Normalize the result.
00176     sa /= s.integer_lcoeff();
00177     sb /= s.integer_lcoeff();
00178
00179     // Lift the result to modulo p^n with p-adic Newton's iteration.
00180     poly samodp(sa);
00181     poly sbmodp(sb);
00182     poly term(BHEAD 1);
00183
00184     sa.setmod(0,1);
00185     sb.setmod(0,1);
00186
00187     poly amodp(a,a.modp,1);
00188     poly bmodp(b,a.modp,1);
00189     poly error(poly(BHEAD 1) - sa*a - sb*b);
00190     poly p(BHEAD a.modp);
00191
00192     for (int n=1; n<a.modn && !error.is_zero(); n++) {
00193         error /= p;
00194         term *= p;
00195         poly errormodp(error,a.modp,1);
00196         poly dsa((samodp * errormodp) % bmodp);
00197         // equivalent, but faster than the symmetric
00198         // poly dsb((sbmodp * errormodp) % amodp);
00199         poly dsb((errormodp - dsa*amodp) / bmodp);
00200         sa += term * dsa;
00201         sb += term * dsb;
00202         error -= a*dsa + b*dsb;
00203     }
00204
00205     sa.setmod(a.modp,a.modn);
00206     sb.setmod(a.modp,a.modn);
00207
00208     // Output the result
00209     vector<poly> res;
00210     res.push_back(sa);
00211     res.push_back(sb);
00212
00213 #ifdef DEBUGALL
00214     cout << "*** [" << thetime() << "]" RES : extended_Euclidean_lifted("«a«","«b«") = "«res«endl;
00215 #endif
00216
00217     return res;
00218 }

```

```

00219
00220 /*
00221 #] extended_gcd_Euclidean_lifted :
00222 #[ solve_Diophantine_univariate :
00223 */
00224
00256 const vector<poly> polyfact::solve_Diophantine_univariate (const vector<poly> &a, const poly &b) {
00257
00258 #ifdef DEBUGALL
00259     cout << "*** [" << thetime() << "]" CALL: solve_Diophantine_univariate(" «a«", "«b«")" << endl;
00260 #endif
00261
00262     POLY_GETIDENTITY(b);
00263
00264     vector<poly> s(1,b);
00265
00266     for (int i=0; i+1<(int)a.size(); i++) {
00267         poly A(BHEAD 1,b.modp,b.modn);
00268         for (int j=i+1; j<(int)a.size(); j++) A *= a[j];
00269
00270         vector<poly> t(extended_gcd_Euclidean_lifted(a[i],A));
00271         poly prev(s.back());
00272         s.back() = t[1] * prev % a[i];
00273         s.push_back(t[0] * prev % A);
00274     }
00275
00276 #ifdef DEBUGALL
00277     cout << "*** [" << thetime() << "]" RES : solve_Diophantine_univariate(" «a«", "«b«") = "«s«<< endl;
00278 #endif
00279
00280     return s;
00281 }
00282
00283 /*
00284 #] solve_Diophantine_univariate :
00285 #[ solve_Diophantine_multivariate :
00286 */
00287
00319 const vector<poly> polyfact::solve_Diophantine_multivariate (const vector<poly> &a, const poly &b,
const vector<int> &x, const vector<int> &c, int d) {
00320
00321 #ifdef DEBUGALL
00322     cout << "*** [" << thetime() << "]" CALL: solve_Diophantine_multivariate("
«a«", "«b«", "«x«", "«c«", "«d«")" << endl;
00323 #endif
00324
00325     POLY_GETIDENTITY(b);
00326
00327     if (b.is_zero()) return vector<poly>(a.size(), poly(BHEAD 0));
00328
00329     if (x.size() == 1) return solve_Diophantine_univariate(a,b);
00330
00331     // Reduce the polynomial with the homomorphism <xm-c{m-1}>
00332     poly simple(poly::simple_poly(BHEAD x.back(),c.back()));
00333
00334     vector<poly> ared(a);
00335     for (int i=0; i<(int)ared.size(); i++)
00336         ared[i] %= simple;
00337
00338     poly bred(b % simple);
00339     vector<int> xred(x.begin(),x.end()-1);
00340     vector<int> cred(c.begin(),c.end()-1);
00341
00342     // Solve the equation in one less variable
00343     vector<poly> s(solve_Diophantine_multivariate(ared,bred,xred,cred,d));
00344     if (s == vector<poly>()) return vector<poly>();
00345
00346     // Cache the Ai = product(aj | j!=i).
00347     vector<poly> A(a.size(), poly(BHEAD 1,b.modp,b.modn));
00348     for (int i=0; i<(int)a.size(); i++)
00349         for (int j=0; j<(int)a.size(); j++)
00350             if (i!=j) A[i] *= a[j];
00351
00352     // Add the powers (xm-c{m-1})^k with ideal-adic Newton iteration.
00353     poly term(BHEAD 1,b.modp,b.modn);
00354
00355     poly error(b);
00356     for (int i=0; i<(int)A.size(); i++)
00357         error -= s[i] * A[i];
00358
00359     for (int deg=1; deg<=d; deg++) {
00360
00361         if (error.is_zero()) break;
00362
00363         error /= simple;
00364         term *= simple;
00365

```

```

00366         vector<poly> ds(solve_Diophantine_multivariate(ared, error%simple, xred, cred, d));
00367         if (ds == vector<poly>()) return vector<poly>();
00368
00369         for (int i=0; i<(int)s.size(); i++) {
00370             s[i] += ds[i] * term;
00371             error -= ds[i] * A[i];
00372         }
00373     }
00374
00375     if (!error.is_zero()) return vector<poly>();
00376
00377 #ifdef DEBUGALL
00378     cout << "*** [" << thetime() << "]" RES : solve_Diophantine_multivariate("
00379     <<a<<","<b<<","<x<<","<c<<","<d<<") = "<s<<endl;
00380 #endif
00381     return s;
00382 }
00383
00384 /*
00385 #] solve_Diophantine_multivariate :
00386 #[ lift_coefficients :
00387 */
00388
00421 const vector<poly> polyfact::lift_coefficients (const poly &_A, const vector<poly> &_a) {
00422
00423 #ifdef DEBUG
00424     cout << "*** [" << thetime() << "]" CALL: lift_coefficients("<_A<<","<_a<<")<<endl;
00425 #endif
00426
00427     POLY_GETIDENTITY(_A);
00428
00429     poly A(_A);
00430     vector<poly> a(_a);
00431     poly term(BHEAD 1);
00432
00433     int x = A.first_variable();
00434
00435     // Replace the leading term of all factors with lterm(A) mod p
00436     poly lead(A.integer_lcoeff());
00437     for (int i=0; i<(int)a.size(); i++) {
00438         a[i] *= lead / a[i].integer_lcoeff();
00439         if (i>0) A*=lead;
00440     }
00441
00442     // Solve Diophantine equation
00443     vector<poly> s(solve_Diophantine_univariate(a,poly(BHEAD 1,A.modp,1)));
00444
00445     // Replace the leading term of all factors with lterm(A) mod p^n
00446     for (int i=0; i<(int)a.size(); i++) {
00447         a[i].setmod(A.modp,A.modn);
00448         a[i] += (lead - a[i].integer_lcoeff()) * poly::simple_poly(BHEAD x,0,a[i].degree(x));
00449     }
00450
00451     // Calculate the error, express it in terms of ai and add corrections.
00452     for (int k=2; k<=A.modn; k++) {
00453         term *= poly(BHEAD A.modp);
00454
00455         poly error(BHEAD -1);
00456         for (int i=0; i<(int)a.size(); i++) error *= a[i];
00457         error += A;
00458         if (error.is_zero()) break;
00459
00460         error /= term;
00461         error.setmod(A.modp,1);
00462
00463         for (int i=0; i<(int)a.size(); i++)
00464             a[i] += term * (error * s[i] % a[i]);
00465     }
00466
00467     // Fix leading coefficients by dividing out integer contents.
00468     for (int i=0; i<(int)a.size(); i++)
00469         a[i] /= polygcd::integer_content(poly(a[i],0,1));
00470
00471 #ifdef DEBUG
00472     cout << "*** [" << thetime() << "]" RES : lift_coefficients("<_A<<","<_a<<") = "<a<<endl;
00473 #endif
00474     return a;
00475 }
00476
00477 /*
00478 #] lift_coefficients :
00479 #[ predetermine :
00480 */
00481
00482

```

```

00497 void polyfact::predetermine (int dep, const vector<vector<int> > &state, vector<vector<vector<int> > >
    &terms, vector<int> &term, int sumdeg) {
00498     // store the term
00499     if (dep == (int)state.size()) {
00500         terms[sumdeg].push_back(term);
00501         return;
00502     }
00503
00504     // recursively create new terms
00505     term.push_back(0);
00506
00507     for (int deg=0; sumdeg+deg<(int)state[dep].size(); deg++)
00508         if (state[dep][deg] > 0) {
00509             term.back() = deg;
00510             predetermine(dep+1, state, terms, term, sumdeg+deg);
00511         }
00512
00513     term.pop_back();
00514 }
00515
00516 /*
00517     #] predetermine :
00518     #[ lift_variables :
00519 */
00520
00558 const vector<poly> polyfact::lift_variables (const poly &A, const vector<poly> &a, const vector<int>
    &x, const vector<int> &c, const vector<poly> &lc) {
00559
00560 #ifdef DEBUG
00561     cout << "*** [" << thetime() << " ] CALL: lift_variables(" << A << ", " << a << ", " << x << ", " << c << ", " << lc << ") \n";
00562 #endif
00563
00564     // If univariate, don't lift
00565     if (x.size() <= 1) return _a;
00566
00567     POLY_GETIDENTITY(A);
00568
00569     vector<poly> a(_a);
00570
00571     // First method: predetermine coefficients
00572
00573     // check feasibility, otherwise it tries too many possibilities
00574     int cnt = POLYFACT_MAX_PREDETERMINATION;
00575     for (int i=0; i<(int)a.size(); i++) {
00576         if (a[i].number_of_terms() == 0) return vector<poly>();
00577         cnt /= a[i].number_of_terms();
00578     }
00579
00580     if (cnt>0) {
00581         // state[n][d]: coefficient of x^d in a[n] is
00582         // 0: non-existent, 1: undetermined, 2: determined
00583         int D = A.degree(x[0]);
00584         vector<vector<int> > state(a.size(), vector<int>(D+1, 0));
00585
00586         for (int i=0; i<(int)a.size(); i++)
00587             for (int j=1; j<a[i][0]; j+=a[i][j])
00588                 state[i][a[i][j+1+x[0]]] = j==1 ? 2 : 1;
00589
00590         // collect all products of terms
00591         vector<vector<vector<int> > > terms(D+1);
00592         vector<int> term;
00593         predetermine(0, state, terms, term);
00594
00595         // count the number of undetermined coefficients
00596         vector<int> num_undet(terms.size(), 0);
00597         for (int i=0; i<(int)terms.size(); i++)
00598             for (int j=0; j<(int)terms[i].size(); j++)
00599                 for (int k=0; k<(int)terms[i][j].size(); k++)
00600                     if (state[k][terms[i][j][k]] == 1) num_undet[i]++;
00601
00602         // replace the current leading coefficients by the correct ones
00603         for (int i=0; i<(int)a.size(); i++)
00604             a[i] += (lc[i] - a[i].lcoeff_univar(x[0])) * poly::simple_poly(BHEAD
x[0], 0, a[i].degree(x[0]));
00605
00606         bool changed;
00607         do {
00608             changed = false;
00609
00610             for (int i=0; i<(int)terms.size(); i++) {
00611                 // is only one coefficient in a equation is undetermined, solve
00612                 // the equation to determine this coefficient
00613                 if (num_undet[i] == 1) {
00614                     // generate equation
00615                     poly lhs(BHEAD 0), rhs(A.coefficient(x[0], i), A.modp, A.modn);
00616                     int which_idx=-1, which_deg=-1;
00617                     for (int j=0; j<(int)terms[i].size(); j++) {

```

```

00618         poly coeff(BHEAD 1, A.modp, A.modn);
00619         bool undet=false;
00620         for (int k=0; k<(int)terms[i][j].size(); k++) {
00621             if (state[k][terms[i][j][k]] == 1) {
00622                 undet = true;
00623                 which_idx=k;
00624                 which_deg=terms[i][j][k];
00625             }
00626             else
00627                 coeff *= a[k].coefficient(x[0], terms[i][j][k]);
00628         }
00629         if (undet)
00630             lhs = coeff;
00631         else
00632             rhs -= coeff;
00633     }
00634     // solve equation
00635     if (A.modn > 1) rhs.setmod(0,1);
00636     if (lhs.is_zero() || !(rhs%lhs).is_zero()) return vector<poly>();
00637     a[which_idx] += (rhs / lhs - a[which_idx].coefficient(x[0],which_deg)) *
poly::simple_poly(BHEAD x[0],0,which_deg);
00638     state[which_idx][which_deg] = 2;
00639
00640     // update number of undetermined coefficients
00641     for (int j=0; j<(int)terms.size(); j++)
00642         for (int k=0; k<(int)terms[j].size(); k++)
00643             if (terms[j][k][which_idx] == which_deg)
00644                 num_undet[j]--;
00645
00646     changed = true;
00647 }
00648 }
00649 }
00650 while (changed);
00651
00652 // if this is the complete result, skip lifting
00653 poly check(BHEAD 1, A.modn>1?A.modp, 1);
00654 for (int i=0; i<(int)a.size(); i++)
00655     check *= a[i];
00656
00657 if (check == A) return a;
00658 }
00659
00660 // Second method: Hensel lifting
00661
00662 // Calculate A and lc's modulo Ii = <xi-c{i-1},...,xm-c{m-1}> (for i=2,...,m)
00663 vector<poly> simple(x.size(), poly(BHEAD 0));
00664 for (int i=(int)x.size()-2; i>=0; i--)
00665     simple[i] = poly::simple_poly(BHEAD x[i+1],c[i],1);
00666
00667 // Calculate the maximum degree of A in x2,...,xm
00668 int maxdegA=0;
00669 for (int i=1; i<(int)x.size(); i++)
00670     maxdegA = MaX(maxdegA, A.degree(x[i]));
00671
00672 // Iteratively add the variables x2,...,xm
00673 for (int xi=1; xi<(int)x.size(); xi++) {
00674     // replace the current leading coefficients by the correct ones
00675     for (int i=0; i<(int)a.size(); i++)
00676         a[i] += (lc[i] - a[i].lccoeff_univar(x[0])) * poly::simple_poly(BHEAD
x[0],0,a[i].degree(x[0]));
00677
00678     vector<poly> anew(a);
00679     for (int i=0; i<(int)anew.size(); i++)
00680         for (int j=xi-1; j<(int)c.size(); j++)
00681             anew[i] %= simple[j];
00682
00683     vector<int> xnew(x.begin(), x.begin()+xi);
00684     vector<int> cnew(c.begin(), c.begin()+xi-1);
00685     poly term(BHEAD 1,A.modp,A.modn);
00686
00687     // Iteratively add the powers xi^k
00688     for (int deg=1, maxdeg=A.degree(x[xi]); deg<=maxdeg; deg++) {
00689
00690         term *= simple[xi-1];
00691
00692         // Calculate the error, express it in terms of ai and add corrections.
00693         poly error(BHEAD -1,A.modp,A.modn);
00694         for (int i=0; i<(int)a.size(); i++) error += a[i];
00695         error += A;
00696         for (int i=xi; i<(int)c.size(); i++) error %= simple[i];
00697
00698         if (error.is_zero()) break;
00699
00700         error /= term;
00701         error %= simple[xi-1];
00702

```

```

00703         vector<poly> s(solve_Diophantine_multivariate(anew,error,xnew,cnew,maxdegA));
00704         if (s == vector<poly>()) return vector<poly>();
00705
00706         for (int i=0; i<(int)a.size(); i++)
00707             a[i] += s[i] * term;
00708     }
00709
00710     // check whether PRODUCT(a[i]) = A mod <xi-c{i-1},...,xm-c{m-1}> over the integers or ZZ/p
00711     poly check(BHEAD -1, A.modn>1?0:A.modp, 1);
00712     for (int i=0; i<(int)a.size(); i++) check *= a[i];
00713     check += A;
00714     for (int i=xi; i<(int)c.size(); i++) check %= simple[i];
00715     if (!check.is_zero()) return vector<poly>();
00716 }
00717
00718 #ifdef DEBUG
00719     cout << "*** [" << thetime() << " ] RES : lift_variables("«A«", "«_a«", "«x«", "«c«", "«lc«") = " << a <<
00720     endl;
00721 #endif
00722     return a;
00723 }
00724
00725 /*
00726 #] lift_variables :
00727 #[ choose_prime :
00728 */
00729
00741 WORD polyfact::choose_prime (const poly &a, const vector<int> &x, WORD p) {
00742
00743     #ifdef DEBUG
00744         cout << "*** [" << thetime() << " ] CALL: choose_prime("«a«", "«x«", "«p«")"«endl;
00745     #endif
00746
00747     POLY_GETIDENTITY(a);
00748
00749     poly icont_lcoeff(polygcd::integer_content(a.lcoeff_univar(x[0])));
00750
00751     if (p==0) p = POLYFACT_FIRST_PRIME;
00752
00753     poly icont_lcoeff_modp(BHEAD 0);
00754
00755     do {
00756
00757         bool is_prime;
00758
00759         do {
00760             p += 2;
00761             is_prime = true;
00762             for (int d=2; d*d<=p; d++)
00763                 if (p%d==0) { is_prime=false; break; }
00764         }
00765         while (!is_prime);
00766
00767         icont_lcoeff_modp = icont_lcoeff;
00768         icont_lcoeff_modp.setmod(p,1);
00769     }
00770     while (icont_lcoeff_modp.is_zero());
00771
00772     #ifdef DEBUG
00773         cout << "*** [" << thetime() << " ] RES : choose_prime("«a«", "«x«", "?") = "«p«endl;
00774     #endif
00775
00776     return p;
00777 }
00778
00779 /*
00780 #] choose_prime :
00781 #[ choose_prime_power :
00782 */
00783
00798 WORD polyfact::choose_prime_power (const poly &a, WORD p) {
00799
00800     #ifdef DEBUG
00801         cout << "*** [" << thetime() << " ] CALL: choose_prime_power("«a«", "«p«")"«endl;
00802     #endif
00803
00804     POLY_GETIDENTITY(a);
00805
00806     // analyse the polynomial for calculating the bound
00807     double maxdegree=0, maxlogcoeff=0, numterms=0;
00808
00809     for (int i=1; i<a[0]; i+=a[i]) {
00810         for (int j=0; j<AN.poly_num_vars; j++)
00811             maxdegree = MaX(maxdegree, a[i+1+j]);
00812
00813         maxlogcoeff = MaX(maxlogcoeff,

```

```

00814                                     log(1.0+(UWORD)a[i+a[i]-2]) +           // most
significant digit + 1
00815                                     BITSINWORD*log(2.0)*(ABS(a[i+a[i]-1])-1)); // number of
digits
00816         numterms++;
00817     }
00818
00819     WORD res = (WORD)ceil((log((sqrt(5.0)+1)/2)*maxdegree + maxlogcoeff + 0.5*log(numterms)) /
log((double)p));
00820
00821 #ifdef DEBUG
00822     cout << "*** [" << thetime() << "] CALL: choose_prime_power(" <<a<<","<p<<") = "<res<<endl;
00823 #endif
00824
00825     return res;
00826 }
00827
00828 /*
00829 #] choose_prime_power :
00830 #[ choose_ideal :
00831 */
00832
00860 const vector<int> polyfact::choose_ideal (const poly &a, int p, const factorized_poly &lc, const
vector<int> &x) {
00861
00862 #ifdef DEBUG
00863     cout << "*** [" << thetime() << "] CALL: polyfact::choose_ideal("
00864         <<a<<","<p<<","<lc<<","<x<<")"<<endl;
00865 #endif
00866
00867     if (x.size()==1) return vector<int>();
00868
00869     POLY_GETIDENTITY(a);
00870
00871     vector<int> c(x.size()-1);
00872
00873     int dega = a.degree(x[0]);
00874     poly amodI(a);
00875
00876     // choose random c
00877     for (int i=0; i<(int)c.size(); i++) {
00878         c[i] = 1 + wranf(BHEAD0) % ((p-1) / POLYFACT_IDEAL_FRACTION);
00879         amodI %= poly::simple_poly(BHEAD x[i+1],c[i],1);
00880     }
00881
00882     poly amodIp(amodI);
00883     amodIp.setmod(p,1);
00884
00885     // check if leading coefficient is non-zero [equivalent to degree=old_degree]
00886     if (amodIp.degree(x[0]) != dega)
00887         return c = vector<int>();
00888
00889     // check if leading coefficient is squarefree [equivalent to gcd(a,a')==const]
00890     if (!polygcd::gcd_Euclidean(amodIp, amodIp.derivative(x[0])).is_integer())
00891         return c = vector<int>();
00892
00893     if (a.modp>0 && a.modn==1) return c;
00894
00895     // check for unique prime factors in each factor lc[i] of the leading coefficient
00896     vector<poly> d(1, polygcd::integer_content(amodI));
00897
00898     for (int i=0; i<(int)lc.factor.size(); i++) {
00899         // constant factor
00900         if (i==0 && lc.factor[i].is_integer()) {
00901             d[0] *= lc.factor[i];
00902             continue;
00903         }
00904
00905         // factor modulo I
00906         poly q(lc.factor[i]);
00907         for (int j=0; j<(int)c.size(); j++)
00908             q %= poly::simple_poly(BHEAD x[j+1],c[j]);
00909         if (q.sign() == -1) q *= poly(BHEAD -1);
00910
00911         // divide out common factors
00912         for (int j=(int)d.size()-1; j>=0; j--) {
00913             poly r(d[j]);
00914             while (!r.is_one()) {
00915                 r = polygcd::integer_gcd(r,q);
00916                 q /= r;
00917             }
00918         }
00919
00920         // check whether there is some factor left
00921         if (q.is_one()) return vector<int>();
00922         d.push_back(q);
00923     }

```



```

00924
00925 #ifdef DEBUG
00926     cout << "*** [" << thetime() << "]" RES : polyfact::choose_ideal("
00927         <<a<<","<p<<","<lc<<","<x<<") = "<c<<endl;
00928 #endif
00929
00930     return c;
00931 }
00932
00933 /*
00934 #] choose_ideal :
00935 #[ squarefree_factors_Yun :
00936 */
00937
00944 const factorized_poly polyfact::squarefree_factors_Yun (const poly &a) {
00945     factorized_poly res;
00946     poly a(_a);
00947
00948     int pow = 1;
00949     int x = a.first_variable();
00950
00951     poly b(a.derivative(x));
00952     poly c(polygcd::gcd(a,b));
00953
00954     while (true) {
00955         a /= c;
00956         b /= c;
00957         b -= a.derivative(x);
00958         if (b.is_zero()) break;
00959         c = polygcd::gcd(a,b);
00960         if (!c.is_one()) res.add_factor(c,pow);
00961         pow++;
00962     }
00963
00964     if (!a.is_one()) res.add_factor(a,pow);
00965     return res;
00966 }
00967
00968 /*
00969 #] squarefree_factors_Yun :
00970 #[ squarefree_factors_modp :
00971 */
00972
00973 const factorized_poly polyfact::squarefree_factors_modp (const poly &a) {
00974     factorized_poly res;
00975     poly a(_a);
00976
00977     int pow = 1;
00978     int x = a.first_variable();
00979     poly b(a.derivative(x));
00980
00981     // poly contains terms of the form c(x)^n (n!=c*p)
00982     if (!b.is_zero()) {
00983         poly c(polygcd::gcd(a,b));
00984         a /= c;
00985
00986         while (!a.is_one()) {
00987             b = polygcd::gcd(a,c);
00988             a /= b;
00989             if (!a.is_one()) res.add_factor(a,pow);
00990             pow++;
00991             a = b;
00992             c /= a;
00993         }
00994
00995         a = c;
00996     }
00997
00998     // polynomial contains terms of the form c(x)^p
00999     if (!a.is_one()) {
01000         for (int i=1; i<a[1]; i+=a[i])
01001             a[i+1+x] /= a.modp;
01002         factorized_poly res2(squarefree_factors(a));
01003         for (int i=0; i<(int)res2.factor.size(); i++) {
01004             res.factor.push_back(res2.factor[i]);
01005             res.power.push_back(a.modp*res2.power[i]);
01006         }
01007     }
01008
01009     return res;
01010 }
01011
01012 /*
01013 #] squarefree_factors_modp :
01014 #[ squarefree_factors :

```

```

01023 */
01024
01045 const factorized_poly polyfact::squarefree_factors (const poly &a) {
01046
01047 #ifdef DEBUG
01048     cout << "*** [" << thetime() << "]" CALL: squarefree_factors("«a«")\n";
01049 #endif
01050
01051     if (a.is_one()) return factorized_poly();
01052
01053     factorized_poly res;
01054
01055     if (a.modp==0)
01056         res = squarefree_factors_Yun(a);
01057     else
01058         res = squarefree_factors_modp(a);
01059
01060 #ifdef DEBUG
01061     cout << "*** [" << thetime() << "]" RES : squarefree_factors("«a«") = "«res«"\n";
01062 #endif
01063
01064     return res;
01065 }
01066
01067 /*
01068 #] squarefree_factors :
01069 #[ Berlekamp_Qmatrix :
01070 */
01071
01094 const vector<vector<WORD> > polyfact::Berlekamp_Qmatrix (const poly &a) {
01095
01096 #ifdef DEBUG
01097     cout << "*** [" << thetime() << "]" CALL: Berlekamp_Qmatrix("«a«")\n";
01098 #endif
01099
01100     if (_a.all_variables() == vector<int>())
01101         return vector<vector<WORD> >(0);
01102
01103     POLY_GETIDENTITY(_a);
01104
01105     poly a(_a);
01106     int x = a.first_variable();
01107     int n = a.degree(x);
01108     int p = a.modp;
01109
01110     poly lc(a.integer_lcoeff());
01111     a /= lc;
01112
01113     vector<vector<WORD> > Q(n, vector<WORD>(n));
01114
01115     // c is the vector of coefficients of the polynomial a
01116     vector<WORD> c(n+1,0);
01117     for (int j=1; j<a[0]; j+=a[j])
01118         c[a[j+1+x]] = a[j+1+AN.poly_num_vars] * a[j+2+AN.poly_num_vars];
01119
01120     // d is the vector of coefficients of x^i mod a, starting with i=0
01121     vector<WORD> d(n,0);
01122     d[0]=1;
01123
01124     for (int i=0; i<=(n-1)*p; i++) {
01125         // store the coefficients of x^(i*p) mod a
01126         if (i%p==0) Q[i/p] = d;
01127
01128         // transform d=x^i mod a into d=x^(i+1) mod a
01129         vector<WORD> e(n);
01130         for (int j=0; j<n; j++) {
01131             e[j] = (-(LONG)d[n-1]*c[j] + (j>0?d[j-1]:0)) % p;
01132             if (e[j]<0) e[j]+=p;
01133         }
01134
01135         d=e;
01136     }
01137
01138     // Q = Q - I
01139     for (int i=0; i<n; i++)
01140         Q[i][i] = (Q[i][i] - 1 + p) % p;
01141
01142     // Gaussian elimination
01143     for (int i=0; i<n; i++) {
01144         // Find pivot
01145         int ii=i; while (ii<n && Q[ii][ii]==0) ii++;
01146         if (ii==n) continue;
01147
01148         for (int k=0; k<n; k++)
01149             swap(Q[k][ii],Q[k][i]);
01150
01151         // normalize row i, which becomes the pivot

```

```

01152     WORD mul;
01153     GetModInverses (Q[i][i], p, &mul, NULL);
01154     vector<int> idx;
01155
01156     for (int k=0; k<n; k++) if (Q[k][i] != 0) {
01157         // store indices of non-zero elements for sparse matrices
01158         idx.push_back(k);
01159         Q[k][i] = ((LONG)Q[k][i] * mul) % p;
01160     }
01161
01162     // reduce
01163     for (int j=0; j<n; j++)
01164         if (j!=i && Q[i][j]!=0) {
01165             LONG mul = Q[i][j];
01166             for (int k=0; k<(int)idx.size(); k++) {
01167                 Q[idx[k]][j] = (Q[idx[k]][j] - mul*Q[idx[k]][i]) % p;
01168                 if (Q[idx[k]][j] < 0) Q[idx[k]][j] += p;
01169             }
01170         }
01171     }
01172
01173     for (int i=0; i<n; i++) {
01174         // Q = Q - I
01175         Q[i][i] = Q[i][i]-1;
01176
01177         // reduce all coefficients in the range 0,1,...,p-1
01178         for (int j=0; j<n; j++) {
01179             Q[i][j] = -Q[i][j]%p;
01180             if (Q[i][j]<0) Q[i][j] += p;
01181         }
01182     }
01183
01184 #ifdef DEBUG
01185     cout << "*** [" << thetime() << "] RES : Berlekamp_Qmatrix(\"_a\") = \"Q\"\\n";
01186 #endif
01187     return Q;
01188 }
01189
01190 /*
01191 #] Berlekamp_Qmatrix :
01192 #[ Berlekamp_find_factors :
01193 */
01194
01219 const vector<poly> polyfact::Berlekamp_find_factors (const poly &a, const vector<vector<WORD> > &q)
01220 {
01221 #ifdef DEBUG
01222     cout << "*** [" << thetime() << "] CALL: Berlekamp_find_factors(\"_a\", \"_q\")\\n";
01223 #endif
01224
01225     if (_a.all_variables() == vector<int>()) return vector<poly>(1, _a);
01226
01227     POLY_GETIDENTITY(_a);
01228
01229     vector<vector<WORD> > q=_q;
01230
01231     int rank=0;
01232     for (int i=0; i<(int)q.size(); i++)
01233         if (q[i]!=vector<WORD>(q[i].size(),0)) rank++;
01234
01235     poly a(_a);
01236     int x = a.first_variable();
01237     int n = a.degree(x);
01238     int p = a.modp;
01239
01240     a /= a.integer_lcoeff();
01241
01242     // Vector of factors, represented as dense polynomials mod p
01243     vector<vector<WORD> > fac(1, vector<WORD>(n+1,0));
01244
01245     fac[0] = poly::to_coefficient_list(a);
01246     bool finished=false;
01247
01248     // Loop over the columns of q + constant, i.e., an exhaustive list of possible factors
01249     for (int i=1; i<n && !finished; i++) {
01250         if (q[i] == vector<WORD>(n,0)) continue;
01251
01252         for (int s=0; s<p && !finished; s++) {
01253             for (int j=0; j<(int)fac.size() && !finished; j++) {
01254                 vector<WORD> c = polygcd::coefficient_list_gcd(fac[j], q[i], p);
01255
01256                 // If a non-trivial factor is found, add it to the list
01257                 if (c.size()!=1 && c.size()!=fac[j].size()) {
01258                     fac.push_back(c);
01259                     fac[j] = poly::coefficient_list_divmod(fac[j], c, p, 0);
01260                     if ((int)fac.size() == rank) finished=true;

```

```

01261         }
01262     }
01263
01264     // Increase the constant term by one
01265     q[i][0] = (q[i][0]+1) % p;
01266 }
01267 }
01268
01269 // Convert the densely represented polynomials to sparse ones
01270 vector<poly> res(fac.size(),poly(BHEAD 0, p));
01271 for (int i=0; i<(int)fac.size(); i++)
01272     res[i] = poly::from_coefficient_list(BHEAD fac[i],x,p);
01273
01274 #ifdef DEBUG
01275     cout << "*** [" << thetime() << "] RES : Berlekamp_find_factors("«_a«","«_q«") = "«res«"\n";
01276 #endif
01277
01278     return res;
01279 }
01280
01281 /*
01282  #] Berlekamp_find_factors :
01283  #[ combine_factors :
01284  */
01285
01307 const vector<poly> polyfact::combine_factors (const poly &a, const vector<poly> &f) {
01308
01309 #ifdef DEBUG
01310     cout << "*** [" << thetime() << "] CALL: combine_factors("«a«","«f«")\n";
01311 #endif
01312
01313     POLY_GETIDENTITY(a);
01314
01315     poly a0(a,0,1);
01316     vector<poly> res;
01317
01318     int num_used = 0;
01319     vector<bool> used(f.size(), false);
01320
01321     // Loop over all bitmasks with num=1,2,...,size(factors)/2 bits
01322     // set, that contain only unused factors
01323     for (int num=1; num<=(int)(f.size() - num_used)/2; num++) {
01324         vector<int> next(f.size() - num_used, 0);
01325         for (int i=0; i<num; i++) next[next.size()-1-i] = 1;
01326
01327         do {
01328             poly fac(BHEAD 1,a.modp,a.modn);
01329             for (int i=0, j=0; i<(int)f.size(); i++)
01330                 if (!used[i] && next[j++]) fac *= f[i];
01331             fac /= fac.integer_lcoeff();
01332             fac *= a.integer_lcoeff();
01333             fac /= polygcd::integer_content(poly(fac,0,1));
01334
01335             if ((a0 % fac).is_zero()) {
01336                 res.push_back(fac);
01337                 for (int i=0, j=0; i<(int)f.size(); i++)
01338                     if (!used[i]) used[i] = next[j++];
01339                 num_used += num;
01340                 num--;
01341                 break;
01342             }
01343         } while (next_permutation(next.begin(), next.end()));
01344     }
01345
01346     // All unused factors together form one more factor
01347     if (num_used != (int)f.size()) {
01348         poly fac(BHEAD 1,a.modp,a.modn);
01349         for (int i=0; i<(int)f.size(); i++)
01350             if (!used[i]) fac *= f[i];
01351         fac /= fac.integer_lcoeff();
01352         fac *= a.integer_lcoeff();
01353         fac /= polygcd::integer_content(poly(fac,0,1));
01354         res.push_back(fac);
01355     }
01356 }
01357
01358 #ifdef DEBUG
01359     cout << "*** [" << thetime() << "] RES : combine_factors("«a«","«f«") = "«res«"\n";
01360 #endif
01361
01362     return res;
01363 }
01364
01365 /*
01366  #] combine_factors :
01367  #[ factorize_squarefree :
01368  */

```

```

01369
01388 const vector<poly> polyfact::factorize_squarefree (const poly &a, const vector<int> &x) {
01389
01390 #ifdef DEBUG
01391     cout << "*** [" << thetime() << "]" CALL: factorize_squarefree("<a<<")\n";
01392 #endif
01393     POLY_GETIDENTITY(a);
01394
01395     WORD p=a.modp, n=a.modn;
01396
01397     try_again:
01398
01399     int bestp=0;
01400     int min_factors = INT_MAX;
01401     poly amodI(BHEAD 0), bestamodI(BHEAD 0);
01402     vector<int> c,d,bestc,bestd;
01403     vector<vector<WORD>> > q,bestq;
01404
01405     // Factorize leading coefficient
01406     factorized_poly lc(factorize(a.lcoeff_univar(x[0])));
01407
01408     // Try a number of primes
01409     int prime_tries = 0;
01410
01411     while (prime_tries<POLYFACT_NUM_CONFIRMATIONS && min_factors>1) {
01412         if (a.modp == 0) {
01413             p = choose_prime(a,x,p);
01414             n = 0;
01415             if (a.degree(x[0]) % p == 0) continue;
01416
01417             // Univariate case: check whether the polynomial mod p is squarefree
01418             // Multivariate case: this check is done after choosing I (for efficiency)
01419             if (x.size()==1) {
01420                 poly amodp(a,p,1);
01421                 if (polygcd::gcd_Euclidean(amodp, amodp.derivative(x[0])).degree(x[0]) != 0)
01422                     continue;
01423             }
01424         }
01425
01426         // Try a number of ideals
01427         if (x.size()>1)
01428             for (int ideal_tries=0; ideal_tries<POLYFACT_MAX_IDEAL_TRIES; ideal_tries++) {
01429                 c = choose_ideal(a,p,lc,x);
01430                 if (c.size()>0) break;
01431             }
01432
01433         if (x.size()==1 || c.size()>0) {
01434             amodI = a;
01435             for (int i=0; i<(int)c.size(); i++)
01436                 amodI %= poly::simple_poly(BHEAD x[i+1],c[i]);
01437
01438             // Determine Q-matrix and its rank. Smaller rank is better.
01439             q = Berlekamp_Qmatrix(poly(amodI,p,1));
01440             int rank=0;
01441             for (int i=0; i<(int)q.size(); i++)
01442                 if (q[i]!=vector<WORD>(q[i].size(),0)) rank++;
01443
01444             if (rank<min_factors) {
01445                 bestp=p;
01446                 bestc=c;
01447                 bestq=q;
01448                 bestamodI=amodI;
01449                 min_factors = rank;
01450                 prime_tries = 0;
01451             }
01452
01453             if (rank==min_factors)
01454                 prime_tries++;
01455         }
01456
01457 #ifdef DEBUG
01458         cout << "*** [" << thetime() << "]" (A) : factorize_squarefree("<a<<
01459             ") try p=" << p << " #factors=" << rank << " (min="<min_factors<<"x"<prime_tries<<")" << endl;
01460 #endif
01461     }
01462 }
01463
01464 p=bestp;
01465 c=bestc;
01466 q=bestq;
01467 amodI=bestamodI;
01468
01469 // Determine to which power of p to lift
01470 if (n==0) {
01471     n = choose_prime_power(amodI,p);
01472     n = MaX(n, choose_prime_power(a,p));
01473 }

```

```

01474
01475     amodI.setmod(p,n);
01476
01477 #ifdef DEBUG
01478     cout << "*** [" << thetime() << "]" (B) : factorize_squarefree("«a«
01479     ") chosen c = " << c << ", p^n = " << p^n << "»»endl;
01480     cout << "*** [" << thetime() << "]" (C) : factorize_squarefree("«a«
01481     ") #factors = " << min_factors << endl;
01482 #endif
01483
01484 // Find factors
01485 vector<poly> f(Berlekamp_find_factors(poly(amodI,p,1),q));
01486
01487 // Lift coefficients
01488 if (f.size() > 1) {
01489     f = lift_coefficients(amodI,f);
01490     if (f==vector<poly>()) {
01491 #ifdef DEBUG
01492         cout << "factor_squarefree failed (lift_coeff step) : " << endl;
01493 #endif
01494         goto try_again;
01495     }
01496
01497 // Combine factors
01498 if (a.modp==0) f = combine_factors(amodI,f);
01499 }
01500
01501 // Lift variables
01502 if (f.size() == 1)
01503     f = vector<poly>(1, a);
01504
01505 if (x.size() > 1 && f.size() > 1) {
01506
01507     // The correct leading coefficients of the factors can be
01508     // reconstructed from prime number factors of the leading
01509     // coefficients modulo I. This is possible since all factors of
01510     // the leading coefficient have unique prime factors for the ideal
01511     // I is chosen as such.
01512
01513     poly amodI(a);
01514     for (int i=0; i<(int)c.size(); i++)
01515         amodI %= poly::simple_poly(BHEAD x[i+1],c[i]);
01516     poly delta(polygcd::integer_content(amodI));
01517
01518     vector<poly> lmodI(lc.factor.size(), poly(BHEAD 0));
01519     for (int i=0; i<(int)lc.factor.size(); i++) {
01520         lmodI[i] = lc.factor[i];
01521         for (int j=0; j<(int)c.size(); j++)
01522             lmodI[i] %= poly::simple_poly(BHEAD x[j+1],c[j]);
01523     }
01524
01525     vector<poly> correct_lc(f.size(), poly(BHEAD 1,p,n));
01526
01527     for (int j=0; j<(int)f.size(); j++) {
01528         poly lc_f(f[j].integer_lcoeff() * delta);
01529         WORD nlc_f = lc_f[lc_f[1]];
01530         poly quo(BHEAD 0),rem(BHEAD 0);
01531         WORD nquo,nrem;
01532
01533         for (int i=(int)lmodI.size()-1; i>=0; i--) {
01534
01535             if (i==0 && lc.factor[i].is_integer()) continue;
01536
01537             do {
01538                 DivLong((UWORD *)&lc_f[2+AN.poly_num_vars], nlc_f,
01539                     (UWORD *)&lmodI[i][2+AN.poly_num_vars], lmodI[i][lmodI[i][1]],
01540                     (UWORD *)&quo[0], &nquo,
01541                     (UWORD *)&rem[0], &nrem);
01542
01543                 if (nrem == 0) {
01544                     correct_lc[j] *= lc.factor[i];
01545                     lc_f.termscopy(&quo[0], 2+AN.poly_num_vars, ABS(nquo));
01546                     nlc_f = nquo;
01547                 }
01548             }
01549             while (nrem == 0);
01550         }
01551     }
01552
01553     for (int i=0; i<(int)correct_lc.size(); i++) {
01554         poly correct_modI(correct_lc[i]);
01555         for (int j=0; j<(int)c.size(); j++)
01556             correct_modI %= poly::simple_poly(BHEAD x[j+1],c[j]);
01557
01558         poly d(polygcd::integer_gcd(correct_modI, f[i].integer_lcoeff()));
01559         correct_lc[i] *= f[i].integer_lcoeff() / d;
01560         delta /= correct_modI / d;

```

```

01561         f[i] *= correct_modI / d;
01562     }
01563
01564     // increase n, because of multiplying with delta
01565     if (!delta.is_one()) {
01566         poly deltapow(BHEAD 1);
01567         for (int i=1; i<(int)correct_lc.size(); i++)
01568             deltapow *= delta;
01569         while (!deltapow.is_zero()) {
01570             deltapow /= poly(BHEAD p);
01571             n++;
01572         }
01573
01574         for (int i=0; i<(int)f.size(); i++) {
01575             f[i].modn = n;
01576             correct_lc[i].modn = n;
01577         }
01578     }
01579
01580     poly aa(a,p,n);
01581
01582     for (int i=0; i<(int)correct_lc.size(); i++) {
01583         correct_lc[i] *= delta;
01584         f[i] *= delta;
01585         if (i>0) aa *= delta;
01586     }
01587
01588     f = lift_variables(aa,f,x,c,correct_lc);
01589
01590     for (int i=0; i<(int)f.size(); i++)
01591         if (a.modp == 0)
01592             f[i] /= polygcd::integer_content(poly(f[i],0,1));
01593         else
01594             f[i] /= polygcd::content_univar(f[i], x[0]);
01595
01596     if (f==vector<poly>()) {
01597 #ifdef DEBUG
01598         cout << "factor_squarefree failed (lift_var step)" << endl;
01599 #endif
01600         goto try_again;
01601     }
01602 }
01603
01604 // set modulus of the factors correctly
01605 if (a.modp==0)
01606     for (int i=0; i<(int)f.size(); i++)
01607         f[i].setmod(0,1);
01608
01609 // Final check (not sure if this is necessary, but it doesn't hurt)
01610 poly check(BHEAD 1,a.modp,a.modn);
01611 for (int i=0; i<(int)f.size(); i++)
01612     check *= f[i];
01613
01614 if (check != a) {
01615 #ifdef DEBUG
01616     cout << "factor_squarefree failed (final check) : " << f << endl;
01617 #endif
01618     goto try_again;
01619 }
01620
01621 #ifdef DEBUG
01622     cout << "*** [" << thetime() << "] RES : factorize_squarefree(" <<a<<","<<x<<","<<c<<) = "<<f<<"\n";
01623 #endif
01624
01625     return f;
01626 }
01627
01628 /*
01629 #] factorize_squarefree :
01630 #[ factorize :
01631 */
01632
01633 const factorized_poly polyfact::factorize (const poly &a) {
01634
01635 #ifdef DEBUG
01636     cout << "*** [" << thetime() << "] CALL: factorize("<<a<<")\n";
01637 #endif
01638     vector<int> x = a.all_variables();
01639
01640     // No variables, so just one factor
01641     if (x.size() == 0) {
01642         factorized_poly res;
01643         if (a.is_one()) return res;
01644         res.add_factor(a,1);
01645         return res;
01646     }
01647 }
01648
01649

```

```

01656 // Remove content
01657 poly conta(polygcd::content_univar(a,x[0]));
01658
01659 factorized_poly faca(factorize(conta));
01660
01661 poly ppa(a / conta);
01662
01663 // Find a squarefree factorization
01664 factorized_poly b(squarefree_factors(ppa));
01665
01666 #ifdef DEBUG
01667 cout << "*** [" << thetime() << "]" ... : factorize("«a«") : SFF = "«b«"\n";
01668 #endif
01669
01670 // Factorize each squarefree factor and build the "factorized_poly"
01671 for (int i=0; i<(int)b.factor.size(); i++) {
01672     vector<poly> c(factorize_squarefree(b.factor[i], x));
01673
01674     for (int j=0; j<(int)c.size(); j++) {
01675         faca.factor.push_back(c[j]);
01676         faca.power.push_back(b.power[i]);
01677     }
01678 }
01679
01680
01681 #ifdef DEBUG
01682 cout << "*** [" << thetime() << "]" RES : factorize("«a«") = "«faca«"\n";
01683 #endif
01684
01685 return faca;
01686 }
01687
01688 /*
01689 #] factorize :
01690 */

```

4.105 polyfact.h

```

00001 #pragma once
00002 /* #] License : */
00003 /*
00004 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00005 * When using this file you are requested to refer to the publication
00006 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00007 * This is considered a matter of courtesy as the development was paid
00008 * for by FOM the Dutch physics granting agency and we would like to
00009 * be able to track its scientific use to convince FOM of its value
00010 * for the community.
00011 *
00012 * This file is part of FORM.
00013 *
00014 * FORM is free software: you can redistribute it and/or modify it under the
00015 * terms of the GNU General Public License as published by the Free Software
00016 * Foundation, either version 3 of the License, or (at your option) any later
00017 * version.
00018 *
00019 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00020 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00021 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00022 * details.
00023 *
00024 * You should have received a copy of the GNU General Public License along
00025 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00026 */
00027 /* #] License : */
00028
00029 #include <vector>
00030 #include <string>
00031
00032 // First prime modulo which factorization is tried. Too small results
00033 // in more unsuccessful attempts; too large is slower.
00034 const int POLYFACT_FIRST_PRIME = 17;
00035
00036 // Fraction of [1,p) that is used for substitutions of variables. Too
00037 // small results in more unsuccessful attempts; too large is slower.
00038 const int POLYFACT_IDEAL_FRACTION = 5;
00039
00040 // Number of ideals that are tried before failure due to unlucky
00041 // choices is accepted.
00042 const int POLYFACT_MAX_IDEAL_TRIES = 3;
00043
00044 // Number of confirmations for the minimal number of factors before
00045 // Hensel lifting is started.

```



```

00046 const int POLYFACT_NUM_CONFIRMATIONS = 3;
00047
00048 // Maximum number of equations for predetermination of coefficients
00049 // for multivariate Hensel lifting
00050 const int POLYFACT_MAX_PREDETERMINATION = 10000;
00051
00052 class poly;
00053
00054 class factorized_poly {
00055     /* Class for representing a factorized polynomial
00056      * The polynomial is given by: PRODUCT(factor[i] ^ power[i])
00057      */
00058 public:
00059     std::vector<poly> factor;
00060     std::vector<int> power;
00061
00062     void add_factor(const poly &f, int p=1);
00063     const std::string toString () const;
00064     friend std::ostream& operator<< (std::ostream &out, const poly &p);
00065 };
00066
00067 std::ostream& operator<< (std::ostream &out, const factorized_poly &a);
00068
00069 namespace polyfact {
00070
00071     // factorization routine
00072     const factorized_poly factorize (const poly &a);
00073
00074     // methods for squarefree factorization
00075     const factorized_poly squarefree_factors (const poly &a);
00076     const factorized_poly squarefree_factors_Yun (const poly &a);
00077     const factorized_poly squarefree_factors_modp (const poly &a);
00078
00079     // methods for choosing suitable reductions
00080     const std::vector<poly> factorize_squarefree (const poly &a, const std::vector<int> &x);
00081     WORD choose_prime (const poly &a, const std::vector<int> &x, WORD p=0);
00082     WORD choose_prime_power (const poly &a, WORD p);
00083     const std::vector<int> choose_ideal (const poly &a, int p, const factorized_poly &lc, const
std::vector<int> &x);
00084
00085     // methods for univariate factorization
00086     const std::vector<std::vector<WORD> > Berlekamp_Qmatrix (const poly &a);
00087     const std::vector<poly> Berlekamp_find_factors (const poly &a, const std::vector<std::vector<WORD>
> &Q);
00088     const std::vector<poly> combine_factors (const poly &a, const std::vector<poly> &f);
00089
00090     // methods for Hensel lifting
00091     const std::vector<poly> extended_gcd_Euclidean_lifted (const poly &a, const poly &b);
00092     const std::vector<poly> solve_Diophantine_univariate (const std::vector<poly> &a, const poly &b);
00093     const std::vector<poly> solve_Diophantine_multivariate (const std::vector<poly> &a, const poly &b,
const std::vector<int> &x, const std::vector<int> &c, int d);
00094     const std::vector<poly> lift_coefficients (const poly &a, const std::vector<poly> &f);
00095     const std::vector<poly> lift_variables (const poly &a, const std::vector<poly> &f, const
std::vector<int> &x, const std::vector<int> &c, const std::vector<poly> &lc);
00096     void predetermine (int dep, const std::vector<std::vector<int> > &state,
std::vector<std::vector<std::vector<int> > > &terms, std::vector<int> &term, int sumdeg=0);
00097
00098 }

```

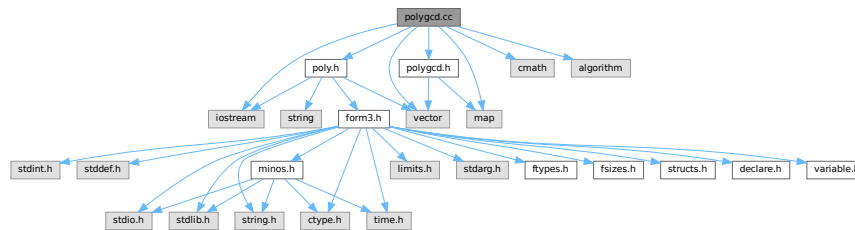
4.106 polygcd.cc File Reference

```

#include "poly.h"
#include "polygcd.h"
#include <iostream>
#include <vector>
#include <cmath>
#include <map>
#include <algorithm>

```

Include dependency graph for polygcd.cc:



Data Structures

- struct [BracketInfo](#)

Functions

- bool [gcd_heuristic_possible](#) (const [poly](#) &a)
- const [poly](#) [gcd_linear_helper](#) (const [poly](#) &a, const [poly](#) &b)

4.106.1 Detailed Description

Contains the routines for calculating greatest commons divisors of multivariate polynomials

Definition in file [polygcd.cc](#).

4.106.2 Function Documentation

4.106.2.1 gcd_heuristic_possible()

```
bool gcd_heuristic_possible (
    const poly & a )
```

Heuristic greatest common divisor of multivariate polynomials

4.106.3 Description

Checks whether the heuristic seems possible by estimating

$\text{MAX\_}\{\text{terms}\} \cdot (\text{coeff} \wedge \text{PROD\_}\{i=1..\#\text{vars}\} (\text{pow\_}i+1))$

and comparing this with GCD_HEURISTIC_MAX_DIGITS.

4.106.4 Notes

- For small polynomials, this consumes time and never triggers.

Definition at line 1142 of file [polygcd.cc](#).

4.106.4.1 gcd_linear_helper()

```
const poly gcd_linear_helper (
    const poly & a,
    const poly & b )
```

Definition at line 1403 of file [polygcd.cc](#).

4.107 polygcd.cc

[Go to the documentation of this file.](#)

```
00001
00006 /* #[ License : */
00007 /*
00008  * Copyright (C) 1984-2026 J.A.M. Vermaseren
00009  * When using this file you are requested to refer to the publication
00010  * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011  * This is considered a matter of courtesy as the development was paid
00012  * for by FOM the Dutch physics granting agency and we would like to
00013  * be able to track its scientific use to convince FOM of its value
00014  * for the community.
00015  *
00016  * This file is part of FORM.
00017  *
00018  * FORM is free software: you can redistribute it and/or modify it under the
00019  * terms of the GNU General Public License as published by the Free Software
00020  * Foundation, either version 3 of the License, or (at your option) any later
00021  * version.
00022  *
00023  * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024  * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025  * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00026  * details.
00027  *
00028  * You should have received a copy of the GNU General Public License along
00029  * with FORM. If not, see <http://www.gnu.org/licenses/>.
00030  */
00031 /* #] License : */
00032 /*
00033  * #[ include :
00034  */
00035
00036 #include "poly.h"
00037 #include "polygcd.h"
00038
00039 #include <iostream>
00040 #include <vector>
00041 #include <cmath>
00042 #include <map>
00043 #include <algorithm>
00044
00045 //#define DEBUG
00046 //#define DEBUGALL
00047
00048 #ifdef DEBUG
00049 #include "mytime.h"
00050 #endif
00051
00052 using namespace std;
00053
00054 /*
00055  * #] include :
00056  * #[ ostream operator :
00057  */
00058
```

```

00059 #ifdef DEBUG
00060 // ostream operator for outputting vector<T>s for debugging purposes
00061 template<class T> ostream& operator<< (ostream& out, const vector<T> &x) {
00062     out<< "{";
00063     for (int i=0; i<(int)x.size(); i++) {
00064         if (i>0) out<< ",";
00065         out<< x[i];
00066     }
00067     out<< "}";
00068     return out;
00069 }
00070 #endif
00071
00072 /*
00073 #] ostream operator :
00074 #[ integer_gcd :
00075 */
00076
00091 const poly polygcd::integer_gcd (const poly &a, const poly &b) {
00092
00093 #ifdef DEBUGALL
00094     cout << "*** [" << thetime() << " ] CALL: integer_gcd(" << a << ", " << b << ")" << endl;
00095 #endif
00096
00097     POLY_GETIDENTITY(a);
00098
00099     if (a.is_zero()) return b;
00100     if (b.is_zero()) return a;
00101
00102     poly c(BHEAD 0, a.modp, a.modn);
00103     WORD nc;
00104
00105     GcdLong(BHEAD
00106             (UWORD *)&a[AN.poly_num_vars+2], a[a[0]-1],
00107             (UWORD *)&b[AN.poly_num_vars+2], b[b[0]-1],
00108             (UWORD *)&c[AN.poly_num_vars+2], &nc);
00109
00110     WORD x = 2 + AN.poly_num_vars + ABS(nc);
00111     c[1] = x; // term length
00112     c[0] = x+1; // total length
00113     c[x] = nc; // coefficient length
00114
00115     for (int i=0; i<AN.poly_num_vars; i++)
00116         c[2+i] = 0; // powers
00117
00118 #ifdef DEBUGALL
00119     cout << "*** [" << thetime() << " ] RES : integer_gcd(" << a << ", " << b << ") = " << c << endl;
00120 #endif
00121
00122     return c;
00123 }
00124
00125 /*
00126 #] integer_gcd :
00127 #[ integer_content :
00128 */
00129
00143 const poly polygcd::integer_content (const poly &a) {
00144
00145 #ifdef DEBUGALL
00146     cout << "*** [" << thetime() << " ] CALL: integer_content(" << a << ")" << endl;
00147 #endif
00148
00149     POLY_GETIDENTITY(a);
00150
00151     if (a.modp>0) return a.integer_lcoeff();
00152
00153     poly c(BHEAD 0, 0, 1);
00154     WORD *d = (WORD *)NumberMalloc("polygcd::integer_content");
00155     WORD nc=0;
00156
00157     for (int i=0; i<AN.poly_num_vars; i++)
00158         c[2+i] = 0;
00159
00160     for (int i=1; i<a[0]; i+=a[i]) {
00161
00162         WCOPY(d, &c[2+AN.poly_num_vars], nc);
00163
00164         GcdLong(BHEAD (UWORD *)d, nc,
00165                 (UWORD *)&a[i+1+AN.poly_num_vars], a[i+a[i]-1],
00166                 (UWORD *)&c[2+AN.poly_num_vars], &nc);
00167
00168         WORD x = 2 + AN.poly_num_vars + ABS(nc);
00169         c[1] = x; // term length
00170         c[0] = x+1; // total length
00171         c[x] = nc; // coefficient length
00172     }

```

```

00173
00174     if (a.sign() != c.sign()) c *= poly(BHEAD -1);
00175
00176     NumberFree(d,"polygcd::integer_content");
00177
00178 #ifdef DEBUGALL
00179     cout << "*** [" << thetime() << "]" RES : integer_content(" << a << ") = " << c << endl;
00180 #endif
00181
00182     return c;
00183 }
00184
00185 /*
00186 #] integer_content :
00187 #[ content_univar :
00188 */
00189
00205 const poly polygcd::content_univar (const poly &a, int x) {
00206
00207 #ifdef DEBUG
00208     cout << "*** [" << thetime() << "]" CALL: content_univar(" << a << ", " << string(1,'a'+x) << ") << endl;
00209 #endif
00210
00211     POLY_GETIDENTITY(a);
00212
00213     poly res(BHEAD 0, a.modp, a.modn);
00214
00215     for (int i=1; i<a[0];) {
00216         poly b(BHEAD 0, a.modp, a.modn);
00217         int deg = a[i+1+x];
00218
00219         for (; i<a[0] && a[i+1+x]==deg; i+=a[i]) {
00220             b.check_memory(b[0]+a[i]);
00221             b.termscopy(&a[i],b[0],a[i]);
00222             b[b[0]+1+x] = 0;
00223             b[0] += a[i];
00224         }
00225
00226         res = gcd(res, b);
00227
00228         if (res.is_integer()) {
00229             res = integer_content(a);
00230             break;
00231         }
00232     }
00233
00234     if (a.sign() != res.sign()) res *= poly(BHEAD -1);
00235
00236 #ifdef DEBUG
00237     cout << "*** [" << thetime() << "]" RES : content_univar(" << a << ", " << string(1,'a'+x) << ") = " << res
00238 << endl;
00239 #endif
00240     return res;
00241 }
00242
00243 /*
00244 #] content_univar :
00245 #[ content_multivar :
00246 */
00247
00263 const poly polygcd::content_multivar (const poly &a, int x) {
00264
00265 #ifdef DEBUGALL
00266     cout << "*** [" << thetime() << "]" CALL: content_multivar(" << a << ", " << string(1,'a'+x) << ") <<
00267     endl;
00268 #endif
00269
00270     POLY_GETIDENTITY(a);
00271
00272     poly res(BHEAD 0, a.modp, a.modn);
00273
00274     for (int i=1,j; i<a[0]; i=j) {
00275         poly b(BHEAD 0, a.modp, a.modn);
00276
00277         for (j=i; j<a[0]; j+=a[j]) {
00278             bool same_powers = true;
00279             for (int k=0; k<AN.poly_num_vars; k++)
00280                 if (k!=x && a[i+1+k]!=a[j+1+k]) {
00281                     same_powers = false;
00282                     break;
00283                 }
00284             if (!same_powers) break;
00285
00286             b.check_memory(b[0]+a[j]);
00287             b.termscopy(&a[j],b[0],a[j]);
00288             for (int k=0; k<AN.poly_num_vars; k++)

```

```

00288         if (k!=x) b[b[0]+1+k]=0;
00289
00290         b[0] += a[j];
00291     }
00292
00293     res = gcd_Euclidean(res, b);
00294
00295     if (res.is_integer()) {
00296         res = poly(BHEAD a.sign(),a.modp,a.modn);
00297         break;
00298     }
00299 }
00300
00301 #ifdef DEBUGALL
00302     cout << "*** [" << thetime() << "]" RES : content_multivar(" <a < ", " < string(1,'a'+x) < ") = " <
res << endl;
00303 #endif
00304
00305     return res;
00306 }
00307
00308 /*
00309     #] content_multivar :
00310     #[ coefficient_list_gcd :
00311 */
00312
00325 const vector<WORD> polygcd::coefficient_list_gcd (const vector<WORD> &a, const vector<WORD> &b, WORD
p) {
00326
00327 #ifdef DEBUGALL
00328     cout << "*** [" << thetime() << "]" CALL: coefficient_list_gcd("<a<","<b<","<p<")<<endl;
00329 #endif
00330
00331     vector<WORD> a(_a), b(_b);
00332
00333     while (b.size() != 0) {
00334         a = poly::coefficient_list_divmod(a,b,p,1);
00335         swap(a,b);
00336     }
00337
00338     while (a.back()==0) a.pop_back();
00339
00340     WORD inv;
00341     GetModInverses(a.back() + (a.back()<0?p:0), p, &inv, NULL);
00342
00343     for (int i=0; i<(int)a.size(); i++)
00344         a[i] = (LONG)inv*a[i] % p;
00345
00346 #ifdef DEBUGALL
00347     cout << "*** [" << thetime() << "]" RES : coefficient_list_gcd("<a<","<b<","<p<") = "<a<<endl;
00348 #endif
00349
00350     return a;
00351 }
00352
00353 /*
00354     #] coefficient_list_gcd :
00355     #[ gcd_Euclidean :
00356 */
00357
00374 const poly polygcd::gcd_Euclidean (const poly &a, const poly &b) {
00375
00376 #ifdef DEBUG
00377     cout << "*** [" << thetime() << "]" CALL: gcd_Euclidean("<a<","<b<")<<endl;
00378 #endif
00379
00380     POLY_GETIDENTITY(a);
00381
00382     if (a.is_zero()) return b;
00383     if (b.is_zero()) return a;
00384     if (a.is_integer() || b.is_integer()) return integer_gcd(a,b);
00385
00386     poly res(BHEAD 0);
00387
00388     if (a.is_dense_univariate()>=-1 && b.is_dense_univariate()>=-1) {
00389         vector<WORD> coeff = coefficient_list_gcd(poly::to_coefficient_list(a),
00390 poly::to_coefficient_list(b), a.modp);
00391         res = poly::from_coefficient_list(BHEAD coeff, a.first_variable(), a.modp);
00392     }
00393     else {
00394         res = a;
00395         poly rem(b);
00396         while (!rem.is_zero())
00397             swap(res%rem, rem);
00398         res /= res.integer_lcoeff();
00399     }

```

```

00400
00401 #ifdef DEBUG
00402     cout << "*** [" << thetime() << "]" RES : gcd_Euclidean("a<<","b<<") = "res<<endl;
00403 #endif
00404
00405     return res;
00406 }
00407
00408 /*
00409 #] gcd_Euclidean :
00410     #[ chinese_remainder :
00411 */
00412
00426 const poly polygcd::chinese_remainder (const poly &a1, const poly &m1, const poly &a2, const poly &m2)
00427 {
00428 #ifdef DEBUGALL
00429     cout << "*** [" << thetime() << "]" CALL: chinese_remainder(" <a1 < "," <m1 < "," <a2 < "," <m2 <
00430 ") << endl;
00431 #endif
00432     POLY_GETIDENTITY(a1);
00433
00434     WORD nx,ny,nz;
00435     UWORD *x = (UWORD *)NumberMalloc("polygcd::chinese_remainder");
00436     UWORD *y = (UWORD *)NumberMalloc("polygcd::chinese_remainder");
00437     UWORD *z = (UWORD *)NumberMalloc("polygcd::chinese_remainder");
00438
00439     GetLongModInverses (BHEAD (UWORD *)&m1[2+AN.poly_num_vars], m1[m1[1]],
00440                         (UWORD *)&m2[2+AN.poly_num_vars], m2[m2[1]],
00441                         (UWORD *)x, &nx, NULL, NULL);
00442
00443     AddLong((UWORD *)&a2[2+AN.poly_num_vars], a2.is_zero() ? 0 : a2[a2[1]],
00444            (UWORD *)&a1[2+AN.poly_num_vars], a1.is_zero() ? 0 : -a1[a1[1]],
00445            y, &ny);
00446
00447     MulLong (x,nx,y,ny,z,&nz);
00448     MulLong (z,nz,(UWORD *)&m1[2+AN.poly_num_vars],m1[m1[1]],x,&nx);
00449
00450     AddLong (x,nx,(UWORD *)&a1[2+AN.poly_num_vars], a1.is_zero() ? 0 : a1[a1[1]],y,&ny);
00451
00452     MulLong ((UWORD *)&m1[2+AN.poly_num_vars], m1[m1[1]],
00453            (UWORD *)&m2[2+AN.poly_num_vars], m2[m2[1]],
00454            (UWORD *)z,&nz);
00455
00456     TakeNormalModulus (y,&ny,(UWORD *)z,nz,NOUNPACK);
00457
00458     poly res(BHEAD y,ny);
00459
00460     NumberFree(x,"polygcd::chinese_remainder");
00461     NumberFree(y,"polygcd::chinese_remainder");
00462     NumberFree(z,"polygcd::chinese_remainder");
00463
00464 #ifdef DEBUGALL
00465     cout << "*** [" << thetime() << "]" RES : chinese_remainder(" <a1 < "," <m1 < "," <a2 < "," <m2 <
00466 ") = " <res << endl;
00467 #endif
00468     return res;
00469 }
00470
00471 /*
00472 #] chinese_remainder :
00473     #[ substitute :
00474 */
00475
00488 const poly polygcd::substitute(const poly &a, int x, int c) {
00489
00490     POLY_GETIDENTITY(a);
00491
00492     poly b(BHEAD 0);
00493
00494     if (a.is_zero()) {
00495         return b;
00496     }
00497
00498     bool zero=true;
00499     int bi=1;
00500
00501     // cache size is bounded by the degree in x, twice the number of terms of
00502     // the polynomial and a constant
00503     vector<WORD> cache(min(a.degree(x)+1,min(2*a.number_of_terms(),
00504     POLYgcd_RAISPOWMOD_CACHE_SIZE)), 0);
00505
00506     for (int ai=1; ai<=a[0]; ai+=a[ai]) {
00507         // last term or different power, then add term to b iff non-zero

```

```

00508     if (!zero) {
00509         bool add=false;
00510         if (ai==a[0])
00511             add=true;
00512         else {
00513             for (int i=0; i<AN.poly_num_vars; i++)
00514                 if (i!=x && a[ai+1+i]!=b[bi+1+i]) {
00515                     zero=true;
00516                     add=true;
00517                     break;
00518                 }
00519             }
00520
00521         if (add) {
00522             if (b[bi+AN.poly_num_vars+1] < 0)
00523                 b[bi+AN.poly_num_vars+1] += a.modp;
00524             bi+=b[bi];
00525         }
00526
00527         if (ai==a[0]) break;
00528     }
00529
00530     b.check_memory(bi);
00531
00532     // create new term in b
00533     if (zero) {
00534         b[bi] = 3+AN.poly_num_vars;
00535         for (int i=0; i<AN.poly_num_vars; i++)
00536             b[bi+1+i] = a[ai+1+i];
00537         b[bi+1+x] = 0;
00538         b[bi+AN.poly_num_vars+1] = 0;
00539         b[bi+AN.poly_num_vars+2] = 1;
00540     }
00541
00542     // add term of a to the current term in b
00543     LONG coeff = a[ai+1+AN.poly_num_vars] * a[ai+2+AN.poly_num_vars];
00544     int pow = a[ai+1+x];
00545
00546     if (pow<(int)cache.size()) {
00547         if (cache[pow]==0)
00548             cache[pow] = RaisPowMod(c, pow, a.modp);
00549         coeff = (coeff * cache[pow]) % a.modp;
00550     }
00551     else {
00552         coeff = (coeff * RaisPowMod(c, pow, a.modp)) % a.modp;
00553     }
00554
00555     b[bi+AN.poly_num_vars+1] = (coeff + b[bi+AN.poly_num_vars+1]) % a.modp;
00556     if (b[bi+AN.poly_num_vars+1] != 0) zero=false;
00557 }
00558
00559 b[0]=bi;
00560 b.setmod(a.modp);
00561
00562 return b;
00563 }
00564
00565 /*
00566     #] substitute :
00567     #[ sparse_interpolation helper functions :
00568 */
00569
00570 // Returns a list of size #terms(a) with entries PROD(ci^powi, i=2..n)
00571 const vector<int> polygcd::sparse_interpolation_get_mul_list (const poly &a, const vector<int> &x,
00572 const vector<int> &c) {
00573     // cache size for variable x is bounded by the degree in x, twice
00574     // the number of terms of the polynomial and a constant
00575     vector<vector<WORD>> > cache(c.size());
00576     int max_cache_size = min(2*a.number_of_terms(),POLYGCD_RAISPOWMOD_CACHE_SIZE);
00577     for (int i=0; i<(int)c.size(); i++)
00578         cache[i] = vector<WORD>(min(a.degree(x[i+1])+1,max_cache_size), 0);
00579
00580     vector<int> res;
00581     for (int i=1; i<a[0]; i+=a[i]) {
00582         LONG coeff=1;
00583         for (int j=0; j<(int)c.size(); j++) {
00584             int pow = a[i+1+x[j+1]];
00585             if (pow<(int)cache[j].size()) {
00586                 if (cache[j][pow]==0)
00587                     cache[j][pow] = RaisPowMod(c[j], pow, a.modp);
00588                 coeff = (coeff * cache[j][pow]) % a.modp;
00589             }
00590             else {
00591                 coeff = (coeff * RaisPowMod(c[j], pow, a.modp)) % a.modp;
00592             }
00593         }
00594         res.push_back(coeff);
00595     }

```



```

00594     }
00595     return res;
00596 }
00597
00598 // Multiplies the coefficients of a with the entries of mul
00599 void polygcd::sparse_interpolation_mul_poly (poly &a, const vector<int> &mul) {
00600     for (int i=1, j=0; i<a[0]; i+=a[i], j++)
00601         a[i+a[i]-2] = ((LONG)a[i+a[i]-2]*mul[j]) % a.modp;
00602 }
00603
00604 // Sets all coefficients to the range 0..modp-1 and the powers of x2...xn to 0
00605 const poly polygcd::sparse_interpolation_reduce_poly (const poly &a, const vector<int> &x) {
00606     poly res(a);
00607     for (int i=1; i<a[0]; i+=a[i]) {
00608         for (int j=1; j<(int)x.size(); j++)
00609             res[i+1+x[j]]=0;
00610         if (res[i+a[i]-1]==-1) {
00611             res[i+a[i]-1]=1;
00612             res[i+a[i]-2]=a.modp-res[i+a[i]-2];
00613         }
00614     }
00615     return res;
00616 }
00617
00618 // Collects entries with equal powers, so that the result is a proper polynomial
00619 const poly polygcd::sparse_interpolation_fix_poly (const poly &a, int x) {
00620     POLY_GETIDENTITY(a);
00621     poly res(BHEAD 0, a.modp, 1);
00622     int j=1;
00623     bool newterm=true;
00624     for (int i=1; i<a[0]; i+=a[i]) {
00625         if (newterm)
00626             res.termscopy(&a[i], j, a[i]);
00627         else
00628             res[j+res[j]-2] = ((LONG)res[j+res[j]-2] + a[i+a[i]-2]) % a.modp;
00629         newterm = i+a[i] == a[0] || res[j+1+x] != a[i+a[i]+1+x];
00630         if (newterm && res[j+res[j]-2]!=0) j += res[j];
00631     }
00632     res[0]=j;
00633     return res;
00634 }
00635
00636 /*
00637 #] sparse_interpolation helper functions :
00638 #[ gcd_modular_sparse_interpolation :
00639 */
00640
00641 const poly polygcd::gcd_modular_sparse_interpolation (const poly &origa, const poly &origb, const
vector<int> &x, const poly &s) {
00642     #ifdef DEBUG
00643         cout << "*** [" << thetime() << "]" CALL: gcd_modular_sparse_interpolation("
00644             << origa << ", " << origb << ", " << x << ", " << s << ")" << endl;
00645     #endif
00646     POLY_GETIDENTITY(origa);
00647     // strip multivariate content
00648     poly conta(content_multivar(origa, x.back()));
00649     poly contb(content_multivar(origb, x.back()));
00650     poly gcdconts(gcd_Euclidean(conta, contb));
00651     const poly& a = conta.is_one() ? origa : origa/conta;
00652     const poly& b = contb.is_one() ? origb : origb/contb;
00653     // for non-monic cases, we need to normalize with the gcd of the lcoeffs of a poly in x[0]
00654     // or else the shape fitting does not work.
00655     // FIXME: the current implementation still rejects some valid shapes.
00656     poly lcgcd(BHEAD 1, a.modp);
00657     if (!s.lcoeff_univar(x[0]).is_integer()) {
00658         lcgcd = gcd_modular_dense_interpolation(a.lcoeff_univar(x[0]), b.lcoeff_univar(x[0]), x,
poly(BHEAD 0));
00659     }
00660     // reduce polynomials
00661     poly ared(sparse_interpolation_reduce_poly(a, x));
00662     poly bred(sparse_interpolation_reduce_poly(b, x));
00663     poly sred(sparse_interpolation_reduce_poly(s, x));
00664     poly lred(sparse_interpolation_reduce_poly(lcgcd, x));
00665     // set all coefficients to 1
00666     for (int i=1; i<sred[0]; i+=sred[i]) {
00667         sred[i+sred[i]-2] = sred[i+sred[i]-1] = 1;
00668     }

```

```

00710     }
00711
00712     // generate random numbers and check there this set doesn't result
00713     // in a singular matrix
00714     vector<int> c(x.size()-1);
00715     vector<int> smul;
00716
00717     bool duplicates;
00718     do {
00719         for (int i=0; i<(int)c.size(); i++)
00720             c[i] = 1 + wranf(BHEAD0) % (a.modp-1);
00721         smul = sparse_interpolation_get_mul_list(s,x,c);
00722
00723         duplicates = false;
00724
00725         int fr=0,to=0;
00726         for (int i=1; i<s[0];) {
00727             int pow = s[i+1+x[0]];
00728             while (i<s[0] && s[i+1+x[0]]==pow) i+=s[i], to++;
00729             for (int j=fr; j<to; j++)
00730                 for (int k=fr; k<j; k++)
00731                     if (smul[j] == smul[k])
00732                         duplicates = true;
00733             fr=to;
00734         }
00735     } while (duplicates);
00736
00737     // get the lists to multiply the polynomials with every iteration
00738     vector<int> amul(sparse_interpolation_get_mul_list(a,x,c));
00739     vector<int> bmul(sparse_interpolation_get_mul_list(b,x,c));
00740     vector<int> lmul(sparse_interpolation_get_mul_list(lcgcd,x,c));
00741
00742     vector<vector<vector<LONG> > > M;
00743     vector<vector<LONG> > V;
00744
00745     int maxMsize=0;
00746
00747     // create (empty) matrices
00748     for (int i=1; i<s[0]; i+=s[i]) {
00749         if (i==1 || s[i+1+x[0]]!=s[i+1+x[0]-s[i]]) {
00750             M.push_back(vector<vector<LONG> >());
00751             V.push_back(vector<LONG>());
00752         }
00753         M.back().push_back(vector<LONG>());
00754         V.back().push_back(0);
00755         maxMsize = max(maxMsize, (int)M.back().size());
00756     }
00757
00758     // generate linear equations
00759     for (int numg=0; numg<maxMsize; numg++) {
00760         poly amodI(sparse_interpolation_fix_poly(ared,x[0]));
00761         poly bmodI(sparse_interpolation_fix_poly(bred,x[0]));
00762         poly lmodI(sparse_interpolation_fix_poly(lred,x[0]));
00763
00764         // A fix for non-monic gcDs. This could be slow if lmodI has many terms,
00765         // since it overfits the gcd now. Another gcd has to be run to remove the
00766         // extra terms.
00767         poly gcd(lmodI * gcd_Euclidean(amodI,bmodI));
00768
00769         // if correct gcd
00770         if (!gcd.is_zero() && gcd[2+x[0]]==sred[2+x[0]]) {
00771             // for each power in the gcd, generate an equation if needed
00772             int gi=1, midx=0;
00773
00774             for (int si=1; si<s[0];) {
00775                 // if the term exists, set Vi=coeff, otherwise Vi remains 0
00776                 if (gi<gcd[0] && gcd[gi+1+x[0]]==sred[si+1+x[0]]) {
00777                     if (numg < (int)V[midx].size())
00778                         V[midx][numg] = gcd[gi+gcd[gi]-1]*gcd[gi+gcd[gi]-2];
00779                     gi += gcd[gi];
00780                 }
00781
00782                 // add the coefficients of s to the matrix M
00783                 for (int i=0; i<(int)M[midx].size(); i++) {
00784                     if (numg < (int)M[midx].size())
00785                         M[midx][numg].push_back(sred[si+1+AN.poly_num_vars]);
00786                     si += s[si];
00787                 }
00788                 midx++;
00789             }
00790         }
00791     }
00792     else {
00793         // incorrect gcd

```

```

00797         if (!gcd.is_zero() && gcd[2+x[0]]<sred[2+x[0]])
00798             return poly(BHEAD 0);
00799         numg--;
00800     }
00801
00802     // multiply polynomials by the lists to obtain new ones
00803     sparse_interpolation_mul_poly(ared, amul);
00804     sparse_interpolation_mul_poly(bred, bmul);
00805     sparse_interpolation_mul_poly(sred, smul);
00806     sparse_interpolation_mul_poly(lred, lmul);
00807 }
00808
00809 // solve the linear equations
00810 for (int i=0; i<(int)M.size(); i++) {
00811     int n = M[i].size();
00812
00813     // Gaussian elimination
00814     for (int j=0; j<n; j++) {
00815         for (int k=0; k<j; k++) {
00816             LONG x = M[i][j][k];
00817             for (int l=k; l<n; l++)
00818                 M[i][j][l] = (M[i][j][l] - M[i][k][l]*x) % a.modp;
00819             V[i][j] = (V[i][j] - V[i][k]*x) % a.modp;
00820         }
00821
00822         // normalize row
00823         WORD x = M[i][j][j]; // WORD for GetModInverses
00824         GetModInverses(x + (x<0?a.modp:0), a.modp, &x, NULL);
00825         for (int k=0; k<n; k++)
00826             M[i][j][k] = (M[i][j][k]*x) % a.modp;
00827         V[i][j] = (V[i][j]*x) % a.modp;
00828     }
00829
00830     // solve
00831     for (int j=n-1; j>=0; j--)
00832         for (int k=j+1; k<n; k++)
00833             V[i][j] = (V[i][j] - M[i][j][k]*V[i][k]) % a.modp;
00834 }
00835
00836 // create coefficient list
00837 vector<LONG> coeff;
00838 for (int i=0; i<(int)V.size(); i++)
00839     for (int j=0; j<(int)V[i].size(); j++)
00840         coeff.push_back(V[i][j]);
00841
00842 // create resulting polynomial
00843 poly res(BHEAD 0);
00844 int ri=1, i=0;
00845 for (int si=1; si<s[0]; si+=s[si]) {
00846     res.check_memory(ri);
00847     res[ri] = 3 + AN.poly_num_vars; // term length
00848     for (int j=0; j<AN.poly_num_vars; j++)
00849         res[ri+1+j] = s[si+1+j]; // powers
00850     res[ri+1+AN.poly_num_vars] = ABS(coeff[i]); // coefficient
00851     res[ri+2+AN.poly_num_vars] = SGN(coeff[i]); // coefficient length
00852     i++;
00853     ri += res[ri];
00854 }
00855 res[0]=ri; // total length
00856 res.setmod(a.modp,1);
00857
00858 if (!poly::divides(res, lcgcd * a) || !poly::divides(res, lcgcd * b)) {
00859     return poly(BHEAD 0); // bad shape
00860 } else {
00861     // refine gcd
00862     if (!poly::divides(res, a))
00863         res = gcd_modular_dense_interpolation(res, a, x, poly(BHEAD 0));
00864     if (!poly::divides(res, b))
00865         res = gcd_modular_dense_interpolation(res, b, x, poly(BHEAD 0));
00866 }
00867
00868 #ifdef DEBUG
00869     cout << "*** [" << thetime() << " ] RES : gcd_modular_sparse_interpolation("
00870           << a << ", " << b << ", " << x << ", " << " << s << ") = " << res << endl;
00871 #endif
00872
00873     return gcdconts * res;
00874 }
00875
00876 /*
00877 #] gcd_modular_sparse_interpolation :
00878 #[ gcd_modular_dense_interpolation :
00879 */
00880
00881 const poly polygcd::gcd_modular_dense_interpolation (const poly &a, const poly &b, const vector<int>
&x, const poly &s) {
00900

```

```

00901 #ifdef DEBUG
00902     cout << "*** [" << thetime() << "]" CALL: gcd_modular_dense_interpolation(" <a << ", " <b << ", " <x <<
    ", " <s << ")" << endl;
00903 #endif
00904
00905     POLY_GETIDENTITY(a);
00906
00907     // if univariate, then use Euclidean algorithm
00908     if (x.size() == 1) {
00909         return gcd_Euclidean(a,b);
00910     }
00911
00912     // if shape is known, use sparse interpolation
00913     if (!s.is_zero()) {
00914         poly res = gcd_modular_sparse_interpolation (a,b,x,s);
00915         if (!res.is_zero()) return res;
00916         // apparently the shape was not correct. continue.
00917     }
00918
00919     // divide out multivariate content in last variable
00920     int X = x.back();
00921
00922     poly conta(content_multivar(a,X));
00923     poly contb(content_multivar(b,X));
00924     poly gcdconts(gcd_Euclidean(conta,contb));
00925     const poly& ppa = conta.is_one() ? a : poly(a/conta);
00926     const poly& ppb = contb.is_one() ? b : poly(b/contb);
00927
00928     // gcd of leading coefficients
00929     poly lcoeffa(ppa.lcoeff_multivar(X));
00930     poly lcoeffb(ppb.lcoeff_multivar(X));
00931     poly gcdlcoeffs(gcd_Euclidean(lcoeffa,lcoeffb));
00932
00933     // calculate the degree bound for each variable
00934     int m = Min(ppa.degree(x[x.size() - 2]),ppb.degree(x[x.size() - 2]));
00935
00936     poly res(BHEAD 0);
00937     poly oldres(BHEAD 0);
00938     poly newshape(BHEAD 0);
00939     poly modpoly(BHEAD 1,a.modp);
00940
00941     while (true) {
00942         // generate random constants and substitute it
00943         int c = 1 + wranf(BHEAD0) % (a.modp-1);
00944         if (substitute(gcdlcoeffs,X,c).is_zero()) continue;
00945         if (substitute(modpoly,X,c).is_zero()) continue;
00946
00947         poly amodc(substitute(ppa,X,c));
00948         poly bmodc(substitute(ppb,X,c));
00949
00950         // calculate gcd recursively
00951         poly gcdmodc(gcd_modular_dense_interpolation(amodc,bmodc,vector<int>(x.begin(),x.end()-1),
newshape));
00952         int n = gcdmodc.degree(x[x.size() - 2]);
00953
00954         // normalize
00955         gcdmodc = (gcdmodc * substitute(gcdlcoeffs,X,c)) / gcdmodc.integer_lcoeff();
00956         poly simple(poly::simple_poly(BHEAD X,c,1,a.modp)); // (X-c) mod p
00957
00958         // if power is smaller, the old one was wrong
00959         if ((res.is_zero() && n == m) || n < m) {
00960             m = n;
00961             res = gcdmodc;
00962             newshape = gcdmodc; // set a new shape (interpolation does not change it)
00963             modpoly = simple;
00964         }
00965         else if (n == m) {
00966             oldres = res;
00967             // equal powers, so interpolate results
00968             poly coeff_poly(substitute(modpoly,X,c));
00969             WORD coeff_word = coeff_poly[2+AN.poly_num_vars] * coeff_poly[3+AN.poly_num_vars];
00970             if (coeff_word < 0) coeff_word += a.modp;
00971
00972             GetModInverses(coeff_word, a.modp, &coeff_word, NULL);
00973
00974             res.setmod(a.modp); // make sure the mod is set before substituting
00975             res += poly(BHEAD coeff_word, a.modp, 1) * modpoly * (gcdmodc - substitute(res,X,c));
00976             modpoly *= simple;
00977         }
00978
00979         // check whether this is the complete gcd
00980         if (!res.is_zero() && res == oldres) {
00981             poly nres = res / content_multivar(res, X);
00982             if (poly::divides(nres,ppa) && poly::divides(nres,ppb)) {
00983 #ifdef DEBUG
00984                 cout << "*** [" << thetime() << "]" RES : gcd_modular_dense_interpolation(" <a << ", " <b
    << ", "

```

```

00985             « x « ", " « ", " « s « ") = " « gcdconts * nres « endl;
00986 #endif
00987             return gcdconts * nres;
00988         }
00989
00990         // At this point, the gcd may be too large due to bad luck
00991         // TODO: create an efficient fail state that tries to find a smaller
00992         // polynomial without interpolating bad ones?
00993         newshape = poly(BHEAD 0); // reset the shape, important!
00994     }
00995 }
00996 }
00997
00998 /*
00999 #] gcd_modular_dense_interpolation : :
01000 #[ gcd_modular :
01001 */
01002
01019 const poly polygcd::gcd_modular (const poly &origa, const poly &origb, const vector<int> &x) {
01020
01021 #ifdef DEBUG
01022     cout << "*** [" << thetime() << "]" CALL: gcd_modular(" « origa « ", " « origb « ", " « x « ")" << endl;
01023 #endif
01024
01025     POLY_GETIDENTITY(origa);
01026
01027     if (origa.is_zero()) return origa.modp==0 ? origb : origb / origb.integer_lcoeff();
01028     if (origb.is_zero()) return origa.modp==0 ? origa : origa / origa.integer_lcoeff();
01029     if (origa==origb) return origa.modp==0 ? origa : origa / origa.integer_lcoeff();
01030
01031     poly ac = integer_content(origa);
01032     poly bc = integer_content(origb);
01033     const poly& a = ac.is_one() ? origa : poly(origa/ac);
01034     const poly& b = bc.is_one() ? origb : poly(origb/bc);
01035     poly ic = integer_gcd(ac, bc);
01036     poly g = integer_gcd(a.integer_lcoeff(), b.integer_lcoeff());
01037
01038     int pnum=0;
01039
01040     poly d(BHEAD 0);
01041     poly m1(BHEAD 1);
01042     int mindeg=MAXPOSITIVE;
01043
01044     while (true) {
01045         // choose a prime and solve modulo the prime
01046         WORD p = NextPrime(BHEAD pnum++);
01047         if (poly(a.integer_lcoeff(),p).is_zero()) continue;
01048         if (poly(b.integer_lcoeff(),p).is_zero()) continue;
01049
01050         poly c(gcd_modular_dense_interpolation(poly(a,p),poly(b,p),x,poly(d,p)));
01051         c = (c * poly(g,p)) / c.integer_lcoeff(); // normalize so that lcoeff(c) = g mod p
01052
01053         if (c.is_zero()) {
01054             // unlucky choices somewhere, so start all over again
01055             d = poly(BHEAD 0);
01056             m1 = poly(BHEAD 1);
01057             mindeg = MAXPOSITIVE;
01058             continue;
01059         }
01060
01061         if (!(poly(a,p)%c).is_zero()) continue;
01062         if (!(poly(b,p)%c).is_zero()) continue;
01063
01064         int deg = c.degree(x[0]);
01065
01066         if (deg < mindeg) {
01067             // small degree, so the old one is wrong
01068             d=c;
01069             d.modp=a.modp;
01070             d.modn=a.modn;
01071             m1 = poly(BHEAD p);
01072             mindeg=deg;
01073         }
01074         else if (deg == mindeg) {
01075             // same degree, so use Chinese Remainder Algorithm
01076             poly newd(BHEAD 0);
01077
01078             for (int ci=1,di=1; ci<c[0]||di<d[0]; ) {
01079                 int comp = ci==c[0] ? -1 : di==d[0] ? +1 : poly::monomial_compare(BHEAD
01080 &c[ci],&d[di]);
01081                 poly a1(BHEAD 0),a2(BHEAD 0);
01082
01083                 newd.check_memory(newd[0]);
01084
01085                 if (comp <= 0) {
01086                     newd.termscopy(&d[di],newd[0],1+AN.poly_num_vars);
01087                     a1 = poly(BHEAD (UWORD *)&d[di+1+AN.poly_num_vars],d[di+d[di]-1]);

```

```

01087         di+=d[di];
01088     }
01089     if (comp >= 0) {
01090         newd.termscopy(&c[ci],newd[0],1+AN.poly_num_vars);
01091         a2 = poly(BHEAD (UWORD *)&c[ci+1+AN.poly_num_vars],c[ci+c[ci]-1]);
01092         ci+=c[ci];
01093     }
01094
01095     poly e(chinese_remainder(a1,m1,a2,poly(BHEAD p)));
01096     newd.termscopy(&e[2+AN.poly_num_vars], newd[0]+1+AN.poly_num_vars, ABS(e[e[1]])+1);
01097     newd[newd[0]] = 2 + AN.poly_num_vars + ABS(e[e[1]]);
01098     newd[0] += newd[newd[0]];
01099 }
01100
01101 m1 *= poly(BHEAD p);
01102 d=newd;
01103 }
01104
01105 // divide out spurious integer content
01106 poly ppd(d / integer_content(d));
01107
01108 // check whether this is the complete gcd
01109 if (poly::divides(ppd,a) && poly::divides(ppd,b)) {
01110 #ifdef DEBUG
01111     cout << "*** [" << thetime() << "]" RES : gcd_modular(" << origa << "," << origb << "," << x << ")
01112 = "
01113         << ic * ppd << endl;
01114 #endif
01115     return ic * ppd;
01116 }
01117 #ifdef DEBUG
01118     cout << "*** [" << thetime() << "]" Retrying modular_gcd with new prime" << endl;
01119 #endif
01120 }
01121
01122 /*
01123 #] gcd_modular :
01124 #[ gcd_heuristic_possible :
01125 */
01126
01142 bool gcd_heuristic_possible (const poly &a) {
01143
01144     POLY_GETIDENTITY(a);
01145
01146     double prod_deg = 1;
01147     for (int j=0; j<AN.poly_num_vars; j++)
01148         prod_deg *= a[2+j]+1;
01149
01150     double digits = ABS(a[1+a[1]-1]);
01151     double lead = a[1+1+AN.poly_num_vars];
01152
01153     return prod_deg*(digits-1+log(2*ABS(lead))/log(2.0)/(BITSINWORD/2)) <
POLYGCD_HEURISTIC_MAX_DIGITS;
01154 }
01155
01156 /*
01157 #] gcd_heuristic_possible :
01158 #[ gcd_heuristic :
01159 */
01160
01188 const poly polygcd::gcd_heuristic (const poly &a, const poly &b, const vector<int> &x, int max_tries)
01189 {
01190 #ifdef DEBUG
01191     cout << "*** [" << thetime() << "]" CALL: gcd_heuristic("<a<","<b<","<x<")\n";
01192 #endif
01193
01194     if (a.is_integer()) return integer_gcd(a,integer_content(b));
01195     if (b.is_integer()) return integer_gcd(integer_content(a),b);
01196
01197     POLY_GETIDENTITY(a);
01198
01199     // Calculate xi = 2*min(max(coefficients a),max(coefficients b))+2
01200
01201     UWORD *pxi=NULL;
01202     WORD nxi=0;
01203
01204     for (int i=1; i<a[0]; i+=a[i]) {
01205         WORD na = ABS(a[i+a[i]-1]);
01206         if (BigLong((UWORD *)&a[i+a[i]-1-na], na, pxi, nxi) > 0) {
01207             pxi = (UWORD *)&a[i+a[i]-1-na];
01208             nxi = na;
01209         }
01210     }
01211

```

```

01212     for (int i=1; i<b[0]; i+=b[i]) {
01213         WORD nb = ABS(b[i+b[i]-1]);
01214         if (BigLong((UWORD *)&b[i+b[i]-1-nb], nb, pxi, nxi) > 0) {
01215             pxi = (UWORD *)&b[i+b[i]-1-nb];
01216             nxi = nb;
01217         }
01218     }
01219
01220     poly xi(BHEAD pxi,nxi);
01221
01222     // Addition of another random factor gives better performance
01223     xi = xi*poly(BHEAD 2) + poly(BHEAD 2 + wranf(BHEAD0)%POLYgcd_HEURISTIC_MAX_ADD_RANDOM);
01224
01225     // If degree*digits(xi) is too large, throw exception
01226     if (max(a.degree(x[0]),b.degree(x[0])) * xi[xi[1]] >= Min(AM.MaxTal,
POLYgcd_HEURISTIC_MAX_DIGITS)) {
01227 #ifdef DEBUG
01228     cout << "*** [" << thetime() << "] RES : gcd_heuristic("«a«","«b«","«x«") = overflow\n";
01229 #endif
01230     throw(gcd_heuristic_failed());
01231 }
01232
01233     for (int times=0; times<max_tries; times++) {
01234
01235         // Recursively calculate the gcd
01236
01237         poly gamma(gcd_heuristic(a % poly::simple_poly(BHEAD x[0],xi),
01238                                 b % poly::simple_poly(BHEAD x[0],xi),
01239                                 vector<int>(x.begin()+1,x.end()),1));
01240
01241         // If a gcd is found, reconstruct the powers of x
01242         if (!gamma.is_zero()) {
01243             // res is construct is reverse order. idx/len are for reversing
01244             // it in the correct order
01245             poly res(BHEAD 0), c(BHEAD 0);
01246             vector<int> idx, len;
01247
01248             for (int power=0; !gamma.is_zero(); power++) {
01249
01250                 // calculate c = gamma % xi (c and gamma are polynomials, xi is integer)
01251                 c = gamma;
01252                 c.coefficients_modulo((UWORD *)&xi[2+AN.poly_num_vars], xi[xi[0]-1], false);
01253
01254                 // Add the terms c * x^power to res
01255                 res.check_memory(res[0]+c[0]);
01256                 res.termscopy(&c[1],res[0],c[0]-1);
01257                 for (int i=1; i<c[0]; i+=c[i])
01258                     res[res[0]-1+i+1+x[0]] = power;
01259
01260                 // Store idx/len for reversing
01261                 if (!c.is_zero()) {
01262                     idx.push_back(res[0]);
01263                     len.push_back(c[0]-1);
01264                 }
01265
01266                 res[0] += c[0]-1;
01267
01268                 // Divide gamma by xi
01269                 gamma = (gamma - c) / xi;
01270             }
01271
01272             // Reverse the resulting polynomial
01273             poly rev(BHEAD 0);
01274             rev.check_memory(res[0]);
01275
01276             rev[0] = 1;
01277             for (int i=idx.size()-1; i>=0; i--) {
01278                 rev.termscopy(&res[idx[i]], rev[0], len[i]);
01279                 rev[0] += len[i];
01280             }
01281             res = rev;
01282
01283             poly ppres = res / integer_content(res);
01284
01285             if ((a%ppres).is_zero() && (b%ppres).is_zero()) {
01286 #ifdef DEBUG
01287             cout << "*** [" << thetime() << "] RES : gcd_heuristic("«a«","«b«","«x«") = "«res«"\n";
01288 #endif
01289             return res;
01290         }
01291     }
01292
01293     // Next xi by multiplying with the golden ratio to avoid correlated errors
01294     xi = xi * poly(BHEAD 28657) / poly(BHEAD 17711) + poly(BHEAD wranf(BHEAD0) %
POLYgcd_HEURISTIC_MAX_ADD_RANDOM);
01295 }
01296

```

```

01297 #ifdef DEBUG
01298     cout << "*** [" << thetime() << " ] RES : gcd_heuristic("«a«", "«b«", "«x«") = failed\n";
01299 #endif
01300
01301     return poly(BHEAD 0);
01302 }
01303
01304 /*
01305     #] gcd_heuristic :
01306     #[ bracket :
01307 */
01308
01309 const map<vector<int>,poly> polygcd::full_bracket(const poly &a, const vector<int>& filter) {
01310     POLY_GETIDENTITY(a);
01311
01312     map<vector<int>,poly> bracket;
01313     for (int ai=1; ai<a[0]; ai+=a[ai]) {
01314         vector<int> varpattern(AN.poly_num_vars);
01315         for (int i=0; i<AN.poly_num_vars; i++)
01316             if (filter[i] == 1 && a[ai + i + 1] > 0)
01317                 varpattern[i] = a[ai + i + 1];
01318
01319         // create monomial
01320         poly mon(BHEAD 1);
01321         mon.setmod(a.modp);
01322         mon[0] = a[ai] + 1;
01323         for (int i=0; i<a[ai]; i++)
01324             if (i > 0 && i <= AN.poly_num_vars && varpattern[i - 1])
01325                 mon[1+i] = 0;
01326             else
01327                 mon[1+i] = a[ai+i];
01328
01329         map<vector<int>,poly>::iterator i = bracket.find(varpattern);
01330         if (i == bracket.end()) {
01331             bracket.insert(std::make_pair(varpattern, mon));
01332         } else {
01333             i->second += mon;
01334         }
01335     }
01336
01337     return bracket;
01338 }
01339
01340 const poly polygcd::bracket(const poly &a, const vector<int>& pattern, const vector<int>& filter) {
01341     POLY_GETIDENTITY(a);
01342
01343     poly bracket(BHEAD 0);
01344     for (int ai=1; ai<a[0]; ai+=a[ai]) {
01345         bool ispat = true;
01346         for (int i=0; i<AN.poly_num_vars; i++)
01347             if (filter[i] == 1 && pattern[i] != a[ai + i + 1]) {
01348                 ispat = false;
01349                 break;
01350             }
01351
01352         if (ispat) {
01353             poly mon(BHEAD 1);
01354             mon.setmod(a.modp);
01355             mon[0] = a[ai] + 1;
01356             for (int i=0; i<a[ai]; i++)
01357                 if (i > 0 && i <= AN.poly_num_vars && pattern[i - 1])
01358                     mon[1+i] = 0;
01359                 else
01360                     mon[1+i] = a[ai+i];
01361             bracket += mon;
01362         }
01363     }
01364
01365     return bracket;
01366 }
01367
01368 const map<vector<int>,int> polygcd::bracket_count(const poly &a, const vector<int>& filter) {
01369     POLY_GETIDENTITY(a);
01370
01371     map<vector<int>,int> bracket;
01372     for (int ai=1; ai<a[0]; ai+=a[ai]) {
01373         vector<int> varpattern(AN.poly_num_vars);
01374         for (int i=0; i<AN.poly_num_vars; i++)
01375             if (filter[i] == 1 && a[ai + i + 1] > 0)
01376                 varpattern[i] = a[ai + i + 1];
01377
01378         map<vector<int>,int>::iterator i = bracket.find(varpattern);
01379         if (i == bracket.end()) {
01380             bracket.insert(std::make_pair(varpattern, 0));
01381         } else {
01382             i->second++;
01383         }
01384     }

```



```

01384     }
01385
01386     return bracket;
01387 }
01388
01389 struct BracketInfo {
01390     std::vector<int> pattern;
01391     int num_terms, dummy = 0;
01392     const poly* p;
01393
01394     BracketInfo(const std::vector<int>& pattern, int num_terms, const poly* p) : pattern(pattern),
num_terms(num_terms), p(p) {}
01395     bool operator<(const BracketInfo& rhs) const { return num_terms > rhs.num_terms; } // biggest
should be first!
01396 };
01397
01398 /*
01399     #] bracket :
01400     #[ gcd_linear:
01401 */
01402
01403 const poly gcd_linear_helper (const poly &a, const poly &b) {
01404     POLY_GETIDENTITY(a);
01405
01406     for (int i = 0; i < AN.poly_num_vars; i++)
01407         if (a.degree(i) == 1) {
01408             vector<int> filter(AN.poly_num_vars);
01409             filter[i] = 1;
01410
01411             // bracket the linear variable
01412             map<vector<int>,poly> ba = polygcd::full_bracket(a, filter);
01413
01414             poly subgcd(BHEAD 1);
01415             if (ba.size() == 2)
01416                 subgcd = polygcd::gcd_linear(ba.begin()->second, (++ba.begin()->second);
01417             else
01418                 subgcd = ba.begin()->second;
01419
01420             poly linfac = a / subgcd;
01421             if (poly::divides(linfac,b))
01422                 return linfac * polygcd::gcd_linear(subgcd, b / linfac);
01423
01424             return polygcd::gcd_linear(subgcd, b);
01425         }
01426
01427     return poly(BHEAD 0);
01428 }
01429
01430 const poly polygcd::gcd_linear (const poly &a, const poly &b) {
01431     #ifdef DEBUG
01432     cout << "*** [" << thetime() << " ] CALL: gcd_linear(\"a<<\", \"b<<\")\n";
01433     #endif
01434
01435     POLY_GETIDENTITY(a);
01436
01437     if (a.is_zero()) return a.modp==0 ? b : b / b.integer_lcoeff();
01438     if (b.is_zero()) return a.modp==0 ? a : a / a.integer_lcoeff();
01439     if (a==b) return a.modp==0 ? a : a / a.integer_lcoeff();
01440
01441     if (a.is_integer() || b.is_integer()) {
01442         if (a.modp > 0) return poly(BHEAD 1,a.modp,a.modn);
01443         return poly(integer_gcd(integer_content(a),integer_content(b)),0,1);
01444     }
01445
01446     poly h = gcd_linear_helper(a, b);
01447     if (!h.is_zero()) return h;
01448
01449     h = gcd_linear_helper(b, a);
01450     if (!h.is_zero()) return h;
01451
01452     vector<int> xa = a.all_variables();
01453     vector<int> xb = b.all_variables();
01454
01455     vector<int> used(AN.poly_num_vars,0);
01456     for (int i=0; i<(int)xa.size(); i++) used[xa[i]]++;
01457     for (int i=0; i<(int)xb.size(); i++) used[xb[i]]++;
01458     vector<int> x;
01459     for (int i=0; i<AN.poly_num_vars; i++)
01460         if (used[i] > 1) x.push_back(i);
01461
01462     return gcd_modular(a,b,x);
01463 }
01464
01465 /*
01466     #] gcd_linear:
01467     #[ gcd :
01468 */

```

```

01473
01494 const poly polygcd::gcd (const poly &a, const poly &b) {
01495
01496 #ifdef DEBUG
01497     cout << "*** [" << thetime() << " ] CALL: gcd(" << a << ", " << b << ") \n";
01498 #endif
01499
01500     POLY_GETIDENTITY(a);
01501
01502     if (a.is_zero()) return a.modp==0 ? b : b / b.integer_lcoeff();
01503     if (b.is_zero()) return a.modp==0 ? a : a / a.integer_lcoeff();
01504     if (a==b) return a.modp==0 ? a : a / a.integer_lcoeff();
01505
01506     if (a.is_integer() || b.is_integer()) {
01507         if (a.modp > 0) return poly(BHEAD 1, a.modp, a.modn);
01508         return poly(integer_gcd(integer_content(a), integer_content(b)), 0, 1);
01509     }
01510
01511     // Generate a list of variables of a and b
01512     vector<int> xa = a.all_variables();
01513     vector<int> xb = b.all_variables();
01514
01515     vector<int> used(AN.poly_num_vars, 0);
01516     for (int i=0; i<(int)xa.size(); i++) used[xa[i]]++;
01517     for (int i=0; i<(int)xb.size(); i++) used[xb[i]]++;
01518     vector<int> x;
01519     for (int i=0; i<AN.poly_num_vars; i++)
01520         if (used[i]) x.push_back(i);
01521
01522     if (a.is_monomial() || b.is_monomial()) {
01523
01524         poly res(BHEAD 1, a.modp, a.modn);
01525         if (a.modp == 0) res = integer_gcd(integer_content(a), integer_content(b));
01526
01527         for (int i=0; i<(int)x.size(); i++)
01528             res[2+x[i]] = 1<<(BITSINWORD-2);
01529
01530         for (int i=1; i<a[0]; i+=a[i])
01531             for (int j=0; j<(int)x.size(); j++)
01532                 res[2+x[j]] = MiN(res[2+x[j]], a[i+1+x[j]]);
01533
01534         for (int i=1; i<b[0]; i+=b[i])
01535             for (int j=0; j<(int)x.size(); j++)
01536                 res[2+x[j]] = MiN(res[2+x[j]], b[i+1+x[j]]);
01537
01538         return res;
01539     }
01540
01541     // Calculate the contents, their gcd and the primitive parts
01542     poly conta(x.size()==1 ? integer_content(a) : content_univar(a, x[0]));
01543     poly contb(x.size()==1 ? integer_content(b) : content_univar(b, x[0]));
01544     poly gcdconts(x.size()==1 ? integer_gcd(conta, contb) : gcd(conta, contb));
01545     const poly& ppa = conta.is_one() ? a : poly(a/conta);
01546     const poly& ppb = contb.is_one() ? b : poly(b/contb);
01547
01548     if (ppa == ppb)
01549         return ppa * gcdconts;
01550
01551     poly res(BHEAD 0);
01552
01553 #ifdef POLY_GCD_USE_HEURISTIC
01554     // Try the heuristic gcd algorithm
01555     if (a.modp==0 && gcd_heuristic_possible(a) && gcd_heuristic_possible(b)) {
01556         try {
01557             res = gcd_heuristic(ppa, ppb, x);
01558             if (!res.is_zero()) res /= integer_content(res);
01559         }
01560         catch (gcd_heuristic_failed) {}
01561     }
01562 #endif
01563
01564     // If gcd==0, the heuristic algorithm failed, so we do more extensive checks.
01565     // First, we filter out variables that appear in only one of the expressions.
01566     if (res.is_zero()) {
01567         bool unusedVars = false;
01568         for (unsigned int i = 0; i < used.size(); i++) {
01569             if (used[i] == 1) {
01570                 unusedVars = true;
01571                 break;
01572             }
01573         }
01574
01575         // if there are no unused variables, go to the linear routine directly
01576         if (!unusedVars) {
01577             res = gcd_linear(ppa, ppb);
01578 #ifdef DEBUG
01579             cout << "New GCD attempt (unused vars): " << res << endl;

```

```

01580 #endif
01581     }
01582
01583     // if res is not the gcd, it is 0 or larger than the gcd.
01584     // we bracket the expression in all the variables that appear only in one expr.
01585     // and we refine the gcd.
01586     bool diva = !res.is_zero() && poly::divides(res, ppa);
01587     bool divb = !res.is_zero() && poly::divides(res, ppb);
01588     if (!diva || !divb) {
01589         vector<BracketInfo> bracketinfo;
01590
01591         if (!diva) {
01592             map<vector<int>,int> ba = bracket_count(ppa, used);
01593             for(map<vector<int>,int>::iterator it = ba.begin(); it != ba.end(); it++)
01594                 bracketinfo.push_back(BracketInfo(it->first, it->second, &ppa));
01595         }
01596
01597         if (!divb) {
01598             map<vector<int>,int> bb = bracket_count(ppb, used);
01599             for(map<vector<int>,int>::iterator it = bb.begin(); it != bb.end(); it++)
01600                 bracketinfo.push_back(BracketInfo(it->first, it->second, &ppb));
01601         }
01602
01603         // sort so that the smallest bracket will be last
01604         sort(bracketinfo.begin(), bracketinfo.end());
01605
01606         if (res.is_zero()) {
01607             res = bracket(*bracketinfo.back().p, bracketinfo.back().pattern, used);
01608             bracketinfo.pop_back();
01609         }
01610
01611         while (bracketinfo.size() > 0) {
01612             poly subpoly(bracket(*bracketinfo.back().p, bracketinfo.back().pattern, used));
01613             if (!poly::divides(res, subpoly)) {
01614                 // if we can filter out more variables, call gcd again
01615                 if (res.all_variables() != subpoly.all_variables())
01616                     res = gcd(subpoly, res);
01617                 else
01618                     res = gcd_linear(subpoly, res);
01619             }
01620             bracketinfo.pop_back();
01621         }
01622     }
01623
01624     if (res.is_zero() || !poly::divides(res, ppa) || !poly::divides(res, ppb)) {
01625         MesPrint("Bad gcd found.");
01626         std::cout << "Bad gcd:" << res << " for " << ppa << " " << ppb << std::endl;
01627         Terminate(1);
01628     }
01629 }
01630
01631 res *= gcdconts * poly(BHEAD res.sign());
01632
01633 #ifdef DEBUG
01634     cout << "*** [" << thetime() << "] RES : gcd(" << a << ", " << b << ") = " << res << endl;
01635 #endif
01636 return res;
01637 }
01638
01639 /*
01640 #] gcd :
01641 */

```

4.108 polygcd.h

```

00001 #pragma once
00002 /* #[ License : */
00003 /*
00004 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00005 *   When using this file you are requested to refer to the publication
00006 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00007 *   This is considered a matter of courtesy as the development was paid
00008 *   for by FOM the Dutch physics granting agency and we would like to
00009 *   be able to track its scientific use to convince FOM of its value
00010 *   for the community.
00011 *
00012 *   This file is part of FORM.
00013 *
00014 *   FORM is free software: you can redistribute it and/or modify it under the
00015 *   terms of the GNU General Public License as published by the Free Software
00016 *   Foundation, either version 3 of the License, or (at your option) any later

```

```

00017 *   version.
00018 *
00019 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00020 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00021 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00022 *   details.
00023 *
00024 *   You should have received a copy of the GNU General Public License along
00025 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00026 */
00027 /* #] License : */
00028
00029 #include <vector>
00030 #include <map>
00031
00032 class poly; // polynomial class
00033 class gcd_heuristic_failed {}; // class for throwing exceptions
00034
00035 // whether or not to use the heuristic before Zippel's algorithm
00036 #define POLYGCD_USE_HEURISTIC
00037
00038 // maximum number of words in a coefficient for gcd_heuristic to continue
00039 const int POLYGCD_HEURISTIC_MAX_DIGITS = 1000;
00040
00041 // maximum number of retries after the heuristic has failed
00042 const int POLYGCD_HEURISTIC_MAX_TRIES = 10;
00043
00044 // a fudge factor, which improves efficiency
00045 const int POLYGCD_HEURISTIC_MAX_ADD_RANDOM = 10;
00046
00047 // maximum cached power in substitute_last and sparse_interpolation_get_mul_list
00048 const int POLYGCD_RAISPOWMOD_CACHE_SIZE = 1000;
00049
00050 namespace polygcd {
00051
00052     // functions to call the gcd routines
00053     const poly integer_gcd (const poly &a, const poly &b);
00054     const poly integer_content (const poly &a);
00055
00056     const poly gcd (const poly &a, const poly &b);
00057     const poly content_univar (const poly &a, int x);
00058     const poly content_multivar (const poly &a, int x);
00059
00060     const std::vector<WORD> coefficient_list_gcd (const std::vector<WORD> &a, const std::vector<WORD>
&b, WORD p);
00061
00062     // internal functions
00063     const poly gcd_heuristic (const poly &a, const poly &b, const std::vector<int> &x, int
max_tries=POLYGCD_HEURISTIC_MAX_TRIES);
00064     const poly gcd_Euclidean (const poly &a, const poly &b);
00065     const poly gcd_modular (const poly &a, const poly &b, const std::vector<int> &x);
00066     const poly gcd_modular_dense_interpolation (const poly &a, const poly &b, const std::vector<int>
&x, const poly &s);
00067     const poly gcd_modular_sparse_interpolation (const poly &a, const poly &b, const std::vector<int>
&x, const poly &s);
00068
00069     const std::vector<int> sparse_interpolation_get_mul_list (const poly &a, const std::vector<int>
&x, const std::vector<int> &c);
00070     void sparse_interpolation_mul_poly (poly &a, const std::vector<int> &m);
00071     const poly sparse_interpolation_reduce_poly (const poly &a, const std::vector<int> &x);
00072     const poly sparse_interpolation_fix_poly (const poly &a, int x);
00073
00074     const poly chinese_remainder (const poly &a1, const poly &m1, const poly &a2, const poly &m2);
00075     const poly substitute(const poly &a, int x, int c);
00076     const std::map<std::vector<int>, poly> full_bracket(const poly &a, const std::vector<int> &filter);
00077     const poly bracket(const poly &a, const std::vector<int> &pattern, const std::vector<int> &
filter);
00078     const std::map<std::vector<int>, int> bracket_count(const poly &a, const std::vector<int> &filter);
00079     const poly gcd_linear (const poly &a, const poly &b);
00080 }

```

4.109 polywrap.cc File Reference

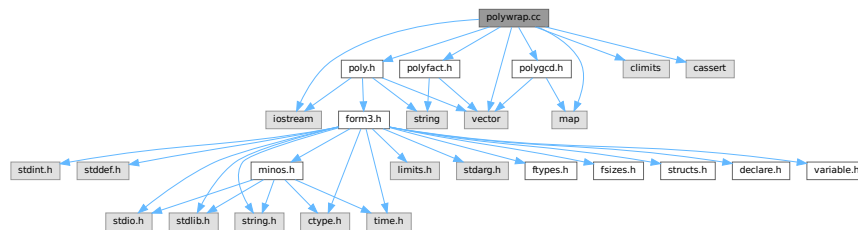
```

#include "poly.h"
#include "polygcd.h"
#include "polyfact.h"
#include <iostream>
#include <vector>
#include <map>

```

```
#include <climits>
#include <cassert>
```

Include dependency graph for polywrap.cc:



Functions

- WORD [poly_determine_modulus](#) (PHEAD bool multi_error, bool is_fun_arg, string message)
- WORD * [poly_gcd](#) (PHEAD WORD *a, WORD *b, WORD fit)
- WORD * [poly_divmod](#) (PHEAD WORD *a, WORD *b, int divmod, WORD fit)
- WORD * [poly_div](#) (PHEAD WORD *a, WORD *b, WORD fit)
- WORD * [poly_rem](#) (PHEAD WORD *a, WORD *b, WORD fit)
- void [poly_ratfun_read](#) (WORD *a, [poly](#) &num, [poly](#) &den)
- void [poly_sort](#) (PHEAD WORD *a)
- WORD * [poly_ratfun_add](#) (PHEAD WORD *t1, WORD *t2)
- int [poly_ratfun_normalize](#) (PHEAD WORD *term)
- void [poly_fix_minus_signs](#) ([factorized_poly](#) &a)
- WORD * [poly_factorize](#) (PHEAD WORD *argin, WORD *argout, bool with_arghead, bool is_fun_arg)
- int [poly_factorize_argument](#) (PHEAD WORD *argin, WORD *argout)
- WORD * [poly_factorize_dollar](#) (PHEAD WORD *argin)
- int [poly_factorize_expression](#) ([EXPRESSIONS](#) expr)
- int [poly_unfactorize_expression](#) ([EXPRESSIONS](#) expr)
- WORD * [poly_inverse](#) (PHEAD WORD *arga, WORD *argb)
- WORD * [poly_mul](#) (PHEAD WORD *a, WORD *b)
- void [poly_free_poly_vars](#) (PHEAD const char *text)

Variables

- const int [POLYWRAP_DENOMPOWER_INCREASE_FACTOR](#) = 2

4.109.1 Detailed Description

Contains methods to call the polynomial methods (written in C++) from the rest of Form (written in C). These include polynomial gcd computation, factorization and polyratfuns.

Definition in file [polywrap.cc](#).

4.109.2 Function Documentation

4.109.2.1 `poly_determine_modulus()`

```
WORD poly_determine_modulus (
    PHEAD bool multi_error,
    bool is_fun_arg,
    string message )
```

Modulus for polynomial algebra

4.109.3 Description

This method determines whether polynomial algebra is done with a modulus or not. This depends on AC.ncmod. If only_funargs is set it also depends on (AC.modmode & ALSOFUNARGS).

The program terminates if the feature is not implemented. Polynomial algebra modulo $M > \text{WORDSIZE}$ is not implemented. If multi_error is set, multivariate algebra mod M is not implemented.

4.109.4 Notes

- If AC.ncmod>0 and only_funargs=true and AC.modmode&ALSOFUNARGS=false, AN.ncmod is set to zero, for otherwise RaisPow calculates mod M .

Definition at line 79 of file [polywrap.cc](#).

Referenced by [poly_factorize\(\)](#), [poly_factorize_expression\(\)](#), [poly_gcd\(\)](#), [poly_ratfun_add\(\)](#), and [poly_ratfun_normalize\(\)](#).

4.109.4.1 `poly_gcd()`

```
WORD * poly_gcd (
    PHEAD WORD * a,
    WORD * b,
    WORD fit )
```

Polynomial gcd

4.109.5 Description

This method calculates the greatest common divisor of two polynomials, given by two zero-terminated Form-style term lists.

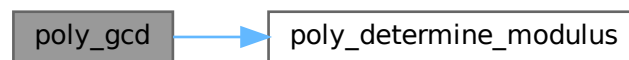
4.109.6 Notes

- The result is written at newly allocated memory
- Called from [ratio.c](#)
- Calls `polygcd::gcd`

Definition at line [124](#) of file [polywrap.cc](#).

References [poly_determine_modulus\(\)](#).

Here is the call graph for this function:



4.109.6.1 poly_divmod()

```
WORD * poly_divmod (  
    PHEAD WORD * a,  
    WORD * b,  
    int divmod,  
    WORD fit )
```

Definition at line [203](#) of file [polywrap.cc](#).

4.109.6.2 poly_div()

```
WORD * poly_div (  
    PHEAD WORD * a,  
    WORD * b,  
    WORD fit )
```

Definition at line [435](#) of file [polywrap.cc](#).

4.109.6.3 poly_rem()

```
WORD * poly_rem (  
    PHEAD WORD * a,  
    WORD * b,  
    WORD fit )
```

Definition at line [463](#) of file [polywrap.cc](#).

4.109.6.4 poly_ratfun_read()

```
void poly_ratfun_read (
    WORD * a,
    poly & num,
    poly & den )
```

Read a PolyRatFun

4.109.7 Description

This method reads a polyratfun starting at the pointer a. The resulting numerator and denominator are written in num and den. If MUSTCLEANPRF, the result is normalized.

4.109.8 Notes

- Calls polygcd::gcd

Definition at line 496 of file [polywrap.cc](#).

Referenced by [poly_ratfun_add\(\)](#), and [poly_ratfun_normalize\(\)](#).

4.109.8.1 poly_sort()

```
void poly_sort (
    PHEAD WORD * a )
```

Sort the polynomial terms

4.109.9 Description

Sorts the terms of a polynomial in Form [poly\(rat\)](#)fun order, i.e. lexicographical order with highest degree first.

4.109.10 Notes

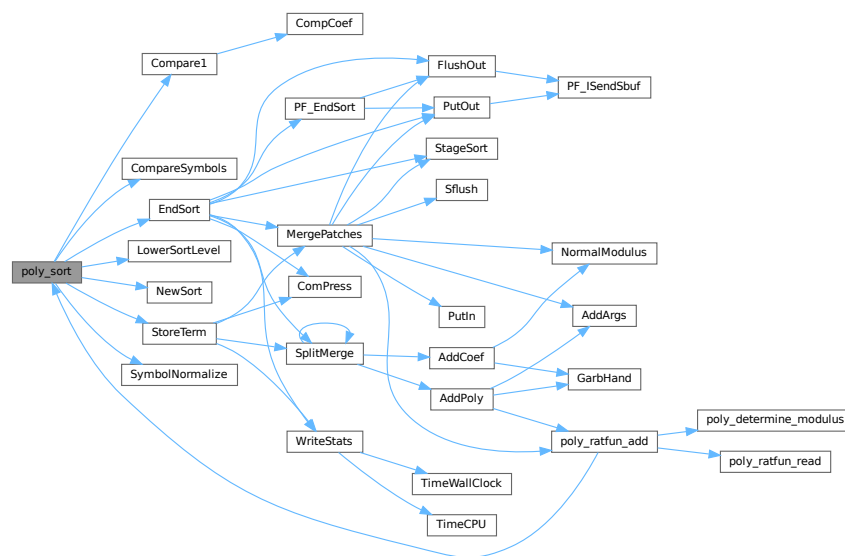
- Uses Form sort routines with custom compare

Definition at line 590 of file [polywrap.cc](#).

References [Compare1\(\)](#), [CompareSymbols\(\)](#), [EndSort\(\)](#), [LowerSortLevel\(\)](#), [NewSort\(\)](#), [StoreTerm\(\)](#), and [SymbolNormalize\(\)](#).

Referenced by [poly_ratfun_add\(\)](#), and [poly_ratfun_normalize\(\)](#).

Here is the call graph for this function:



4.109.10.1 poly_ratfun_add()

```
WORD * poly_ratfun_add (
    PHEAD WORD * t1,
    WORD * t2 )
```

Addition of PolyRatFuns

4.109.11 Description

This method gets two pointers to polyratfuns with up to two arguments each and calculates the sum.

4.109.14 Notes

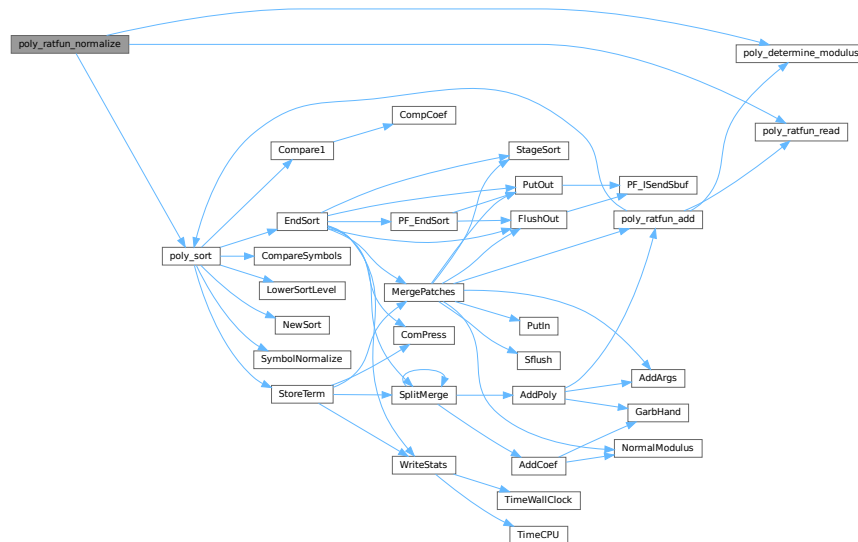
- The result overwrites the original term
- Called from [proces.c](#)
- Calls `poly::operators` and `polygcd::gcd`

Definition at line 769 of file [polywrap.cc](#).

References [poly_determine_modulus\(\)](#), [poly_ratfun_read\(\)](#), and [poly_sort\(\)](#).

Referenced by [PolyFunMul\(\)](#), and [PrepPoly\(\)](#).

Here is the call graph for this function:



4.109.14.1 poly_fix_minus_signs()

```
void poly_fix_minus_signs (
    factorized_poly & a )
```

Definition at line 921 of file [polywrap.cc](#).

4.109.14.2 poly_factorize()

```
WORD * poly_factorize (
    PHEAD WORD * argin,
    WORD * argout,
    bool with_arghead,
    bool is_fun_arg )
```

Factorization of function arguments / dollars

4.109.15 Description

This method factorizes a Form style argument or zero-terminated term list.

4.109.16 Notes

- Called from `poly_factorize_{argument,dollar}`
- Calls `polyfact::factorize`

Definition at line 986 of file `polywrap.cc`.

References `poly_determine_modulus()`.

Referenced by `poly_factorize_argument()`, and `poly_factorize_dollar()`.

Here is the call graph for this function:



4.109.16.1 poly_factorize_argument()

```
int poly_factorize_argument (
    PHEAD WORD * argin,
    WORD * argout )
```

Factorization of function arguments

4.109.17 Description

This method factorizes the Form-style argument `argin`.

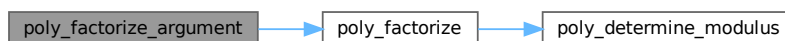
4.109.18 Notes

- The result is written at argout
- Called from [argument.c](#)
- Calls `poly_factorize`

Definition at line 1112 of file [polywrap.cc](#).

References [poly_factorize\(\)](#).

Here is the call graph for this function:



4.109.18.1 poly_factorize_dollar()

```
WORD * poly_factorize_dollar (
    PHEAD WORD * argin )
```

Factorization of dollar variables

4.109.19 Description

This method factorizes a dollar variable.

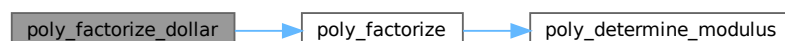
4.109.20 Notes

- The result is written at newly allocated memory.
- Called from [dollar.c](#)
- Calls `poly_factorize`

Definition at line 1146 of file [polywrap.cc](#).

References [poly_factorize\(\)](#).

Here is the call graph for this function:



```
int poly_factorize_expression (
    EXPRESSIONS expr )
```

4.109.21 Description

4.109.22 Notes

- Definition at line 1178 of file polywrap.cc.

Referenced by [PF_Processor\(\)](#), and [Processor\(\)](#).

```
int poly_unfactorize_expression (
    EXPRESSIONS expr )
```

4.109.23 Description

4.109.24 Notes

- Referenced by [PF_Processor\(\)](#), and [Processor\(\)](#).

[illegible]

4.109.24.1 poly_inverse()

```
WORD * poly_inverse (
    PHEAD WORD * arga,
    WORD * argb )
```

Definition at line 1743 of file [polywrap.cc](#).

4.109.24.2 poly_mul()

```
WORD * poly_mul (
    PHEAD WORD * a,
    WORD * b )
```

Definition at line 1948 of file [polywrap.cc](#).

4.109.24.3 poly_free_poly_vars()

```
void poly_free_poly_vars (
    PHEAD const char * text )
```

Definition at line 2014 of file [polywrap.cc](#).

4.109.25 Variable Documentation

4.109.25.1 POLYWRAP_DENOMPOWER_INCREASE_FACTOR

```
const int POLYWRAP_DENOMPOWER_INCREASE_FACTOR = 2
```

Definition at line 52 of file [polywrap.cc](#).

4.110 polywrap.cc

[Go to the documentation of this file.](#)

```
00001
00008 /* # [ License : */
00009 /*
00010 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00011 * When using this file you are requested to refer to the publication
00012 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013 * This is considered a matter of courtesy as the development was paid
00014 * for by FOM the Dutch physics granting agency and we would like to
00015 * be able to track its scientific use to convince FOM of its value
00016 * for the community.
00017 *
00018 * This file is part of FORM.
00019 *
00020 * FORM is free software: you can redistribute it and/or modify it under the
00021 * terms of the GNU General Public License as published by the Free Software
00022 * Foundation, either version 3 of the License, or (at your option) any later
00023 * version.
00024 *
00025 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00028 * details.
00029 *
```



```

00030 *   You should have received a copy of the GNU General Public License along
00031 *   with FORM.  If not, see <http://www.gnu.org/licenses/>.
00032 */
00033 /* #] License : */
00034
00035 #include "poly.h"
00036 #include "polygcd.h"
00037 #include "polyfact.h"
00038
00039 #include <iostream>
00040 #include <vector>
00041 #include <map>
00042 #include <climits>
00043 #include <cassert>
00044
00045 // #define DEBUG
00046
00047 #ifdef DEBUG
00048 #include "mytime.h"
00049 #endif
00050
00051 // denompower is increased by this factor when divmod_heap fails
00052 const int POLYWRAP_DENOMPOWER_INCREASE_FACTOR = 2;
00053
00054 using namespace std;
00055
00056 /*
00057  #[ poly_determine_modulus :
00058 */
00059
00079 WORD poly_determine_modulus (PHEAD bool multi_error, bool is_fun_arg, string message) {
00080
00081     if (AC.ncmod==0) return 0;
00082
00083     if (!is_fun_arg || (AC.modmode & ALSOFUNARGS)) {
00084
00085         if (ABS(AC.ncmod)>1) {
00086             MLOCK(ErrorMessageLock);
00087             MesPrint ((char*)"ERROR: %s with modulus > WORDSIZE not implemented",message.c_str());
00088             MUNLOCK(ErrorMessageLock);
00089             Terminate(-1);
00090         }
00091
00092         if (multi_error && AN.poly_num_vars>1) {
00093             MLOCK(ErrorMessageLock);
00094             MesPrint ((char*)"ERROR: multivariate %s with modulus not implemented",message.c_str());
00095             MUNLOCK(ErrorMessageLock);
00096             Terminate(-1);
00097         }
00098
00099         return *AC.cmod;
00100     }
00101
00102     AN.ncmod = 0;
00103     return 0;
00104 }
00105
00106 /*
00107  #] poly_determine_modulus :
00108  #[ poly_gcd :
00109 */
00110
00124 WORD *poly_gcd(PHEAD WORD *a, WORD *b, WORD fit) {
00125
00126 #ifdef WITHFLINT
00127     if ( AC.FlintPolyFlag && AC.ncmod==0 ) {
00128         WORD *ret = flint_gcd(BHEAD a, b, fit);
00129         return ret;
00130     }
00131 #endif
00132
00133 #ifdef DEBUG
00134     cout << "*** [" << thetime() << "]" << " CALL : poly_gcd" << endl;
00135 #endif
00136
00137 //
00138 //MesPrint("Calling poly_gcd with:");
00139 //{
00140 // WORD *at = a;
00141 // MesPrint("      a:");
00142 // while ( *at ) {
00143 //     MesPrint("      %a",*at,at);
00144 //     at += *at;
00145 // }
00146 // MesPrint("      b:");
00147 // at = b;
00148 // while ( *at ) {

```

```

00149 //      MesPrint("    %a",*at,at);
00150 //      at += *at;
00151 //  }
00152 //}
00153
00154 // Extract variables
00155 vector<WORD *> e;
00156 e.reserve(2);
00157 e.push_back(a);
00158 e.push_back(b);
00159 poly::get_variables(BHEAD e, false, true);
00160
00161 // Check for modulus calculus
00162 WORD modp=poly_determine_modulus(BHEAD true, true, "polynomial GCD");
00163
00164 // Convert to polynomials
00165 poly pa(poly::argument_to_poly(BHEAD a, false, true), modp, 1);
00166 poly pb(poly::argument_to_poly(BHEAD b, false, true), modp, 1);
00167
00168 // Calculate gcd
00169 poly gcd(polygcd::gcd(pa,pb));
00170
00171 // Allocate new memory and convert to Form notation
00172 int newsize = (gcd.size_of_form_notation()+1);
00173 WORD *res;
00174 if ( fit ) {
00175     if ( newsize*sizeof(WORD) >= (size_t)(AM.MaxTer) ) {
00176         MLOCK(ErrorMessageLock);
00177         MesPrint("poly_gcd: Term too complex (%d words). Maybe increasing MaxTermSize (%d words)
can help",
00178                 newsize, AM.MaxTer/sizeof(WORD));
00179         MUNLOCK(ErrorMessageLock);
00180         Terminate(-1);
00181     }
00182     res = TermMalloc("poly_gcd");
00183 }
00184 else {
00185     res = (WORD *)Malloca1(newsize*sizeof(WORD), "poly_gcd");
00186 }
00187 poly::poly_to_argument(gcd, res, false);
00188
00189 poly_free_poly_vars(BHEAD "AN.poly_vars_qcd");
00190
00191 // reset modulo calculation
00192 AN.ncmod = AC.ncmod;
00193 return res;
00194 }
00195
00196 /*
00197 #] poly_gcd :
00198 #[ poly_divmod :
00199
00200 if fit == 1 the answer must fit inside a term.
00201 */
00202
00203 WORD *poly_divmod(PHEAD WORD *a, WORD *b, int divmod, WORD fit) {
00204
00205 #ifdef DEBUG
00206     cout << "*** [" << thetime() << "]" CALL : poly_divmod" << endl;
00207 #endif
00208
00209 // check for modulus calculus
00210 WORD modp=poly_determine_modulus(BHEAD false, true, "polynomial division");
00211
00212 // get variables
00213 vector<WORD *> e;
00214 e.reserve(2);
00215 e.push_back(a);
00216 e.push_back(b);
00217 poly::get_variables(BHEAD e, false, false);
00218
00219 // add extra variables to keep track of denominators
00220 const int DENOMSYMBOL = MAXPOSITIVE;
00221
00222 // WORD *new_poly_vars = (WORD *)Malloca1((AN.poly_num_vars+1)*sizeof(WORD), "AN.poly_vars");
00223 // WCOPY(new_poly_vars, AN.poly_vars, AN.poly_num_vars);
00224 // new_poly_vars[AN.poly_num_vars] = DENOMSYMBOL;
00225 // if (AN.poly_num_vars > 0)
00226 //     M_free(AN.poly_vars, "AN.poly_vars");
00227 // AN.poly_num_vars++;
00228 // AN.poly_vars = new_poly_vars;
00229 // AN.poly_vars[AN.poly_num_vars++] = DENOMSYMBOL;
00230
00231 // convert to polynomials
00232 poly dena(BHEAD 0);
00233 poly denb(BHEAD 0);
00234 poly pa(poly::argument_to_poly(BHEAD a, false, true, &dena), modp, 1);

```

```

00235     poly pb(poly::argument_to_poly(BHEAD b, false, true, &denb), modp, 1);
00236
00237     // remove contents
00238     poly numres(polygcd::integer_content(pa));
00239     poly denres(polygcd::integer_content(pb));
00240     pa /= numres;
00241     pb /= denres;
00242
00243     if (divmod==0) {
00244         numres *= denb;
00245         denres *= dena;
00246     }
00247     else {
00248         denres = dena;
00249     }
00250
00251     poly gcdres(polygcd::integer_gcd(numres,denres));
00252     numres /= gcdres;
00253     denres /= gcdres;
00254
00255     // determine lcoeff(b)
00256     poly lcoeffb(pb.integer_lcoeff());
00257
00258     poly pres(BHEAD 0);
00259
00260     if (!lcoeffb.is_one()) {
00261
00262         if (AN.poly_num_vars > 2) {
00263             // the original polynomial is multivariate (one dummy variable has
00264             // been added), so it is not trivial to determine which power of
00265             // lcoeff(b) can be in the answer
00266
00267             int denompower = 0, prevdenompower = 0;
00268             poly pq(BHEAD 0);
00269             poly pr(BHEAD 0);
00270
00271             // try denompower = 0, if this fails increase denompower until division succeeds
00272             bool div_fail = true;
00273             do
00274             {
00275                 if(denompower < prevdenompower)
00276                 {
00277                     // denompower increased beyond INT_MAX
00278                     MLOCK(ErrorMessageLock);
00279                     MesPrint ((char*)"ERROR: pseudo-division failed in poly_divmod (denompower >
INT_MAX)");
00280                     MUNLOCK(ErrorMessageLock);
00281                     Terminate(1);
00282                 }
00283
00284                 if(denompower != 0)
00285                 {
00286                     // multiply a by lcoeffb^(denompower-prevdenompower)
00287                     WORD n = lcoeffb[lcoeffb[1]];
00288                     RaisPow(BHEAD (UWORD *)&lcoeffb[2+AN.poly_num_vars], &n,
denompower-prevdenompower);
00289                     lcoeffb[1] = 2 + AN.poly_num_vars + ABS(n);
00290                     lcoeffb[0] = 1 + lcoeffb[1];
00291                     lcoeffb[lcoeffb[1]] = n;
00292
00293                     pa *= lcoeffb;
00294                     denres *= lcoeffb;
00295                 }
00296
00297                 // try division
00298                 poly ppow(BHEAD 0);
00299                 poly::divmod_heap(pa,pb,pq,pr,false,true,div_fail); // sets div_fail
00300
00301                 // increase denompower for next iteration
00302                 prevdenompower = denompower;
00303                 denompower = (denompower==0 ? 1 : denompower*POLYWRAP_DENOMPPOWER_INCREASE_FACTOR+1 );
00304             }
00305             while(div_fail);
00306
00307             pres = (divmod==0 ? pq : pr);
00308
00309         }
00310         else {
00311             // one variable, so the power is the difference of the degrees
00312
00313             int denompower = MaX(0, pa.degree(0) - pb.degree(0) + 1);
00314
00315             // multiply a by that power
00316             WORD n = lcoeffb[lcoeffb[1]];
00317             RaisPow(BHEAD (UWORD *)&lcoeffb[2+AN.poly_num_vars], &n, denompower);
00318             lcoeffb[1] = 2 + AN.poly_num_vars + ABS(n);

```

```

00319         lcoeffb[0] = 1 + lcoeffb[1];
00320         lcoeffb[lcoeffb[1]] = n;
00321
00322         pa *= lcoeffb;
00323         denres *= lcoeffb;
00324
00325         pres = (divmod==0 ? pa/pb : pa%pb);
00326
00327     }
00328 }
00329 }
00330 else {
00331
00332     pres = (divmod==0 ? pa/pb : pa%pb);
00333
00334 }
00335
00336 // convert to Form notation
00337 // NOTE: this part can be rewritten with poly::size_of_form_notation_with_den()
00338 // and poly::poly_to_argument_with_den().
00339 WORD *res;
00340
00341 // special case: a=0
00342 if (pres[0]==1) {
00343     if ( fit ) {
00344         res = TermMalloc("poly_divmod");
00345     }
00346     else {
00347         res = (WORD *)Malloca(sizeof(WORD), "poly_divmod");
00348     }
00349     res[0] = 0;
00350 }
00351 else {
00352     pres *= numres;
00353
00354     WORD nden;
00355     UWORD *den = (UWORD *)NumberMalloc("poly_divmod");
00356
00357     // allocate the memory; note that this overestimates the size,
00358     // since the estimated denominators are too large
00359     int ressize = pres.size_of_form_notation() + pres.number_of_terms()*2*ABS(denres[denres[1]]) +
1;
00360
00361     if ( fit ) {
00362         if ( ressize*sizeof(WORD) > (size_t)(AM.MaxTer) ) {
00363             MLOCK(ErrorMessageLock);
00364             MesPrint("poly_divmod: Term too complex (%d words). Maybe increasing MaxTermSize (%d
words) can help",
00365                     ressize, AM.MaxTer/sizeof(WORD));
00366             MUNLOCK(ErrorMessageLock);
00367             Terminate(-1);
00368         }
00369         res = TermMalloc("poly_divmod");
00370     }
00371     else {
00372         res = (WORD *)Malloca(ressize*sizeof(WORD), "poly_divmod");
00373     }
00374     int L=0;
00375
00376     for (int i=1; i!=pres[0]; i+=pres[i]) {
00377
00378         res[L]=1; // length
00379         bool first = true;
00380
00381         for (int j=0; j<AN.poly_num_vars; j++)
00382             if (pres[i+1+j] > 0) {
00383                 if (first) {
00384                     first = false;
00385                     res[L+1] = 1; // symbols
00386                     res[L+2] = 2; // length
00387                 }
00388                 res[L+1+res[L+2]++] = AN.poly_vars[j]; // symbol
00389                 res[L+1+res[L+2]++] = pres[i+1+j]; // power
00390             }
00391
00392         if (!first) res[L] += res[L+2]; // fix length
00393
00394         // numerator
00395         WORD nnum = pres[i+pres[i]-1];
00396         WCOPY(&res[L+res[L]], &pres[i+pres[i]-1-ABS(nnum)], ABS(nnum));
00397
00398         // calculate denominator
00399         nden = denres[denres[1]];
00400         WCOPY(den, &denres[2+AN.poly_num_vars], ABS(nden));
00401
00402         if (nden!=1 || den[0]!=1)
00403             Simplify(BHEAD (UWORD *)&res[L+res[L]], &nnum, den, &nden); // gcd(num,den)

```

```

00404         Pack((UWORD *)&res[L+res[L]], &nnum, den, nden);           // format
00405         res[L] += 2*ABS(nnum)+1;                                     // fix length
00406         res[L+res[L]-1] = SGN(nnum)*(2*ABS(nnum)+1);               // length of coefficient
00407         L += res[L];                                               // fix length
00408     }
00409
00410     res[L] = 0;
00411
00412     NumberFree(den,"poly_divmod");
00413 }
00414
00415 // clean up
00416 poly_free_poly_vars(BHEAD "AN.poly_vars_divmod");
00417
00418 // reset modulo calculation
00419 AN.ncmod = AC.ncmod;
00420
00421 return res;
00422 }
00423
00424 /*
00425 #] poly_divmod :
00426 #[ poly_div :
00427
00428 Routine divides the expression in arg1 by the expression in arg2.
00429 We did not take out special cases.
00430 The arguments are zero terminated sequences of term(s).
00431 The action is to divide arg1 by arg2: [arg1/arg2].
00432 The answer should be a buffer (allocated by Malloc1) with a zero
00433 terminated sequence of terms (or just zero).
00434 */
00435 WORD *poly_div(PHEAD WORD *a, WORD *b, WORD fit) {
00436
00437 #ifdef WITHFLINT
00438     if ( AC.FlintPolyFlag && AC.ncmod==0 ) {
00439         WORD *ret = flint_div(BHEAD a, b, fit);
00440         return ret;
00441     }
00442 #endif
00443
00444 #ifdef DEBUG
00445     cout << "*** [" << thetime() << "]" CALL : poly_div" << endl;
00446 #endif
00447
00448     return poly_divmod(BHEAD a, b, 0, fit);
00449 }
00450
00451 /*
00452 #] poly_div :
00453 #[ poly_rem :
00454
00455 Routine divides the expression in arg1 by the expression in arg2
00456 and takes the remainder.
00457 We did not take out special cases.
00458 The arguments are zero terminated sequences of term(s).
00459 The action is to divide arg1 by arg2 and take the remainder: [arg1%arg2].
00460 The answer should be a buffer (allocated by Malloc1) with a zero
00461 terminated sequence of terms (or just zero).
00462 */
00463 WORD *poly_rem(PHEAD WORD *a, WORD *b, WORD fit) {
00464
00465 #ifdef WITHFLINT
00466     if ( AC.FlintPolyFlag && AC.ncmod==0 ) {
00467         WORD *ret = flint_rem(BHEAD a, b, fit);
00468         return ret;
00469     }
00470 #endif
00471
00472 #ifdef DEBUG
00473     cout << "*** [" << thetime() << "]" CALL : poly_rem" << endl;
00474 #endif
00475
00476     return poly_divmod(BHEAD a, b, 1, fit);
00477 }
00478
00479 /*
00480 #] poly_rem :
00481 #[ poly_ratfun_read :
00482 */
00483
00496 void poly_ratfun_read (WORD *a, poly &nnum, poly &den) {
00497
00498 #ifdef DEBUG
00499     cout << "*** [" << thetime() << "]" CALL : poly_ratfun_read" << endl;
00500 #endif
00501
00502     POLY_GETIDENTITY(nnum);

```

```

00503
00504     int modp = num.modp;
00505
00506     WORD *astop = a+a[1];
00507
00508     bool clean = (a[2] & MUSTCLEANPRF) == 0;
00509
00510     a += FUNHEAD;
00511     if (a >= astop) {
00512         MLOCK(ErrorMessageLock);
00513         MesPrint ((char*)"ERROR: PolyRatFun cannot have zero arguments");
00514         MUNLOCK(ErrorMessageLock);
00515         Terminate(-1);
00516     }
00517
00518     poly den_num(BHEAD 1), den_den(BHEAD 1);
00519
00520     num = poly::argument_to_poly(BHEAD a, true, !clean, &den_num);
00521     num.setmod(modp,1);
00522     NEXTARG(a);
00523
00524     if (a < astop) {
00525         den = poly::argument_to_poly(BHEAD a, true, !clean, &den_den);
00526         den.setmod(modp,1);
00527         NEXTARG(a);
00528     }
00529     else {
00530         den = poly(BHEAD 1, modp, 1);
00531     }
00532
00533     if (a < astop) {
00534         MLOCK(ErrorMessageLock);
00535         MesPrint ((char*)"ERROR: PolyRatFun cannot have more than two arguments");
00536         MUNLOCK(ErrorMessageLock);
00537         Terminate(-1);
00538     }
00539
00540     // JD: At this point, num and den are certainly sorted into the correct order by
00541     // poly::argument_to_poly, but we can't rely on the clean flag to know if there
00542     // are any negative powers. Check for them, and set clean = false if there are any.
00543     vector<WORD> minpower(AN.poly_num_vars, MAXPOSITIVE);
00544
00545     for (int i=1; i<num[0]; i+=num[i]) {
00546         for (int j=0; j<AN.poly_num_vars; j++) {
00547             minpower[j] = Min(minpower[j], num[i+1+j]);
00548         }
00549     }
00550     for (int i=1; i<den[0]; i+=den[i]) {
00551         for (int j=0; j<AN.poly_num_vars; j++) {
00552             minpower[j] = Min(minpower[j], den[i+1+j]);
00553             if ( minpower[j] < 0 ) clean = false;
00554         }
00555     }
00556
00557     if (!clean) {
00558         for (int i=1; i<num[0]; i+=num[i])
00559             for (int j=0; j<AN.poly_num_vars; j++)
00560                 num[i+1+j] -= minpower[j];
00561         for (int i=1; i<den[0]; i+=den[i])
00562             for (int j=0; j<AN.poly_num_vars; j++)
00563                 den[i+1+j] -= minpower[j];
00564
00565         num *= den_den;
00566         den *= den_num;
00567
00568         poly gcd = polygcd::gcd(num,den);
00569         num /= gcd;
00570         den /= gcd;
00571     }
00572 }
00573
00574 /*
00575 #] poly_ratfun_read :
00576 #[ poly_sort :
00577 */
00578
00590 void poly_sort(PHEAD WORD *a) {
00591
00592 #ifdef DEBUG
00593     cout << "*** [" << thetime() << "]" CALL : poly_sort" << endl;
00594 #endif
00595     if (NewSort(BHEAD0)) { Terminate(-1); }
00596     AR.CompareRoutine = (COMPAREDUMMY)(&CompareSymbols);
00597
00598     for (int i=ARGHEAD; i<a[0]; i+=a[i]) {
00599         if (SymbolNormalize(a+i)<0 || StoreTerm(BHEAD a+i)) {
00600             AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);

```

```

00601         LowerSortLevel();
00602         Terminate(-1);
00603     }
00604 }
00605
00606 if (EndSort(BHEAD a+ARGHEAD,1) < 0) {
00607     AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
00608     Terminate(-1);
00609 }
00610
00611 AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
00612 a[1] = 0; // set dirty flag to zero
00613 }
00614
00615 /*
00616 #] poly_sort :
00617 #[ poly_ratfun_add :
00618 */
00619
00633 WORD *poly_ratfun_add (PHEAD WORD *t1, WORD *t2) {
00634
00635 #ifdef WITHFLINT
00636     if ( AC.FlintPolyFlag && AC.ncmod==0 ) {
00637         WORD *ret = flint_ratfun_add(BHEAD t1, t2);
00638         return ret;
00639     }
00640 #endif
00641
00642     if ( AR.PolyFunExp == 1 ) return PolyRatFunSpecial(BHEAD t1, t2);
00643
00644 #ifdef DEBUG
00645     cout << "*** [" << thetime() << " ] CALL : poly_ratfun_add" << endl;
00646 #endif
00647
00648     WORD *oldworkpointer = AT.WorkPointer;
00649
00650     // Extract variables
00651     vector<WORD *> e;
00652     e.reserve(4);
00653
00654     for (WORD *t=t1+FUNHEAD; t<t1+t1[1];) {
00655         e.push_back(t);
00656         NEXTARG(t);
00657     }
00658     for (WORD *t=t2+FUNHEAD; t<t2+t2[1];) {
00659         e.push_back(t);
00660         NEXTARG(t);
00661     }
00662
00663     assert(e.size() == 4);
00664
00665     poly::get_variables(BHEAD e, true, true);
00666
00667     // Check for modulus calculus
00668     WORD modp=poly_determine_modulus(BHEAD true, true, "PolyRatFun");
00669
00670     // Find numerators / denominators
00671     poly num1(BHEAD 0,modp,1), den1(BHEAD 0,modp,1), num2(BHEAD 0,modp,1), den2(BHEAD 0,modp,1);
00672
00673     poly_ratfun_read(t1, num1, den1);
00674     poly_ratfun_read(t2, num2, den2);
00675
00676     poly num(BHEAD 0),den(BHEAD 0),gcd(BHEAD 0);
00677
00678     // Calculate result
00679     if (den1 != den2) {
00680         gcd = polygcd::gcd(den1,den2);
00681 #ifdef OLDADDITION
00682         num = num1*(den2/gcd) + num2*(den1/gcd);
00683         den = (den1/gcd)*den2;
00684         gcd = polygcd::gcd(num,den);
00685 #else
00686         den = den1/gcd;
00687         num = num1*(den2/gcd) + num2*den;
00688         den = den*den2;
00689         gcd = polygcd::gcd(num,gcd);
00690 #endif
00691     }
00692     else {
00693         num = num1 + num2;
00694         den = den1;
00695         gcd = polygcd::gcd(num,den);
00696     }
00697
00698     num /= gcd;
00699     den /= gcd;
00700

```

```

00701 // Fix sign
00702 if (den.sign() == -1) { num*=poly(BHEAD -1); den*=poly(BHEAD -1); }
00703
00704 // Check size: include FUNHEAD for the prf itself, an ARGHEAD each for num and den,
00705 // and 3 for the final coeff "1/1". We don't know here what the rest of the term looks like,
00706 // but it certainly has at least its total size (so +1):
00707 if ((num.size_of_form_notation() + den.size_of_form_notation() + FUNHEAD + 2*ARGHEAD + 3 + 1)
00708     > AM.MaxTer/(int)sizeof(WORD)) {
00709
00710     MLOCK(ErrorMessageLock);
00711     MesPrint ("ERROR: PolyRatFun doesn't fit in a term");
00712     MesPrint ("(1) num size = %d, den size = %d, rest = %d, MaxTermSize = %d words",
00713         num.size_of_form_notation()+ARGHEAD,
00714         den.size_of_form_notation()+ARGHEAD,
00715         FUNHEAD + 3 + 1,
00716         AM.MaxTer/sizeof(WORD));
00717     MUNLOCK(ErrorMessageLock);
00718     Terminate(-1);
00719 }
00720
00721 // Format result in Form notation
00722 WORD *t = oldworkpointer;
00723
00724 *t++ = AR.PolyFun; // function
00725 *t++ = 0; // length (to be determined)
00726 // *t++ &= ~MUSTCLEANPRF; // clean polyratfun
00727 *t++ = 0;
00728 FILLFUN3(t); // header
00729 poly::poly_to_argument(num,t, true); // argument 1 (numerator)
00730 if (*t>0 && t[1]==DIRTYFLAG) // to Form order
00731     poly_sort(BHEAD t);
00732 t += (*t>0 ? *t : 2);
00733 poly::poly_to_argument(den,t, true); // argument 2 (denominator)
00734 if (*t>0 && t[1]==DIRTYFLAG) // to Form order
00735     poly_sort(BHEAD t);
00736 t += (*t>0 ? *t : 2);
00737
00738 oldworkpointer[1] = t - oldworkpointer; // length
00739 AT.WorkPointer = t;
00740
00741 poly_free_poly_vars(BHEAD "AN.poly_vars_ratfun_add");
00742
00743 // reset modulo calculation
00744 AN.ncmod = AC.ncmod;
00745
00746 return oldworkpointer;
00747 }
00748
00749 /*
00750 #] poly_ratfun_add :
00751 #[ poly_ratfun_normalize :
00752 */
00753
00769 int poly_ratfun_normalize (PHEAD WORD *term) {
00770
00771 #ifdef WITHFLINT
00772     if ( AC.FlintPolyFlag && AC.ncmod==0 ) {
00773         flint_ratfun_normalize(BHEAD term);
00774         return 0;
00775     }
00776 #endif
00777
00778 #ifdef DEBUG
00779     cout << "*** [" << thetime() << " ] CALL : poly_ratfun_normalize" << endl;
00780 #endif
00781
00782 // Strip coefficient
00783 WORD *tstop = term + *term;
00784 int ncoeff = tstop[-1];
00785 tstop -= ABS(ncoeff);
00786
00787 // if only one clean polyratfun, return immediately
00788 int num_polyratfun = 0;
00789
00790 for (WORD *t=term+1; t<tstop; t+=t[1])
00791     if (*t == AR.PolyFun) {
00792         num_polyratfun++;
00793         if ((t[2] & MUSTCLEANPRF) != 0)
00794             num_polyratfun = INT_MAX;
00795         if (num_polyratfun > 1) break;
00796     }
00797
00798 if (num_polyratfun <= 1) return 0;
00799
00800 WORD oldsorttype = AR.SortType;
00801 AR.SortType = SORTHIGFIRST;
00802

```



```

00803 /*
00804     When there are polyratfun's with only one variable: rename them
00805     temporarily to TMPPOLYFUN.
00806 */
00807 for (WORD *t=term+1; t<tstop; t+=t[1]) {
00808     if (*t == AR.PolyFun && (t[1] == FUNHEAD+t[FUNHEAD]
00809         || t[1] == FUNHEAD+2 ) ) { *t = TMPPOLYFUN; }
00810 }
00811
00812 // Extract all variables in the polyfuns
00813 vector<WORD *> e;
00814
00815 for (WORD *t=term+1; t<tstop; t+=t[1]) {
00816     if (*t == AR.PolyFun)
00817         for (WORD *t2 = t+FUNHEAD; t2<t+t[1];) {
00818             e.push_back(t2);
00819             NEXTARG(t2);
00820         }
00821 }
00822
00823 poly::get_variables(BHEAD e, true, true);
00824
00825 // Check for modulus calculus
00826 WORD modp=poly_determine_modulus(BHEAD true, true, "PolyRatFun");
00827
00828 // Accumulate total denominator/numerator and copy the remaining terms
00829 // We start with 'trivial' polynomials
00830 poly num1(BHEAD (UWORD *)tstop, ncoeff/2, modp, 1);
00831 poly den1(BHEAD (UWORD *)tstop+ABS(ncoeff/2), ABS(ncoeff)/2, modp, 1);
00832
00833 WORD *s = term+1;
00834
00835 for (WORD *t=term+1; t<tstop;)
00836     if (*t == AR.PolyFun) {
00837         poly num2(BHEAD 0, modp, 1);
00838         poly den2(BHEAD 0, modp, 1);
00839         poly_ratfun_read(t, num2, den2);
00840
00841         if ((t[2] & MUSTCLEANPRF) != 0) { // first normalize
00842             poly gcd1(polygcd::gcd(num2, den2));
00843             num2 = num2/gcd1;
00844             den2 = den2/gcd1;
00845         }
00846         t += t[1];
00847         poly gcd1(polygcd::gcd(num1, den2));
00848         poly gcd2(polygcd::gcd(num2, den1));
00849
00850         num1 = (num1 / gcd1) * (num2 / gcd2);
00851         den1 = (den1 / gcd2) * (den2 / gcd1);
00852     }
00853     else {
00854         int i = t[1];
00855         if (s != t) { NCOPY(s, t, i) }
00856         else { t += i; s += i; }
00857     }
00858 }
00859
00860 // Fix sign
00861 if (den1.sign() == -1) { num1*=poly(BHEAD -1); den1*=poly(BHEAD -1); }
00862
00863 // Check size: include FUNHEAD for the prf itself, an ARGHEAD each for num and den,
00864 // s-term for the copied term so far, and 3 for final coeff "1/1"
00865 if ((num1.size_of_form_notation() + den1.size_of_form_notation() + FUNHEAD + 2*ARGHEAD
00866     + s-term + 3) > AM.MaxTer/(int)sizeof(WORD)) {
00867     MLOCK(ErrorMessageLock);
00868     MesPrint ("ERROR: PolyRatFun doesn't fit in a term");
00869     MesPrint ("(2) num size = %d, den size = %d, rest = %d, MaxTermSize = %d words",
00870         num1.size_of_form_notation()+ARGHEAD,
00871         den1.size_of_form_notation()+ARGHEAD,
00872         FUNHEAD + s-term + 3,
00873         AM.MaxTer/sizeof(WORD));
00874     MUNLOCK(ErrorMessageLock);
00875     Terminate(-1);
00876 }
00877
00878 // Format result in Form notation
00879 WORD *t = s;
00880 *t++ = AR.PolyFun; // function
00881 *t++ = 0; // size (to be determined)
00882 *t++ &= ~MUSTCLEANPRF; // clean polyratfun
00883 FILLFUN3(t); // header
00884 poly::poly_to_argument(num1, t, true); // argument 1 (numerator)
00885 if (*t>0 && t[1]==DIRTYFLAG) // to Form order
00886     poly_sort(BHEAD t);
00887 t += (*t>0 ? *t : 2);
00888 poly::poly_to_argument(den1, t, true); // argument 2 (denominator)

```

```

00890     if (*t>0 && t[1]==DIRTYFLAG)           // to Form order
00891         poly_sort(BHEAD t);
00892     t += (*t>0 ? *t : 2);
00893
00894     s[1] = t - s;                           // function length
00895
00896     *t++ = 1;                                // term coefficient
00897     *t++ = 1;
00898     *t++ = 3;
00899
00900     term[0] = t-term;                        // term length
00901
00902     poly_free_poly_vars(BHEAD "AN.poly_vars_ratfun_normalize");
00903
00904     // reset modulo calculation
00905     AN.ncmod = AC.ncmod;
00906
00907     tstop = term + *term; tstop -= ABS(tstop[-1]);
00908     for (WORD *t=term+1; t<tstop; t+=t[1]) {
00909         if (*t == TMPPOLYFUN) *t = AR.PolyFun;
00910     }
00911
00912     AR.SortType = oldsorttype;
00913     return 0;
00914 }
00915
00916 /*
00917 #] poly_ratfun_normalize :
00918 #[ poly_fix_minus_signs :
00919 */
00920
00921 void poly_fix_minus_signs (factorized_poly &a) {
00922
00923     if ( a.factor.empty() ) return;
00924
00925     POLY_GETIDENTITY(a.factor[0]);
00926
00927     int overall_sign = +1;
00928
00929     // find term with maximum power of highest symbol
00930     for (int i=0; i<(int)a.factor.size(); i++) {
00931
00932         int maxvar = -1;
00933         int maxpow = -1;
00934         int sign = +1;
00935
00936         WORD *tstop = a.factor[i].terms; tstop += *tstop;
00937         for (WORD *t=a.factor[i].terms+1; t<tstop; t+=*t)
00938             for (int j=0; j<AN.poly_num_vars; j++) {
00939                 int var = AN.poly_vars[j];
00940                 int pow = t[1+j];
00941                 if (pow>0 && (var>maxvar || (var==maxvar && pow>maxpow))) {
00942                     maxvar = var;
00943                     maxpow = pow;
00944                     sign = SGN(*(t+*t-1));
00945                 }
00946             }
00947
00948         // if negative coefficient, multiply by -1
00949         if (sign== -1) {
00950             a.factor[i] *= poly(BHEAD sign);
00951             if (a.power[i] % 2 == 1) overall_sign*=-1;
00952         }
00953     }
00954
00955     // if overall minus sign
00956     if (overall_sign == -1) {
00957         // look at constant factor and multiply by -1
00958         for (int i=0; i<(int)a.factor.size(); i++)
00959             if (a.factor[i].is_integer()) {
00960                 a.factor[i] *= poly(BHEAD -1);
00961                 return;
00962             }
00963
00964         // otherwise, add a factor of -1
00965         a.add_factor(poly(BHEAD -1), 1);
00966     }
00967 }
00968
00969 /*
00970 #] poly_fix_minus_signs :
00971 #[ poly_factorize :
00972 */
00973
00986 WORD *poly_factorize (PHEAD WORD *argin, WORD *argout, bool with_arghead, bool is_fun_arg) {
00987
00988 #ifdef DEBUG

```

```

00989     cout << "*** [" << thetime() << "]" CALL : poly_factorize" << endl;
00990 #endif
00991
00992     poly::get_variables(BHEAD vector<WORD*>(1,argin), with_arghead, true);
00993     poly den(BHEAD 0);
00994     poly a(poly::argument_to_poly(BHEAD argin, with_arghead, true, &den));
00995
00996     // check for modulus calculus
00997     WORD modp=poly_determine_modulus(BHEAD true, is_fun_arg, "polynomial factorization");
00998     a.setmod(modp,1);
00999
01000     // factorize
01001     factorized_poly f(polyfact::factorize(a));
01002     poly_fix_minus_signs(f);
01003
01004     poly num(BHEAD 1);
01005     for (int i=0; i<(int)f.factor.size(); i++)
01006         if (f.factor[i].is_integer())
01007             num = f.factor[i];
01008
01009     // determine size
01010     int len = with_arghead ? ARGHEAD : 0;
01011
01012     if (!num.is_one() || !den.is_one()) {
01013         len++;
01014         len += MAX(ABS(num[num[1]]), den[den[1]])*2+1;
01015         len += with_arghead ? ARGHEAD : 1;
01016     }
01017
01018     for (int i=0; i<(int)f.factor.size(); i++) {
01019         if (!f.factor[i].is_integer()) {
01020             len += f.power[i] * f.factor[i].size_of_form_notation();
01021             len += f.power[i] * (with_arghead ? ARGHEAD : 1);
01022         }
01023     }
01024
01025     len++;
01026
01027     if (argout != NULL) {
01028         // check size
01029         if (len >= AM.MaxTer) {
01030             MLOCK(ErrorMessageLock);
01031             MesPrint ("ERROR: factorization doesn't fit in a term (len = %d, MaxTermSize = %d words)",
01032                     len/sizeof(WORD), AM.MaxTer/sizeof(WORD));
01033             MUNLOCK(ErrorMessageLock);
01034             Terminate(-1);
01035         }
01036     }
01037     else {
01038         // allocate size
01039         argout = (WORD*) Malloc1(len*sizeof(WORD), "poly_factorize");
01040     }
01041
01042     WORD *old_argout = argout;
01043
01044     // constant factor
01045     if (!num.is_one() || !den.is_one()) {
01046         int n = max(ABS(num[num[1]]), ABS(den[den[1]]));
01047
01048         if (with_arghead) {
01049             *argout++ = ARGHEAD + 2 + 2*n;
01050             for (int i=1; i<ARGHEAD; i++)
01051                 *argout++ = 0;
01052         }
01053
01054         *argout++ = 2 + 2*n;
01055
01056         for (int i=0; i<n; i++)
01057             *argout++ = i<ABS(num[num[1]]) ? num[2+AN.poly_num_vars+i] : 0;
01058         for (int i=0; i<n; i++)
01059             *argout++ = i<ABS(den[den[1]]) ? den[2+AN.poly_num_vars+i] : 0;
01060
01061         *argout++ = SGN(num[num[1]]) * (2*n+1);
01062
01063         if (!with_arghead)
01064             *argout++ = 0;
01065         else {
01066             if (ToFast(old_argout, old_argout))
01067                 argout = old_argout+2;
01068         }
01069     }
01070
01071     // non-constant factors
01072     for (int i=0; i<(int)f.factor.size(); i++)
01073         if (!f.factor[i].is_integer())
01074             for (int j=0; j<f.power[i]; j++) {
01075                 poly::poly_to_argument(f.factor[i], argout, with_arghead);

```

```

01076
01077         if (with_arghead)
01078             argout += *argout > 0 ? *argout : 2;
01079         else {
01080             while (*argout!=0) argout+=*argout;
01081             argout++;
01082         }
01083     }
01084
01085     *argout=0;
01086
01087     poly_free_poly_vars(BHEAD "AN.poly_vars_factorize");
01088
01089     // reset modulo calculation
01090     AN.ncmod = AC.ncmod;
01091
01092     return old_argout;
01093 }
01094
01095 /*
01096 #] poly_factorize :
01097 #[ poly_factorize_argument :
01098 */
01099
01100 int poly_factorize_argument(PHEAD WORD *argin, WORD *argout) {
01101
01102     #ifdef DEBUG
01103         cout << "*** [" << thetime() << " ] CALL : poly_factorize_argument" << endl;
01104     #endif
01105
01106     #ifdef WITHFLINT
01107         if ( AC.FlintPolyFlag && AC.ncmod==0 ) {
01108             flint_factorize_argument(BHEAD argin, argout);
01109             return 0;
01110         }
01111     #endif
01112
01113     poly_factorize(BHEAD argin,argout,true,true);
01114     return 0;
01115 }
01116
01117 /*
01118 #] poly_factorize_argument :
01119 #[ poly_factorize_dollar :
01120 */
01121
01122 WORD *poly_factorize_dollar (PHEAD WORD *argin) {
01123
01124     #ifdef DEBUG
01125         cout << "*** [" << thetime() << " ] CALL : poly_factorize_dollar" << endl;
01126     #endif
01127
01128     #ifdef WITHFLINT
01129         if ( AC.FlintPolyFlag && AC.ncmod==0 ) {
01130             return flint_factorize_dollar(BHEAD argin);
01131         }
01132     #endif
01133
01134     return poly_factorize(BHEAD argin,NULL,false,false);
01135 }
01136
01137 /*
01138 #] poly_factorize_dollar :
01139 #[ poly_factorize_expression :
01140 */
01141
01142 int poly_factorize_expression(EXPRESSIONS expr) {
01143
01144     #ifdef DEBUG
01145         cout << "*** [" << thetime() << " ] CALL : poly_factorize_expression" << endl;
01146     #endif
01147
01148     GETIDENTITY;
01149
01150     if (AT.WorkPointer + AM.MaxTer > AT.WorkTop ) {
01151         MLOCK(ErrorMessageLock);
01152         MesWork();
01153         MUNLOCK(ErrorMessageLock);
01154         Terminate(-1);
01155     }
01156
01157     WORD *term = AT.WorkPointer;
01158     WORD startebuf = cbuf[AT.ebufnum].numrhs;
01159     FILEHANDLE *file;
01160     POSITION pos;
01161
01162     FILEHANDLE *oldinfile = AR.infile;

```

```

01199 FILEHANDLE *oldoutfile = AR.outfile;
01200 WORD oldBracketOn = AR.BracketOn;
01201 WORD *oldBrackBuf = AT.BrackBuf;
01202 WORD oldbracketindexflag = AT.bracketindexflag;
01203 char oldCommercial[COMMERCIALSIZE+2];
01204
01205 strcpy(oldCommercial, (char*)AC.Commercial);
01206 strcpy((char*)AC.Commercial, "factorize");
01207
01208 // locate is the input
01209 if (expr->status == HIDDENEXPRESSION || expr->status == HIDDENLEXPRESSION ||
01210     expr->status == INTOHIDEEXPRESSION || expr->status == INTOHIDELEXPRESSION) {
01211     AR.InHiBuf = 0; file = AR.hidefile; AR.GetFile = 2;
01212 }
01213 else {
01214     AR.InInBuf = 0; file = AR.outfile; AR.GetFile = 0;
01215 }
01216
01217 // read and write to expression file
01218 AR.infile = AR.outfile = file;
01219
01220 // dummy indices are not allowed
01221 if (expr->numdummies > 0) {
01222     MesPrint("ERROR: factorization with dummy indices not implemented");
01223     Terminate(-1);
01224 }
01225
01226 // determine whether the expression is on file or in memory
01227 if (file->handle >= 0) {
01228     pos = expr->onfile;
01229     SeekFile(file->handle, &pos, SEEK_SET);
01230     if (ISNOTEQUALPOS(pos, expr->onfile)) {
01231         MesPrint("ERROR: something wrong in scratch file [poly_factorize_expression]");
01232         Terminate(-1);
01233     }
01234     file->POposition = expr->onfile;
01235     file->POfull = file->PObuffer;
01236     if (expr->status == HIDDENEXPRESSION)
01237         AR.InHiBuf = 0;
01238     else
01239         AR.InInBuf = 0;
01240 }
01241 else {
01242     file->POfill = (WORD *) ((UBYTE *) (file->PObuffer) + BASEPOSITION(expr->onfile));
01243 }
01244
01245 SetScratch(AR.infile, &(expr->onfile));
01246
01247 // read the first header term
01248 WORD size = GetTerm(BHEAD term);
01249 if (size <= 0) {
01250     MesPrint("ERROR: something wrong with expression [poly_factorize_expression]");
01251     Terminate(-1);
01252 }
01253
01254 // store position: this is where the output will go
01255 pos = expr->onfile;
01256 ADDPOS(pos, size*sizeof(WORD));
01257
01258 // use polynomial as buffer, because it is easy to extend
01259 poly buffer(BHEAD 0);
01260 int bufpos = 0;
01261 int sumcommu = 0;
01262
01263 // read all terms
01264 while (GetTerm(BHEAD term)) {
01265     // substitute non-symbols by extra symbols
01266     sumcommu += DoesCommu(term);
01267     if (sumcommu > 1) {
01268         MesPrint("ERROR: Cannot factorize an expression with more than one noncommuting object");
01269         Terminate(-1);
01270     }
01271     buffer.check_memory(bufpos);
01272     if (LocalConvertToPoly(BHEAD term, buffer.terms + bufpos, startebuf, 0) < 0) {
01273         MesPrint("ERROR: in LocalConvertToPoly [factorize_expression]");
01274         Terminate(-1);
01275     }
01276     bufpos += *(buffer.terms + bufpos);
01277 }
01278 buffer[bufpos] = 0;
01279
01280 // parse the polynomial
01281
01282 AN.poly_num_vars = 0;
01283 poly::get_variables(BHEAD vector<WORD*>(1, buffer.terms), false, true);
01284 poly den(BHEAD 0);
01285 poly a(poly::argument_to_poly(BHEAD buffer.terms, false, true, &den));

```

```

01286
01287 // check for modulus calculus
01288 WORD modp=poly_determine_modulus(BHEAD true, false, "polynomial factorization");
01289 a.setmod(modp,1);
01290
01291 // create output
01292 SetScratch(file, &pos);
01293 NewSort(BHEAD0);
01294
01295 CBUF *C = cbuf+AC.cbufnum;
01296 CBUF *CC = cbuf+AT.ebufnum;
01297
01298 // turn brackets on. We force the existence of a bracket index.
01299 WORD nexpr = expr - Expressions;
01300 AR.BracketOn = 1;
01301 AT.BrackBuf = AM.BracketFactors;
01302 AT.bracketindexflag = 1;
01303 ClearBracketIndex(~nexpr-2); // Clears the index made during primary generation
01304 OpenBracketIndex(nexpr); // Set up a new index
01305
01306 if (a.is_zero()) {
01307     expr->numfactors = 1;
01308 }
01309 else if (a.is_one() && den.is_one()) {
01310     expr->numfactors = 1;
01311
01312     term[0] = 8;
01313     term[1] = SYMBOL;
01314     term[2] = 4;
01315     term[3] = FACTORSYMBOL;
01316     term[4] = 1;
01317     term[5] = 1;
01318     term[6] = 1;
01319     term[7] = 3;
01320
01321     AT.WorkPointer += *term;
01322     Generator(BHEAD term, C->numlhs);
01323     AT.WorkPointer = term;
01324 }
01325 else {
01326     factorized_poly fac;
01327     bool iszero = false;
01328
01329     if (!(expr->vflags & ISFACTORIZED)) {
01330         // factorize the polynomial
01331         fac = polyfact::factorize(a);
01332         poly_fix_minus_signs(fac);
01333     }
01334     else {
01335         int factorsymbol=-1;
01336         for (int i=0; i<AN.poly_num_vars; i++)
01337             if (AN.poly_vars[i] == FACTORSYMBOL)
01338                 factorsymbol = i;
01339
01340         poly denpow(BHEAD 1);
01341
01342         // already factorized, so factorize the factors
01343         for (int i=1; i<=expr->numfactors; i++) {
01344             poly origfac(a.coefficient(factorsymbol, i));
01345             factorized_poly fac2;
01346             if (origfac.is_zero())
01347                 iszero=true;
01348             else {
01349                 fac2 = polyfact::factorize(origfac);
01350                 poly_fix_minus_signs(fac2);
01351                 denpow *= den;
01352             }
01353             for (int j=0; j<(int)fac2.power.size(); j++)
01354                 fac.add_factor(fac2.factor[j], fac2.power[j]);
01355         }
01356
01357         // update denominator, since each factor was scaled
01358         den=denpow;
01359     }
01360
01361     expr->numfactors = 0;
01362
01363     // coefficient
01364     poly num(BHEAD 1);
01365     for (int i=0; i<(int)fac.factor.size(); i++)
01366         if (fac.factor[i].is_integer())
01367             num *= fac.factor[i];
01368
01369     poly gcd(polygcd::integer_gcd(num,den));
01370     den/=gcd;
01371     num/=gcd;
01372

```

```

01373         if (iszero)
01374             expr->numfactors++;
01375
01376         if (!iszero || (expr->vflags & KEEPZERO)) {
01377             if (!num.is_one() || !den.is_one()) {
01378                 expr->numfactors++;
01379
01380                 int n = max(ABS(num[num[1]]), ABS(den[den[1]]));
01381
01382                 term[0] = 6 + 2*n;
01383                 term[1] = SYMBOL;
01384                 term[2] = 4;
01385                 term[3] = FACTORSYMBOL;
01386                 term[4] = expr->numfactors;
01387                 for (int i=0; i<n; i++) {
01388                     term[5+i] = i<ABS(num[num[1]]) ? num[2+AN.poly_num_vars+i] : 0;
01389                     term[5+n+i] = i<ABS(den[den[1]]) ? den[2+AN.poly_num_vars+i] : 0;
01390                 }
01391                 term[5+2*n] = SGN(num[num[1]]) * (2*n+1);
01392                 AT.WorkPointer += *term;
01393                 Generator(BHEAD term, C->numlhs);
01394                 AT.WorkPointer = term;
01395             }
01396
01397             vector<poly> fac_arg(fac.factor.size(), poly(BHEAD 0));
01398
01399             // convert the non-constant factors to Form-style arguments
01400             for (int i=0; i<(int)fac.factor.size(); i++)
01401                 if (!fac.factor[i].is_integer()) {
01402                     buffer.check_memory(fac.factor[i].size_of_form_notation()+1);
01403                     poly::poly_to_argument(fac.factor[i], buffer.terms, false);
01404                     NewSort(BHEAD0);
01405
01406                     for (WORD *t=buffer.terms; *t!=0; t+=*t) {
01407                         // substitute extra symbols
01408                         if (ConvertFromPoly(BHEAD t, term, numxsymbol,
01409                             CC->numrhs-startebuf+numxsymbol,
01410                                     startebuf-numxsymbol, 1) <= 0 ) {
01411                             MesPrint("ERROR: in ConvertFromPoly [factorize_expression]");
01412                             Terminate(-1);
01413                             return(-1);
01414                         }
01415
01416                         // store term
01417                         AT.WorkPointer += *term;
01418                         Generator(BHEAD term, C->numlhs);
01419                         AT.WorkPointer = term;
01420                     }
01421
01422                     // sort and store in buffer
01423                     WORD *buffer;
01424                     if (EndSort(BHEAD (WORD *)((void *)&buffer)), 2) < 0) return -1;
01425
01426                     LONG bufsize=0;
01427                     for (WORD *t=buffer; *t!=0; t+=*t)
01428                         bufsize+=*t;
01429
01430                     fac_arg[i].check_memory(bufsize+ARGHEAD+1);
01431
01432                     for (int j=0; j<ARGHEAD; j++)
01433                         fac_arg[i].terms[j] = 0;
01434                     fac_arg[i].terms[0] = ARGHEAD + bufsize;
01435                     WCOPY(fac_arg[i].terms+ARGHEAD, buffer, bufsize);
01436                     M_free(buffer, "polynomial factorization");
01437                 }
01438
01439             // compare and sort the factors in Form notation
01440             vector<int> order;
01441             vector<vector<int>> comp(fac.factor.size(), vector<int>(fac.factor.size(), 0));
01442
01443             for (int i=0; i<(int)fac.factor.size(); i++)
01444                 if (!fac.factor[i].is_integer()) {
01445                     order.push_back(i);
01446
01447                     for (int j=i+1; j<(int)fac.factor.size(); j++)
01448                         if (!fac.factor[j].is_integer()) {
01449                             comp[i][j] = CompArg(fac_arg[j].terms, fac_arg[i].terms);
01450                             comp[j][i] = -comp[i][j];
01451                         }
01452                 }
01453
01454             for (int i=0; i<(int)order.size(); i++)
01455                 for (int j=0; j+1<(int)order.size(); j++)
01456                     if (comp[order[i]][order[j]] == 1)
01457                         swap(order[i], order[j]);
01458
01459             // create the final expression

```

```

01459         for (int i=0; i<(int)order.size(); i++)
01460             for (int j=0; j<fac.power[order[i]]; j++) {
01461
01462                 expr->numfactors++;
01463
01464                 WORD *tstop = fac_arg[order[i]].terms + *fac_arg[order[i]].terms;
01465                 for (WORD *t=fac_arg[order[i]].terms+ARGHEAD; t<tstop; t+=*t) {
01466
01467                     WCOPY(term+4, t, *t);
01468
01469                     // add special symbol "factor_"
01470                     *term = *(term+4) + 4;
01471                     *(term+1) = SYMBOL;
01472                     *(term+2) = 4;
01473                     *(term+3) = FACTORSYMBOL;
01474                     *(term+4) = expr->numfactors;
01475
01476                     // store term
01477                     AT.WorkPointer += *term;
01478                     Generator(BHEAD term, C->numlhs);
01479                     AT.WorkPointer = term;
01480                 }
01481             }
01482     }
01483 }
01484
01485 // final sorting
01486 if (EndSort(BHEAD NULL,0) < 0) {
01487     LowerSortLevel();
01488     Terminate(-1);
01489 }
01490
01491 // set factorized flag
01492 if (expr->numfactors > 0)
01493     expr->vflags |= ISFACTORIZED;
01494
01495 // clean up
01496 AR.infile = oldinfile;
01497 AR.outfile = oldoutfile;
01498 AR.BracketOn = oldBracketOn;
01499 AT.BrackBuf = oldBrackBuf;
01500 AT.bracketindexflag = oldbracketindexflag;
01501 strcpy((char*)AC.Commercial, oldCommercial);
01502
01503 poly_free_poly_vars(BHEAD "AN.poly_vars_factorize_expression");
01504
01505 return 0;
01506 }
01507
01508 /*
01509 #] poly_factorize_expression :
01510 #[ poly_unfactorize_expression :
01511 */
01512
01513 #if ( SUBEXPSIZE == 5 )
01514 static WORD genericterm[] = {38,1,4,FACTORSYMBOL,0
01515 ,EXPRESSION,15,0,1,0,13,10,8,1,4,FACTORSYMBOL,0,1,1,3
01516 ,EXPRESSION,15,0,1,0,13,10,8,1,4,FACTORSYMBOL,0,1,1,3
01517 ,1,1,3,0};
01518 static WORD genericterm2[] = {23,1,4,FACTORSYMBOL,0
01519 ,EXPRESSION,15,0,1,0,13,10,8,1,4,FACTORSYMBOL,0,1,1,3
01520 ,1,1,3,0};
01521 #endif
01522
01523 int poly_unfactorize_expression(EXPRESSIONS expr)
01524 {
01525     GETIDENTITY;
01526     int i, j, nfac = expr->numfactors, nfacp, nexpr = expr - Expressions;
01527     int expriszero = 0;
01528
01529     FILEHANDLE *oldinfile = AR.infile;
01530     FILEHANDLE *oldoutfile = AR.outfile;
01531     char oldCommercial[COMMERCIALSIZE+2];
01532
01533     WORD *oldworkpointer = AT.WorkPointer;
01534     WORD *term = AT.WorkPointer, *t, *w, size;
01535
01536     FILEHANDLE *file;
01537     POSITION pos, oldpos;
01538
01539     WORD oldBracketOn = AR.BracketOn;
01540     WORD *oldBrackBuf = AT.BrackBuf;
01541     CBUF *C = cbuf+AC.cbunum;
01542
01543     if ( ( expr->vflags & ISFACTORIZED ) == 0 ) return(0);
01544
01545     if ( AT.WorkPointer + AM.MaxTer > AT.WorkTop ) {

```



```

01558         MLOCK(ErrorMessageLock);
01559         MesWork();
01560         MUNLOCK(ErrorMessageLock);
01561         Terminate(-1);
01562     }
01563
01564     oldpos = AS.OldOnFile[nexpr];
01565     AS.OldOnFile[nexpr] = expr->onfile;
01566
01567     strcpy(oldCommercial, (char*)AC.Commercial);
01568     strcpy((char*)AC.Commercial, "unfactorize");
01569 /*
01570     locate the input
01571 */
01572     if ( expr->status == HIDDENEXPRESSION || expr->status == HIDDENLEXPRESSION ||
01573         expr->status == INTOHIDEEXPRESSION || expr->status == INTOHIDELEXPRESSION ) {
01574         AR.InHiBuf = 0; file = AR.hidefile; AR.GetFile = 2;
01575     }
01576     else {
01577         AR.InInBuf = 0; file = AR.outfile; AR.GetFile = 0;
01578     }
01579 /*
01580     read and write to expression file
01581 */
01582     AR.infile = AR.outfile = file;
01583 /*
01584     set the input file to the correct position
01585 */
01586     if ( file->handle >= 0 ) {
01587         pos = expr->onfile;
01588         SeekFile(file->handle, &pos, SEEK_SET);
01589         if (ISNOTEQUALPOS(pos, expr->onfile)) {
01590             MesPrint("ERROR: something wrong in scratch file unfactorize_expression");
01591             Terminate(-1);
01592         }
01593         file->POposition = expr->onfile;
01594         file->POfull = file->PObuffer;
01595         if ( expr->status == HIDDENEXPRESSION )
01596             AR.InHiBuf = 0;
01597         else
01598             AR.InInBuf = 0;
01599     }
01600     else {
01601         file->POfill = (WORD *) ((UBYTE *) (file->PObuffer) + BASEPOSITION(expr->onfile));
01602     }
01603     SetScratch(AR.infile, &(expr->onfile));
01604 /*
01605     Test for whether the first factor is zero.
01606 */
01607     if ( GetFirstBracket(term, nexpr) < 0 ) Terminate(-1);
01608     if ( term[4] != 1 || *term != 8 || term[1] != SYMBOL || term[3] != FACTORSYMBOL || term[4] != 1 )
01609     {
01610         expriszero = 1;
01611     }
01612     SetScratch(AR.infile, &(expr->onfile));
01613 /*
01614     Read the prototype. After this we have the file ready for the output at pos.
01615 */
01616     size = GetTerm(BHEAD term);
01617     if ( size <= 0 ) {
01618         MesPrint ("ERROR: something wrong with expression unfactorize_expression");
01619         Terminate(-1);
01620     }
01621     pos = expr->onfile;
01622     ADDPOS(pos, size*sizeof(WORD));
01623 /*
01624     Set the brackets straight
01625 */
01626     AR.BracketOn = 1;
01627     AT.BrackBuf = AM.BracketFactors;
01628     AT.bracketinfo = 0;
01629     while ( nfac > 2 ) {
01630         nfacp = nfac - nfac%2;
01631 /*
01632     Prepare the bracket index. We have:
01633         e->bracketinfo: the old input bracket index
01634         e->newbracketinfo: the bracket index made for our current input
01635     We need to keep e->bracketinfo in case other workers need it (InParallel)
01636     Hence we work with AT.bracketinfo which takes priority.
01637     Note that in Processor we forced a newbracketinfo to be made.
01638 */
01639     if ( AT.bracketinfo != 0 ) ClearBracketIndex(-1);
01640     AT.bracketinfo = expr->newbracketinfo;
01641     OpenBracketIndex(nexpr);
01642 /*
01643     Now emulate the terms:
01644         sum_(i,0,nfacp,2,factor_^(i/2+1)*F[factor_^(i+1)]*F[factor_^(i+2)])

```

```

01644         +factor_^(nfacp/2+1)*F[factor_^nfac]
01645     */
01646     NewSort(BHEAD0);
01647     if ( expriszero == 0 ) {
01648         for ( i = 0; i < nfacp; i += 2 ) {
01649             t = genericterm; w = term = oldworkpointer;
01650             j = *t; NCOPY(w,t,j);
01651             term[4] = i/2+1;
01652             term[7] = nexpr;
01653             term[16] = i+1;
01654             term[22] = nexpr;
01655             term[31] = i+2;
01656             AT.WorkPointer = term + *term;
01657             Generator(BHEAD term, C->numlhs);
01658         }
01659         if ( nfac > nfacp ) {
01660             t = genericterm2; w = term = oldworkpointer;
01661             j = *t; NCOPY(w,t,j);
01662             term[4] = i/2+1;
01663             term[7] = nexpr;
01664             term[16] = nfac;
01665             AT.WorkPointer = term + *term;
01666             Generator(BHEAD term, C->numlhs);
01667         }
01668     }
01669     if ( EndSort(BHEAD AM.S0->sBuffer,0) < 0 ) {
01670         LowerSortLevel();
01671         Terminate(-1);
01672     }
01673     /*
01674     Set the file back into reading position
01675     */
01676     SetScratch(file, &pos);
01677     nfac = (nfac+1)/2;
01678     if ( expriszero ) { nfac = 1; }
01679 }
01680 if ( AT.bracketinfo != 0 ) ClearBracketIndex(-1);
01681 AT.bracketinfo = expr->newbracketinfo;
01682 expr->newbracketinfo = 0;
01683 /*
01684 Reset the brackets to make them ready for the final pass
01685 */
01686 AR.BracketOn = oldBracketOn;
01687 AT.BrackBuf = oldBrackBuf;
01688 if ( AR.BracketOn ) OpenBracketIndex(nexpr);
01689 /*
01690 We distinguish two cases: nfac == 2 and nfac == 1
01691 After preparing the term we skip the factor_ part.
01692 */
01693 NewSort(BHEAD0);
01694 if ( expriszero == 0 ) {
01695     if ( nfac == 1 ) {
01696         t = genericterm2; w = term = oldworkpointer;
01697         j = *t; NCOPY(w,t,j);
01698         term[7] = nexpr;
01699         term[16] = nfac;
01700     }
01701     else if ( nfac == 2 ) {
01702         t = genericterm; w = term = oldworkpointer;
01703         j = *t; NCOPY(w,t,j);
01704         term[7] = nexpr;
01705         term[16] = 1;
01706         term[22] = nexpr;
01707         term[31] = 2;
01708     }
01709     else {
01710         AS.OldOnFile[nexpr] = oldpos;
01711         return(-1);
01712     }
01713     term[4] = term[0]-4;
01714     term += 4;
01715     AT.WorkPointer = term + *term;
01716     Generator(BHEAD term, C->numlhs);
01717 }
01718 if ( EndSort(BHEAD AM.S0->sBuffer,0) < 0 ) {
01719     LowerSortLevel();
01720     Terminate(-1);
01721 }
01722 /*
01723 Final Cleanup
01724 */
01725 expr->numfactors = 0;
01726 expr->vflags &= ~ISFACTORIZED;
01727 if ( AT.bracketinfo != 0 ) ClearBracketIndex(-1);
01728
01729 AR.infile = oldinfile;
01730 AR.outfile = oldoutfile;

```

```

01731     strcpy((char*)AC.Commercial, oldCommercial);
01732     AT.WorkPointer = oldworkpointer;
01733     AS.OldOnFile[nexpr] = oldpos;
01734
01735     return(0);
01736 }
01737
01738 /*
01739 #] poly_unfactorize_expression :
01740 #[ poly_inverse :
01741 */
01742
01743 WORD *poly_inverse(PHEAD WORD *arga, WORD *argb) {
01744
01745 #ifdef WITHFLINT
01746     if ( AC.FlintPolyFlag && AC.ncmod==0 ) {
01747         return flint_inverse(BHEAD arga, argb);
01748     }
01749 #endif
01750
01751 #ifdef DEBUG
01752     cout << "*** [" << thetime() << "]" << " CALL : poly_inverse" << endl;
01753 #endif
01754
01755     // Extract variables
01756     vector<WORD *> e;
01757     e.reserve(2);
01758     e.push_back(arga);
01759     e.push_back(argb);
01760     poly::get_variables(BHEAD e, false, true);
01761
01762     if (AN.poly_num_vars > 1) {
01763         MLOCK(ErrorMessageLock);
01764         MesPrint ((char*)"ERROR: multivariate polynomial inverse is generally impossible");
01765         MUNLOCK(ErrorMessageLock);
01766         Terminate(-1);
01767     }
01768
01769     poly finalden(BHEAD 1), finalres(BHEAD 1);
01770     int ressize = 0;
01771     WORD *res = NULL;
01772
01773     // Convert to polynomials
01774     poly dena(BHEAD 0); // We need to keep the overall denominator of arga, to multiply the result
01775     poly a(poly::argument_to_poly(BHEAD arga, false, true, &dena));
01776     poly b(poly::argument_to_poly(BHEAD argb, false, true));
01777
01778     // Divide out the integer content, FORM has not already done this.
01779     poly content_a(BHEAD 0), content_b(BHEAD 0);
01780     content_a = polygcd::integer_content(a);
01781     content_b = polygcd::integer_content(b);
01782     a /= content_a;
01783     b /= content_b;
01784
01785     poly invamodp(BHEAD 0), invbmodp(BHEAD 0);
01786     WORD modp = 0;
01787
01788     // Special cases:
01789     // Possibly strange that we give 1 for inverse_(x1,1) but here we take MMA's convention.
01790     if ((a.is_one() && b.is_one()) || a.is_one()) {
01791         finalres = poly(BHEAD 1);
01792     }
01793     else if (b.is_one()) {
01794         finalres = poly(BHEAD 0);
01795     }
01796     else {
01797         // Check for modulus calculus
01798         modp=poly_determine_modulus(BHEAD true, true, "polynomial inverse");
01799         const bool mod_calc = modp==0 ? false : true;
01800         a.setmod(modp,1);
01801         b.setmod(modp,1);
01802
01803         // Check the gcd of a,b: if it is != 1, the inverse does not exist.
01804         poly gcd(polygcd::gcd(a,b));
01805         if (!gcd.is_one()) {
01806             MLOCK(ErrorMessageLock);
01807             if (mod_calc) {
01808                 MesPrint ((char*)"ERROR: polynomial inverse does not exist (mod %d)", modp);
01809             }
01810             else {
01811                 MesPrint ((char*)"ERROR: polynomial inverse does not exist");
01812             }
01813             MUNLOCK(ErrorMessageLock);
01814             Terminate(-1);
01815         }
01816
01817         bool inv_exists = true;

```

```

01818     do {
01819         // If we are not using modulus calculus, find a suitable prime for xgcd:
01820         if (!mod_calc) {
01821             vector<int> x(1,0);
01822             modp = polyfact::choose_prime(a.integer_lcoeff()*b.integer_lcoeff(), x, modp);
01823         }
01824
01825         poly amodp(a,modp,1);
01826         poly bmodp(b,modp,1);
01827
01828         // Calculate gcd
01829         vector<poly> xgcd(polyfact::extended_gcd_Euclidean_lifted(amodp,bmodp));
01830         invamodp = poly(xgcd[0]);
01831         invbmodp = poly(xgcd[1]);
01832
01833         inv_exists = ((invamodp * amodp) % bmodp).is_one();
01834         if (!inv_exists && mod_calc) {
01835             // Control should not reach here!
01836             MLOCK(ErrorMessageLock);
01837             MesPrint ((char*)"ERROR: polynomial inverse does not exist (mod %d) B", modp);
01838             MUNLOCK(ErrorMessageLock);
01839             Terminate(-1);
01840         }
01841
01842         // If the inverse does not exist and we are not working in modulus calculus,
01843         // choose a new prime and try again. This loop should always terminate, as
01844         // we have already checked that gcd(a,b) == 1.
01845     } while (!inv_exists);
01846
01847     // estimate of the size of the Form notation; might be extended later
01848     ressize = invamodp.size_of_form_notation()+1;
01849     res = (WORD *)Malloca(ressize*sizeof(WORD), "poly_inverse");
01850
01851     // initialize polynomials to store the result
01852     poly primepower(BHEAD modp);
01853     poly inva(invamodp,modp,1);
01854     poly invb(invbmop,modp,1);
01855
01856     while (true) {
01857         // convert to Form notation
01858         int j=0;
01859         WORD n=0;
01860         for (int i=1; i<inva[0]; i+=inva[i]) {
01861
01862             // check whether res should be extended
01863             while (ressize < j + 2*ABS(inva[i+inva[i]-1]) + (inva[i+1]>0?4:0) + 3) {
01864                 int newressize = 2*ressize;
01865
01866                 WORD *newres = (WORD *)Malloca(newressize*sizeof(WORD), "poly_inverse");
01867                 WCOPY(newres, res, ressize);
01868                 M_free(res, "poly_inverse");
01869                 res = newres;
01870                 ressize = newressize;
01871             }
01872
01873             res[j] = 1;
01874             if (inva[i+1]>0) {
01875                 res[j+res[j]++] = SYMBOL;
01876                 res[j+res[j]++] = 4;
01877                 res[j+res[j]++] = AN.poly_vars[0];
01878                 res[j+res[j]++] = inva[i+1];
01879             }
01880             MakeLongRational(BHEAD (UWORD *)&inva[i+2], inva[i+inva[i]-1],
01881                             (UWORD*)&primepower.terms[3],
01882                             primepower.terms[primepower.terms[1]],
01883                             (UWORD *)&res[j+res[j]], &n);
01884             res[j] += 2*ABS(n);
01885             res[j+res[j]++] = SGN(n)*(2*ABS(n)+1);
01886             j += res[j];
01887         }
01888         res[j]=0;
01889
01890         // if modulus calculus is set, this is the answer
01891         if (a.modp != 0) break;
01892
01893         // otherwise check over integers
01894         poly den(BHEAD 0);
01895         poly check(poly::argument_to_poly(BHEAD res, false, true, &den));
01896         // Shortcut: if b's lcoeff doesn't divide the lcoeff of check*a,
01897         // b certainly doesn't divide check*a:
01898         if (poly::divides(b.integer_lcoeff(), check.integer_lcoeff()*a.integer_lcoeff())) {
01899             check = check*a - den;
01900             if (poly::divides(b, check)) break;
01901         }
01902
01903         // if incorrect, lift with quadratic p-adic Newton's iteration.
01904         poly error((poly(BHEAD 1) - a*inva - b*invb) / primepower);

```

```

01904         poly errormodpp(error, modp, inva.modn);
01905
01906         inva.modn *= 2;
01907         invb.modn *= 2;
01908
01909         poly dinva((inva * errormodpp) % b);
01910         poly dinvb((invb * errormodpp) % a);
01911
01912         inva += dinva * primepower;
01913         invb += dinvb * primepower;
01914
01915         primepower *= primepower;
01916     }
01917
01918     // One more round trip from form -> poly -> form, to multiply by dena in an easy way
01919     finalres = poly(poly::argument_to_poly(BHEAD res, false, true, &finalden));
01920 }
01921
01922 finalres *= dena;
01923 // The overall denominator additionally needs to be multiplied by content_a:
01924 finalden *= content_a;
01925 const WORD finalden_size = finalden.terms[finalden.terms[1]];
01926 const int finalsize = finalres.size_of_form_notation_with_den(finalden_size)+1;
01927 if (ressize < finalsize) {
01928     if (res != NULL) {
01929         M_free(res, "poly_inverse");
01930     }
01931     res = (WORD *)Malloca(finalsize*sizeof(WORD), "poly_inverse");
01932 }
01933 poly::poly_to_argument_with_den(finalres, finalden_size,
01934     (UWORD*)&(finalden.terms[finalden.terms[1] - ABS(finalden_size)]), res, false);
01935
01936 // clean up and reset modulo calculation
01937 poly_free_poly_vars(BHEAD "AN.poly_vars_inverse");
01938
01939 AN.ncmod = AC.ncmod;
01940 return res;
01941 }
01942
01943 /*
01944 #] poly_inverse :
01945 #[ poly_mul :
01946 */
01947
01948 WORD *poly_mul(PHEAD WORD *a, WORD *b) {
01949
01950 #ifdef DEBUG
01951     cout << "*** [" << thetime() << "]" CALL : poly_mul" << endl;
01952 #endif
01953
01954 #ifdef WITHFLINT
01955     if ( AC.FlintPolyFlag && AC.ncmod==0 ) {
01956         return flint_mul(BHEAD a, b);
01957     }
01958 #endif
01959
01960     // Extract variables
01961     vector<WORD *> e;
01962     e.reserve(2);
01963     e.push_back(a);
01964     e.push_back(b);
01965     poly::get_variables(BHEAD e, false, false); // TODO: any performance effect by sort_vars=true?
01966
01967     // Convert to polynomials
01968     poly dena(BHEAD 0);
01969     poly denb(BHEAD 0);
01970     poly pa(poly::argument_to_poly(BHEAD a, false, true, &dena));
01971     poly pb(poly::argument_to_poly(BHEAD b, false, true, &denb));
01972
01973     // Check for modulus calculus
01974     WORD modp = poly_determine_modulus(BHEAD true, true, "polynomial multiplication");
01975     pa.setmod(modp, 1);
01976
01977     // NOTE: mul_ is currently implemented by translating negative powers of
01978     // symbols to extra symbols. For future improvement, it may be better to
01979     // compute
01980     // (content(a) * content(b)) * (a/content(a)) * (b/content(b))
01981     // to avoid introducing extra symbols for "mixed" cases, e.g.,
01982     // (1+x) * (1/x) -> (1+x) * (1+Z1_).
01983     assert(dena.is_integer());
01984     assert(denb.is_integer());
01985     assert(modp == 0 || dena.is_one());
01986     assert(modp == 0 || denb.is_one());
01987
01988     // multiplication
01989     pa *= pb;
01990

```

```

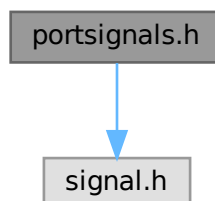
01991 // convert to Form notation
01992 WORD *res;
01993 if (dena.is_one() && denb.is_one()) {
01994     res = (WORD *)Malloca1((pa.size_of_form_notation() + 1) * sizeof(WORD), "poly_mul");
01995     poly::poly_to_argument(pa, res, false);
01996 } else {
01997     dena *= denb;
01998     res = (WORD *)Malloca1((pa.size_of_form_notation_with_den(dena[dena[1]]) + 1) * sizeof(WORD),
01999 "poly_mul");
01999     poly::poly_to_argument_with_den(pa, dena[dena[1]], (const UWORD *)&dena[2+AN.poly_num_vars],
02000 res, false);
02001 }
02002 // clean up and reset modulo calculation
02003 poly_free_poly_vars(BHEAD "AN.poly_vars_mul");
02004 AN.ncmod = AC.ncmod;
02005
02006 return res;
02007 }
02008
02009 /*
02010 #] poly_mul :
02011 #[ poly_free_poly_vars :
02012 */
02013
02014 void poly_free_poly_vars(PHEAD const char *text)
02015 {
02016     if ( AN.poly_vars_type == 0 ) {
02017         TermFree(AN.poly_vars, text);
02018     }
02019     else {
02020         M_free(AN.poly_vars, text);
02021     }
02022     AN.poly_num_vars = 0;
02023     AN.poly_vars = 0;
02024 }
02025
02026 /*
02027 #] poly_free_poly_vars :
02028 */

```

4.111 portsignals.h File Reference

#include <signal.h>

Include dependency graph for portsignals.h:



Macros

- #define [FATAL_SIG_ERROR](#) 4
- #define [NSIG](#) (1024)
- #define [SIGSEGV](#) (NSIG+1)
- #define [SIGFPE](#) (NSIG+2)

- `#define SIGILL (NSIG+3)`
- `#define SIGEMT (NSIG+4)`
- `#define SIGSYS (NSIG+5)`
- `#define SIGPIPE (NSIG+6)`
- `#define SIGLOST (NSIG+7)`
- `#define SIGXCPU (NSIG+8)`
- `#define SIGXFSZ (NSIG+9)`
- `#define SIGTERM (NSIG+10)`
- `#define SIGINT (NSIG+11)`
- `#define SIGQUIT (NSIG+12)`
- `#define SIGHUP (NSIG+13)`
- `#define SIGALRM (NSIG+14)`
- `#define SIGVTALRM (NSIG+15)`
- `#define SIGPROF (NSIG+16)`

4.111.1 Detailed Description

Contains definitions for signals used/intercepted in FORM.

Some systems (especially LINUX) have not enough signals available so some of the (!documented!) signals are not defined. This file contains the definition of all signals used in the program. If the signal is not defined we define it as unused ($>NSIG$).

The include of signal.h must be first, before we try to define undefined signals.

Definition in file [portsignals.h](#).

4.111.2 Macro Definition Documentation

4.111.2.1 FATAL_SIG_ERROR

```
#define FATAL_SIG_ERROR 4
```

Definition at line 47 of file [portsignals.h](#).

4.111.2.2 NSIG

```
#define NSIG (1024)
```

Definition at line 53 of file [portsignals.h](#).

4.111.2.3 SIGSEGV

```
#define SIGSEGV (NSIG+1)
```

Definition at line 57 of file [portsignals.h](#).

4.111.2.4 SIGFPE

```
#define SIGFPE (NSIG+2)
```

Definition at line 60 of file [portsignals.h](#).

4.111.2.5 SIGILL

```
#define SIGILL (NSIG+3)
```

Definition at line 63 of file [portsignals.h](#).

4.111.2.6 SIGEMT

```
#define SIGEMT (NSIG+4)
```

Definition at line 66 of file [portsignals.h](#).

4.111.2.7 SIGSYS

```
#define SIGSYS (NSIG+5)
```

Definition at line 69 of file [portsignals.h](#).

4.111.2.8 SIGPIPE

```
#define SIGPIPE (NSIG+6)
```

Definition at line 72 of file [portsignals.h](#).

4.111.2.9 SIGLOST

```
#define SIGLOST (NSIG+7)
```

Definition at line 75 of file [portsignals.h](#).

4.111.2.10 SIGXCPU

```
#define SIGXCPU (NSIG+8)
```

Definition at line 78 of file [portsignals.h](#).

4.111.2.11 SIGXFSZ

```
#define SIGXFSZ (NSIG+9)
```

Definition at line 81 of file [portsignals.h](#).

4.111.2.12 SIGTERM

```
#define SIGTERM (NSIG+10)
```

Definition at line 84 of file [portsignals.h](#).

4.111.2.13 SIGINT

```
#define SIGINT (NSIG+11)
```

Definition at line 87 of file [portsignals.h](#).

4.111.2.14 SIGQUIT

```
#define SIGQUIT (NSIG+12)
```

Definition at line 90 of file [portsignals.h](#).

4.111.2.15 SIGHUP

```
#define SIGHUP (NSIG+13)
```

Definition at line 93 of file [portsignals.h](#).

4.111.2.16 SIGALRM

```
#define SIGALRM (NSIG+14)
```

Definition at line 96 of file [portsignals.h](#).

4.111.2.17 SIGVTALRM

```
#define SIGVTALRM (NSIG+15)
```

Definition at line 99 of file [portsignals.h](#).

4.111.2.18 SIGPROF

```
#define SIGPROF (NSIG+16)
```

Definition at line 102 of file [portsignals.h](#).

4.112 portsignals.h

[Go to the documentation of this file.](#)

```

00001 #ifndef PORTSIGNAL_H
00002 #define PORTSIGNAL_H
00003
00018 /* #[] License : */
00019 /*
00020  * Copyright (C) 1984-2026 J.A.M. Vermaseren
00021  * When using this file you are requested to refer to the publication
00022  * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00023  * This is considered a matter of courtesy as the development was paid
00024  * for by FOM the Dutch physics granting agency and we would like to
00025  * be able to track its scientific use to convince FOM of its value
00026  * for the community.
00027  *
00028  * This file is part of FORM.
00029  *
00030  * FORM is free software: you can redistribute it and/or modify it under the
00031  * terms of the GNU General Public License as published by the Free Software
00032  * Foundation, either version 3 of the License, or (at your option) any later
00033  * version.
00034  *
00035  * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00036  * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00037  * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00038  * details.
00039  *
00040  * You should have received a copy of the GNU General Public License along
00041  * with FORM. If not, see <http://www.gnu.org/licenses/>.
00042  */
00043 /* #[] License : */
00044
00045 #include <signal.h>
00046
00047 #define FATAL_SIG_ERROR 4
00048
00049 #ifndef NSIG
00050 /*
00051  * The value of NSIG must be enough to fall outside the range of defined signals
00052  */
00053 #define NSIG (1024)
00054 #endif
00055
00056 #ifndef SIGSEGV
00057 #define SIGSEGV (NSIG+1)
00058 #endif
00059 #ifndef SIGFPE
00060 #define SIGFPE (NSIG+2)
00061 #endif
00062 #ifndef SIGILL
00063 #define SIGILL (NSIG+3)
00064 #endif
00065 #ifndef SIGEMT
00066 #define SIGEMT (NSIG+4)
00067 #endif
00068 #ifndef SIGSYS
00069 #define SIGSYS (NSIG+5)
00070 #endif
00071 #ifndef SIGPIPE
00072 #define SIGPIPE (NSIG+6)
00073 #endif
00074 #ifndef SIGLOST
00075 #define SIGLOST (NSIG+7)
00076 #endif
00077 #ifndef SIGXCPU
00078 #define SIGXCPU (NSIG+8)
00079 #endif
00080 #ifndef SIGXFSZ
00081 #define SIGXFSZ (NSIG+9)
00082 #endif
00083 #ifndef SIGTERM
00084 #define SIGTERM (NSIG+10)
00085 #endif
00086 #ifndef SIGINT
00087 #define SIGINT (NSIG+11)
00088 #endif
00089 #ifndef SIGQUIT
00090 #define SIGQUIT (NSIG+12)
00091 #endif
00092 #ifndef SIGHUP
00093 #define SIGHUP (NSIG+13)
00094 #endif
00095 #ifndef SIGALRM
00096 #define SIGALRM (NSIG+14)

```

```

00097 #endif
00098 #ifndef SIGVTALRM
00099 #define SIGVTALRM (NSIG+15)
00100 #endif
00101 #ifndef SIGPROF
00102 #define SIGPROF (NSIG+16)
00103 #endif
00104
00105 #endif

```

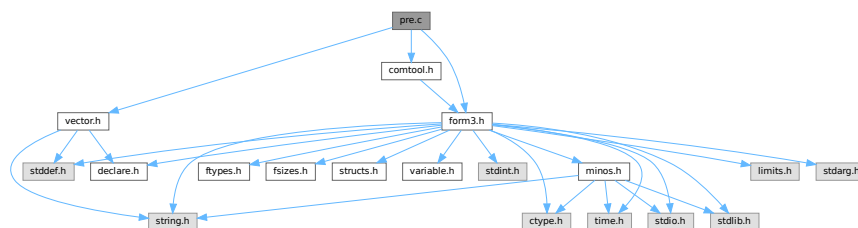
4.113 pre.c File Reference

```

#include "form3.h"
#include "comtool.h"
#include "vector.h"

```

Include dependency graph for pre.c:



Macros

- #define [STRINGIFY](#)(x) [STRINGIFY__](#)(x)
- #define [STRINGIFY__](#)(x) #x
- #define [SKIPBUFSIZE](#) 20
- #define [KILL](#) "kill"
- #define [KILLALL](#) "killall"
- #define [DAEMON](#) "daemon"
- #define [SHELL](#) "shell"
- #define [STDERR](#) "stderr"
- #define [TRUE_EXPR](#) "true"
- #define [FALSE_EXPR](#) "false"
- #define [NOSHELL](#) "noshell"
- #define [TERMINAL](#) "terminal"

Functions

- UBYTE [GetInput](#) (void)
- void [ClearPushback](#) (void)
- UBYTE [GetChar](#) (int level)
- void [CharOut](#) (UBYTE c)
- void [UnsetAllowDelay](#) (void)
- UBYTE * [GetPreVar](#) (UBYTE *name, int flag)
- int [PutPreVar](#) (UBYTE *name, UBYTE *value, UBYTE *args, int mode)
- void [PopPreVars](#) (int tonumber)
- void [IniModule](#) (int type)

- void [IniSpecialModule](#) (int type)
- void [PreProcessor](#) (void)
- int [PreProInstruction](#) (void)
- int [LoadInstruction](#) (int mode)
- int [LoadStatement](#) (int type)
- int [ExpandTripleDots](#) (int par)
- [KEYWORD](#) * [FindKeyWord](#) (UBYTE *theword, [KEYWORD](#) *table, int size)
- [KEYWORD](#) * [FindInKeyWord](#) (UBYTE *theword, [KEYWORD](#) *table, int size)
- int [TheDefine](#) (UBYTE *s, int mode)
- int [DoCommentChar](#) (UBYTE *s)
- int [DoPreAssign](#) (UBYTE *s)
- int [DoDefine](#) (UBYTE *s)
- int [DoRedefine](#) (UBYTE *s)
- int [ClearMacro](#) (UBYTE *name)
- int [TheUndefine](#) (UBYTE *name)
- int [DoUndefine](#) (UBYTE *s)
- int [DoInclude](#) (UBYTE *s)
- int [DoReverseInclude](#) (UBYTE *s)
- int [Include](#) (UBYTE *s, int type)
- int [DoPreExchange](#) (UBYTE *s)
- int [DoCall](#) (UBYTE *s)
- int [DoDebug](#) (UBYTE *s)
- int [DoTerminate](#) (UBYTE *s)
- int [DoContinueDo](#) (UBYTE *s)
- int [DoDo](#) (UBYTE *s)
- int [DoBreakDo](#) (UBYTE *s)
- int [DoElse](#) (UBYTE *s)
- int [DoElseif](#) (UBYTE *s)
- int [DoEnddo](#) (UBYTE *s)
- int [DoEndif](#) (UBYTE *s)
- int [DoEndprocedure](#) (UBYTE *s)
- int [Dolf](#) (UBYTE *s)
- int [Dolfdef](#) (UBYTE *s, int par)
- int [Dolfydef](#) (UBYTE *s)
- int [Dolfndef](#) (UBYTE *s)
- int [DoInside](#) (UBYTE *s)
- int [DoEndInside](#) (UBYTE *s)
- int [DoMessage](#) (UBYTE *s)
- int [DoPipe](#) (UBYTE *s)
- int [DoPrcExtension](#) (UBYTE *s)
- int [DoPreOut](#) (UBYTE *s)
- int [DoPrePrintTimes](#) (UBYTE *s)
- int [DoPreSortReallocate](#) (UBYTE *s)
- int [DoPreAppend](#) (UBYTE *s)
- int [DoPreCreate](#) (UBYTE *s)
- int [DoPreRemove](#) (UBYTE *s)
- int [DoPreClose](#) (UBYTE *s)
- int [DoPreWrite](#) (UBYTE *s)
- int [DoProcedure](#) (UBYTE *s)
- int [DoPreBreak](#) (UBYTE *s)
- int [DoPreCase](#) (UBYTE *s)
- int [DoPreDefault](#) (UBYTE *s)
- int [DoPreEndSwitch](#) (UBYTE *s)
- int [DoPreSwitch](#) (UBYTE *s)
- int [DoPreShow](#) (UBYTE *s)

- int [DoSystem](#) (UBYTE *s)
- int [PreLoad](#) ([PRELOAD](#) *p, UBYTE *start, UBYTE *stop, int mode, char *message)
- int [PreSkip](#) (UBYTE *start, UBYTE *stop, int mode)
- void [StartPrepro](#) (void)
- int [EvalPrelf](#) (UBYTE *s)
- UBYTE * [PrelfEval](#) (UBYTE *s, int *value)
- int [PreCmp](#) (int type, int val, UBYTE *t, int type2, int val2, UBYTE *t2, int cmpop)
- int [PreEq](#) (int type, int val, UBYTE *t, int type2, int val2, UBYTE *t2, int eqop)
- UBYTE * [pParseObject](#) (UBYTE *s, int *type, LONG *val2)
- UBYTE * [PreCalc](#) (void)
- UBYTE * [PreEval](#) (UBYTE *s, LONG *x)
- void [AddToPreTypes](#) (int type)
- void [MessPreNesting](#) (int par)
- int [DoPreAddSeparator](#) (UBYTE *s)
- int [DoPreRmSeparator](#) (UBYTE *s)
- int [DoExternal](#) (UBYTE *s)
- int [DoPrompt](#) (UBYTE *s)
- int [DoSetExternal](#) (UBYTE *s)
- int [DoSetExternalAttr](#) (UBYTE *s)
- int [DoRmExternal](#) (UBYTE *s)
- int [DoFromExternal](#) (UBYTE *s)
- int [DoToExternal](#) (UBYTE *s)
- UBYTE * [defineChannel](#) (UBYTE *s, [HANDLERS](#) *h)
- int [writeToChannel](#) (int wtype, UBYTE *s, [HANDLERS](#) *h)
- int [DoFactDollar](#) (UBYTE *s)
- WORD [GetDollarNumber](#) (UBYTE **inp, [DOLLARS](#) d)
- int [DoSetRandom](#) (UBYTE *s)
- int [DoOptimize](#) (UBYTE *s)
- int [DoClearOptimize](#) (UBYTE *s)
- int [DoSkipExtraSymbols](#) (UBYTE *s)
- int [DoPreReset](#) (UBYTE *s)
- int [DoPreAppendPath](#) (UBYTE *s)
- int [DoPrePrependPath](#) (UBYTE *s)
- int [DoTimeOutAfter](#) (UBYTE *s)
- int [DoNamespace](#) (UBYTE *s)
- int [DoEndNamespace](#) (UBYTE *s)
- UBYTE * [SkipName](#) (UBYTE *s)
- UBYTE * [ConstructName](#) (UBYTE *s, UBYTE type)
- int [DoUse](#) (UBYTE *s)
- int [UserFlags](#) (UBYTE *s, int par)
- int [DoClearUserFlag](#) (UBYTE *s)
- int [DoSetUserFlag](#) (UBYTE *s)

4.113.1 Detailed Description

This is the preprocessor and all its routines.

Definition in file [pre.c](#).

4.113.2 Macro Definition Documentation

4.113.2.1 STRINGIFY

```
#define STRINGIFY(  
    x ) STRINGIFY__(x)
```

Definition at line 4329 of file [pre.c](#).

4.113.2.2 STRINGIFY__

```
#define STRINGIFY__(  
    x ) #x
```

Definition at line 4330 of file [pre.c](#).

4.113.2.3 SKIPBUFSIZE

```
#define SKIPBUFSIZE 20
```

Definition at line 4453 of file [pre.c](#).

4.113.2.4 KILL

```
#define KILL "kill"
```

Definition at line 5673 of file [pre.c](#).

4.113.2.5 KILLALL

```
#define KILLALL "killall"
```

Definition at line 5674 of file [pre.c](#).

4.113.2.6 DAEMON

```
#define DAEMON "daemon"
```

Definition at line 5675 of file [pre.c](#).

4.113.2.7 SHELL

```
#define SHELL "shell"
```

Definition at line 5676 of file [pre.c](#).

4.113.2.8 STDERR

```
#define STDERR "stderr"
```

Definition at line 5677 of file [pre.c](#).

4.113.2.9 TRUE_EXPR

```
#define TRUE_EXPR "true"
```

Definition at line 5679 of file [pre.c](#).

4.113.2.10 FALSE_EXPR

```
#define FALSE_EXPR "false"
```

Definition at line 5680 of file [pre.c](#).

4.113.2.11 NOSHELL

```
#define NOSHELL "noshell"
```

Definition at line 5681 of file [pre.c](#).

4.113.2.12 TERMINAL

```
#define TERMINAL "terminal"
```

Definition at line 5682 of file [pre.c](#).

4.113.3 Function Documentation

4.113.3.1 GetInput()

```
UBYTE GetInput (  
    void )
```

Definition at line 138 of file [pre.c](#).

4.113.3.2 ClearPushback()

```
void ClearPushback (  
    void )
```

Definition at line 167 of file [pre.c](#).

4.113.3.3 GetChar()

```
UBYTE GetChar (
    int level )
```

Definition at line 185 of file [pre.c](#).

4.113.3.4 CharOut()

```
void CharOut (
    UBYTE c )
```

Definition at line 495 of file [pre.c](#).

4.113.3.5 UnsetAllowDelay()

```
void UnsetAllowDelay (
    void )
```

Definition at line 516 of file [pre.c](#).

4.113.3.6 GetPreVar()

```
UBYTE * GetPreVar (
    UBYTE * name,
    int flag )
```

Definition at line 542 of file [pre.c](#).

4.113.3.7 PutPreVar()

```
int PutPreVar (
    UBYTE * name,
    UBYTE * value,
    UBYTE * args,
    int mode )
```

Inserts/Updates a preprocessor variable in the name administration.

Parameters

<i>name</i>	Character string with the variable name.
<i>value</i>	Character string with a possible value. Special case: if this argument is zero, then we have no value. Note: This is different from having an empty argument! This should only occur when the name starts with a ?
<i>args</i>	Character string with possible arguments.
<i>mode</i>	=0: always create a new name entry, =1: try to do a redefinition if possible.

Returns

Index of used entry in name list.

Definition at line 724 of file [pre.c](#).

Referenced by [ClearOptimize\(\)](#), [Generator\(\)](#), [Optimize\(\)](#), [PF_BroadcastRedefinedPreVars\(\)](#), [StartVariables\(\)](#), and [TheDefine\(\)](#).

4.113.3.8 PopPreVars()

```
void PopPreVars (
    int tonumber )
```

Definition at line 826 of file [pre.c](#).

4.113.3.9 IniModule()

```
void IniModule (
    int type )
```

Definition at line 841 of file [pre.c](#).

4.113.3.10 IniSpecialModule()

```
void IniSpecialModule (
    int type )
```

Definition at line 943 of file [pre.c](#).

4.113.3.11 PreProcessor()

```
void PreProcessor (
    void )
```

Definition at line 953 of file [pre.c](#).

4.113.3.12 PreProInstruction()

```
int PreProInstruction (
    void )
```

Definition at line 1172 of file [pre.c](#).

4.113.3.13 LoadInstruction()

```
int LoadInstruction (
    int mode )
```

Definition at line 1260 of file [pre.c](#).

4.113.3.14 LoadStatement()

```
int LoadStatement (
    int type )
```

Definition at line 1463 of file [pre.c](#).

4.113.3.15 ExpandTripleDots()

```
int ExpandTripleDots (
    int par )
```

Definition at line 1601 of file [pre.c](#).

4.113.3.16 FindKeyWord()

```
KEYWORD * FindKeyWord (
    UBYTE * theword,
    KEYWORD * table,
    int size )
```

Definition at line 1969 of file [pre.c](#).

4.113.3.17 FindInKeyWord()

```
KEYWORD * FindInKeyWord (
    UBYTE * theword,
    KEYWORD * table,
    int size )
```

Definition at line 2000 of file [pre.c](#).

4.113.3.18 TheDefine()

```
int TheDefine (
    UBYTE * s,
    int mode )
```

Preprocessor assignment. Possible arguments and values are treated and the new preprocessor variable is put into the name administration.

Parameters

<i>s</i>	Pointer to the character string following the preprocessor command.
<i>mode</i>	Bitmask. 0-bit clear: always create a new name entry, 0-bit set: try to redefine an existing name, 1-bit set: ignore preprocessor if/switch status.

Returns

zero: no errors, negative number: errors.

Definition at line 2030 of file [pre.c](#).

References [PutPreVar\(\)](#).

Here is the call graph for this function:

**4.113.3.19 DoCommentChar()**

```
int DoCommentChar (
    UBYTE * s )
```

Definition at line 2103 of file [pre.c](#).

4.113.3.20 DoPreAssign()

```
int DoPreAssign (
    UBYTE * s )
```

Definition at line 2134 of file [pre.c](#).

4.113.3.21 DoDefine()

```
int DoDefine (
    UBYTE * s )
```

Definition at line 2165 of file [pre.c](#).

4.113.3.22 DoRedefine()

```
int DoRedefine (
    UBYTE * s )
```

Definition at line 2175 of file [pre.c](#).

4.113.3.23 ClearMacro()

```
int ClearMacro (
    UBYTE * name )
```

Definition at line 2187 of file [pre.c](#).

4.113.3.24 TheUndefine()

```
int TheUndefine (
    UBYTE * name )
```

Definition at line 2215 of file [pre.c](#).

4.113.3.25 DoUndefine()

```
int DoUndefine (
    UBYTE * s )
```

Definition at line 2269 of file [pre.c](#).

4.113.3.26 DoInclude()

```
int DoInclude (
    UBYTE * s )
```

Definition at line 2321 of file [pre.c](#).

4.113.3.27 DoReverseInclude()

```
int DoReverseInclude (
    UBYTE * s )
```

Definition at line 2328 of file [pre.c](#).

4.113.3.28 Include()

```
int Include (
    UBYTE * s,
    int type )
```

Definition at line 2335 of file [pre.c](#).

4.113.3.29 DoPreExchange()

```
int DoPreExchange (
    UBYTE * s )
```

Definition at line 2496 of file [pre.c](#).

4.113.3.30 DoCall()

```
int DoCall (
    UBYTE * s )
```

Definition at line 2557 of file [pre.c](#).

4.113.3.31 DoDebug()

```
int DoDebug (
    UBYTE * s )
```

Definition at line 2729 of file [pre.c](#).

4.113.3.32 DoTerminate()

```
int DoTerminate (
    UBYTE * s )
```

Definition at line 2766 of file [pre.c](#).

4.113.3.33 DoContinueDo()

```
int DoContinueDo (
    UBYTE * s )
```

Jumps forward to the corresponding `#enddo` of the specified number of outer `#do` loops.

Syntax:

```
#continuedo [<number>=1]
```

If `number` is omitted, it defaults to 1. If `number` is zero then the instruction has no effect.

Definition at line 2800 of file [pre.c](#).

4.113.3.34 DoDo()

```
int DoDo (
    UBYTE * s )
```

Definition at line 2864 of file [pre.c](#).

4.113.3.35 DoBreakDo()

```
int DoBreakDo (
    UBYTE * s )
```

Definition at line 3079 of file [pre.c](#).

4.113.3.36 DoElse()

```
int DoElse (
    UBYTE * s )
```

Definition at line 3157 of file [pre.c](#).

4.113.3.37 DoElseif()

```
int DoElseif (
    UBYTE * s )
```

Definition at line 3194 of file [pre.c](#).

4.113.3.38 DoEnddo()

```
int DoEnddo (
    UBYTE * s )
```

Definition at line 3229 of file [pre.c](#).

4.113.3.39 DoEndif()

```
int DoEndif (
    UBYTE * s )
```

Definition at line 3375 of file [pre.c](#).

4.113.3.40 DoEndprocedure()

```
int DoEndprocedure (
    UBYTE * s )
```

Definition at line 3403 of file [pre.c](#).

4.113.3.41 Dolf()

```
int DoIf (
    UBYTE * s )
```

Definition at line 3430 of file [pre.c](#).

4.113.3.42 Dolfdef()

```
int DoIfdef (
    UBYTE * s,
    int par )
```

Definition at line 3454 of file [pre.c](#).

4.113.3.43 DoIfydef()

```
int DoIfydef (  
    UBYTE * s )
```

Definition at line [3479](#) of file [pre.c](#).

4.113.3.44 DoIfndef()

```
int DoIfndef (  
    UBYTE * s )
```

Definition at line [3489](#) of file [pre.c](#).

4.113.3.45 DoInside()

```
int DoInside (  
    UBYTE * s )
```

Definition at line [3514](#) of file [pre.c](#).

4.113.3.46 DoEndInside()

```
int DoEndInside (  
    UBYTE * s )
```

Definition at line [3607](#) of file [pre.c](#).

4.113.3.47 DoMessage()

```
int DoMessage (  
    UBYTE * s )
```

Definition at line [3739](#) of file [pre.c](#).

4.113.3.48 DoPipe()

```
int DoPipe (  
    UBYTE * s )
```

Definition at line [3753](#) of file [pre.c](#).

4.113.3.49 DoPrExtension()

```
int DoPrExtension (  
    UBYTE * s )
```

Definition at line [3776](#) of file [pre.c](#).

4.113.3.50 DoPreOut()

```
int DoPreOut (
    UBYTE * s )
```

Definition at line [3810](#) of file [pre.c](#).

4.113.3.51 DoPrePrintTimes()

```
int DoPrePrintTimes (
    UBYTE * s )
```

Definition at line [3833](#) of file [pre.c](#).

4.113.3.52 DoPreSortReallocate()

```
int DoPreSortReallocate (
    UBYTE * s )
```

Definition at line [3847](#) of file [pre.c](#).

4.113.3.53 DoPreAppend()

```
int DoPreAppend (
    UBYTE * s )
```

Definition at line [3867](#) of file [pre.c](#).

4.113.3.54 DoPreCreate()

```
int DoPreCreate (
    UBYTE * s )
```

Definition at line [3910](#) of file [pre.c](#).

4.113.3.55 DoPreRemove()

```
int DoPreRemove (
    UBYTE * s )
```

Definition at line [3950](#) of file [pre.c](#).

4.113.3.56 DoPreClose()

```
int DoPreClose (
    UBYTE * s )
```

Definition at line [3983](#) of file [pre.c](#).

4.113.3.57 DoPreWrite()

```
int DoPreWrite (
    UBYTE * s )
```

Definition at line [4027](#) of file [pre.c](#).

4.113.3.58 DoProcedure()

```
int DoProcedure (
    UBYTE * s )
```

Definition at line [4068](#) of file [pre.c](#).

4.113.3.59 DoPreBreak()

```
int DoPreBreak (
    UBYTE * s )
```

Definition at line [4125](#) of file [pre.c](#).

4.113.3.60 DoPreCase()

```
int DoPreCase (
    UBYTE * s )
```

Definition at line [4149](#) of file [pre.c](#).

4.113.3.61 DoPreDefault()

```
int DoPreDefault (
    UBYTE * s )
```

Definition at line [4188](#) of file [pre.c](#).

4.113.3.62 DoPreEndSwitch()

```
int DoPreEndSwitch (
    UBYTE * s )
```

Definition at line [4212](#) of file [pre.c](#).

4.113.3.63 DoPreSwitch()

```
int DoPreSwitch (
    UBYTE * s )
```

Definition at line [4239](#) of file [pre.c](#).

4.113.3.64 DoPreShow()

```
int DoPreShow (
    UBYTE * s )
```

Definition at line [4293](#) of file [pre.c](#).

4.113.3.65 DoSystem()

```
int DoSystem (
    UBYTE * s )
```

Definition at line [4332](#) of file [pre.c](#).

4.113.3.66 PreLoad()

```
int PreLoad (
    PRELOAD * p,
    UBYTE * start,
    UBYTE * stop,
    int mode,
    char * message )
```

Definition at line [4372](#) of file [pre.c](#).

4.113.3.67 PreSkip()

```
int PreSkip (
    UBYTE * start,
    UBYTE * stop,
    int mode )
```

Definition at line [4455](#) of file [pre.c](#).

4.113.3.68 StartPrepro()

```
void StartPrepro (
    void )
```

Definition at line [4510](#) of file [pre.c](#).

4.113.3.69 EvalPrelf()

```
int EvalPreIf (
    UBYTE * s )
```

Definition at line [4538](#) of file [pre.c](#).

4.113.3.70 PreIfEval()

```
UBYTE * PreIfEval (
    UBYTE * s,
    int * value )
```

Definition at line [4573](#) of file [pre.c](#).

4.113.3.71 PreCmp()

```
int PreCmp (
    int type,
    int val,
    UBYTE * t,
    int type2,
    int val2,
    UBYTE * t2,
    int cmpop )
```

Definition at line [4702](#) of file [pre.c](#).

4.113.3.72 PreEq()

```
int PreEq (
    int type,
    int val,
    UBYTE * t,
    int type2,
    int val2,
    UBYTE * t2,
    int eqop )
```

Definition at line [4726](#) of file [pre.c](#).

4.113.3.73 pParseObject()

```
UBYTE * pParseObject (
    UBYTE * s,
    int * type,
    LONG * val2 )
```

Definition at line [4755](#) of file [pre.c](#).

4.113.3.74 PreCalc()

```
UBYTE * PreCalc (
    void )
```

Definition at line [5191](#) of file [pre.c](#).

4.113.3.75 PreEval()

```
UBYTE * PreEval (  
    UBYTE * s,  
    LONG * x )
```

Definition at line 5269 of file [pre.c](#).

4.113.3.76 AddToPreTypes()

```
void AddToPreTypes (  
    int type )
```

Definition at line 5419 of file [pre.c](#).

4.113.3.77 MessPreNesting()

```
void MessPreNesting (  
    int par )
```

Definition at line 5437 of file [pre.c](#).

4.113.3.78 DoPreAddSeparator()

```
int DoPreAddSeparator (  
    UBYTE * s )
```

Definition at line 5460 of file [pre.c](#).

4.113.3.79 DoPreRmSeparator()

```
int DoPreRmSeparator (  
    UBYTE * s )
```

Definition at line 5485 of file [pre.c](#).

4.113.3.80 DoExternal()

```
int DoExternal (  
    UBYTE * s )
```

Definition at line 5502 of file [pre.c](#).

4.113.3.81 DoPrompt()

```
int DoPrompt (  
    UBYTE * s )
```

Definition at line 5584 of file [pre.c](#).

4.113.3.82 DoSetExternal()

```
int DoSetExternal (
    UBYTE * s )
```

Definition at line 5617 of file [pre.c](#).

4.113.3.83 DoSetExternalAttr()

```
int DoSetExternalAttr (
    UBYTE * s )
```

Definition at line 5687 of file [pre.c](#).

4.113.3.84 DoRmExternal()

```
int DoRmExternal (
    UBYTE * s )
```

Definition at line 5804 of file [pre.c](#).

4.113.3.85 DoFromExternal()

```
int DoFromExternal (
    UBYTE * s )
```

Definition at line 5869 of file [pre.c](#).

4.113.3.86 DoToExternal()

```
int DoToExternal (
    UBYTE * s )
```

Definition at line 6064 of file [pre.c](#).

4.113.3.87 defineChannel()

```
UBYTE * defineChannel (
    UBYTE * s,
    HANDLERS * h )
```

Definition at line 6111 of file [pre.c](#).

4.113.3.88 writeToChannel()

```
int writeToChannel (
    int wtype,
    UBYTE * s,
    HANDLERS * h )
```

Definition at line 6146 of file [pre.c](#).

4.113.3.89 DoFactDollar()

```
int DoFactDollar (
    UBYTE * s )
```

Definition at line 6711 of file [pre.c](#).

4.113.3.90 GetDollarNumber()

```
WORD GetDollarNumber (
    UBYTE ** inp,
    DOLLARS d )
```

Definition at line 6748 of file [pre.c](#).

4.113.3.91 DoSetRandom()

```
int DoSetRandom (
    UBYTE * s )
```

Definition at line 6838 of file [pre.c](#).

4.113.3.92 DoOptimize()

```
int DoOptimize (
    UBYTE * s )
```

Definition at line 6881 of file [pre.c](#).

4.113.3.93 DoClearOptimize()

```
int DoClearOptimize (
    UBYTE * s )
```

Definition at line 7014 of file [pre.c](#).

4.113.3.94 DoSkipExtraSymbols()

```
int DoSkipExtraSymbols (
    UBYTE * s )
```

Definition at line 7034 of file [pre.c](#).

4.113.3.95 DoPreReset()

```
int DoPreReset (
    UBYTE * s )
```

Definition at line 7073 of file [pre.c](#).

4.113.3.96 DoPreAppendPath()

```
int DoPreAppendPath (
    UBYTE * s )
```

Appends the given path (absolute or relative to the current file directory) to the FORM path.

Syntax: #appendpath <path>

Definition at line 7231 of file [pre.c](#).

4.113.3.97 DoPrePrependPath()

```
int DoPrePrependPath (
    UBYTE * s )
```

Prepends the given path (absolute or relative to the current file directory) to the FORM path.

Syntax: #prependpath <path>

Definition at line 7248 of file [pre.c](#).

4.113.3.98 DoTimeOutAfter()

```
int DoTimeOutAfter (
    UBYTE * s )
```

Definition at line 7260 of file [pre.c](#).

4.113.3.99 DoNamespace()

```
int DoNamespace (
    UBYTE * s )
```

Definition at line 7312 of file [pre.c](#).

4.113.3.100 DoEndNamespace()

```
int DoEndNamespace (
    UBYTE * s )
```

Definition at line 7360 of file [pre.c](#).

4.113.3.101 SkipName()

```
UBYTE * SkipName (
    UBYTE * s )
```

Definition at line 7383 of file [pre.c](#).

4.113.3.102 ConstructName()

```
UBYTE * ConstructName (  
    UBYTE * s,  
    UBYTE type )
```

Definition at line [7483](#) of file [pre.c](#).

4.113.3.103 DoUse()

```
int DoUse (  
    UBYTE * s )
```

Definition at line [7575](#) of file [pre.c](#).

4.113.3.104 UserFlags()

```
int UserFlags (  
    UBYTE * s,  
    int par )
```

Definition at line [7623](#) of file [pre.c](#).

4.113.3.105 DoClearUserFlag()

```
int DoClearUserFlag (  
    UBYTE * s )
```

Definition at line [7730](#) of file [pre.c](#).

4.113.3.106 DoSetUserFlag()

```
int DoSetUserFlag (  
    UBYTE * s )
```

Definition at line [7740](#) of file [pre.c](#).

4.114 pre.c

[Go to the documentation of this file.](#)

```

00001
00005 /* #[] License : */
00006 /*
00007 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00008 *   When using this file you are requested to refer to the publication
00009 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010 *   This is considered a matter of courtesy as the development was paid
00011 *   for by FOM the Dutch physics granting agency and we would like to
00012 *   be able to track its scientific use to convince FOM of its value
00013 *   for the community.
00014 *
00015 *   This file is part of FORM.
00016 *
00017 *   FORM is free software: you can redistribute it and/or modify it under the
00018 *   terms of the GNU General Public License as published by the Free Software
00019 *   Foundation, either version 3 of the License, or (at your option) any later
00020 *   version.
00021 *
00022 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00025 *   details.
00026 *
00027 *   You should have received a copy of the GNU General Public License along
00028 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00029 */
00030 /* #] License : */
00031 /*
00032    #[] Includes :
00033 */
00034 #include "form3.h"
00035 #include "comtool.h"
00036 #ifdef WITHFLOAT
00037 #include "math.h"
00038 #endif
00039
00040 static UBYTE pushbackchar = 0;
00041 static int oldmode = 0;
00042 static int stopdelay = 0;
00043 static STREAM *oldstream = 0;
00044 static UBYTE underscore[2] = {'_', 0};
00045 static PREVAR *ThePreVar = 0;
00046
00047 static int ExitDoLoops(int, const char *);
00048
00049 static KEYWORD precommands[] = {
00050     {"add", DoPreAdd, 0, 0}
00051     , {"addseparator", DoPreAddSeparator, 0, 0}
00052     , {"append", DoPreAppend, 0, 0}
00053     , {"appendpath", DoPreAppendPath, 0, 0}
00054     , {"assign", DoPreAssign, 0, 0}
00055     , {"break", DoPreBreak, 0, 0}
00056     , {"breakdo", DoBreakDo, 0, 0}
00057     , {"call", DoCall, 0, 0}
00058     , {"case", DoPreCase, 0, 0}
00059     , {"clearflag", DoClearUserFlag, 0, 0}
00060     , {"clearoptimize", DoClearOptimize, 0, 0}
00061     , {"close", DoPreClose, 0, 0}
00062     , {"closedictionary", DoPreCloseDictionary, 0, 0}
00063     , {"commentchar", DoCommentChar, 0, 0}
00064     , {"continuedo", DoContinueDo, 0, 0}
00065     , {"create", DoPreCreate, 0, 0}
00066     , {"debug", DoDebug, 0, 0}
00067     , {"default", DoPreDefault, 0, 0}
00068     , {"define", DoDefine, 0, 0}
00069     , {"do", DoDo, 0, 0}
00070     , {"else", DoElse, 0, 0}
00071     , {"elseif", DoElseif, 0, 0}
00072     , {"enddo", DoEnddo, 0, 0}
00073     , {"endfloat", DoEndFloat, 0, 0}
00074     , {"endif", DoEndif, 0, 0}
00075     , {"endinside", DoEndInside, 0, 0}
00076     , {"endnamespace", DoEndNamespace, 0, 0}
00077     , {"endprocedure", DoEndprocedure, 0, 0}
00078     , {"endswitch", DoPreEndSwitch, 0, 0}
00079     , {"exchange", DoPreExchange, 0, 0}
00080     , {"external", DoExternal, 0, 0}
00081     , {"factdollar", DoFactDollar, 0, 0}
00082     , {"fromexternal", DoFromExternal, 0, 0}
00083     , {"if", DoIf, 0, 0}

```

```

00086     ,{"ifdef"           , DoIfydef           , 0, 0}
00087     ,{"ifndef"          , DoIfndef          , 0, 0}
00088     ,{"include"          , DoInclude          , 0, 0}
00089     ,{"inside"           , DoInside           , 0, 0}
00090     ,{"message"          , DoMessage          , 0, 0}
00091     ,{"namespace"        , DoNamespace        , 0, 0}
00092     ,{"opendictionary"   , DoPreOpenDictionary, 0, 0}
00093     ,{"optimize"         , DoOptimize         , 0, 0}
00094     ,{"pipe"             , DoPipe             , 0, 0}
00095     ,{"preout"           , DoPreOut           , 0, 0}
00096     ,{"prependpath"      , DoPrePrependPath   , 0, 0}
00097     ,{"printtimes"       , DoPrePrintTimes    , 0, 0}
00098     ,{"procedure"        , DoProcedure         , 0, 0}
00099     ,{"procedureextension", DoPrceExtension    , 0, 0}
00100     ,{"prompt"           , DoPrompt           , 0, 0}
00101     ,{"redefine"         , DoRedefine         , 0, 0}
00102     ,{"remove"           , DoPreRemove        , 0, 0}
00103     ,{"reset"            , DoPreReset         , 0, 0}
00104     ,{"reverseinclude"   , DoReverseInclude   , 0, 0}
00105     ,{"rmexternal"       , DoRmExternal       , 0, 0}
00106     ,{"rmseparator"      , DoPreRmSeparator   , 0, 0}
00107     ,{"setexternal"      , DoSetExternal      , 0, 0}
00108     ,{"setexternalattr"  , DoSetExternalAttr  , 0, 0}
00109     ,{"setflag"          , DoSetUserFlag      , 0, 0}
00110     ,{"setrandom"        , DoSetRandom        , 0, 0}
00111     ,{"show"             , DoPreShow          , 0, 0}
00112     ,{"skipextrasymbols" , DoSkipExtraSymbols , 0, 0}
00113     ,{"sortreallocate"   , DoPreSortReallocate, 0, 0}
00114     #ifdef WITHFLOAT
00115     ,{"startfloat"       , DoStartFloat       , 0, 0}
00116     #endif
00117     ,{"switch"           , DoPreSwitch        , 0, 0}
00118     ,{"system"           , DoSystem           , 0, 0}
00119     ,{"terminate"        , DoTerminate        , 0, 0}
00120     ,{"timeoutafter"     , DoTimeOutAfter     , 0, 0}
00121     ,{"toexternal"       , DoToExternal       , 0, 0}
00122     ,{"undefine"         , DoUndefine         , 0, 0}
00123     ,{"use"              , DoUse              , 0, 0}
00124     ,{"usedictionary"    , DoPreUseDictionary, 0, 0}
00125     ,{"write"            , DoPreWrite         , 0, 0}
00126 };
00127
00128 /*
00129     #] Includes :
00130     # [ PreProcessor :
00131     #[ GetInput :
00132
00133         Gets one input character. If we reach the end of a stream
00134         we pop to the previous stream and try again.
00135         If there are no more streams we let this be known.
00136     */
00137
00138 UBYTE GetInput(void)
00139 {
00140     UBYTE c;
00141     while ( AC.CurrentStream ) {
00142         c = GetFromStream(AC.CurrentStream);
00143         if ( c != ENDOFSTREAM ) {
00144             #ifdef WITHMPI
00145                 if ( PF.me == MASTER
00146                     && AC.NoShowInput <= 0
00147                     && AC.CurrentStream->type != PREVARSTREAM )
00148             #else
00149                 if ( AC.NoShowInput <= 0 && AC.CurrentStream->type != PREVARSTREAM )
00150             #endif
00151                 CharOut(c);
00152                 return(c);
00153             }
00154             AC.CurrentStream = CloseStream(AC.CurrentStream);
00155             if ( stopdelay && AC.CurrentStream == oldstream ) {
00156                 stopdelay = 0; AP.AllowDelay = 1;
00157             }
00158         }
00159         return(ENDOFINPUT);
00160     }
00161
00162     /*
00163         #] GetInput :
00164         #[ ClearPushback :
00165     */
00166
00167 void ClearPushback(void)
00168 {
00169     pushbackchar = 0;
00170 }
00171
00172 /*

```

```

00173      #] ClearPushback :
00174      #[ GetChar :
00175
00176      Reads one character. If it encounters a quote it immediately
00177      takes the whole preprocessor variable and opens a stream
00178      for it and starts reading the stream.
00179      Note that we have to take special precautions for escaped quotes.
00180      That is why we remember the previous character. We allow the
00181      (dubious?) construction of ending a stream with a backslash and
00182      then using it to escape an object in the parent stream.
00183  */
00184
00185  UBYTE GetChar(int level)
00186  {
00187      UBYTE namebuf[MAXPRENAMESIZE+2], c, *s, *t;
00188      static UBYTE lastchar, charinbuf = 0;
00189      int i, j, raiselow, olddelay;
00190      STREAM *stream;
00191      if ( level > 0 ) {
00192          lastchar = ' ';
00193          goto higherlevel;
00194      }
00195      if ( pushbackchar ) { c = pushbackchar; pushbackchar = 0; return(c); }
00196      if ( charinbuf ) { c = charinbuf; charinbuf = 0; return(c); }
00197      c = GetInput();
00198      for(;;) {
00199          if ( c == '\\\\' ) {
00200              charinbuf = GetInput();
00201              if ( charinbuf != LINEFEED ) {
00202                  pushbackchar = charinbuf;
00203                  charinbuf = 0;
00204                  break;
00205              }
00206              charinbuf = 0; /* Escaped linefeed -> skip leading blanks */
00207              while ( ( c = GetInput() ) == ' ' || c == '\t' ) {}
00208          }
00209          else if ( c == '\"' || c == '\'' ) {
00210              if ( AP.DelayPrevar == 1 && c == '\"' ) {
00211                  AP.DelayPrevar = 0;
00212                  break;
00213              }
00214              lastchar = c;
00215              higherlevel:
00216              c = GetInput();
00217              if ( c == '!' && lastchar == '\'' ) {
00218                  if ( stopdelay == 0 ) oldstream = AC.CurrentStream;
00219                  AP.AllowDelay = 0;
00220                  stopdelay = 1;
00221                  c = GetInput();
00222              }
00223              if ( c == '~' && lastchar == '\'' ) {
00224                  if ( AP.AllowDelay ) {
00225                      pushbackchar = c;
00226                      c = lastchar;
00227                      AP.DelayPrevar = 1;
00228                      break;
00229                  }
00230              }
00231              else {
00232                  pushbackchar = c;
00233              }
00234              olddelay = AP.DelayPrevar;
00235              AP.DelayPrevar = 0;
00236              i = 0; lastchar = 0;
00237              for(;;) {
00238                  if ( pushbackchar ) { c = pushbackchar; pushbackchar = 0; }
00239                  else { c = GetInput(); }
00240                  if ( c == ENDOFINPUT || ( ( c == '\"' || c == LINEFEED )
00241                      && lastchar != '\\\\' ) ) {
00242                      break;
00243                  }
00244                  if ( c == '{' ) { /* Try the preprocessor calculator */
00245                      if ( PreCalc() == 0 ) Terminate(-1);
00246                      c = GetInput(); /* This is either a { or a number */
00247                      if ( c == '{' ) {
00248                          MesPrint("@Illegal set inside preprocessor variable name");
00249                          Terminate(-1);
00250                      }
00251                  }
00252                  if ( c == '~' && lastchar != '\\\\' ) {
00253                      c = GetChar(1);
00254                      if ( c == ENDOFINPUT || ( ( c == '\"' || c == LINEFEED )
00255                          && lastchar != '\\\\' ) ) {
00256                          break;
00257                      }
00258                  }
00259                  if ( lastchar == '\\\\' ) { i--; lastchar = 0; }

```

```

00260         else lastchar = c;
00261         namebuf[i++] = c;
00262         if ( i > MAXPRENAMESIZE ) {
00263             namebuf[i] = 0;
00264             Error1("Preprocessor variable name too long: ",namebuf);
00265         }
00266     }
00267     namebuf[i++] = 0;
00268     if ( c != '"' ) {
00269         Error1("Unmatched quotes for preprocessor variable",namebuf);
00270     }
00271     AP.DelayPrevar = olddelay;
00272     if ( namebuf[0] == '$' ) {
00273         raiselow = PRENOACTION;
00274         if ( AP.PreproFlag && *AP.preStart ) {
00275             s = EndOfToken(AP.preStart);
00276             c = *s; *s = 0;
00277             if ( ( StrICmp(AP.preStart, (UBYTE *) "ifdef" ) == 0
00278                 || StrICmp(AP.preStart, (UBYTE *) "ifndef" ) == 0 )
00279                 && GetDollar(namebuf+1) < 0 ) {
00280                 *s = c; c = ' ';
00281                 break;
00282             }
00283             *s = c;
00284         }
00285         else {
00286             s = EndOfToken(namebuf+1);
00287             if ( *s == '[' ) { while ( *s ) s++; }
00288         }
00289         if ( *s == '-' && s[1] == '-' && s[2] == 0 )
00290             raiselow = PRELOWERAFTER;
00291         else if ( *s == '+' && s[1] == '+' && s[2] == 0 )
00292             raiselow = PRERAISEAFTER;
00293         c = *s; *s = 0;
00294         if ( OpenStream(namebuf+1,DOLLARSTREAM,0,raiselow) == 0 ) {
00295             *s = c;
00296             MesPrint("@Undefined variable %s used as preprocessor variable",
00297                     namebuf);
00298             Terminate(-1);
00299         }
00300         *s = c;
00301     }
00302     else {
00303         raiselow = PRENOACTION;
00304         if ( AP.PreproFlag && *AP.preStart ) {
00305             s = EndOfToken(AP.preStart);
00306             c = *s; *s = 0;
00307             if ( ( StrICmp(AP.preStart, (UBYTE *) "ifdef" ) == 0
00308                 || StrICmp(AP.preStart, (UBYTE *) "ifndef" ) == 0 )
00309                 && GetPreVar(namebuf,WITHOUTERROR) == 0 ) {
00310                 *s = c; c = ' ';
00311                 break;
00312             }
00313             *s = c;
00314         }
00315         s = EndOfToken(namebuf);
00316         if ( *s == '-' ) s++;
00317         if ( *s == '-' && s[1] == '-' && s[2] == 0 )
00318             raiselow = PRELOWERAFTER;
00319         else if ( *s == '+' && s[1] == '+' && s[2] == 0 )
00320             raiselow = PRERAISEAFTER;
00321         else if ( *s == '(' && namebuf[i-2] == ')' ) {
00322             /*
00323              Now count the arguments and separate them by zeroes
00324              Check on the ?var construction and if present, reset
00325              some comma's.
00326              Make the assignments of the variables
00327              Run the macro.
00328              Undefine the variables
00329             */
00330             int nargs = 1;
00331             PREVAR *p;
00332             size_t p_offset;
00333             *s++ = 0; namebuf[i-2] = 0;
00334             if ( StrICmp(namebuf, (UBYTE *) "random_" ) == 0 ) {
00335                 UBYTE *ranvalue;
00336                 ranvalue = PreRandom(s);
00337                 PutPreVar(namebuf,ranvalue,0,1);
00338                 M_free(ranvalue,"PreRandom");
00339                 goto dostream;
00340             }
00341             else if ( StrICmp(namebuf, (UBYTE *) "tolower_" ) == 0 ) {
00342                 UBYTE *ss = s;
00343                 while ( *ss ) { *ss = (UBYTE)(tolower(*ss)); ss++; }
00344                 PutPreVar(namebuf,s,0,1);
00345                 goto dostream;
00346             }

```

```

00347     else if ( StrICmp(namebuf, (UBYTE *)"toupper_") == 0 ) {
00348         UBYTE *ss = s;
00349         while ( *ss ) { *ss = (UBYTE)(toupper(*ss)); ss++; }
00350         PutPreVar(namebuf,s,0,1);
00351         goto dostream;
00352     }
00353     else if ( StrICmp(namebuf, (UBYTE *)"takeleft_") == 0 ) {
00354         UBYTE *ss = s;
00355         int x = 0, nsize;
00356         while ( *ss != ',' && *ss ) ss++;
00357         nsize = ss-s;
00358         if ( *ss ) {
00359             *ss++ = 0;
00360             while ( FG.cTable[*ss] == 1 ) x = 10*x + (*ss++ - '0');
00361             if ( x > nsize ) x = nsize;
00362         }
00363         else x = 0;
00364         PutPreVar(namebuf,s+x,0,1);
00365         goto dostream;
00366     }
00367     else if ( StrICmp(namebuf, (UBYTE *)"takeright_") == 0 ) {
00368         UBYTE *ss = s;
00369         int x = 0, nsize;
00370         while ( *ss != ',' && *ss ) ss++;
00371         nsize = ss-s;
00372         if ( *ss ) {
00373             *ss++ = 0;
00374             while ( FG.cTable[*ss] == 1 ) x = 10*x + (*ss++ - '0');
00375             if ( x > nsize ) x = nsize;
00376         }
00377         else x = 0;
00378         x = nsize - x;
00379         s[x] = 0;
00380         PutPreVar(namebuf,s,0,1);
00381         goto dostream;
00382     }
00383     else if ( StrICmp(namebuf, (UBYTE *)"keepleft_") == 0 ) {
00384         UBYTE *ss = s;
00385         int x = 0, nsize;
00386         while ( *ss != ',' && *ss ) ss++;
00387         nsize = ss-s;
00388         if ( *ss ) {
00389             *ss++ = 0;
00390             while ( FG.cTable[*ss] == 1 ) x = 10*x + (*ss++ - '0');
00391             if ( x > nsize ) x = nsize;
00392         }
00393         else x = nsize;
00394         s[x] = 0;
00395         PutPreVar(namebuf,s,0,1);
00396         goto dostream;
00397     }
00398     else if ( StrICmp(namebuf, (UBYTE *)"kepright_") == 0 ) {
00399         UBYTE *ss = s;
00400         int x = 0, nsize;
00401         while ( *ss != ',' && *ss ) ss++;
00402         nsize = ss-s;
00403         if ( *ss ) {
00404             *ss++ = 0;
00405             while ( FG.cTable[*ss] == 1 ) x = 10*x + (*ss++ - '0');
00406             if ( x > nsize ) x = nsize;
00407         }
00408         else x = nsize;
00409         x = nsize-x;
00410         PutPreVar(namebuf,s+x,0,1);
00411         goto dostream;
00412     }
00413     while ( *s ) {
00414         if ( *s == '\\\\' ) s++;
00415         if ( *s == ',' ) { *s = 0; nargs++; }
00416         s++;
00417     }
00418     GetPreVar(namebuf,WITHEXERROR);
00419     p = ThePreVar;
00420     if ( p == 0 ) {
00421         MesPrint("@Illegal use of arguments in preprocessor variable %s",namebuf);
00422         Terminate(-1);
00423     }
00424     if ( p->nargs <= 0 || ( p->wildarg == 0 && nargs != p->nargs )
00425         || ( p->wildarg > 0 && nargs < p->nargs-1 ) ) {
00426         MesPrint("@Arguments of macro %s do not match",namebuf);
00427         Terminate(-1);
00428     }
00429     if ( p->wildarg > 0 ) {
00430         /*
00431             Change some zeroes into commas
00432         */
00433         s = namebuf;

```

```

00434         for ( j = 0; j < p->wildarg; j++ ) {
00435             while ( *s ) s++;
00436             s++;
00437         }
00438         for ( j = 0; j < nargs-p->nargs; j++ ) {
00439             while ( *s ) s++;
00440             *s++ = ',';
00441         }
00442     }
00443 /*
00444     Now we can make the assignments
00445 */
00446     s = namebuf;
00447     while ( *s ) s++;
00448     s++;
00449     t = p->argnames;
00450     p_offset = p - PreVar;
00451     for ( j = 0; j < p->nargs; j++ ) {
00452         if ( ( nargs == p->nargs-1 ) && ( *t == '?' ) ) {
00453             PutPreVar(t,0,0,0);
00454         }
00455         else {
00456             PutPreVar(t,s,0,0);
00457             while ( *s ) s++;
00458             s++;
00459         }
00460         p = PreVar + p_offset;
00461         while ( *t ) t++;
00462         t++;
00463     }
00464 }
00465 dostream;;
00466     if ( ( stream = OpenStream(namebuf,PREVARSTREAM,0,raiselow) ) == 0 ) {
00467 /*
00468     Eat comma before or after. This is `no value'
00469 */
00470     }
00471     else if ( stream->inbuffer == 0 ) {
00472         c = GetInput();
00473         if ( level > 0 && c == '\\' ) return(c);
00474         goto endofloop;
00475     }
00476 }
00477     c = GetInput();
00478 }
00479     else if ( c == '{' ) { /* Try the preprocessor calculator */
00480         if ( PreCalc() == 0 ) Terminate(-1);
00481         c = GetInput(); /* This is either a { or a number */
00482         break;
00483     }
00484     else break;
00485 endofloop;;
00486 }
00487     return(c);
00488 }
00489
00490 /*
00491     #] GetChar :
00492     #[ CharOut :
00493 */
00494
00495 void CharOut(UBYTE c)
00496 {
00497     if ( c == LINEFEED ) {
00498         AM.OutBuffer[AP.InOutBuf++] = c;
00499         WriteString(INPUTOUT,AM.OutBuffer,AP.InOutBuf);
00500         AP.InOutBuf = 0;
00501     }
00502     else {
00503         if ( AP.InOutBuf >= AM.OutBufSize || c == LINEFEED ) {
00504             WriteString(INPUTOUT,AM.OutBuffer,AP.InOutBuf);
00505             AP.InOutBuf = 0;
00506         }
00507         AM.OutBuffer[AP.InOutBuf++] = c;
00508     }
00509 }
00510
00511 /*
00512     #] CharOut :
00513     #[ UnsetAllowDelay :
00514 */
00515
00516 void UnsetAllowDelay(void)
00517 {
00518     if ( ThePreVar != 0 ) {
00519         if ( ThePreVar->nargs > 0 ) AP.AllowDelay = 0;
00520     }

```

```

00521 }
00522
00523 /*
00524     #] UnsetAllowDelay :
00525     #[ GetPreVar :
00526
00527     We use the model of a heap. If the same name has been used more
00528     than once the last definition is used. This gives the impression
00529     of local variables.
00530
00531     There are two types: The regular ones and the expression variables.
00532     The last ones are like UNCHANGED_exprname and ZERO_exprname or
00533     UNCHANGED_ and ZERO_.
00534 */
00535
00536 static UBYTE *yes = (UBYTE *) "1";
00537 static UBYTE *no = (UBYTE *) "0";
00538 static UBYTE numintopolynomial[12];
00539 #include "vector.h"
00540 static Vector(UBYTE, exprstr); /* Used for numactiveexprs_ and activeexprnames_. */
00541
00542 UBYTE *GetPreVar(UBYTE *name, int flag)
00543 {
00544     GETIDENTITY
00545     int i, mode;
00546     WORD number;
00547     UBYTE *t, c = 0, *tt = 0;
00548     t = name; while ( *t ) t++;
00549     if ( t[-1] == '-' && t[-2] == '-' && t-2 > name && t[-3] != '_' ) {
00550         t -= 2; c = *t; *t = 0; tt = t;
00551     }
00552     else if ( t[-1] == '+' && t[-2] == '+' && t-2 > name && t[-3] != '_' ) {
00553         t -= 2; c = *t; *t = 0; tt = t;
00554     }
00555     else if ( StrICmp(name, (UBYTE *) "time_") == 0 ) {
00556         UBYTE millibuf[24];
00557         LONG millitime, timepart;
00558         int timepart1, timepart2;
00559         static char timestring[40];
00560         /* millitime = TimeCPU(1); */
00561         millitime = GetRunningTime();
00562         timepart = millitime%1000;
00563         millitime /= 1000;
00564         timepart /= 10;
00565         timepart1 = timepart / 10;
00566         timepart2 = timepart % 10;
00567         NumToStr(millibuf, millitime);
00568         snprintf(timestring, 40, "%s.%ld%ld", millibuf, timepart1, timepart2);
00569         return((UBYTE *) timestring);
00570     }
00571     else if ( ( StrICmp(name, (UBYTE *) "timer_") == 0 )
00572         || ( StrICmp(name, (UBYTE *) "stopwatch_") == 0 ) ) {
00573         static char timestring[40];
00574         snprintf(timestring, 40, "%ld", (long int) (GetRunningTime() - AP.StopWatchZero));
00575         return((UBYTE *) timestring);
00576     }
00577     else if ( StrICmp(name, (UBYTE *) "numactiveexprs_") == 0 ) {
00578         /* the number of active expressions */
00579         int n = 0;
00580         for ( i = 0; i < NumExpressions; i++ ) {
00581             EXPRESSIONS e = Expressions + i;
00582             switch ( e->status ) {
00583                 case LOCALEXPRESSION:
00584                 case GLOBALEXPRESSION:
00585                 case UNHIDELEXPRESSION:
00586                 case UNHIDEGEXPRESSION:
00587                 case INTOHIDELEXPRESSION:
00588                 case INTOHIDEGEXPRESSION:
00589                     n++;
00590                     break;
00591             }
00592         }
00593         VectorReserve(exprstr, 41); /* up to 128-bit */
00594         LongCopy(n, (char *) VectorPtr(exprstr));
00595         return VectorPtr(exprstr);
00596     }
00597     else if ( StrICmp(name, (UBYTE *) "activeexprnames_") == 0 ) {
00598         /* the list of active expressions separated by commas */
00599         int j = 0;
00600         VectorReserve(exprstr, 16); /* at least 1 character for '\0' */
00601         for ( i = 0; i < NumExpressions; i++ ) {
00602             UBYTE *p, *s;
00603             int len, k;
00604             EXPRESSIONS e = Expressions + i;
00605             switch ( e->status ) {
00606                 case LOCALEXPRESSION:
00607                 case GLOBALEXPRESSION:

```

```

00608         case UNHIDELEXPRESSION:
00609         case UNHIDEGEXPRESSION:
00610         case INTOHIDELEXPRESSION:
00611         case INTOHIDEGEXPRESSION:
00612             s = AC.exprnames->namebuffer + e->name;
00613             len = StrLen(s);
00614             VectorSize(exprstr) = j; /* j bytes must be copied in extending the buffer. */
00615             VectorReserve(exprstr, j + len * 2 + 1);
00616             p = VectorPtr(exprstr);
00617             if ( j > 0 ) p[j++] = ',';
00618             for ( k = 0; k < len; k++ ) {
00619                 if ( s[k] == ',' || s[k] == '|' ) p[j++] = '\\';
00620                 p[j++] = s[k];
00621             }
00622             break;
00623     }
00624 }
00625 VectorPtr(exprstr)[j] = '\\0';
00626 return VectorPtr(exprstr);
00627 }
00628 else if ( StrICmp(name, (UBYTE *) "path_") == 0 ) {
00629     /* the current FORM path (for debugging both in .c and .frm) */
00630     if ( AM.Path ) {
00631         return(AM.Path);
00632     }
00633     else {
00634         return((UBYTE *) "");
00635     }
00636 }
00637 t = name;
00638 while ( *t && *t != '_' ) t++;
00639 for ( i = NumPre-1; i >= 0; i-- ) {
00640     if ( *t == '_' && ( StrICmp(name, PreVar[i].name) == 0 ) ) {
00641         if ( c ) *tt = c;
00642         ThePreVar = PreVar+i;
00643         return(PreVar[i].value);
00644     }
00645     else if ( StrCmp(name, PreVar[i].name) == 0 ) {
00646         if ( c ) *tt = c;
00647         ThePreVar = PreVar+i;
00648         return(PreVar[i].value);
00649     }
00650 }
00651 if ( *t == '_' ) {
00652     if ( StrICmp(name, (UBYTE *) "EXTRASYNBOLS_") == 0 ) goto extrashort;
00653     *t = 0;
00654     if ( StrICmp(name, (UBYTE *) "UNCHANGED") == 0 ) mode = 1;
00655     else if ( StrICmp(name, (UBYTE *) "ZERO") == 0 ) mode = 0;
00656     else if ( StrICmp(name, (UBYTE *) "SHOWINPUT") == 0 ) {
00657         *t++ = '_';
00658         if ( c ) *tt = c;
00659         if ( AC.NoShowInput > 0 ) return(no);
00660         else return(yes);
00661     }
00662     else if ( StrICmp(name, (UBYTE *) "EXTRASYNBOLS") == 0 ) {
00663         *t++ = '_';
00664 extrashort:;
00665         number = cbuf[AM.sbufnum].numrhs;
00666         t = numintopolynomial;
00667         NumCopy(number, t);
00668         return(numintopolynomial);
00669     }
00670     else mode = -1;
00671     *t++ = '_';
00672     if ( mode >= 0 ) {
00673         ThePreVar = 0;
00674         if ( *t ) {
00675             if ( GetName(AC.exprnames, t, &number, NOAUTO) == CEXPRESSION ) {
00676                 if ( c ) *tt = c;
00677                 if ( ( Expressions[number].vflags & ( 1 << mode ) ) != 0 )
00678                     return(yes);
00679                 else return(no);
00680             }
00681         }
00682         else {
00683             /*
00684              Here we have to test all active results.
00685              These are in 'negative' so the flags have to be zero.
00686             */
00687             if ( c ) *tt = c;
00688             if ( ( AR.expflags & ( 1 << mode ) ) == 0 ) return(yes);
00689             else return(no);
00690         }
00691     }
00692 }
00693 if ( ( t = (UBYTE *) (getenv((char *) (name))) ) != 0 ) {
00694     if ( c ) *tt = c;

```



```

00695     ThePreVar = 0;
00696     return(t);
00697 }
00698 if ( c ) *tt = c;
00699 if ( flag == WITHEERROR ) {
00700     Error1("Undefined preprocessor variable",name);
00701 }
00702 return(0);
00703 }
00704
00705 /*
00706     #] GetPreVar :
00707     #[ PutPreVar :
00708 */
00709
00724 int PutPreVar(UBYTE *name, UBYTE *value, UBYTE *args, int mode)
00725 {
00726     int i, ii, num = 2, nnum = 2, numargs = 0;
00727     UBYTE *s, *t, *u = 0;
00728     PREVAR *p;
00729     if ( value == 0 && name[0] != '?' ) {
00730         MesPrint("@Illegal empty value for preprocessor variable %s",name);
00731         Terminate(-1);
00732     }
00733     if ( args ) {
00734         s = args; num++;
00735         while ( *s ) {
00736             if ( *s != ' ' && *s != '\t' ) num++;
00737             s++;
00738         }
00739     }
00740     if ( mode == 1 ) {
00741         i = NumPre;
00742         while ( --i >= 0 ) {
00743             if ( StrCmp(name,PreVar[i].name) == 0 ) {
00744                 u = PreVar[i].name;
00745                 break;
00746             }
00747         }
00748     }
00749     else i = -1;
00750     if ( i < 0 ) { p = (PREVAR *)FromList(&AP.PreVarList); ii = p - PreVar; }
00751     else { p = &(PreVar[i]); ii = i; }
00752     if ( value ) {
00753         s = value; while ( *s ) { s++; num++; }
00754     }
00755     else num = 1;
00756     if ( i >= 0 ) {
00757         if ( p->value ) {
00758             s = p->value;
00759             while ( *s ) { s++; nnum++; }
00760         }
00761         else nnum = 1;
00762         if ( nnum >= num ) {
00763             /*
00764                 We can keep this in place
00765             */
00766             if ( value && p->value ) {
00767                 s = value;
00768                 t = p->value;
00769                 while ( *s ) *t++ = *s++;
00770                 *t = 0;
00771             }
00772             else p->value = 0;
00773             return(ii);
00774         }
00775     }
00776     s = name; while ( *s ) { s++; num++; }
00777     t = (UBYTE *)Malloc1(num,"PreVariable");
00778     p->name = t;
00779     s = name; while ( *s ) *t++ = *s++; *t++ = 0;
00780     if ( value ) {
00781         p->value = t;
00782         s = value; while ( *s ) *t++ = *s++; *t = 0;
00783         if ( AM.atstartup && t[-1] == '\n' ) t[-1] = 0;
00784     }
00785     else p->value = 0;
00786     p->wildarg = 0;
00787     if ( args ) {
00788         int first = 1;
00789         t++; p->argnames = t;
00790         s = args;
00791         while ( *s ) {
00792             if ( *s == ' ' || *s == '\t' ) { s++; continue; }
00793             if ( *s == ',' ) {
00794                 s++; *t++ = 0; numargs++;
00795                 while ( *s == ' ' || *s == '\t' ) s++;

```

```

00796         if ( *s == '?' ) {
00797             if ( p->wildarg > 0 ) {
00798                 Error0("More than one ?var in #define");
00799             }
00800             p->wildarg = numargs;
00801         }
00802     }
00803     else if ( *s == '?' && first ) {
00804         p->wildarg = 1; *t++ = *s++;
00805     }
00806     else { *t++ = *s++; }
00807     first = 0;
00808 }
00809 *t = 0;
00810 numargs++;
00811 p->nargs = numargs;
00812 }
00813 else {
00814     p->nargs = 0;
00815     p->argnames = 0;
00816 }
00817 if ( u ) M_free(u,"replace PreVar value");
00818 return(ii);
00819 }
00820
00821 /*
00822     #] PutPreVar :
00823     #[ PopPreVars :
00824 */
00825
00826 void PopPreVars(int tonumber)
00827 {
00828     PREVAR *p = &(PreVar[NumPre]);
00829     while ( NumPre > tonumber ) {
00830         NumPre--; p--;
00831         M_free(p->name,"popping PreVar");
00832         p->name = p->value = 0;
00833     }
00834 }
00835
00836 /*
00837     #] PopPreVars :
00838     #[ IniModule :
00839 */
00840
00841 void IniModule(int type)
00842 {
00843     GETIDENTITY
00844     WORD **w, i;
00845     CBUF *C = cbuf+AC.cbunum;
00846     /*[05nov2003 mt]:*/
00847 #ifdef WITHMPI
00848     /* To prevent
00849      * (1) FlushOut() and PutOut() on the slaves to send a mess to the master
00850      * compiling a module,
00851      * (2) EndSort() called from poly_factorize_expression() on the master
00852      * waits for the slaves.
00853      */
00854     PF.parallel=0;
00855     /*BTW, this was the bug preventing usage of more than 1 expression!*/
00856 #endif
00857     AR.BracketOn = 0;
00858     AR.StoreData.dirtyflag = 0;
00859     AC.bracketindexflag = 0;
00860     AT.bracketindexflag = 0;
00861
00862     /*[06nov2003 mt]:*/
00863 #ifdef WITHMPI
00864     /* This flag may be set in the procedure tokenize(). */
00865     AC.RhsExprInModuleFlag = 0;
00866     /*[20oct2009 mt]:*/
00867     PF.mkSlaveInfile=0;
00868     PF.slavebuf.PObuffer=NULL;
00869     for(i=0; i<NumExpressions; i++)
00870         Expressions[i].vflags &= ~ISINRHS;
00871     /*[20oct2009 mt]:*/
00872 #endif
00873 /*[06nov2003 mt]:*/
00874
00875     /*[19nov2003 mt]:*/
00876     /*The module counter:*/
00877     (AC.CModule)++;
00878     /*[19nov2003 mt]:*/
00879
00880     if ( !type ) {
00881         if ( C->rhs ) {

```

```

00883         w = C->rhs; i = C->maxrhs;
00884         do { *w++ = 0; } while ( --i > 0 );
00885     }
00886     if ( C->lhs ) {
00887         w = C->lhs; i = C->maxlhs;
00888         do { *w++ = 0; } while ( --i > 0 );
00889     }
00890 }
00891 C->numlhs = C->numrhs = 0;
00892 ClearTree(AC.cbufnum);
00893 while ( AC.NumLabels > 0 ) {
00894     AC.NumLabels--;
00895     if ( AC.LabelNames[AC.NumLabels] ) M_free(AC.LabelNames[AC.NumLabels], "LabelName");
00896 }
00897
00898 C->Pointer = C->Buffer;
00899
00900 AC.Commercial[0] = 0;
00901
00902 AC.IfStack = AC.IfHeap;
00903 AC.arglevel = 0;
00904 AC.termlevel = 0;
00905 AC.IfLevel = 0;
00906 AC.WhileLevel = 0;
00907 AC.RepLevel = 0;
00908 AC.insidelevel = 0;
00909 AC.dolooplevel = 0;
00910 AC.MustTestTable = 0;
00911 AO.PrintType = 0; /* Otherwise statistics can get spoiled */
00912 AC.ComDefer = 0;
00913 AC.CollectFun = 0;
00914 AM.S0->PolyWise = 0;
00915 AC.SymChangeFlag = 0;
00916 AP.lhdollarerror = 0;
00917 AR.PolyFun = AC.lPolyFun;
00918 AR.PolyFunInv = AC.lPolyFunInv;
00919 AR.PolyFunType = AC.lPolyFunType;
00920 AR.PolyFunExp = AC.lPolyFunExp;
00921 AR.PolyFunVar = AC.lPolyFunVar;
00922 AR.PolyFunPow = AC.lPolyFunPow;
00923 AC.mparallelflag = AC.parallelflag | AM.hparallelflag;
00924 AC.inparallelflag = 0;
00925 AC.mProcessBucketSize = AC.ProcessBucketSize;
00926 NumPotModdollars = 0;
00927 AC.topolynomialflag = 0;
00928 #ifdef WITHPTHREADS
00929     if ( AM.totalnumberofthreads > 1 ) AS.MultiThreaded = 1;
00930     else AS.MultiThreaded = 0;
00931     for ( i = 1; i < AM.totalnumberofthreads; i++ ) {
00932         AB[i]->T.S0->PolyWise = 0;
00933     }
00934 #endif
00935     OpenTemp();
00936 }
00937
00938 /*
00939     #] IniModule :
00940     #[ IniSpecialModule :
00941 */
00942
00943 void IniSpecialModule(int type)
00944 {
00945     DUMMYUSE(type);
00946 }
00947
00948 /*
00949     #] IniSpecialModule :
00950     #[ PreProcessor :
00951 */
00952
00953 void PreProcessor(void)
00954 {
00955     int moduletype = FIRSTMODULE;
00956     int specialtype = 0;
00957     int error1 = 0, error2 = 0, retcode, retval;
00958     UBYTE c, *t, *s;
00959     AP.StopWatchZero = GetRunningTime();
00960     AC.compiletype = 0;
00961     AP.PreContinuation = 0;
00962     AP.PreAssignLevel = 0;
00963     AP.gNumPre = NumPre;
00964     AC.iPointer = AC.iBuffer;
00965     AC.iPointer[0] = 0;
00966
00967     if ( AC.CheckpointFlag == -1 ) DoRecovery(&moduletype);
00968     AC.CheckpointStamp = Timer(0);
00969 }

```

```

00970     for(;;) {
00971     /*      if ( A.StatisticsFlag ) CharOut(LINEFEED); */
00972
00973         IniModule(moduletype);
00974
00975         /*Re-define preprocessor variable CMODULE_ as a current module number, starting from 1*/
00976         /*The module counter is AC.CModule, it is incremented in IniModule*/
00977         {
00978             UBYTE buf[24];/*64/Log_2[10] = 19.3, this is enough for any integer*/
00979             NumToStr(buf,AC.CModule);
00980             PutPreVar((UBYTE *) "CMODULE_",buf,0,1);
00981         }
00982
00983         if ( specialtype ) IniSpecialModule(specialtype);
00984
00985         for(;;) { /* Read a single line/statement */
00986             c = GetChar(0);
00987             if ( c == AP.ComChar ) { /* This line is commentary */
00988                 LoadInstruction(5);
00989                 if ( AC.CurrentStream->FoldName ) {
00990                     t = AP.preStart;
00991                     if ( *t && t[1] && t[2] == '#' && t[3] == ']' ) {
00992                         t += 4;
00993                         while ( *t == ' ' || *t == '\t' ) t++;
00994                         s = AC.CurrentStream->FoldName;
00995                         while ( *s == *t ) { s++; t++; }
00996                         if ( *s == 0 && ( *t == ' ' || *t == '\t'
00997                             || *t == ':' ) ) {
00998                             while ( *t == ' ' || *t == '\t' ) t++;
00999                             if ( *t == ':' ) {
01000                                 AC.CurrentStream = CloseStream(AC.CurrentStream);
01001                             }
01002                         }
01003                     }
01004                 }
01005                 *AP.preStart = 0;
01006                 continue;
01007             }
01008             while ( c == ' ' || c == '\t' ) c = GetChar(0);
01009             if ( c == LINEFEED ) continue;
01010             if ( c == ENDOFINPUT ) {
01011                 /*      CharOut(LINEFEED); */
01012                 Warning(".end instruction generated");
01013                 moduletype = ENDMODULE; specialtype = 0;
01014                 goto endmodule; /* Fake one */
01015             }
01016             if ( c == '#' ) {
01017                 if ( PreProInstruction() ) { error1++; error2++; AP.preError++; }
01018                 *AP.preStart = 0;
01019             }
01020             else if ( c == '.' ) {
01021                 if ( ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) ||
01022                     ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) ) {
01023                     LoadInstruction(1);
01024                     continue;
01025                 }
01026                 if ( ModuleInstruction(&moduletype,&specialtype) ) { error2++; AP.preError++; }
01027                 if ( specialtype ) SetSpecialMode(moduletype,specialtype);
01028                 if ( AP.PreInsideLevel != 0 ) {
01029                     MesPrint("@end of module instructions may not be used inside");
01030                     MesPrint("@the scope of a %#inside %#endinside construction.");
01031                     Terminate(-1);
01032                 }
01033                 if ( AC.RepLevel > 0 ) {
01034                     MesPrint("&EndRepeat statement(s) missing");
01035                     error2++; AP.preError++;
01036                 }
01037                 if ( AC.tablecheck == 0 ) {
01038                     AC.tablecheck = 1;
01039                     if ( TestTables() ) { error2++; AP.preError++; }
01040                 }
01041                 if ( AP.PreContinuation ) {
01042                     error1++; error2++;
01043                     MesPrint("&Unfinished statement. Missing ;?");
01044                 }
01045                 if ( moduletype == GLOBALMODULE ) MakeGlobal();
01046                 else {
01047                     endmodule:
01048                     if ( error2 == 0 && AM.qError == 0 ) {
01049                         retcode = ExecModule(moduletype);
01050                         #ifdef WITHMPI
01051                         if(PF.slavebuf.PObuffer!=NULL){
01052                             M_free(PF.slavebuf.PObuffer,"PF inbuf");
01053                             PF.slavebuf.PObuffer=NULL;
01054                         }
01055                         #endif
01056                         UpdatePositions();
01057                         if ( retcode < 0 ) error1++;

```

```

01057         if ( retcode ) { error2++; AP.preError++; }
01058     }
01059     else {
01060         EXPRESSIONS e;
01061         WORD j;
01062         for ( j = 0, e = Expressions; j < NumExpressions; j++, e++ ) {
01063             if ( e->replace == NEWLYDEFINEDEXPRESSION ) e->replace =
REGULAREXPRESSION;
01064         }
01065     }
01066     switch ( moduletype ) {
01067     case STOREMODULE:
01068         if ( ExecStore() ) error1++;
01069         break;
01070     case CLEARMODULE:
01071         FullCleanUp();
01072         error1 = error2 = AP.preError = 0;
01073         AM.atstartup = 1;
01074         PutPreVar((UBYTE *) "DATE_", (UBYTE *) MakeDate(), 0, 1);
01075         AM.atstartup = 0;
01076         if ( AM.resetTimeOnClear ) {
01077             #ifdef WITHPTHREADS
01078                 ClearAllThreads();
01079             #endif
01080                 AM.SumTime += TimeCPU(1);
01081                 TimeCPU(0);
01082             }
01083             AP.StopWatchZero = GetRunningTime();
01084             break;
01085     case ENDMODULE:
01086         Terminate( -( error1 | error2 ) );
01087     }
01088     }
01089     AC.tablecheck = 0;
01090     AC.compiletype = 0;
01091     if ( AC.exprfillwarning > 0 ) {
01092         AC.exprfillwarning = 0;
01093     }
01094     if ( AC.CheckpointFlag && error1 == 0 && error2 == 0 ) DoCheckpoint(moduletype);
01095     break; /* start a new module */
01096 }
01097 else {
01098     if ( ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) ||
01099         ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) ) {
01100         pushbackchar = c;
01101         LoadInstruction(5);
01102         continue;
01103     }
01104     UngetChar(c);
01105     if ( AP.PreContinuation ) {
01106         retval = LoadStatement(OLDSTATEMENT);
01107     }
01108     else {
01109         AC.CurrentStream->prevline = AC.CurrentStream->linenumber;
01110         retval = LoadStatement(NEWSTATEMENT);
01111     }
01112     if ( retval < 0 ) {
01113         error1++;
01114         if ( retval == -1 ) AP.PreContinuation = 0;
01115         else AP.PreContinuation = 1;
01116         TryRecover(0);
01117     }
01118     else if ( retval > 0 ) AP.PreContinuation = 0;
01119     else AP.PreContinuation = 1;
01120     if ( error1 == 0 && !AP.PreContinuation ) {
01121         if ( ( AP.PreDebug & PREPROONLY ) == 0 ) {
01122             int onpmd = NumPotModdollars;
01123             #ifdef WITHMPI
01124             WORD oldRhsExprInModuleFlag = AC.RhsExprInModuleFlag;
01125             if ( AP.PreAssignFlag ) AC.RhsExprInModuleFlag = 0;
01126             #endif
01127             if ( AP.PreOut || ( AP.PreDebug & DUMPTOCOMPILER )
01128                 == DUMPTOCOMPILER )
01129                 MesPrint(" %s", AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]);
01130             retcode = CompileStatement(AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]);
01131             if ( retcode < 0 ) error1++;
01132             if ( retcode ) { error2++; AP.preError++; }
01133             if ( AP.PreAssignFlag ) {
01134                 if ( retcode == 0 ) {
01135                     if ( ( retcode = CatchDollar(0) ) < 0 ) error1++;
01136                     else if ( retcode > 0 ) { error2++; AP.preError++; }
01137                 }
01138                 else CatchDollar(-1);
01139                 POPPREASSIGNLEVEL;
01140                 if ( AP.PreAssignLevel <= 0 )
01141                     AP.PreAssignFlag = 0;
01142                 NumPotModdollars = onpmd;

```

```

01143 #ifdef WITHMPI
01144             AC.RhsExprInModuleFlag = oldRhsExprInModuleFlag;
01145 #endif
01146         }
01147     }
01148     else {
01149         MesPrint(" %s",AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]);
01150     }
01151 }
01152 else if ( !AP.PreContinuation ) {
01153     if ( AP.PreAssignLevel > 0 ) {
01154         POPPREASSIGNLEVEL;
01155         if ( AP.PreAssignLevel <=0 )
01156             AP.PreAssignFlag = 0;
01157     }
01158 }
01159 /*
01160         if ( !AP.PreContinuation ) AP.PreAssignFlag = 0;
01161 */
01162     }
01163 }
01164 }
01165 }
01166
01167 /*
01168     #] PreProcessor :
01169     #[ PreProInstruction :
01170 */
01171
01172 int PreProInstruction(void)
01173 {
01174     UBYTE *s, *t;
01175     KEYWORD *key;
01176     AP.PreproFlag = 1;
01177     AP.preFill = 0;
01178     AP.AllowDelay = 0;
01179     AP.DelayPrevar = 0;
01180
01181     oldmode = 0;
01182     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) {
01183         LoadInstruction(3);
01184         if ( ( StrICmp(AP.preStart,(UBYTE *)"case") == 0
01185             || StrICmp(AP.preStart,(UBYTE *)"default") == 0 )
01186             && AP.PreSwitchModes[AP.PreSwitchLevel] == SEARCHINGPRECASE ) {
01187             LoadInstruction(0);
01188         }
01189         else if ( StrICmp(AP.preStart,(UBYTE *)"assign ") == 0 ) {}
01190         else { LoadInstruction(1); }
01191     }
01192     else if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) {
01193         LoadInstruction(3);
01194         if ( ( StrICmp(AP.preStart,(UBYTE *)"else") == 0
01195             || StrICmp(AP.preStart,(UBYTE *)"elseif") == 0 )
01196             && AP.PreIfStack[AP.PreIfLevel] == LOOKINGFORELSE ) {
01197             LoadInstruction(0);
01198         }
01199         else if ( StrICmp(AP.preStart,(UBYTE *)"assign ") == 0 ) {}
01200         else {
01201             LoadInstruction(1);
01202         }
01203     }
01204     else {
01205         LoadInstruction(0);
01206     }
01207     AP.PreproFlag = 0;
01208     t = AP.preStart;
01209     if ( *t == '-' ) {
01210         if ( AP.PreSwitchModes[AP.PreSwitchLevel] == EXECUTINGPRESWITCH
01211             && AP.PreIfStack[AP.PreIfLevel] == EXECUTINGIF )
01212             AC.NoShowInput = 1;
01213     }
01214     else if ( *t == '+' ) {
01215         if ( AP.PreSwitchModes[AP.PreSwitchLevel] == EXECUTINGPRESWITCH
01216             && AP.PreIfStack[AP.PreIfLevel] == EXECUTINGIF )
01217             AC.NoShowInput = 0;
01218     }
01219     else if ( *t == ':' ) {}
01220     else {
01221         retry:;
01222         key = FindKeyWord(t,precommands,sizeof(precommands)/sizeof(KEYWORD));
01223         s = EndOfToken(t);
01224         if ( key == 0 ) {
01225             if ( *s == ';' ) {
01226                 *s = 0; goto retry;
01227             }
01228             else {
01229                 *s = 0;

```

```

01230         MesPrint("@Unrecognized preprocessor instruction: %s",t);
01231         return(-1);
01232     }
01233 }
01234 while ( *s == ' ' || *s == '\t' || *s == ',' ) s++;
01235 t = s;
01236 while ( *t ) t++;
01237 while ( ( t[-1] == ';' ) && ( t[-2] != '\\\ ' ) ) {
01238     t--; *t = 0;
01239 }
01240 return((key->func)(s));
01241 }
01242 return(0);
01243 }
01244
01245 /*
01246     #] PreProInstruction :
01247     #[ LoadInstruction :
01248
01249     0: preprocessor instruction that may involve matching of brackets
01250     1: runs straight to end-of-line
01251     2: runs to ;
01252     3: only gets one word without ` ' interpretation.
01253     5: with pushbackchar, but inside commentary. -> 1
01254
01255 To be added:
01256     In define, redefine, call and listed do we may have delayed substitution
01257     of preprocessor variables.
01258 */
01259
01260 int LoadInstruction(int mode)
01261 {
01262     UBYTE *s, *sstart, *t, c, cp;
01263     LONG position, fillpos = 0;
01264     int bralevel = 0, parlevel = 0, first = 1;
01265     int quotelevel = 0;
01266     if ( AP.preFill ) {
01267         s = AP.preFill;
01268         AP.preFill = 0;
01269         if ( s[1] != LINEFEED && s[1] != ENDOFINPUT ) {
01270             s[0] = s[1]; s++;
01271         }
01272         else { oldmode = mode; return(0); }
01273     }
01274     else { s = AP.preStart; }
01275     sstart = s; *s = 0;
01276     for(;;) {
01277         if ( ( mode & 1 ) == 1 ) {
01278             if ( pushbackchar && ( mode == 3 || mode == 5 ) ) {
01279                 c = pushbackchar; pushbackchar = 0;
01280             }
01281             else c = GetInput();
01282         }
01283         else {
01284             c = GetChar(0);
01285         }
01286
01287         if ( mode == 2 && c == ';' ) break;
01288         if ( ( mode == 1 || mode == 5 ) && c == LINEFEED ) break;
01289         if ( mode == 3 && FG.cTable[c] != 0 ) {
01290             if ( c == '$' ) {
01291                 pushbackchar = '$';
01292                 *s++ = 'a'; *s++ = 's'; *s++ = 's'; *s++ = 'i';
01293                 *s++ = 'g'; *s++ = 'n'; *s++ = ' '; *s = 0;
01294             }
01295             if ( c == '\\' || c == '`' ) { /* we do not expand preprocessor variables */
01296                 mode = 1;
01297             }
01298             else {
01299                 AP.preFill = s; *s++ = 0; *s = c;
01300                 oldmode = mode;
01301                 return(0);
01302             }
01303         }
01304         if ( mode == 0 && first ) {
01305             if ( c == '$' ) {
01306                 dodollar:
01307                 s = sstart;
01308                 *s++ = 'a'; *s++ = 's'; *s++ = 's'; *s++ = 'i';
01309                 *s++ = 'g'; *s++ = 'n'; *s = 0;
01310                 pushbackchar = c;
01311                 oldmode = mode;
01312                 return(0);
01313             }
01314             if ( c == ' ' || c == '\t' || c == ',' ) {}
01315             else first = 0;
01316         }
01317         else if ( mode == 1 && first && c == '$' && oldmode == 3 ) goto dodollar;

```

```

01317         if ( c == ENDOFINPUT || ( c == LINEFEED
01318 /*      && bralevel == 0 */
01319         && quotelevel == 0 ) ) {
01320             if ( mode == 2 && c == ENDOFINPUT ) {
01321                 MesPrint("@Unexpected end of instruction");
01322                 oldmode = mode;
01323                 return(-1);
01324             }
01325 /*
01326             if ( mode == 0 && bralevel ) {
01327                 MesPrint("@Unmatched brackets");
01328                 oldmode = mode;
01329                 return(-1);
01330             }
01331 */
01332             if ( mode != 2 ) break;
01333         }
01334         if ( quotelevel ) {
01335             if ( c == '\\\\' ) {
01336                 if ( ( mode == 1 ) || ( mode == 5 ) ) c = GetInput();
01337                 else {
01338                     c = GetChar(0);
01339                 }
01340                 if ( c == ENDOFINPUT ) {
01341                     MesPrint("@Unmatched \"");
01342                     if ( mode == 2 && c == ENDOFINPUT ) {
01343                         MesPrint("@Unexpected end of instruction");
01344                     }
01345 /*
01346                     if ( mode == 0 && bralevel ) {
01347                         MesPrint("@Unmatched brackets");
01348                     }
01349 */
01350                     oldmode = mode;
01351                     return(-1);
01352                 }
01353                 else if ( c == LINEFEED ) {}
01354                 else if ( c == '"' ) { *s++ = '\\\\'; }
01355                 else {
01356                     *s++ = '\\\\';
01357                 }
01358             }
01359             else if ( c == '"' ) {
01360                 quotelevel = 0;
01361                 AP.AllowDelay = 0;
01362             }
01363         }
01364         else if ( c == '\\\\' ) {
01365             if ( ( mode == 1 ) || ( mode == 5 ) ) cp = GetInput();
01366             else {
01367                 cp = GetChar(0);
01368             }
01369             if ( cp == LINEFEED ) continue;
01370             if ( mode != 2 || cp != ';' ) *s++ = c;
01371             c = cp;
01372         }
01373         else if ( c == '"' ) {
01374 /*
01375             Now look back in the buffer and determine what the keyword is.
01376             If it is define or redefine, put AllowDelay to 1.
01377 */
01378             t = AP.preStart;
01379             while ( FG.cTable[*t] <= 1 ) t++;
01380             cp = *t; *t = 0;
01381             if ( ( StrICmp(AP.preStart, (UBYTE *) "define") == 0 )
01382                 || ( StrICmp(AP.preStart, (UBYTE *) "redefine") == 0 ) ) {
01383                 AP.AllowDelay = 1;
01384                 oldstream = AC.CurrentStream;
01385             }
01386             *t = cp;
01387             quotelevel = 1;
01388         }
01389         else if ( quotelevel == 0 && bralevel == 0 && c == '(' ) {
01390             t = AP.preStart;
01391             while ( FG.cTable[*t] <= 1 ) t++;
01392             cp = *t; *t = 0;
01393             if ( ( parlevel == 0 )
01394                 && ( StrICmp(AP.preStart, (UBYTE *) "call") == 0 ) ) {
01395                 AP.AllowDelay = 1;
01396                 oldstream = AC.CurrentStream;
01397             }
01398             *t = cp;
01399             parlevel++;
01400         }
01401         else if ( quotelevel == 0 && bralevel == 0 && c == ')' ) {
01402             parlevel--;
01403         }

```



```

01404     else if ( quotelevel == 0 && parlevel == 0 && c == '{' ) {
01405         t = AP.preStart;
01406         while ( FG.cTable[*t] <= 1 ) t++;
01407         cp = *t; *t = 0;
01408         if ( ( bralevel == 0 )
01409             && ( ( StrICmp(AP.preStart, (UBYTE *) "call") == 0 )
01410                 || ( StrICmp(AP.preStart, (UBYTE *) "do") == 0 ) ) ) {
01411             AP.AllowDelay = 1;
01412             oldstream = AC.CurrentStream;
01413         }
01414         *t = cp;
01415         bralevel++;
01416     }
01417     else if ( quotelevel == 0 && parlevel == 0 && c == '}' ) {
01418         bralevel--;
01419         if ( bralevel < 0 ) {
01420             if ( mode != 5 ) {
01421                 MesPrint("@Unmatched brackets");
01422                 oldmode = mode;
01423                 return(-1);
01424             }
01425             bralevel = 0;
01426         }
01427     }
01428     if ( s >= (AP.preStop-1) ) {
01429         UBYTE **ppp;
01430         position = s - AP.preStart;
01431         if ( AP.preFill ) fillpos = AP.preFill - AP.preStart;
01432         ppp = &(AP.preStart); /* to avoid a compiler warning */
01433         if ( DoubleLList((void ***) ppp, &AP.pSize, sizeof(UBYTE),
01434             "instruction buffer") ) { *s = 0; oldmode = mode; return(-1); }
01435         AP.preStop = AP.preStart + AP.pSize-3;
01436         s = AP.preStart + position;
01437         if ( AP.preFill ) AP.preFill = fillpos + AP.preStart;
01438     }
01439     *s++ = c;
01440 }
01441 *s = 0;
01442 oldmode = mode;
01443 if ( mode == 0 ) {
01444     if ( ExpandTripleDots(1) < 0 ) return(-1);
01445 }
01446 return(0);
01447 }
01448
01449 /*
01450     #] LoadInstruction :
01451     #[ LoadStatement :
01452
01453     Puts the current string together in the input buffer.
01454     Does things like placing comma's where needed and expand ...
01455     We force a comma after the keyword. Before 8-sep-2009 the program might
01456     not put a comma if a + or - followed. And then the compiler ate
01457     the + or - and we needed repair code in the routines that used the
01458     + or - (Print, modulus, multiply and (a)bracket). This worked but
01459     the problem was with statements like Dimension -4; which then would
01460     be processed as Dimension 4; (JV)
01461 */
01462
01463 int LoadStatement(int type)
01464 {
01465     UBYTE *s, c, cp;
01466     int retval = 0, stringlevel = 0, newstatement = 0;
01467     if ( type == NEWSTATEMENT ) { AP.eat = 1; newstatement = 1;
01468         s = AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]; }
01469     else { s = AC.iPointer; *s = 0; c = ' '; goto blank; }
01470     *s = 0;
01471     for(;;) {
01472         c = GetChar(0);
01473         if ( c == ENDOFINPUT ) { retval = -1; break; }
01474         if ( stringlevel == 0 ) {
01475             if ( c == LINEFEED ) {
01476                 if ( AP.eat < 0 ) { s--; AP.eat = 0; }
01477                 retval = 0; break;
01478             }
01479             if ( c == ';' ) {
01480                 if ( AP.eat < 0 ) { s--; AP.eat = 0; }
01481                 while ( ( c = GetChar(0) ) == ' ' || c == '\t' ) {}
01482                 if ( c != LINEFEED ) UngetChar(c);
01483                 retval = 1;
01484                 break;
01485             }
01486         }
01487         if ( c == '\\\\' ) {
01488             cp = GetChar(0);
01489             if ( cp == LINEFEED ) continue;
01490             *s++ = c;

```

```

01491         c = cp;
01492     }
01493     if ( c == '"' ) {
01494         if ( stringlevel == 0 ) stringlevel = 1;
01495         else stringlevel = 0;
01496         AP.eat = 0;
01497     }
01498     else if ( stringlevel == 0 ) {
01499         if ( c == '\\t' ) c = ' ';
01500         if ( c == '\\n' ) {
01501             blank:
01502                 if ( newstatement < 0 ) newstatement = 0;
01503                 if ( AP.eat && ( newstatement == 0 ) ) continue;
01504                 c = ' ';
01505                 AP.eat = -2;
01506                 if ( newstatement > 0 ) newstatement = -1;
01507         }
01508         else if ( chartype[c] <= 3 ) {
01509             AP.eat = 0;
01510             if ( newstatement < 0 ) newstatement = 0;
01511         }
01512         else if ( c == ',' ) {
01513             if ( newstatement > 0 ) {
01514                 newstatement = -1;
01515                 AP.eat = -2;
01516             }
01517             else if ( AP.eat == -2 ) { s--; } /*
01518             else if ( AP.eat == -2 ) { AP.eat = 1; continue; }
01519             else { goto doall; }
01520         }
01521         else {
01522             doall::
01523                 if ( AP.eat < 0 ) {
01524                     if ( newstatement == 0 ) s--;
01525                     else { newstatement = 0; }
01526                 }
01527                 else if ( newstatement == 1 ) newstatement = 0;
01528                 AP.eat = 1;
01529                 if ( c == '*' && s > AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel] && s[-1] == '*' )
01530                 {
01531                     s[-1] = '^';
01532                     continue;
01533                 }
01534             }
01535         }
01536         if ( s >= AC.iStop ) {
01537             if ( !AP.iBufError ) {
01538                 LONG position = s - AC.iBuffer;
01539                 LONG position2 = AC.iPointer - AC.iBuffer;
01540                 UBYTE **ppp = &(AC.iBuffer); /* to avoid a compiler warning */
01541                 if ( DoubleLList((void ***)ppp,&AC.iBufferSize
01542                     ,sizeof(UBYTE),"statement buffer") ) {
01543                     *s = 0; retval = -1; AP.iBufError = 1;
01544                 }
01545                 AC.iPointer = AC.iBuffer + position2;
01546                 AC.iStop = AC.iBuffer + AC.iBufferSize-2;
01547                 s = AC.iBuffer + position;
01548             }
01549             if ( AP.iBufError ) {
01550                 for(;;){
01551                     c = GetChar(0);
01552                     if ( c == ENDOFINPUT ) { retval = -1; break; }
01553                     if ( c == '"' ) {
01554                         if ( stringlevel > 0 ) stringlevel = 0;
01555                         else stringlevel = 1;
01556                     }
01557                     else if ( c == LINEFEED && !stringlevel ) { retval = -2; break; }
01558                     else if ( c == ';' && !stringlevel ) {
01559                         while ( ( c = GetChar(0) ) == ' ' || c == '\\t' ) {}
01560                         if ( c != LINEFEED ) UngetChar(c);
01561                         retval = -1;
01562                         break;
01563                     }
01564                     else if ( c == '\\\\' ) c = GetChar(0);
01565                 }
01566                 break;
01567             }
01568         }
01569         *s++ = c;
01570     }
01571     AC.iPointer = s;
01572     *s = 0;
01573     if ( stringlevel > 0 ) {
01574         MesPrint("@Unbalanced \". Runaway string");
01575         retval = -1;
01576     }
01577     if ( retval == 1 ) {
01578         if ( ExpandTripleDots(0) < 0 ) retval = -1;
01579     }

```

```

01577     return(retval);
01578 }
01579
01580 /*
01581     #] LoadStatement :
01582     #[ ExpandTripleDots :
01583 */
01584
01585 static inline int IsSignChar(UBYTE c)
01586 {
01587     return c == '+' || c == '-';
01588 }
01589
01590 static inline int IsAlphanumericChar(UBYTE c)
01591 {
01592     return FG.cTable[c] == 0 || FG.cTable[c] == 1;
01593 }
01594
01595 static inline int CanParseSignedNumber(const UBYTE *s)
01596 {
01597     while ( IsSignChar(*s) ) s++;
01598     return FG.cTable[*s] == 1;
01599 }
01600
01601 int ExpandTripleDots(int par)
01602 {
01603     UBYTE *s, *s1, *s2, *n1, *n2, *t1, *t2, *startp, operator1, operator2, c, cc;
01604     UBYTE *nBuffer, *strngs, *Buffer, *Stop;
01605     LONG withquestion, x1, x2, y1, y2, number, inc, newsize, pow, fullsize;
01606     int i, error = 0, i1, i2, ii, *nums = 0;
01607
01608     if ( par == 0 ) {
01609         Buffer = AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]; Stop = AC.iStop;
01610     }
01611     else {
01612         Buffer = AP.preStart; Stop = AP.preStop;
01613     }
01614     s = Buffer; while ( *s ) s++;
01615     fullsize = s - Buffer;
01616     if ( fullsize < 7 ) return(error);
01617
01618     s = Buffer+2;
01619     while ( *s ) {
01620         if ( *s != '.' || ( s[-1] != ',' && FG.cTable[s[-1]] != 5 ) )
01621             { s++; continue; }
01622         if ( s[-1] == '%' || s[-1] == '^' || s[1] != '.' || s[2] != '.' )
01623             { s++; continue; }
01624         s1 = s - 2;
01625         s += 3;
01626         if ( *s != s[-4] && ( *s != '+' || s[-4] != '-' )
01627             && ( *s != '-' || s[-4] != '+' ) ) {
01628             MesPrint("&Improper operators for ...");
01629             error = -1;
01630         }
01631         operator1 = s[-4];
01632         operator2 = *s++;
01633         if ( operator1 == ':' ) operator1 = '.';
01634         if ( operator2 == ':' ) operator2 = '.';
01635     /*
01636         We have now O1...O2 (O stands for operator)
01637         Full syntax is
01638         [str]#1[?]O1...O2[str]#2[?] (Special case)
01639         in which both strings are identical and if one ? then also the other.
01640         <pattern1>O1...O2<pattern2> (General case)
01641         in which the difference in the patterns is just numerical.
01642     */
01643     s2 = s; /* the beginning of the second string */
01644     if ( *s2 != '<' || *s1 != '>' ) { /* Special case */
01645         startp = s1+1;
01646         withquestion = ( *s1 == '?' ); s1--;
01647         while ( FG.cTable[*s1] == 1 && s1 >= Buffer ) s1--;
01648         n1 = s1+1; /* Beginning of first number */
01649         if ( FG.cTable[*n1] != 1 ) {
01650             MesPrint("&No first number in ... operator");
01651             error = -1;
01652         }
01653         while ( FG.cTable[*s1] <= 1 && s1 >= Buffer ) s1--;
01654         s1++;
01655     /*
01656         We have now the first string from s1 to n1, number from n1
01657     */
01658     t1 = s1; t2 = s2;
01659     while ( t1 < n1 && *t1 == *t2 ) { t1++; t2++; }
01660     n2 = t2;
01661     if ( FG.cTable[*t2] != 1 ) {
01662         MesPrint("&No second number in ... operator");
01663         error = -1;

```

```

01664     }
01665     x2 = 0;
01666     while ( FG.cTable[*t2] == 1 ) x2 = 10*x2 + *t2++ - '0';
01667     x1 = 0;
01668     while ( FG.cTable[*t1] == 1 ) x1 = 10*x1 + *t1++ - '0';
01669     if ( withquestion != ( *t2 == '?' ) ) {
01670         MesPrint("&Improper use of ? in ... operator");
01671         if ( *t2 == '?' ) t2++;
01672         error = -1;
01673     }
01674     else if ( withquestion ) t2++;
01675     if ( FG.cTable[*t2] <= 2 ) {
01676         MesPrint("&Illegal object after ... construction");
01677         error = -1;
01678     }
01679     c = *nl; *nl = 0; s = t2;
01680     if ( error ) continue;
01681 /*
01682     At this point the syntax has been fulfilled. We have
01683     str in s1.
01684     x1,x2 are #1,#2
01685     operator1,operator2 are the two operators.
01686     s points at whatever comes after.
01687     Expansion will have to be computed.
01688 */
01689     if ( x2 < x1 ) { number = x1-x2; inc = -1; y1 = x2; y2 = x1; }
01690     else { number = x2-x1; inc = 1; y1 = x1; y2 = x2; }
01691     newsize = (number+1)*(nl-s1) /* the strings */
01692             + number /* the operators */
01693             +(number+1)*(withquestion?1:0) /* questionmarks */
01694             +(number+1); /* last digits */
01695     pow = 10;
01696     for ( i = 1; i < 10; i++, pow *= 10 ) {
01697         if ( y1 >= pow ) newsize += number+1;
01698         else if ( y2 >= pow ) newsize += y2-pow+1;
01699         else break;
01700     }
01701     while ( Buffer+(fullsize+newsize-(s-s1)) >= Stop ) {
01702         LONG strpos = s1-Buffer;
01703         LONG endstr = nl-Buffer;
01704         LONG startq = startp - Buffer;
01705         LONG position = s - Buffer;
01706         UBYTE *ppp;
01707         if ( par == 0 ) {
01708             LONG position2 = AC.iPointer - AC.iBuffer;
01709             ppp = &(AC.iBuffer); /* to avoid a compiler warning */
01710             if ( DoubleLList((void ***)ppp,&AC.iBufferSize
01711                             ,sizeof(UBYTE),"statement buffer") ) {
01712                 Terminate(-1);
01713             }
01714             AC.iPointer = AC.iBuffer + position2;
01715             AC.iStop = AC.iBuffer + AC.iBufferSize-2;
01716             Buffer = AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]; Stop = AC.iStop;
01717         }
01718         else {
01719             LONG fillpos = 0;
01720             if ( AP.preFill ) fillpos = AP.preFill - AP.preStart;
01721             ppp = &(AP.preStart); /* to avoid a compiler warning */
01722             if ( DoubleLList((void ***)ppp,&AP.pSize,sizeof(UBYTE),
01723                             "instruction buffer") ) {
01724                 Terminate(-1);
01725             }
01726             AP.preStop = AP.preStart + AP.pSize-3;
01727             if ( AP.preFill ) AP.preFill = fillpos + AP.preStart;
01728             Buffer = AP.preStart; Stop = AP.preStop;
01729         }
01730         s = Buffer + position;
01731         nl = Buffer + endstr;
01732         s1 = Buffer + strpos;
01733         startp = Buffer + startq;
01734     }
01735 /*
01736     We have space for the expansion in the buffer.
01737     There are two cases: new size > old size
01738                        old size >= new size
01739     Note that wherever we move things, it will be at least startp.
01740 */
01741     if ( newsize > (s-s1) ) {
01742         t2 = Buffer + fullsize;
01743         t1 = t2 + (newsize - (s-s1));
01744         *t1 = 0;
01745         while ( t2 > s ) { *--t1 = *--t2; }
01746     }
01747     else if ( newsize < (s-s1) ) {
01748         t1 = s1 + newsize; t2 = s; s = t1;
01749         while ( *t2 ) *t1++ = *t2++;
01750         *t1 = 0;

```

```

01751     }
01752     for ( x1 += inc, t1 = startp; number > 0; number--, x1 += inc ) {
01753         *t1++ = operator1;
01754         cc = operator1; operator1 = operator2; operator2 = cc;
01755         t2 = s1; while ( *t2 ) *t1++ = *t2++;
01756         x2 = x1; n2 = t1;
01757         do {
01758             *t1++ = '0' + x2 % 10;
01759             x2 /= 10;
01760         } while ( x2 );
01761         s2 = t1 - 1;
01762         while ( s2 > n2 ) { cc = *s2; *s2 = *n2; *n2++ = cc; s2--; }
01763         if ( withquestion ) *t1++ = '?';
01764     }
01765     fullsize += newsize - ( s - s1 );
01766     *n1 = c;
01767 }
01768 else { /* General case. Find the patterns first */
01769     t1 = s1; s1--;
01770     while ( s1 > Buffer ) {
01771         if ( *s1 == '<' ) break;
01772         s1--;
01773     }
01774     t2 = s2;
01775     while ( *t2 ) {
01776         if ( *t2 == '>' ) break;
01777         t2++;
01778     }
01779     if ( *s1 != '<' || *t2 != '>' ) {
01780         MesPrint("&Illegal attempt to use ... operator");
01781         return(-1);
01782     }
01783     s1++; s2++; /* Pointers to the patterns */
01784     nums = (int *)Malloca1((t1-s1)*2*(sizeof(int)+sizeof(UBYTE))
01785         , "Expand ...");
01786     strngs = (UBYTE *) (nums + 2*(t1-s1));
01787     n1 = s1; n2 = s2; ii = -1; i = 0;
01788     s = strngs;
01789     while ( n1 < t1 || n2 < t2 ) {
01790         /* Check the next characters can be parsed as numbers including signs. */
01791         if ( CanParseSignedNumber(n1) && CanParseSignedNumber(n2) ) {
01792             /*
01793              * Don't allow the cases that one has the sign and the other doesn't,
01794              * and the meaning changes without the sign. For example,
01795              * <f(1)>+...+<f(3)> Allowed
01796              * <f(-2)>+...+<f(2)> Allowed
01797              * <f(x-2)>+...+<f(x+2)> Allowed
01798              * <f(x-2)>+...+<f(x2)> Not allowed
01799              */
01800             int sign1 = IsSignChar(*n1);
01801             int sign2 = IsSignChar(*n2);
01802             int inword1 = s1 < n1 && IsAlphanumericChar(n1[-1]);
01803             int inword2 = s2 < n2 && IsAlphanumericChar(n2[-1]);
01804             if ( ( sign1 ^ sign2 ) && ( inword1 || inword2 ) ) break; /* Not allowed. */
01805             if ( sign1 || sign2 ) {
01806                 *s++ = '+'; /* Marker indicating we need the sign. */
01807             }
01808         } else {
01809             /* If they are not numbers, they should be same. */
01810             if ( *n1 == *n2 ) { *s++ = *n1++; n2++; continue; }
01811             else break;
01812         }
01813         ParseSignedNumber(x1, n1);
01814         ParseSignedNumber(x2, n2);
01815         if ( x1 == x2 ) {
01816             if ( s != strngs && ( s[-1] == '+' || s[-1] == '-' ) ) {
01817                 /* We need the sign. */
01818                 s--;
01819                 if ( x1 >= 0 ) {
01820                     *s++ = '+';
01821                 }
01822             }
01823             s = NumCopy(x1, s);
01824         }
01825         else {
01826             nums[2*i] = x1; nums[2*i+1] = x2;
01827             i++; *s++ = 0;
01828         }
01829     }
01830     if ( n1 < t1 || n2 < t2 ) {
01831         MesPrint("&Improper use of ... operator.");
01832 theend: M_free(nums, "Expand ...");
01833         return(-1);
01834     }
01835     *s = 0;
01836     if ( i == 0 ) ii = 0;
01837     else {

```

```

01838         ii = nums[0] - nums[1];
01839         if ( ii < 0 ) ii = -ii;
01840         for ( x1 = 1; x1 < i; x1++ ) {
01841             x2 = nums[2*x1]-nums[2*x1+1];
01842             if ( x2 < 0 ) x2 = -x2;
01843             if ( x2 != ii ) {
01844                 MesPrint("&Improper synchronization of numbers in ... operator");
01845                 goto theend;
01846             }
01847         }
01848     }
01849     ii++;
01850 /*
01851     We have now proper syntax.
01852     There are i+1 strings in strngs and i pairs of numbers
01853     in nums. Each time a start value and a finish value.
01854     We have ii steps. If ii <= 2, it will fit in the existing
01855     allocation. But this is hardly useful.
01856     We make a new allocation and copy from the old.
01857     Compute space.
01858 */
01859     x2 = s - strngs - i; /* -1 for end-of-string and +1 for the operator*/
01860     for ( i1 = 0; i1 < i; i1++ ) {
01861         i2 = nums[2*i1];
01862         x1 = nums[2*i1+1];
01863         if ( i2 < 0 ) i2 = -i2;
01864         if ( x1 < 0 ) x1 = -x1;
01865         if ( x1 > i2 ) i2 = x1;
01866         x1 = 2;
01867         while ( i2 > 0 ) { i2 /= 10; x1++; }
01868         x2 += x1;
01869     }
01870     x2 *= ii; /* Space for the expanded string (a bit more) */
01871     x2 += fullsize;
01872     x2 += 5; /* This will definitely hold everything */
01873     x2 += sizeof(UBYTE *);
01874     x2 = x2 - (x2 & (sizeof(UBYTE *)-1));
01875
01876     nBuffer = (UBYTE *)Malloc1(x2,"input buffer");
01877     n1 = nBuffer; s = Buffer; s1--;
01878     while ( s < s1 ) *n1++ = *s++;
01879 /*
01880     Solution of the special case that no comma was generated
01881     due to the presence of < to start the pattern.
01882     We get a comma when the word before ends in an alphanumeric
01883     character, a _ or a ] and the word inside starts with an
01884     alphanumeric character, a [ (or an _ (for future considerations))
01885 */
01886     if ( ( ( n1 > nBuffer ) && ( ( FG.cTable[n1[-1]] <= 1 )
01887     || ( n1[-1] == '_' ) || ( n1[-1] == ']' ) ) ) &&
01888     ( ( FG.cTable[strngs[0]] <= 1 ) || ( strngs[0] == '[' )
01889     || ( strngs[0] == '_' ) ) ) *n1++ = ',';
01890
01891     for ( i1 = 0; i1 < ii; i1++ ) {
01892         s = strngs; while ( *s ) *n1++ = *s++;
01893         for ( i2 = 0; i2 < i; i2++ ) {
01894             if ( n1 > nBuffer && IsSignChar(n1[-1]) ) {
01895                 /* We need the sign of counters. */
01896                 n1--;
01897                 if ( nums[2*i2] >= 0 ) {
01898                     *n1++ = '+';
01899                 }
01900             }
01901             n1 = NumCopy((WORD)(nums[2*i2]),n1);
01902             if ( nums[2*i2] > nums[2*i2+1] ) nums[2*i2]--;
01903             else nums[2*i2]++;
01904             s++; while ( *s ) *n1++ = *s++;
01905         }
01906         if ( ( i1 & 1 ) == 0 ) *n1++ = operator1;
01907         else *n1++ = operator2;
01908     }
01909     n1--; /* drop the trailing operator */
01910     s = t2 + 1; n2 = n1;
01911 /*
01912     Similar extra comma
01913 */
01914     if ( ( ( ( FG.cTable[n1[-1]] <= 1 )
01915     || ( n1[-1] == '_' ) || ( n1[-1] == ']' ) ) ) &&
01916     ( ( FG.cTable[s[0]] <= 1 ) || ( s[0] == '[' )
01917     || ( s[0] == '_' ) ) ) *n1++ = ',';
01918
01919     while ( *s ) *n1++ = *s++;
01920     *n1 = 0;
01921     if ( par == 0 ) {
01922         LONG nnn1 = n1-nBuffer;
01923         LONG nnn2 = n2-nBuffer;
01924         LONG nnn3;

```

```

01925         while ( AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel] + x2 >= AC.iStop ) {
01926             LONG position = s-Buffer;
01927             LONG position2 = AC.iPointer - AC.iBuffer;
01928             UBYTE **ppp;
01929             ppp = &(AC.iBuffer); /* to avoid a compiler warning */
01930             if ( DoubleLList((void ***)ppp,&AC.iBufferSize
01931                 ,sizeof(UBYTE),"statement buffer") ) {
01932                 Terminate(-1);
01933             }
01934             AC.iPointer = AC.iBuffer + position2;
01935             AC.iStop = AC.iBuffer + AC.iBufferSize-2;
01936             Buffer = AC.iBuffer+AP.PreAssignStack[AP.PreAssignLevel]; Stop = AC.iStop;
01937             s = Buffer + position;
01938         }
01939     /*
01940         This can be improved. We only have to start from the first term.
01941     */
01942     for ( nnn3 = 0; nnn3 < nnn1; nnn3++ ) Buffer[nnn3] = nBuffer[nnn3];
01943     Buffer[nnn3] = 0;
01944     n1 = Buffer + nnn1;
01945     n2 = Buffer + nnn2;
01946     M_free(nBuffer,"input buffer");
01947     M_free(nums,"Expand ...");
01948 }
01949 else { /* Comes here only inside a real preprocessor instruction */
01950     AP.preStop = nBuffer + x2 - 2;
01951     AP.pSize = x2;
01952     M_free(AP.preStart,"input buffer");
01953     M_free(nums,"Expand ...");
01954     AP.preStart = nBuffer;
01955     Buffer = AP.preStart; Stop = AP.preStop;
01956 }
01957 fullsize = n1 - Buffer;
01958 s = n2;
01959 }
01960 }
01961 return(error);
01962 }
01963
01964 /*
01965     #] ExpandTripleDots :
01966     #[ FindKeyWord :
01967 */
01968
01969 KEYWORD *FindKeyWord(UBYTE *theword, KEYWORD *table, int size)
01970 {
01971     int low,med,hi;
01972     UBYTE *s1, *s2;
01973     low = 0;
01974     hi = size-1;
01975     while ( hi >= low ) {
01976         med = (hi+low)/2;
01977         s1 = (UBYTE *) (table[med].name);
01978         s2 = theword;
01979         while ( *s1 && tolower(*s1) == tolower(*s2) ) { s1++; s2++; }
01980         if ( *s1 == 0 &&
01981             /*[30apr2004 mt]:*/
01982             /* The bug!!:
01983                 FG.cTable[*s2] != 1 && FG.cTable[*s2] != 2
01984             */
01985                 FG.cTable[*s2] != 0 && FG.cTable[*s2] != 1
01986             /*
01987                 ( *s2 == ' ' || *s2 == '\t' || *s2 == 0 || *s2 == ',' || *s2 == '(' ) */
01988             )
01989             return(table+med);
01990         if ( tolower(*s2) > tolower(*s1) ) low = med+1;
01991         else hi = med - 1;
01992     }
01993     return(0);
01994 }
01995
01996 /*
01997     #] FindKeyWord :
01998     #[ FindInKeyWord :
01999 */
02000 KEYWORD *FindInKeyWord(UBYTE *theword, KEYWORD *table, int size)
02001 {
02002     int i;
02003     UBYTE *s1, *s2;
02004     for ( i = 0; i < size; i++ ) {
02005         s1 = (UBYTE *) (table[i].name);
02006         s2 = theword;
02007         while ( *s1 && tolower(*s1) == tolower(*s2) ) { s1++; s2++; }
02008         if ( *s2 == 0 || *s2 == ' ' || *s2 == ',' || *s2 == '\t' )
02009             return(table+i);
02010     }
02011     return(0);

```

```

02012 }
02013
02014 /*
02015     #] FindInKeyWord :
02016     #[ TheDefine :
02017 */
02018
02030 int TheDefine(UBYTE *s, int mode)
02031 {
02032     UBYTE *name, *value, *valpoin, *args = 0, c;
02033     if ( ( mode & 2 ) == 0 ) {
02034         if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02035         if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02036     }
02037     else { mode &= ~2; }
02038     name = s;
02039     if ( chartype[*s] != 0 ) goto illname;
02040     s++;
02041     while ( chartype[*s] <= 1 ) s++;
02042     value = s;
02043     while ( *s == ' ' || *s == '\t' ) s++;
02044     c = *s; *value = 0;
02045     if ( c == 0 ) {
02046         if ( PutPreVar(name, (UBYTE *) "1", 0, mode) < 0 ) return(-1);
02047         return(0);
02048     }
02049     if ( c == '(' ) { /* arguments. scan for correctness */
02050         s++; args = s;
02051         for (;;) {
02052             if ( chartype[*s] != 0 ) goto illarg;
02053             s++;
02054             while ( chartype[*s] <= 1 ) s++;
02055             while ( *s == ' ' || *s == '\t' ) s++;
02056             if ( *s == ')' ) break;
02057             if ( *s != ',' ) goto illargs;
02058             s++;
02059             while ( *s == ' ' || *s == '\t' ) s++;
02060         }
02061         *s++ = 0;
02062         while ( *s == ' ' || *s == '\t' ) s++;
02063         c = *s;
02064     }
02065     if ( c == '"' ) {
02066         s++; valpoin = value = s;
02067         while ( *s != '"' ) {
02068             if ( *s == '\\' ) {
02069                 if ( s[1] == 'n' ) { *valpoin++ = LINEFEED; s += 2; }
02070                 else if ( s[1] == '"' ) { *valpoin++ = '"'; s += 2; }
02071                 else if ( s[1] == 0 ) goto illval;
02072                 else { *valpoin++ = *s++; *valpoin++ = *s++; }
02073             }
02074             else *valpoin++ = *s++;
02075         }
02076         *valpoin = 0;
02077         if ( PutPreVar(name, value, args, mode) < 0 ) return(-1);
02078     }
02079     else {
02080         MesPrint("@Illegal string for preprocessor variable %s. Forgotten double quotes (\") ?", name);
02081         return(-1);
02082     }
02083     return(0);
02084 illname:;
02085     MesPrint("@Illegally formed name of preprocessor variable");
02086     return(-1);
02087 illarg:;
02088     MesPrint("@Illegally formed name of argument of preprocessor definition");
02089     return(-1);
02090 illargs:;
02091     MesPrint("@Illegally formed arguments of preprocessor definition");
02092     return(-1);
02093 illval:;
02094     MesPrint("@Illegal valpoin for preprocessor variable %s", name);
02095     return(-1);
02096 }
02097
02098 /*
02099     #] TheDefine :
02100     #[ DoCommentChar :
02101 */
02102
02103 int DoCommentChar(UBYTE *s)
02104 {
02105     UBYTE c;
02106     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02107     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02108     while ( *s == ' ' || *s == '\t' ) s++;
02109     if ( *s == 0 || *s == '\n' ) {

```



```

02110         MesPrint("@No valid comment character specified");
02111         return(-1);
02112     }
02113     c = *s++;
02114     while ( *s == ' ' || *s == '\t' ) s++;
02115     if ( *s != 0 && *s != '\n' ) {
02116         MesPrint("@Comment character should be a single valid character");
02117         return(-1);
02118     }
02119     AP.ComChar = c;
02120     return(0);
02121 }
02122
02123 /*
02124     #] DoCommentChar :
02125     #[ DoPreAssign :
02126
02127     Routine assigns a 'value' to a $variable.
02128     Syntax: #assign
02129           next line(s) a statement of the type
02130           $name = expression;
02131     Note: at the moment of the assign there cannot be an 'open' statement.
02132 */
02133
02134 int DoPreAssign(UBYTE *s)
02135 {
02136     int error = 0;
02137     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) {
02138         return(0);
02139     }
02140     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) {
02141         return(0);
02142     }
02143     if ( *s ) {
02144         MesPrint("@Illegal characters in #assign instruction");
02145         error = 1;
02146     }
02147     PUSHPREASSIGNLEVEL;
02148     AP.PreAssignFlag = 1;
02149 /*
02150     if ( AP.PreContinuation ) {
02151         MesPrint("@Assign instructions cannot occur inside statements");
02152         MesPrint("@Missing ; ?");
02153         AP.PreContinuation = 0;
02154         error = 1;
02155     }
02156 */
02157     return(error);
02158 }
02159
02160 /*
02161     #] DoPreAssign :
02162     #[ DoDefine :
02163 */
02164
02165 int DoDefine(UBYTE *s)
02166 {
02167     return(TheDefine(s,0));
02168 }
02169
02170 /*
02171     #] DoDefine :
02172     #[ DoRedefine :
02173 */
02174
02175 int DoRedefine(UBYTE *s)
02176 {
02177     return(TheDefine(s,1));
02178 }
02179
02180 /*
02181     #] DoRedefine :
02182     #[ ClearMacro :
02183
02184     Undefines the arguments of a macro after its use.
02185 */
02186
02187 int ClearMacro(UBYTE *name)
02188 {
02189     int i;
02190     PREVAR *p;
02191     UBYTE *s;
02192     for ( i = NumPre-1, p = &(PreVar[NumPre-1]); i >= 0; i--, p-- ) {
02193         if ( StrCmp(name,p->name) == 0 ) break;
02194     }
02195     if ( i < 0 ) return(-1);
02196     if ( p->nargs <= 0 ) return(0);

```

```

02197     s = p->argnames;
02198     for ( i = 0; i < p->nargs; i++ ) {
02199         TheUndefine(s);
02200         while ( *s ) s++;
02201         s++;
02202     }
02203     return(0);
02204 }
02205
02206 /*
02207     #] ClearMacro :
02208     #[ TheUndefine :
02209
02210     There is a complication here. If there are redefine statements
02211     they will be pointing at the wrong variable if their number is
02212     greater than the number of the variable we pop.
02213 */
02214
02215 int TheUndefine(UBYTE *name)
02216 {
02217     int i, inum, error = 0;
02218     PREVAR *p;
02219     for ( i = NumPre-1, p = &(PreVar[NumPre-1]); i >= 0; i--, p-- ) {
02220         if ( StrCmp(name,p->name) == 0 ) {
02221             M_free(p->name,"undefining PreVar");
02222             NumPre--;
02223             inum = i;
02224             while ( i < NumPre ) {
02225                 p->name = p[1].name;
02226                 p->value = p[1].value;
02227                 p++; i++;
02228             }
02229             p->name = 0; p->value = 0;
02230             {
02231                 CBUF *CC = cbuf + AC.cbunum;
02232                 int j, k;
02233                 for ( j = 1; j <= CC->numlhs; j++ ) {
02234                     if ( CC->lhs[j][0] == TYPEREDEFPRE ) {
02235                         if ( CC->lhs[j][2] > inum ) CC->lhs[j][2]--;
02236                         else if ( CC->lhs[j][2] == inum ) {
02237                             for ( k = inum - 1; k >= 0; k-- )
02238                                 if ( StrCmp(name, PreVar[k].name) == 0 ) break;
02239                             if ( k >= 0 ) CC->lhs[j][2] = k;
02240                             else {
02241                                 MesPrint("@Conflict between undefining a preprocessor variable and a
02242                                 redefine statement");
02243                                 error = 1;
02244                             }
02245                         }
02246                     }
02247                 }
02248                 #ifdef PARALLELCODE
02249                 for ( j = 0; j < AC.numfirstnum; j++ ) {
02250                     if ( AC.pfirstnum[j] > inum ) AC.pfirstnum[j]--;
02251                     else if ( AC.pfirstnum[j] == inum ) {
02252                         for ( k = inum - 1; k >= 0; k-- )
02253                             if ( StrCmp(name, PreVar[k].name) == 0 ) break;
02254                         if ( k >= 0 ) AC.pfirstnum[j] = k;
02255                     }
02256                 }
02257                 #endif
02258                 break;
02259             }
02260         }
02261     }
02262     return(error);
02263 }
02264
02265 /*
02266     #] TheUndefine :
02267     #[ DoUndefine :
02268 */
02269 int DoUndefine(UBYTE *s)
02270 {
02271     UBYTE *name, *t;
02272     int error = 0, retval;
02273     /*
02274     int i;
02275     PREVAR *p;
02276 */
02277     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02278     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02279     name = s;
02280     if ( chartype[*s] != 0 ) goto illname;
02281     s++;
02282     while ( chartype[*s] <= 1 ) s++;

```

```

02283     t = s;
02284     if ( *s && *s != ' ' && *s != '\t' ) goto illname;
02285     while ( *s == ' ' || *s == '\t' ) s++;
02286     if ( *s ) {
02287         MesPrint("@Undefine should just have a variable name");
02288         error = -1;
02289     }
02290     *t = 0;
02291     if ( ( retval = TheUndefine(name) ) != 0 ) {
02292         if ( error == 0 ) return(retval);
02293         if ( error > 0 ) error = retval;
02294     }
02295     /*
02296     for ( i = NumPre-1, p = &(PreVar[NumPre-1]); i >= 0; i--, p-- ) {
02297         if ( StrCmp(name,p->name) == 0 ) {
02298             M_free(p->name,"undefining PreVar");
02299             NumPre--;
02300             while ( i < NumPre ) {
02301                 p->name = p[i].name;
02302                 p->value = p[i].value;
02303                 p++; i++;
02304             }
02305             p->name = 0; p->value = 0;
02306             break;
02307         }
02308     }
02309     */
02310     return(error);
02311 illname:;
02312     MesPrint("@Illegally formed name of preprocessor variable");
02313     return(-1);
02314 }
02315
02316 /*
02317     #] DoUndefine :
02318     #[ DoInclude :
02319 */
02320
02321 int DoInclude(UBYTE *s) { return(Include(s,FILESTREAM)); }
02322
02323 /*
02324     #] DoInclude :
02325     #[ DoReverseInclude :
02326 */
02327
02328 int DoReverseInclude(UBYTE *s) { return(Include(s,REVERSEFILESTREAM)); }
02329
02330 /*
02331     #] DoReverseInclude :
02332     #[ Include :
02333 */
02334
02335 int Include(UBYTE *s, int type)
02336 {
02337     UBYTE *name = s, *fold, *t, c, c1 = 0, c2 = 0, c3 = 0;
02338     int strloffset, withnolist = AC.NoShowInput;
02339     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02340     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02341     if ( *s == '-' || *s == '+' ) {
02342         if ( *s == '-' ) withnolist = 1;
02343         else withnolist = 0;
02344         s++;
02345         while ( *s == ' ' || *s == '\t' ) s++;
02346         name = s;
02347     }
02348     if ( *s == '"' ) {
02349         while ( *s && *s != '"' ) {
02350             if ( *s == '\\' ) s++;
02351             s++;
02352         }
02353         t = s++;
02354     }
02355     else {
02356         while ( *s && *s != ' ' && *s != '\t' ) {
02357             if ( *s == '\\' ) s++;
02358             s++;
02359         }
02360         t = s;
02361     }
02362     while ( *s == ' ' || *s == '\t' ) s++;
02363     if ( *s == '#' ) {
02364         *t = 0;
02365         s++;
02366         while ( *s == ' ' || *s == '\t' ) s++;
02367         fold = s;
02368         if ( *s == 0 ) {
02369             MesPrint("@Empty fold name");

```

```

02370         return(-1);
02371     }
02372     continue_fold:
02373     while ( *s && *s != ' ' && *s != '\t' ) {
02374         if ( *s == '\\' ) s++;
02375         s++;
02376     }
02377     t = s;
02378     while ( *s == ' ' || *s == '\t' ) s++;
02379     if ( *s ) {
02380         /*
02381          * A non-whitespace character is found. Continue parsing the fold.
02382          */
02383         goto continue_fold;
02384     }
02385 }
02386 else if ( *s == 0 ) {
02387     fold = 0;
02388 }
02389 else {
02390     MesPrint("@Improper syntax for file name");
02391     return(-1);
02392 }
02393 *t = 0;
02394 if ( fold ) {
02395     fold = strDup1(fold,"foldname");
02396 }
02397 /*
02398  We have the name of the file in 'name' and the fold in 'fold' (or NULL)
02399 */
02400 if ( OpenStream(name,type,0,PRENOACTION) == 0 ) {
02401     if ( fold ) { M_free(fold,"foldname"); fold = 0; }
02402     return(-1);
02403 }
02404 if ( fold ) {
02405     LONG position = -1;
02406     int foldopen = 0;
02407     LONG linenum = 0, prevline = 0;
02408     name = strDup1(name,"name of include file");
02409     AC.CurrentStream->FoldName = strDup1(fold,"name of fold");
02410     AC.NoShowInput++;
02411     for(;;) {
02412         c = GetFromStream(AC.CurrentStream);
02413         if ( c == EOFSTREAM ) {
02414             AC.CurrentStream = CloseStream(AC.CurrentStream);
02415             goto nofold;
02416         }
02417         if ( c == AP.ComChar ) {
02418             strloffset = AC.CurrentStream-AC.Streams;
02419             LoadInstruction(1);
02420             if ( AC.CurrentStream != strloffset+AC.Streams ) {
02421                 c = EOFSTREAM;
02422             }
02423             else {
02424                 t = AP.preStart;
02425                 if ( t[2] == '#' && ( t[3] == '[' && !foldopen )
02426                     || ( t[3] == ']' && foldopen ) ) {
02427                     t += 4;
02428                     while ( *t == ' ' || *t == '\t' ) t++;
02429                     s = AC.CurrentStream->FoldName;
02430                     while ( *s == *t ) { s++; t++; }
02431                     if ( *s == 0 && ( *t == ' ' || *t == '\t'
02432                         || *t == ':' ) ) {
02433                         while ( *t == ' ' || *t == '\t' ) t++;
02434                         if ( *t == ':' ) {
02435                             if ( foldopen == 0 ) {
02436                                 foldopen = 1;
02437                                 position = GetStreamPosition(AC.CurrentStream);
02438                                 linenum = AC.CurrentStream->linenum;
02439                                 prevline = AC.CurrentStream->prevline;
02440                                 c3 = AC.CurrentStream->isnextchar;
02441                                 c1 = AC.CurrentStream->nextchar[0];
02442                                 c2 = AC.CurrentStream->nextchar[1];
02443                             }
02444                             else {
02445                                 foldopen = 0;
02446                                 PositionStream(AC.CurrentStream,position);
02447                                 AC.CurrentStream->linenum = linenum;
02448                                 AC.CurrentStream->prevline = prevline;
02449                                 AC.CurrentStream->eqnum = 1;
02450                                 AC.NoShowInput--;
02451                                 AC.CurrentStream->isnextchar = c3;
02452                                 AC.CurrentStream->nextchar[0] = c1;
02453                                 AC.CurrentStream->nextchar[1] = c2;
02454                                 break;
02455                             }
02456                         }

```

```

02457     }
02458     }
02459     }
02460     }
02461     else {
02462         while ( c != LINEFEED && c != ENDOFSTREAM ) {
02463             c = GetFromStream(AC.CurrentStream);
02464             if ( c == ENDOFSTREAM ) {
02465                 AC.CurrentStream = CloseStream(AC.CurrentStream);
02466                 break;
02467             }
02468         }
02469     }
02470     if ( c == ENDOFSTREAM ) {
02471 nofold:
02472         MesPrint("@Cannot find fold %s in file %s",fold,name);
02473         UngetChar(c);
02474         AC.NoShowInput--;
02475         M_free(name,"name of include file");
02476         Terminate(-1);
02477     }
02478 }
02479 M_free(name,"name of include file");
02480 }
02481 AC.NoShowInput = withnolist;
02482 if ( fold ) { M_free(fold,"foldname"); fold = 0; }
02483 return(0);
02484 }
02485
02486 /*
02487 #] Include :
02488 #[ DoPreExchange :
02489
02490 Exchanges the names of expressions or the contents of dollars
02491 Syntax:
02492     #exchange expr1,expr2
02493     #exchange $var1,$var2
02494 */
02495
02496 int DoPreExchange(UBYTE *s)
02497 {
02498     int error = 0;
02499     UBYTE *s1, *s2;
02500     WORD num1, num2;
02501     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02502     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02503     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
02504     if ( *s == '$' ) {
02505         s++; s1 = s; while ( FG.cTable[*s] <= 1 ) s++;
02506         if ( *s != ',' && *s != ' ' && *s != '\t' ) goto syntax;
02507         *s++ = 0;
02508         while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
02509         if ( *s != '$' ) goto syntax;
02510         s++; s2 = s; while ( FG.cTable[*s] <= 1 ) s++;
02511         if ( *s != 0 && *s != ';' ) goto syntax;
02512         *s = 0;
02513         if ( ( num1 = GetDollar(s1) ) <= 0 ) {
02514             MesPrint("@$%s has not been defined (yet)",s1);
02515             error = 1;
02516         }
02517         if ( ( num2 = GetDollar(s2) ) <= 0 ) {
02518             MesPrint("@$%s has not been defined (yet)",s2);
02519             error = 1;
02520         }
02521         if ( error == 0 ) {
02522             ExchangeDollars((int)num1,(int)num2);
02523         }
02524     }
02525     else {
02526         s1 = s; s = SkipAName(s);
02527         if ( *s != ',' && *s != ' ' && *s != '\t' ) goto syntax;
02528         *s++ = 0;
02529         while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
02530         if ( FG.cTable[*s] != 0 && *s != '[' ) goto syntax;
02531         s2 = s; s = SkipAName(s);
02532         if ( *s != 0 && *s != ';' ) goto syntax;
02533         *s = 0;
02534         if ( GetName(AC.exprnames,s1,&num1,NOAUTO) != CEXPRESSION ) {
02535             MesPrint("@%s is not an expression",s1);
02536             error = 1;
02537         }
02538         if ( GetName(AC.exprnames,s2,&num2,NOAUTO) != CEXPRESSION ) {
02539             MesPrint("@%s is not an expression",s2);
02540             error = 1;
02541         }
02542         if ( error == 0 ) {
02543             ExchangeExpressions((int)num1,(int)num2);

```

```

02544     }
02545     }
02546     return(error);
02547 syntax:
02548     MesPrint("@Proper syntax: %#exchange expr1,expr2 or %#exchange $var1,$var2");
02549     return(1);
02550 }
02551
02552 /*
02553     #] DoPreExchange :
02554     #[ DoCall :
02555 */
02556
02557 int DoCall(UBYTE *s)
02558 {
02559     UBYTE *t, *u, *v, *name, c, cp, *args1, *args2, *t1, *t2, *wild = 0;
02560     int bratype = 0, wildargs = 0, inwildargs = 0, nwildargs = 0;
02561     PROCEDURE *p;
02562     int streamoffset;
02563     int i, namesize, narg1, narg2, bralevel, numpre;
02564     LONG i1, i2;
02565     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02566     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02567 /*
02568     1: Get the name of the procedure.
02569     2: Locate the procedure.
02570 */
02571     name = s; s = EndOfToken(s); c = *s; *s = 0;
02572     for ( i = NumProcedures-1; i >= 0; i-- ) {
02573         if ( StrCmp(Procedures[i].name,name) == 0 ) break;
02574     }
02575     // AP.ProcList.num = NumProcedures is incremented inside FromList
02576     p = (PROCEDURE *)FromList(&AP.ProcList);
02577     if ( i < 0 ) { /* Try to find a file */
02578         namesize = 0;
02579         t = name;
02580         while ( *t ) { t++; namesize++; }
02581         t = AP.procedureExtension;
02582         while ( *t ) { t++; namesize++; }
02583         t = p->name = (UBYTE *)Malloc1(namesize+2,"procedure");
02584         u = name;
02585         while ( *u ) *t++ = *u++;
02586         *t++ = '.';
02587         v = AP.procedureExtension;
02588         while ( *v ) *t++ = *v++;
02589         *t = 0;
02590         p->loadmode = 0; /* buffer should be freed at end */
02591         p->mustfree = 1;
02592         p->p.buffer = LoadInputFile(p->name,PROCEDUREFILE);
02593         if ( p->p.buffer == 0 ) return(-1);
02594         t[-4] = 0;
02595     }
02596     else {
02597         p->p.buffer = Procedures[i].p.buffer;
02598         p->name = Procedures[i].name;
02599         p->loadmode = 1;
02600         p->mustfree = 0; // this is just a copy of pointers to a permanently stored procedure
02601     }
02602     t = p->p.buffer;
02603     SKIPBLANKS(t)
02604     if ( *t++ != '#' ) goto wrongfile;
02605     SKIPBLANKS(t)
02606     t += 9;
02607     SKIPBLANKS(t)
02608     u = EndOfToken(t);
02609     cp = *u; *u = 0;
02610     if ( StrCmp(t,name) != 0 ) goto wrongfile;
02611     *u = cp;
02612     *s = c;
02613 /*
02614     The pointer p points to the contents of the procedure (in memory)
02615     Now we have to match the arguments. u points to after the name
02616     in the 'file', s to after the name in the call statement.
02617 */
02618     bralevel = narg1 = narg2 = 0; args2 = u;
02619     SKIPBLANKS(u)
02620     if ( *u == '(' ) {
02621         u++; SKIPBLANKS(u)
02622         args2 = u;
02623         while ( *u != ')' ) {
02624             if ( *u == '?' ) { wildargs++; u++; nwildargs = narg2+1; }
02625             narg2++; u = EndOfToken(u); SKIPBLANKS(u)
02626             if ( *u == ',' ) { u++; SKIPBLANKS(u) }
02627             else if ( *u != ')' || ( wildargs > 1 ) ) {
02628                 MesPrint("@Illegal argument field in procedure %s",p->name);
02629                 return(-1);
02630             }
02631         }
02632     }

```

```

02631     }
02632 }
02633 while ( *u != LINEFEED ) u++;
02634 SKIPBLANKS(s)
02635 args1 = s+1;
02636 if ( *s == '(' ) bratype = 1;
02637 do {
02638     if ( *s == '{' && bratype == 0 ) bralevel++;
02639     else if ( *s == '(' && bratype == 1 ) bralevel++;
02640     else if ( *s == '}' && bratype == 0 ) {
02641         bralevel--;
02642         if ( bralevel == 0 ) {
02643             *s = 0; nargs1++;
02644             if ( wildargs && nargs1 == nwildargs ) wild = s;
02645         }
02646     }
02647     else if ( *s == ')' && bratype == 1 ) {
02648         bralevel--;
02649         if ( bralevel == 0 ) {
02650             *s = 0; nargs1++;
02651             if ( wildargs && nargs1 == nwildargs ) wild = s;
02652         }
02653     }
02654     /*[12dec2003 mt]:*/
02655     /*else if ( *s == ',' || *s == '|' ) {*/
02656     else if ( set_in(*s,AC.separators) ) { /*Function set_in see in
02657                                         file tools.c*/
02658         /*:[12dec2003 mt]:*/
02659         *s = 0; nargs1++;
02660         if ( wildargs && nargs1 == nwildargs ) wild = s;
02661     }
02662     else if ( *s == '\\\\' ) s++;
02663     s++;
02664 } while ( bralevel > 0 );
02665 if ( wildargs && nargs1 >= nargs2-1 ) {
02666     inwildargs = nargs1-nargs2+1;
02667     if ( inwildargs == 0 ) nwildargs = 0;
02668     else {
02669         while ( inwildargs > 1 ) {
02670             *wild = ',';
02671             while ( *wild ) wild++;
02672             inwildargs--;
02673         }
02674     }
02675 }
02676 else if ( nargs1 != nargs2 && ( nargs2 != 0 || nargs1 != 1 || *args1 != 0 ) ) {
02677     MesPrint("@Arguments of procedure %s are not matching",p->name);
02678     return(-1);
02679 }
02680 numpre = -NumPre-1; /* For the stream */
02681 for ( i = 0; i < nargs2; i++ ) {
02682     t = args2;
02683     if ( *t == '?' ) {
02684         args2++;
02685     }
02686     if ( *t == '?' && inwildargs == 0 ) {
02687         args2 = EndOfToken(args2); c = *args2; *args2 = 0;
02688         if ( PutPreVar(t,(UBYTE *)"",0,0) < 0 ) return(-1);
02689     }
02690     else {
02691         args2 = EndOfToken(args2); c = *args2; *args2 = 0;
02692         t1 = t2 = args1;
02693         while ( *t1 ) {
02694             if ( *t1 == '\\\\' ) t1++;
02695             if ( t1 != t2 ) *t2 = *t1;
02696             t2++; t1++;
02697         }
02698         *t2 = 0;
02699         if ( PutPreVar(t,args1,0,0) < 0 ) return(-1);
02700         args1 = t1+1; /* Next argument */
02701     }
02702     *args2 = c; SKIPBLANKS(args2) /* skip to next name */
02703     args2++; SKIPBLANKS(args2)
02704 }
02705 streamoffset = AC.CurrentStream - AC.Streams;
02706 args1 = AC.CurrentStream->name;
02707 AC.CurrentStream->name = p->name;
02708 i1 = AC.CurrentStream->linenumber;
02709 i2 = AC.CurrentStream->prevline;
02710 AC.CurrentStream->prevline =
02711 AC.CurrentStream->linenumber = 2;
02712 OpenStream(u+1,PREREADSTREAM3,numpre,PRENOACTION);
02713 AC.Streams[streamoffset].name = args1;
02714 AC.Streams[streamoffset].linenumber = i1;
02715 AC.Streams[streamoffset].prevline = i2;
02716 AddToPreTypes(PRETYPEPROCEDURE);
02717 return(0);

```

```

02718 wrongfile;;
02719     if ( i < 0 ) MesPrint("@File %s is not a proper procedure",p->name);
02720     else MesPrint("!!!Internal error with procedure names: %s",name);
02721     return(-1);
02722 }
02723
02724 /*
02725     #] DoCall :
02726     #[ DoDebug :
02727 */
02728
02729 int DoDebug(UBYTE *s)
02730 {
02731     int x;
02732     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02733     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02734     NeedNumber(x,s,nonumber)
02735     if ( x < 0 || x > (PREPROONLY
02736         | DUMPTOCOMPILER
02737         | DUMPOUTTERMS
02738         | DUMPINTERMS
02739         | DUMPTOSORT
02740         | DUMPTOPARALLEL
02741 #ifdef WITHPTHREADS
02742         | THREADSDEBUG
02743 #endif
02744     ) ) goto nonumber;
02745     AP.PreDebug = 0;
02746     if ( ( x & PREPROONLY ) != 0 ) AP.PreDebug |= PREPROONLY; /* 1 */
02747     if ( ( x & DUMPTOCOMPILER ) != 0 ) AP.PreDebug |= DUMPTOCOMPILER; /* 2 */
02748     if ( ( x & DUMPOUTTERMS ) != 0 ) AP.PreDebug |= DUMPOUTTERMS; /* 4 */
02749     if ( ( x & DUMPINTERMS ) != 0 ) AP.PreDebug |= DUMPINTERMS; /* 8 */
02750     if ( ( x & DUMPTOSORT ) != 0 ) AP.PreDebug |= DUMPTOSORT; /* 16 */
02751     if ( ( x & DUMPTOPARALLEL ) != 0 ) AP.PreDebug |= DUMPTOPARALLEL; /* 32 */
02752 #ifdef WITHPTHREADS
02753     if ( ( x & THREADSDEBUG ) != 0 ) AP.PreDebug |= THREADSDEBUG; /* 64 */
02754 #endif
02755     return(0);
02756 nonumber:
02757     MesPrint("@Illegal argument for debug instruction");
02758     return(1);
02759 }
02760
02761 /*
02762     #] DoDebug :
02763     #[ DoTerminate :
02764 */
02765
02766 int DoTerminate(UBYTE *s)
02767 {
02768     int x;
02769     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02770     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02771     if ( *s ) {
02772         NeedNumber(x,s,nonumber)
02773         Terminate(x);
02774     }
02775     else {
02776         Terminate(-1);
02777     }
02778     return(0);
02779 nonumber:
02780     MesPrint("@Illegal argument for terminate instruction");
02781     return(1);
02782 }
02783
02784 /*
02785     #] DoTerminate :
02786     #[ DoContinueDo :
02787 */
02788
02800 int DoContinueDo(UBYTE *s)
02801 {
02802     DOLOOP *loop;
02803     WORD levels;
02804     int result;
02805
02806     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02807     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02808
02809     if ( NumDoLoops <= 0 ) {
02810         MesPrint("@%#continuedo without %#do");
02811         return(1);
02812     }
02813
02814     SkipSpaces(&s);
02815     if ( *s == 0 ) {

```



```

02816         levels = 1;
02817     }
02818     else if ( FG.cTable[*s] == 1 ) {
02819         ParseNumber(levels,s);
02820         SkipSpaces(&s);
02821         if ( *s != 0 ) goto improper;
02822     }
02823     else {
02824 improper:
02825         MesPrint("@Improper syntax of %#continuedo instruction");
02826         return(1);
02827     }
02828
02829     if ( levels > NumDoLoops ) {
02830         MesPrint("@Too many loop levels requested in %#continuedo instruction");
02831         return(1);
02832     }
02833
02834     result = ExitDoLoops(levels-1,"continuedo");
02835     if ( result != 0 ) return(result);
02836
02837     if ( levels <= 0 ) return(0);
02838
02839     if ( AC.CurrentStream->type == PREREADSTREAM3
02840         || AP.PreTypes[AP.NumPreTypes] == PRETYPEPROCEDURE ) {
02841         MesPrint("@Trying to jump out of a procedure with a %#continuedo instruction");
02842         return(1);
02843     }
02844
02845     loop = &(DoLoops[NumDoLoops-1]);
02846     AP.NumPreTypes = loop->NumPreTypes+1;
02847     AP.PreIfLevel = loop->PreIfLevel;
02848     AP.PreSwitchLevel = loop->PreSwitchLevel;
02849
02850     return(DoEnddo(s));
02851 }
02852
02853 /*
02854     #] DoContinueDo :
02855     #[ DoDo :
02856
02857     The do loop has three varieties:
02858     #do i = num1,num2 [,num3]
02859     #do i = {string1,string2,...,stringn}
02860         The | as separator is also allowed for backwards compatibility
02861     #do i = expression      One by one all terms of the expression
02862 */
02863
02864 int DoDo(UBYTE *s)
02865 {
02866     GETIDENTITY
02867     UBYTE *t, c, *u, *uu;
02868     DOLOOP *loop;
02869     WORD expnum;
02870     LONG linenum = AC.CurrentStream->linenumber;
02871     int oldNoShowInput = AC.NoShowInput, i, oldpreassignflag;
02872
02873     if ( ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH )
02874         || ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) ) {
02875         if ( PreSkip((UBYTE *) "do", (UBYTE *) "enddo",1) ) return(-1);
02876         return(0);
02877     }
02878
02879     /*
02880     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
02881     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
02882     */
02883     AddToPreTypes(PRETYPEDO);
02884
02885     loop = (DOLOOP *)FromList(&AP.LoopList);
02886     loop->firstdollar = loop->lastdollar = loop->incdollar = -1;
02887     loop->NumPreTypes = AP.NumPreTypes-1;
02888     loop->PreIfLevel = AP.PreIfLevel;
02889     loop->PreSwitchLevel = AP.PreSwitchLevel;
02890     AC.NoShowInput = 1;
02891     if ( PreLoad(&(loop->p), (UBYTE *) "do", (UBYTE *) "enddo",1,"doloop") ) return(-1);
02892     AC.NoShowInput = oldNoShowInput;
02893     loop->NoShowInput = AC.NoShowInput;
02894     /*
02895     Get now the name. We have to take great care when the name is terminated!
02896     */
02897     s = loop->p.buffer + (s - AP.preStart);
02898     SKIPBLANKS(s)
02899     loop->name = s;
02900     if ( chartype[*s] != 0 ) goto illname;
02901     s++;
02902     while ( chartype[*s] <= 1 ) s++;

```

```

02903     t = s;
02904     while ( *s == ' ' || *s == '\t' ) s++;
02905     if ( *s != '=' ) goto illdo;
02906     s++;
02907     while ( *s == ' ' || *s == '\t' ) s++;
02908     *t = 0;
02909
02910     if ( *s == '{' ) {
02911         loop->type = LISTEDLOOP;
02912         s++; loop->vars = s;
02913         loop->lastnum = 0;
02914         while ( *s != '}' && *s != 0 ) {
02915             if ( set_in(*s,AC.separators) ) { *s = 0; loop->lastnum++; }
02916             else if ( *s == '\\ ' ) s++;
02917             s++;
02918         }
02919         if ( *s == 0 ) goto illdo;
02920         *s++ = 0;
02921         loop->lastnum++;
02922         loop->firstnum = 0;
02923         loop->contents = s;
02924     }
02925     else if ( *s == '-' || *s == '+' || chartype[*s] == 1 || *s == '$' ) {
02926         loop->type = NUMERICLOOP;
02927         t = s;
02928         while ( *s && *s != ',' ) s++;
02929         if ( *s == 0 ) goto illdo;
02930         if ( *t == '$' ) {
02931             c = *s; *s = 0;
02932             if ( GetName(AC.dollarnames,t+1,&loop->firstdollar,NOAUTO) != CDOLLAR ) {
02933                 MesPrint("@%s is undefined in first parameter in %sdo instruction",t);
02934                 return(-1);
02935             }
02936             loop->firstnum = DolToLong(BHEAD loop->firstdollar);
02937             if ( AN.ErrorInDollar ) {
02938                 MesPrint("@%s does not evaluate into a valid loop parameter",t);
02939                 return(-1);
02940             }
02941             *s++ = c;
02942         }
02943         else {
02944             *s = ' ';
02945             if ( PreEval(t,&loop->firstnum) == 0 ) goto illdo;
02946             *s++ = ',';
02947         }
02948         t = s;
02949         while ( *s && *s != ',' && *s != ';' && *s != LINEFEED ) s++;
02950         c = *s;
02951         if ( *t == '$' ) {
02952             *s = 0;
02953             if ( GetName(AC.dollarnames,t+1,&loop->lastdollar,NOAUTO) != CDOLLAR ) {
02954                 MesPrint("@%s is undefined in second parameter in %sdo instruction",t);
02955                 return(-1);
02956             }
02957             loop->lastnum = DolToLong(BHEAD loop->lastdollar);
02958             if ( AN.ErrorInDollar ) {
02959                 MesPrint("@%s does not evaluate into a valid loop parameter",t);
02960                 return(-1);
02961             }
02962             *s++ = c;
02963         }
02964         else {
02965             *s = ' ';
02966             if ( PreEval(t,&loop->lastnum) == 0 ) goto illdo;
02967             *s++ = c;
02968         }
02969         if ( c == ',' ) {
02970             t = s;
02971             while ( *s && *s != ';' && *s != LINEFEED ) s++;
02972             if ( *t == '$' ) {
02973                 c = *s; *s = 0;
02974                 if ( GetName(AC.dollarnames,t+1,&loop->incdollar,NOAUTO) != CDOLLAR ) {
02975                     MesPrint("@%s is undefined in third parameter in %sdo instruction",t);
02976                     return(-1);
02977                 }
02978                 loop->incnum = DolToLong(BHEAD loop->incdollar);
02979                 if ( AN.ErrorInDollar ) {
02980                     MesPrint("@%s does not evaluate into a valid loop parameter",t);
02981                     return(-1);
02982                 }
02983                 *s++ = c;
02984             }
02985             else {
02986                 c = *s; *s = ' ';
02987                 if ( PreEval(t,&loop->incnum) == 0 ) goto illdo;
02988                 *s++ = c;
02989             }
02990         }
02991     }

```

```

02990     }
02991     else loop->incnum = 1;
02992     loop->contents = s;
02993 }
02994 else if ( ( chartye[*s] == 0 ) || ( *s == '[' ) ) {
02995     int oldNumPotModdollars = NumPotModdollars;
02996 #ifdef WITHMPI
02997     WORD oldRhsExprInModuleFlag = AC.RhsExprInModuleFlag;
02998     AC.RhsExprInModuleFlag = 0;
02999 #endif
03000     t = s;
03001     if ( ( s = SkipAName(s) ) == 0 ) goto illdo;
03002     c = *s; *s = 0;
03003     if ( GetName(AC.exprnames,t,&expnum,NOAUTO) == CEXPRESSION ) {
03004         loop->type = ONEEXPRESSION;
03005     /*
03006         We should remember the expression by name for when it gets
03007         renumbered!!! If it gets deleted there will be a crash or at
03008         least the loop terminates.
03009     */
03010         loop->vars = t;
03011     }
03012     else goto illdo;
03013     if ( c == ',' || c == '\t' || c == ';' ) { s++; }
03014     else if ( c != 0 && c != '\n' ) goto illdo;
03015     while ( *s == ',' || *s == '\t' || *s == ';' ) s++;
03016     if ( *s != 0 && *s != '\n' ) goto illdo;
03017     loop->firstnum = 0;
03018     s++;
03019     loop->contents = s;
03020     loop->incnum = 0;
03021     /*
03022         Next determine size of statement and allocate space
03023     */
03024     while ( *t ) t++;
03025     i = t - loop->vars;
03026     t = loop->name;
03027     while ( *t ) { t++; i++; }
03028     i += 4;
03029     loop->dollarname = Malloc1((LONG)i,"do-loop instruction");
03030     /*
03031         Construct the statement
03032     */
03033     u = loop->dollarname;
03034     *u++ = '$'; t = loop->name; while ( *t ) *u++ = *t++;
03035     *u++ = '_'; uu = u; *u++ = '='; t = loop->vars;
03036     while ( *t ) *u++ = *t++;
03037     *t = 0; *u = 0;
03038     /*
03039         Compile and put in dollar variable.
03040         Note that we remember the dollar by name and that this name ends in _
03041     */
03042     oldpreassignflag = AP.PreAssignFlag;
03043     AP.PreAssignFlag = 2;
03044     CompileStatement(loop->dollarname);
03045     if ( CatchDollar(0) ) {
03046         MesPrint("@Cannot load expression in do loop");
03047         return(-1);
03048     }
03049     AP.PreAssignFlag = oldpreassignflag;
03050     NumPotModdollars = oldNumPotModdollars;
03051 #ifdef WITHMPI
03052     AC.RhsExprInModuleFlag = oldRhsExprInModuleFlag;
03053 #endif
03054     *uu = 0;
03055 }
03056 else goto illdo; /* Syntax problems */
03057 loop->errorsinloop = 0;
03058 /* loop->startlinenumber = linenum+1; 5-oct-2000 One too much? */
03059 loop->startlinenumber = linenum;
03060 PutPreVar(loop->name, (UBYTE *)"0",0,0);
03061 loop->firstloopcall = 1;
03062 return(DoEnddo(s));
03063 illname:;
03064 MesPrint("@Improper name for do loop variable");
03065 return(-1);
03066 illdo:;
03067 MesPrint("@Improper syntax in do loop instruction");
03068 return(-1);
03069 }
03070
03071 /*
03072     #] DoDo :
03073     #[ DoBreakDo :
03074
03075     #breakdo [num]
03076     jumps out of num #do-loops (if there are that many) (default is 1)

```

```

03077 */
03078
03079 int DoBreakDo(UBYTE *s)
03080 {
03081     WORD levels;
03082
03083     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03084     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03085
03086     if ( NumDoLoops <= 0 ) {
03087         MesPrint("@%#breakdo without %#do");
03088         return(1);
03089     }
03090 /*
03091     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEDO ) { MessPreNesting(4); return(-1); }
03092 */
03093     while ( *s && ( *s == ' ' || *s == ' ' || *s == '\t' ) ) s++;
03094     if ( *s == 0 ) {
03095         levels = 1;
03096     }
03097     else if ( FG.cTable[*s] == 1 ) {
03098         levels = 0;
03099         while ( *s >= '0' && *s <= '9' ) { levels = 10*levels + *s++ - '0'; }
03100         if ( *s != 0 ) goto improper;
03101     }
03102     else {
03103 improper:
03104         MesPrint("@Improper syntax of %#breakdo instruction");
03105         return(1);
03106     }
03107     if ( levels > NumDoLoops ) {
03108         MesPrint("@Too many loop levels requested in %#breakdo instruction");
03109         Terminate(-1);
03110     }
03111     return(ExitDoLoops(levels, "breakdo"));
03112 }
03113
03121 static int ExitDoLoops(int levels, const char *instruction)
03122 {
03123     DOLOOP *loop;
03124     while ( levels > 0 ) {
03125         while ( AC.CurrentStream->type != PREREADSTREAM
03126             && AC.CurrentStream->type != PREREADSTREAM2
03127             && AC.CurrentStream->type != PREREADSTREAM3 ) {
03128             AC.CurrentStream = CloseStream(AC.CurrentStream);
03129         }
03130         while ( AP.PreTypes[AP.NumPreTypes] != PRETYPEDO
03131             && AP.PreTypes[AP.NumPreTypes] != PRETYPEPROCEDURE ) AP.NumPreTypes--;
03132         if ( AC.CurrentStream->type == PREREADSTREAM3
03133             || AP.PreTypes[AP.NumPreTypes] == PRETYPEPROCEDURE ) {
03134             MesPrint("@Trying to jump out of a procedure with a %s instruction", instruction);
03135             return(1);
03136         }
03137         loop = &(DoLoops[NumDoLoops-1]);
03138         AP.NumPreTypes = loop->NumPreTypes;
03139         AP.PreIfLevel = loop->PreIfLevel;
03140         AP.PreSwitchLevel = loop->PreSwitchLevel;
03141         NumDoLoops--;
03142         DoUndefine(loop->name);
03143         M_free(loop->p.buffer, "loop->p.buffer");
03144         loop->firstloopcall = 0;
03145
03146         AC.CurrentStream = CloseStream(AC.CurrentStream);
03147         levels--;
03148     }
03149     return(0);
03150 }
03151
03152 /*
03153     #] DoBreakDo :
03154     #[ DoElse :
03155 */
03156
03157 int DoElse(UBYTE *s)
03158 {
03159     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEIF ) {
03160         if ( AP.PreIfLevel <= 0 ) MesPrint("@%#else without corresponding %#if");
03161         else MessPreNesting(1);
03162         return(-1);
03163     }
03164     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03165     while ( *s == ' ' ) s++;
03166     if ( tolower(*s) == 'i' && tolower(s[1]) == 'f' && s[2]
03167         && FG.cTable[s[2]] > 1 && s[2] != '_' ) {
03168         s += 2;
03169         while ( *s == ' ' ) s++;
03170         return(DoElseif(s));
03171     }

```

```

03171     }
03172     if ( AP.PreIfLevel <= 0 ) {
03173         MesPrint("@%#else without corresponding %if");
03174         return(-1);
03175     }
03176     switch ( AP.PreIfStack[AP.PreIfLevel] ) {
03177         case EXECUTINGIF:
03178             AP.PreIfStack[AP.PreIfLevel] = LOOKINGFORENDIF;
03179             break;
03180         case LOOKINGFORELSE:
03181             AP.PreIfStack[AP.PreIfLevel] = EXECUTINGIF;
03182             break;
03183         case LOOKINGFORENDIF:
03184             break;
03185     }
03186     return(0);
03187 }
03188
03189 /*
03190     #] DoElse :
03191     #[ DoElseif :
03192 */
03193
03194 int DoElseif(UBYTE *s)
03195 {
03196     int condition;
03197     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEIF ) {
03198         if ( AP.PreIfLevel <= 0 ) MesPrint("@%#elseif without corresponding %if");
03199         else MessPreNesting(2);
03200         return(-1);
03201     }
03202     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03203     if ( AP.PreIfLevel <= 0 ) {
03204         MesPrint("@%#elseif without corresponding %if");
03205         return(-1);
03206     }
03207     switch ( AP.PreIfStack[AP.PreIfLevel] ) {
03208         case EXECUTINGIF:
03209             AP.PreIfStack[AP.PreIfLevel] = LOOKINGFORENDIF;
03210             break;
03211         case LOOKINGFORELSE:
03212             if ( ( condition = EvalPreIf(s) ) < 0 ) return(-1);
03213             AP.PreIfStack[AP.PreIfLevel] = condition;
03214             break;
03215         case LOOKINGFORENDIF:
03216             break;
03217     }
03218     return(0);
03219 }
03220
03221 /*
03222     #] DoElseif :
03223     #[ DoEnddo :
03224
03225     At the first call there is no stream yet.
03226     After that we have to close the stream and start a new one.
03227 */
03228
03229 int DoEnddo(UBYTE *s)
03230 {
03231     GETIDENTITY
03232     DOLOOP *loop;
03233     UBYTE *t, *tt, *value, numstr[16];
03234     LONG xval;
03235     int xsign, retval;
03236     DUMMYUSE(s);
03237     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03238     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03239     /*
03240     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ||
03241         AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) {
03242         if ( AP.PreTypes[AP.NumPreTypes] == PRETYPEPEDO ) AP.NumPreTypes--;
03243         else { MessPreNesting(3); return(-1); }
03244         return(0);
03245     }
03246     */
03247     if ( NumDoLoops <= 0 ) {
03248         MesPrint("@%#enddo without %do");
03249         return(1);
03250     }
03251     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEPEDO ) { MessPreNesting(4); return(-1); }
03252     loop = &(DoLoops[NumDoLoops-1]);
03253     if ( !loop->firstloopcall ) AC.CurrentStream = CloseStream(AC.CurrentStream);
03254
03255     if ( loop->errorsinloop ) {
03256         MesPrint("++++Errors in Loop");
03257         goto finish;

```

```

03258     }
03259     if ( loop->type == LISTEDLOOP ) {
03260         if ( loop->firstnum >= loop->lastnum ) goto finish;
03261         loop->firstnum++;
03262         t = value = loop->vars;
03263         while ( *value ) value++;
03264         value++;
03265         loop->vars = value;
03266         value = tt = t;
03267         while ( *value ) {
03268             if ( *value == '\\\\' ) value++;
03269             *tt++ = *value++;
03270         }
03271         *tt = 0;
03272         PutPreVar(loop->name,t,0,1); /* We overwrite the definition */
03273     }
03274     else if ( loop->type == NUMERICALLOOP ) {
03275
03276         if ( !loop->firstloopcall ) {
03277             /*
03278              Test whether the variable was changed inside the loop into
03279              a different numerical value. If so, adjust.
03280             */
03281             t = GetPreVar(loop->name,WITHOUTERROR);
03282             if ( t ) {
03283                 value = t;
03284                 xsign = 1;
03285                 while ( *value && ( *value == ' ' || *value == '-' || *value == '+' ) ) {
03286                     if ( *value == '-' ) xsign = -xsign;
03287                     value++;
03288                 }
03289                 t = value; xval = 0;
03290                 while ( *value >= '0' && *value <= '9' ) xval = 10*xval + *value++ - '0';
03291                 while ( *value && *value == ' ' ) value++;
03292                 if ( *value == 0 ) {
03293                     /*
03294                      Now we may substitute the loopvalue.
03295                     */
03296                     if ( xsign < 0 ) xval = -xval;
03297                     if ( loop->incdollar >= 0 ) {
03298                         loop->incnum = DolToLong(BHEAD loop->incdollar);
03299                         if ( AN.ErrorInDollar ) {
03300                             MesPrint("@%s does not evaluate into a valid third loop
03301 parameter",DOLLARNAME(Dollars,loop->incdollar));
03302                             return(-1);
03303                         }
03304                     }
03305                     loop->firstnum = xval + loop->incnum;
03306                 }
03307             }
03308             if ( loop->lastdollar >= 0 ) {
03309                 loop->lastnum = DolToLong(BHEAD loop->lastdollar);
03310                 if ( AN.ErrorInDollar ) {
03311                     MesPrint("@%s does not evaluate into a valid second loop
03312 parameter",DOLLARNAME(Dollars,loop->lastdollar));
03313                     return(-1);
03314                 }
03315             }
03316             if ( ( loop->incnum > 0 && loop->firstnum > loop->lastnum )
03317                 || ( loop->incnum < 0 && loop->firstnum < loop->lastnum ) ) goto finish;
03318             NumToStr(numstr,loop->firstnum);
03319             t = numstr;
03320             loop->firstnum += loop->incnum;
03321             PutPreVar(loop->name,t,0,1); /* We overwrite the definition */
03322         }
03323     }
03324     else if ( loop->type == ONEEXPRESSION ) {
03325         /*
03326         Find the dollar expression
03327         */
03328         WORD numdollar = GetDollar(loop->dollarname+1);
03329         DOLLARS d = Dollars + numdollar;
03330         WORD *w, *dw, v, *ww;
03331         if ( (d->where) == 0 ) {
03332             d->type = DOLUNDEFINED;
03333             M_free(loop->dollarname,"do-loop instruction");
03334             goto finish;
03335         }
03336         w = d->where + loop->incnum;
03337         if ( *w == 0 ) {
03338             M_free(d->where,"dollar");
03339             d->where = 0;
03340             d->type = DOLUNDEFINED;
03341             M_free(loop->dollarname,"do-loop instruction");
03342             goto finish;
03343         }
03344     }

```

```

03343         loop->incnum += *w;
03344     /*
03345         Now the term has to be converted to text.
03346     */
03347         ww = w + *w; v = *ww; *ww = 0;
03348         dw = d->where; d->where = w;
03349         t = WriteDollarToBuffer(numdollar,1);
03350         d->where = dw; *ww = v;
03351         PutPreVar(loop->name,t,0,1);    /* We overwrite the definition */
03352         M_free(t,"dollar");
03353     }
03354     if ( loop->firstloopcall ) OpenStream(loop->contents,PREREADSTREAM2,0,PRENOACTION);
03355     else OpenStream(loop->contents,PREREADSTREAM,0,PRENOACTION);
03356     AC.CurrentStream->prevline =
03357     AC.CurrentStream->linenumber = loop->startlinenumber;
03358     AC.CurrentStream->eqnum = 0;
03359     loop->firstloopcall = 0;
03360     return(0);
03361 finish:;
03362     NumDoLoops--;
03363     retval = DoUndefine(loop->name);
03364     M_free(loop->p.buffer,"loop->p.buffer");
03365     loop->firstloopcall = 0;
03366     AP.NumPreTypes--;
03367     return(retval);
03368 }
03369
03370 /*
03371     #] DoEnddo :
03372     #[ DoEndif :
03373 */
03374 int DoEndif(UBYTE *s)
03375 {
03376     DUMMYUSE(s);
03377     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEIF ) {
03378         if ( AP.PreIfLevel <= 0 ) MesPrint("@%#endif without corresponding %#if");
03379         else MessPreNesting(5);
03380         return(-1);
03381     }
03382     AP.NumPreTypes--;
03383     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03384     if ( AP.PreIfLevel <= 0 ) {
03385         MesPrint("@%#endif without corresponding %#if");
03386         return(-1);
03387     }
03388     AP.PreIfLevel--;
03389     return(0);
03390 }
03391 }
03392
03393 /*
03394     #] DoEndif :
03395     #[ DoEndprocedure :
03396
03397     Action is simple: close the current stream if it is still
03398     the stream from which the statement came.
03399     Then pop the current procedure and all its local derivatives.
03400     if loadmode > 1 the procedure was defined locally.
03401 */
03402 int DoEndprocedure(UBYTE *s)
03403 {
03404     DUMMYUSE(s);
03405     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEPROCEDURE ) {
03406         MessPreNesting(6);
03407         return(-1);
03408     }
03409     AP.NumPreTypes--;
03410     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03411     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03412     AC.CurrentStream = CloseStream(AC.CurrentStream);
03413     do {
03414         NumProcedures--;
03415         if ( Procedures[NumProcedures].mustfree == 1 ) {
03416             M_free(Procedures[NumProcedures].p.buffer,"procedures buffer");
03417             M_free(Procedures[NumProcedures].name,"procedures name");
03418         }
03419     } while ( Procedures[NumProcedures].loadmode > 1 );
03420     return(0);
03421 }
03422
03423 /*
03424     #] DoEndprocedure :
03425     #[ DoIf :
03426 */
03427
03428
03429

```

```

03430 int DoIf(UBYTE *s)
03431 {
03432     int condition;
03433     AddToPreTypes(PRETYPEIF);
03434     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03435     if ( AP.PreIfStack[AP.PreIfLevel] == EXECUTINGIF ) {
03436         condition = EvalPreIf(s);
03437         if ( condition < 0 ) return(-1);
03438     }
03439     else condition = LOOKINGFORENDIF;
03440     if ( AP.PreIfLevel+1 >= AP.MaxPreIfLevel ) {
03441         int **ppp = &AP.PreIfStack; /* To avoid a compiler warning */
03442         if ( DoubleList((void ***)ppp,&AP.MaxPreIfLevel,sizeof(int),
03443             "PreIfLevels") ) return(-1);
03444     }
03445     AP.PreIfStack[++AP.PreIfLevel] = condition;
03446     return(0);
03447 }
03448
03449 /*
03450     #] DoIf :
03451     #[ DoIfdef :
03452 */
03453
03454 int DoIfdef(UBYTE *s, int par)
03455 {
03456     int condition;
03457     AddToPreTypes(PRETYPEIF);
03458     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03459     if ( AP.PreIfStack[AP.PreIfLevel] == EXECUTINGIF ) {
03460         while ( *s == ' ' || *s == '\t' ) s++;
03461         if ( ( *s == 0 ) == ( par == 1 ) ) condition = LOOKINGFORELSE;
03462         else condition = EXECUTINGIF;
03463     }
03464     else condition = LOOKINGFORENDIF;
03465     if ( AP.PreIfLevel+1 >= AP.MaxPreIfLevel ) {
03466         int **ppp = &AP.PreIfStack; /* to avoid a compiler warning */
03467         if ( DoubleList((void ***)ppp,&AP.MaxPreIfLevel,sizeof(int),
03468             "PreIfLevels") ) return(-1);
03469     }
03470     AP.PreIfStack[++AP.PreIfLevel] = condition;
03471     return(0);
03472 }
03473
03474 /*
03475     #] DoIfdef :
03476     #[ DoIfydef :
03477 */
03478
03479 int DoIfydef(UBYTE *s)
03480 {
03481     return DoIfdef(s,1);
03482 }
03483
03484 /*
03485     #] DoIfydef :
03486     #[ DoIfndef :
03487 */
03488
03489 int DoIfndef(UBYTE *s)
03490 {
03491     return DoIfdef(s,2);
03492 }
03493
03494 /*
03495     #] DoIfndef :
03496     #[ DoInside :
03497
03498     #inside $var1,...,$varn
03499     statements without .sort
03500     #endinside
03501
03502     executes the statements on the contents of the $ variables as if they
03503     are a module. The results are put back in the dollar variables.
03504     To do this right we need a struct with
03505         old compiler buffer
03506         list of numbers of dollars
03507         length of the list
03508         length of the array containing the list
03509     Because we need to compose statements, the statement buffer must be
03510     empty. This means that we have to test for that. Same at the end. We
03511     must have a completed statement.
03512 */
03513
03514 int DoInside(UBYTE *s)
03515 {
03516     GETIDENTITY

```



```

03517     int numdol, error = 0;
03518     WORD *nb, newsiz, i;
03519     UBYTE *name, c;
03520     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03521     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03522     if ( AP.PreInsideLevel != 0 ) {
03523         MesPrint("@Illegal nesting of %inside/%endinside instructions");
03524         return(-1);
03525     }
03526     /*
03527     if ( AP.PreContinuation ) {
03528         error = -1;
03529         MesPrint("@%inside cannot be inside a regular statement");
03530     }
03531     */
03532     PUSHPREASSIGNLEVEL
03533     /*
03534     Now the dollars to do
03535     */
03536     AP.inside.numdollars = 0;
03537     for(;;) {
03538         while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
03539         if ( *s == 0 ) break;
03540         if ( *s != '$' ) {
03541             MesPrint("@%inside instruction can have only $ variables for parameters");
03542             return(-1);
03543         }
03544         s++;
03545         name = s;
03546         while (chartype[*s] <= 1 ) s++;
03547         c = *s; *s = 0;
03548         if ( ( numdol = GetDollar(name) ) < 0 ) {
03549             MesPrint("@%inside: $%s has not (yet) been defined",name);
03550             *s = c;
03551             error = -1;
03552         }
03553         else {
03554             *s = c;
03555             if ( AP.inside.numdollars >= AP.inside.size ) {
03556                 if ( AP.inside.buffer == 0 ) newsiz = 20;
03557                 else newsiz = 2*AP.inside.size;
03558                 nb = (WORD *)Malloca1(newsiz*sizeof(WORD),"insidebuffer");
03559                 if ( AP.inside.buffer ) {
03560                     for ( i = 0; i < AP.inside.size; i++ ) nb[i] = AP.inside.buffer[i];
03561                     M_free(AP.inside.buffer,"insidebuffer");
03562                 }
03563                 AP.inside.buffer = nb;
03564                 AP.inside.size = newsiz;
03565             }
03566             AP.inside.buffer[AP.inside.numdollars++] = numdol;
03567         }
03568     }
03569     /*
03570     We have to store the configuration of the compiler buffer, so that
03571     we know where to start executing and how to reset the buffer.
03572     */
03573     AP.inside.oldcompletetype = AC.completetype;
03574     AP.inside.oldparallelflag = AC.mparallelflag;
03575     AP.inside.oldnumpotmoddollars = NumPotModdollars;
03576     AP.inside.oldcbuf = AC.cbufnum;
03577     AP.inside.olddrbuf = AM.rdbufnum;
03578     AP.inside.oldcnumlhs = AR.Cnumlhs,
03579     AddToPreTypes(PRETYPEINSIDE);
03580     AP.PreInsideLevel = 1;
03581     AC.cbufnum = AP.inside.inscbuf;
03582     AM.rdbufnum = AP.inside.inscbuf;
03583     clearcbuf(AC.cbufnum);
03584     AC.completetype = 0;
03585     AC.mparallelflag = PARALLELFLAG;
03586     #ifdef WITHMPI
03587     /*
03588     * We use AC.RhsExprInModuleFlag, PotModdollars, and AC.pfirstnum
03589     * in order to check (1) whether there are expression names in RHS,
03590     * (2) which dollar variables can be modified, and (3) which
03591     * preprocessor variables can be redefined, in #inside.
03592     * We store the current values of them, and then reset them.
03593     */
03594     PF_StoreInsideInfo();
03595     AC.RhsExprInModuleFlag = 0;
03596     NumPotModdollars = 0;
03597     AC.numfirstnum = 0;
03598     #endif
03599     return(error);
03600 }
03601
03602 /*
03603     #] DoInside :

```

```

03604         # [ DoEndInside :
03605         */
03606
03607 int DoEndInside(UBYTE *s)
03608 {
03609     GETIDENTITY
03610     WORD numdol, *oldworkpointer = AT.WorkPointer, *term, *t, j, i;
03611     DOLLARS d, nd;
03612     WORD oldbracketon = AR.BracketOn;
03613     WORD *oldcompresspointer = AR.CompressPointer;
03614     int oldmultithreaded = AS.MultiThreaded;
03615     /* int oldmparallelflag = AC.mparallelflag; */
03616     FILEHANDLE *f;
03617 #ifdef WITHMPI
03618     int error = 0;
03619 #endif
03620     DUMMYUSE(s);
03621     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03622     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03623     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPEINSIDE ) {
03624         if ( AP.PreInsideLevel != 1 ) MesPrint("@%#endinside without corresponding %#inside");
03625         else MessPreNesting(11);
03626         return(-1);
03627     }
03628     AP.NumPreTypes--;
03629     if ( AP.PreInsideLevel != 1 ) {
03630         MesPrint("@%#endinside without corresponding %#inside");
03631         return(-1);
03632     }
03633     if ( AP.PreContinuation ) {
03634         MesPrint("@%#endinside: previous statement not terminated.");
03635         Terminate(-1);
03636     }
03637     AC.compiletype = AP.inside.oldcompiletype;
03638     AR.Cnumlhs = cbuf[AM.rbufnum].numlhs;
03639 #ifdef WITHMPI
03640     /*
03641      * If the #inside...#endinside contains expressions in RHS, only the master executes it
03642      * and then broadcasts the result to the all slaves. If not, the all processes execute
03643      * it and in this case no MPI interactions are needed.
03644      */
03645     if ( PF.me == MASTER || !AC.RhsExprInModuleFlag ) {
03646 #endif
03647         AR.BracketOn = 0;
03648         AS.MultiThreaded = 0;
03649         /* AC.mparallelflag = PARALLELFLAG; */
03650         if ( AR.CompressPointer == 0 ) AR.CompressPointer = AR.CompressBuffer;
03651         f = AR.infile; AR.infile = AR.outfile; AR.outfile = f;
03652     /*
03653     Now we have to execute the statements on the proper dollars.
03654     */
03655     for ( i = 0; i < AP.inside.numdollars; i++ ) {
03656         numdol = AP.inside.buffer[i];
03657         nd = d = Dollars + numdol;
03658         if ( d->type != DOLZERO ) {
03659             if ( d->type != DOLTERMS ) nd = DolToTerms(BHEAD numdol);
03660             term = nd->where;
03661             NewSort(BHEAD0);
03662             NewSort(BHEAD0);
03663             AR.MaxDum = AM.IndDum;
03664             while ( *term ) {
03665                 t = oldworkpointer; j = *term;
03666                 NCOPY(t,term,j);
03667                 AT.WorkPointer = t;
03668                 AN.IndDum = AM.IndDum;
03669                 AR.CurDum = ReNumber(BHEAD term);
03670                 if ( Generator(BHEAD oldworkpointer,0) ) {
03671                     MesPrint("@Called from %#endinside");
03672                     MesPrint("@Evaluating variable %s",DOLLARNAME(Dollars,numdol));
03673                     Terminate(-1);
03674                 }
03675             }
03676             AT.WorkPointer = oldworkpointer;
03677             CleanDollarFactors(d);
03678             if ( d->where ) { M_free(d->where,"dollar contents"); d->where = 0; }
03679             EndSort(BHEAD (WORD *) ((void *) (&(d->where))),2);
03680             LowerSortLevel();
03681             term = d->where; while ( *term ) term += *term;
03682             d->size = term - d->where;
03683             if ( nd != d ) M_free(nd,"Copy of dollar variable");
03684             if ( d->where[0] == 0 ) {
03685                 M_free(d->where,"dollar contents"); d->where = 0;
03686                 d->type = DOLZERO;
03687             }
03688         }
03689     }
03690 #ifdef WITHMPI

```

```

03691     }
03692     if ( AC.RhsExprInModuleFlag ) {
03693         /*
03694          * The only master executed the statements in #inside.
03695          * We need to broadcast the result to the all slaves.
03696          */
03697         for ( i = 0; i < AP.inside.numdollars; i++ ) {
03698             /*
03699              * Mark $-variables specified in the #inside instruction as modified
03700              * such that they will be broadcast.
03701              */
03702             AddPotModdollar(AP.inside.buffer[i]);
03703         }
03704         /* Now actual broadcast of modified variables. */
03705         if ( NumPotModdollars > 0 ) {
03706             error = PF_BroadcastModifiedDollars();
03707             if ( error ) goto cleanup;
03708         }
03709         if ( AC.numpfirstnum > 0 ) {
03710             error = PF_BroadcastRedefinedPreVars();
03711             if ( error ) goto cleanup;
03712         }
03713     }
03714 cleanup:
03715 #endif
03716     f = AR.infile; AR.infile = AR.outfile; AR.outfile = f;
03717     AC.cbufnum = AP.inside.oldcbuf;
03718     AM.rbufnum = AP.inside.olldrbuf;
03719     AR.Cnumlhs = AP.inside.oldcnumlhs;
03720     AR.BracketOn = oldbracketon;
03721     AP.PreInsideLevel = 0;
03722     AR.CompressPointer = oldcompresspointer;
03723     AS.MultiThreaded = oldmultithreaded;
03724     AC.mparallelflag = AP.inside.oldparallelflag;
03725     NumPotModdollars = AP.inside.oldnumpotmoddollars;
03726     POPPREASSIGNLEVEL
03727 #ifdef WITHMPI
03728     PF_RestoreInsideInfo();
03729     if ( error ) return error;
03730 #endif
03731     return(0);
03732 }
03733
03734 /*
03735     #] DoEndInside :
03736     #[ DoMessage :
03737 */
03738
03739 int DoMessage(UBYTE *s)
03740 {
03741     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03742     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03743     while ( *s == ' ' || *s == '\t' ) s++;
03744     MesPrint("~~~%s",s);
03745     return(0);
03746 }
03747
03748 /*
03749     #] DoMessage :
03750     #[ DoPipe :
03751 */
03752
03753 int DoPipe(UBYTE *s)
03754 {
03755     #ifndef WITHPIPE
03756         DUMMYUSE(s);
03757     #endif
03758     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03759     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03760     #ifdef WITHPIPE
03761         FLUSHCONSOLE;
03762         while ( *s == ' ' || *s == '\t' ) s++;
03763         if ( OpenStream(s,PIPESTREAM,0,PREENOACTION) == 0 ) return(-1);
03764         return(0);
03765     #else
03766         Error0("Pipes not implemented on this computer/system");
03767         return(-1);
03768     #endif
03769 }
03770
03771 /*
03772     #] DoPipe :
03773     #[ DoPrcExtension :
03774 */
03775
03776 int DoPrcExtension(UBYTE *s)
03777 {

```

```

03778     UBYTE *t, *u, c;
03779     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03780     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03781     while ( *s == ' ' || *s == '\t' ) s++;
03782     if ( *s == 0 || *s == '\n' ) {
03783         MesPrint("@No valid procedure extension specified");
03784         return(-1);
03785     }
03786     if ( FG.cTable[*s] != 0 ) {
03787         MesPrint("@Procedure extension should be a string starting with an alphabetic character. No
whitespace.");
03788         return(-1);
03789     }
03790     t = s;
03791     while ( *s && *s != '\n' && *s != ' ' && *s != '\t' ) s++;
03792     u = s;
03793     while ( *s == ' ' || *s == '\t' ) s++;
03794     if ( *s != 0 && *s != '\n' ) {
03795         MesPrint("@Too many parameters in ProcedureExtension instruction");
03796         return(-1);
03797     }
03798     c = *u; *u = 0;
03799     if ( AP.procedureExtension ) M_free(AP.procedureExtension, "ProcedureExtension");
03800     AP.procedureExtension = strDupl(t, "ProcedureExtension");
03801     *u = c;
03802     return(0);
03803 }
03804
03805 /*
03806     #] DoPrCExtension :
03807     #[ DoPreOut :
03808 */
03809
03810 int DoPreOut(UBYTE *s)
03811 {
03812     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03813     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03814     if ( tolower(*s) == 'o' ) {
03815         if ( tolower(s[1]) == 'n' && s[2] == 0 ) {
03816             AP.PreOut = 1;
03817             return(0);
03818         }
03819         if ( tolower(s[1]) == 'f' && tolower(s[2]) == 'f' && s[3] == 0 ) {
03820             AP.PreOut = 0;
03821             return(0);
03822         }
03823     }
03824     MesPrint("@Illegal option in PreOut instruction");
03825     return(-1);
03826 }
03827
03828 /*
03829     #] DoPreOut :
03830     #[ DoPrePrintTimes :
03831 */
03832
03833 int DoPrePrintTimes(UBYTE *s)
03834 {
03835     DUMMYUSE(s);
03836     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03837     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03838     PrintRunningTime();
03839     return(0);
03840 }
03841
03842 /*
03843     #] DoPrePrintTimes :
03844     #[ DoPreSortReallocate :
03845 */
03846
03847 int DoPreSortReallocate(UBYTE *s)
03848 {
03849     DUMMYUSE(s);
03850     if ( AC.SortReallocateFlag == 0 ) {
03851         /* Currently off, so set to 2. Then the reallocation code knows the flag was
03852            set here, since "On sortreallocate;" sets it to 1. */
03853         AC.SortReallocateFlag = 2;
03854     }
03855     /* If the flag is already on, do nothing. */
03856     return(0);
03857 }
03858
03859 /*
03860     #] DoPreSortReallocate :
03861     #[ DoPreAppend :
03862
03863     Syntax:

```

```

03864         #append <filename>
03865     */
03866
03867 int DoPreAppend(UBYTE *s)
03868 {
03869     UBYTE *name, *to;
03870
03871     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03872     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03873     if ( AP.preError ) return(0);
03874     while ( *s == ' ' || *s == '\t' ) s++;
03875 /*
03876     Determine where to write
03877 */
03878     if ( *s == '<' ) {
03879         s++;
03880         name = to = s;
03881         while ( *s && *s != '>' ) {
03882             if ( *s == '\\' ) s++;
03883             *to++ = *s++;
03884         }
03885         if ( *s == 0 ) {
03886             MesPrint("@Improper termination of filename");
03887             return(-1);
03888         }
03889         s++;
03890         *to = 0;
03891         if ( *name ) { GetAppendChannel((char *)name); }
03892         else goto improper;
03893     }
03894     else {
03895 improper:
03896         MesPrint("@Proper syntax is: %#append <filename>");
03897         return(-1);
03898     }
03899     return(0);
03900 }
03901
03902 /*
03903     #] DoPreAppend :
03904     #[ DoPreCreate :
03905
03906     Syntax:
03907     #create <filename>
03908 */
03909
03910 int DoPreCreate(UBYTE *s)
03911 {
03912     UBYTE *name, *to;
03913
03914     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03915     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03916     if ( AP.preError ) return(0);
03917     while ( *s == ' ' || *s == '\t' ) s++;
03918 /*
03919     Determine where to write
03920 */
03921     if ( *s == '<' ) {
03922         s++;
03923         name = to = s;
03924         while ( *s && *s != '>' ) {
03925             if ( *s == '\\' ) s++;
03926             *to++ = *s++;
03927         }
03928         if ( *s == 0 ) {
03929             MesPrint("@Improper termination of filename");
03930             return(-1);
03931         }
03932         s++;
03933         *to = 0;
03934         if ( *name ) { GetChannel((char *)name,0); }
03935         else goto improper;
03936     }
03937     else {
03938 improper:
03939         MesPrint("@Proper syntax is: %#create <filename>");
03940         return(-1);
03941     }
03942     return(0);
03943 }
03944
03945 /*
03946     #] DoPreCreate :
03947     #[ DoPreRemove :
03948 */
03949
03950 int DoPreRemove(UBYTE *s)

```

```

03951 {
03952     UBYTE *name, *to;
03953     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03954     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03955     if ( AP.preError ) return(0);
03956     while ( *s == ' ' || *s == '\t' ) s++;
03957     if ( *s == '<' ) { s++; }
03958     else {
03959         MesPrint("@Proper syntax is: %#remove <filename>");
03960         return(-1);
03961     }
03962     name = to = s;
03963     while ( *s && *s != '>' ) {
03964         if ( *s == '\\\\' ) s++;
03965         *to++ = *s++;
03966     }
03967     if ( *s == 0 ) {
03968         MesPrint("@Improper filename");
03969         return(-1);
03970     }
03971     s++;
03972     *to = 0;
03973     CloseChannel((char *)name);
03974     remove((char *)name);
03975     return(0);
03976 }
03977
03978 /*
03979     #] DoPreRemove :
03980     #[ DoPreClose :
03981 */
03982
03983 int DoPreClose(UBYTE *s)
03984 {
03985     UBYTE *name, *to;
03986     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
03987     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
03988     if ( AP.preError ) return(0);
03989     while ( *s == ' ' || *s == '\t' ) s++;
03990     if ( *s == '<' ) { s++; }
03991     else {
03992         MesPrint("@Proper syntax is: %#close <filename>");
03993         return(-1);
03994     }
03995     name = to = s;
03996     while ( *s && *s != '>' ) {
03997         if ( *s == '\\\\' ) s++;
03998         *to++ = *s++;
03999     }
04000     if ( *s == 0 ) {
04001         MesPrint("@Improper filename");
04002         return(-1);
04003     }
04004     s++;
04005     *to = 0;
04006     return(CloseChannel((char *)name));
04007 }
04008
04009 /*
04010     #] DoPreClose :
04011     #[ DoPreWrite :
04012
04013     Syntax:
04014     #write [<filename>] "formatstring" [,objects]
04015     The format string can contain the following special objects/codes
04016     \n  newline
04017     \t  tab
04018     \!  if last entry in string: no linefeed at end
04019     \b  put \ in output
04020     %$  $-variable (to be found among the objects)
04021     %e  expression (name to be found among the objects)
04022     %E  expression without ; (name to be found among the objects)
04023     %s  string (to be found among the objects) (with or without "")
04024     %S  subterms (see PrintSubtermList)
04025 */
04026
04027 int DoPreWrite(UBYTE *s)
04028 {
04029     HANDLERS h;
04030
04031     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
04032     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
04033     if ( AP.preError ) return(0);
04034
04035     #ifdef WITHMPI
04036     if ( PF.me != MASTER ) return 0;
04037 #endif

```

```

04038
04039     h.oldsilent      = AM.silent;
04040     h.newlogonly     = h.oldlogonly = AM.FileOnlyFlag;
04041     h.newhandle      = h.oldhandle  = AC.LogHandle;
04042     h.oldprinttype   = AO.PrintType;
04043
04044     while ( *s == ' ' || *s == '\t' ) s++;
04045 /*
04046     Determine where to write
04047 */
04048     if( (s=defineChannel(s,&h))==0 ) return(-1);
04049
04050     return(writeToChannel(WRITEOUT,s,&h));
04051 }
04052
04053 /*
04054     #] DoPreWrite :
04055     #[ DoProcedure :
04056
04057     We have to read this procedure into a buffer.
04058     The only complications are:
04059     1: we have to seek through the file to do this efficiently
04060         the file operations under VMS cannot do this properly
04061         (unless we use the proper ANSI structs?)
04062         This is the reason why we read whole input files under VMS.
04063     2: what to do when the same name is used twice.
04064     Note that we have to do the reading without substitution of
04065     preprocessor variables.
04066 */
04067
04068 int DoProcedure(UBYTE *s)
04069 {
04070     UBYTE c;
04071     PROCEDURE *p;
04072     LONG i;
04073     if ( ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH )
04074     || ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) ) {
04075         if ( PreSkip((UBYTE *) "procedure", (UBYTE *) "endprocedure",1) ) return(-1);
04076         return(0);
04077     }
04078     // AP.ProcList.num = NumProcedures is incremented inside FromList
04079     p = (PROCEDURE *)FromList(&AP.ProcList);
04080     if ( PreLoad(&(p->p), (UBYTE *) "procedure", (UBYTE *) "endprocedure"
04081     ,1,(char *) "procedure") ) return(-1);
04082
04083     // If the procedure below is not local (loadmode 0) or has been called (loadmode 1), this is a
04084     // nested procedure definition. We must free its allocations when we hit its DoEndprocedure.
04085     // If it seems local (loadmode 2) but is tagged as "mustfree", it is multiply nested and we
04086     // similarly must free its allocations.
04087     if ( NumProcedures >= 2 &&
04088         ( Procedures[NumProcedures-2].loadmode != 2 || Procedures[NumProcedures-2].mustfree ) ) {
04089         p->mustfree = 1;
04090     }
04091     // Otherwise, we are defining a local procedure at "ground level", and we keep the definition
04092     // on the procedure stack until FORM terminates.
04093     else {
04094         p->mustfree = 0;
04095     }
04096
04097     p->loadmode = 2;
04098     s = p->p.buffer + 10;
04099     while ( *s == ' ' || *s == LINEFEED ) s++;
04100     if ( chartype[*s] ) {
04101         MesPrint("@Illegal name for procedure");
04102         return(-1);
04103     }
04104     p->name = s++;
04105     while ( chartype[*s] == 0 || chartype[*s] == 1 ) s++;
04106     c = *s; *s = 0;
04107     p->name = strDup1(p->name,"procedure");
04108     *s = c;
04109 /*
04110     Check for double names
04111 */
04112     for ( i = NumProcedures-2; i >= 0; i-- ) {
04113         if ( StrCmp(Procedures[i].name,p->name) == 0 ) {
04114             Error1("Multiple occurrence of procedure name ",p->name);
04115         }
04116     }
04117     return(0);
04118 }
04119
04120 /*
04121     #] DoProcedure :
04122     #[ DoPreBreak :
04123 */
04124

```

```

04125 int DoPreBreak(UBYTE *s)
04126 {
04127     DUMMYUSE(s);
04128     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
04129     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPESWITCH ) {
04130         if ( AP.PreSwitchLevel <= 0 )
04131             MesPrint("@Break without corresponding Switch");
04132         else MessPreNesting(7);
04133         return(-1);
04134     }
04135     if ( AP.PreSwitchLevel <= 0 ) {
04136         MesPrint("@Break without corresponding Switch");
04137         return(-1);
04138     }
04139     if ( AP.PreSwitchModes[AP.PreSwitchLevel] == EXECUTINGPRESWITCH )
04140         AP.PreSwitchModes[AP.PreSwitchLevel] = SEARCHINGPREENDSWITCH;
04141     return(0);
04142 }
04143
04144 /*
04145     #] DoPreBreak :
04146     #[ DoPreCase :
04147 */
04148
04149 int DoPreCase(UBYTE *s)
04150 {
04151     UBYTE *t;
04152     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
04153     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPESWITCH ) {
04154         if ( AP.PreSwitchLevel <= 0 )
04155             MesPrint("@Case without corresponding Switch");
04156         else MessPreNesting(8);
04157         return(-1);
04158     }
04159     if ( AP.PreSwitchLevel <= 0 ) {
04160         MesPrint("@Case without corresponding Switch");
04161         return(-1);
04162     }
04163     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != SEARCHINGPRECASE ) return(0);
04164
04165     SKIPBLANKS(s)
04166     t = s;
04167     while ( *s ) { if ( *s == '\\\\' ) s++; s++; }
04168     while ( s > t && ( s[-1] == ' ' || s[-1] == '\\t' ) && s[-2] != '\\\\' ) {
04169         if ( s[-2] == '\\\\' ) s--;
04170         s--;
04171     }
04172     if ( *t == '"' && s > t+1 && s[-1] == '"' && s[-2] != '\\\\' ) {
04173         t++; s--; *s = 0;
04174     }
04175     else *s = 0;
04176     s = AP.PreSwitchStrings[AP.PreSwitchLevel];
04177     while ( *t == *s && *t ) { s++; t++; }
04178     if ( *t || *s ) return(0); /* case did not match */
04179     AP.PreSwitchModes[AP.PreSwitchLevel] = EXECUTINGPRESWITCH;
04180     return(0);
04181 }
04182
04183 /*
04184     #] DoPreCase :
04185     #[ DoPreDefault :
04186 */
04187
04188 int DoPreDefault(UBYTE *s)
04189 {
04190     DUMMYUSE(s);
04191     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
04192     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPESWITCH ) {
04193         if ( AP.PreSwitchLevel <= 0 )
04194             MesPrint("@Default without corresponding Switch");
04195         else MessPreNesting(9);
04196         return(-1);
04197     }
04198     if ( AP.PreSwitchLevel <= 0 ) {
04199         MesPrint("@Default without corresponding Switch");
04200         return(-1);
04201     }
04202     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != SEARCHINGPRECASE ) return(0);
04203     AP.PreSwitchModes[AP.PreSwitchLevel] = EXECUTINGPRESWITCH;
04204     return(0);
04205 }
04206
04207 /*
04208     #] DoPreDefault :
04209     #[ DoPreEndSwitch :
04210 */
04211

```



```

04212 int DoPreEndSwitch(UBYTE *s)
04213 {
04214     DUMMYUSE(s);
04215     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
04216     if ( AP.PreTypes[AP.NumPreTypes] != PRETYPESWITCH ) {
04217         if ( AP.PreSwitchLevel <= 0 )
04218             MesPrint("@EndSwitch without corresponding Switch");
04219         else MessPreNesting(10);
04220         return(-1);
04221     }
04222     AP.NumPreTypes--;
04223     if ( AP.PreSwitchLevel <= 0 ) {
04224         MesPrint("@EndSwitch without corresponding Switch");
04225         return(-1);
04226     }
04227     M_free(AP.PreSwitchStrings[AP.PreSwitchLevel--], "pre switch string");
04228     return(0);
04229 }
04230
04231 /*
04232     #] DoPreEndSwitch :
04233     #[ DoPreSwitch :
04234
04235     There should be a string after this.
04236     We have to store it somewhere.
04237 */
04238
04239 int DoPreSwitch(UBYTE *s)
04240 {
04241     UBYTE *t, *switchstring, **newstrings;
04242     int newnum, i, *newmodes;
04243     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
04244     SKIPBLANKS(s)
04245     t = s;
04246     while ( *s ) { if ( *s == '\\\\' ) s++; s++; }
04247     while ( s > t && ( s[-1] == ' ' || s[-1] == '\\t' ) && s[-2] != '\\\\' ) {
04248         if ( s[-2] == '\\\\' ) s--;
04249         s--;
04250     }
04251     if ( *t == '"' && s > t+1 && s[-1] == '"' && s[-2] != '\\\\' ) {
04252         t++; s--; *s = 0;
04253     }
04254     else *s = 0;
04255     switchstring = (UBYTE *)Malloc1((s-t)+1, "case string");
04256     s = switchstring;
04257     while ( *t ) {
04258         if ( *t == '\\\\' ) t++;
04259         *s++ = *t++;
04260     }
04261     *s = 0;
04262     if ( AP.PreSwitchLevel >= AP.NumPreSwitchStrings ) {
04263         newnum = 2*AP.NumPreSwitchStrings;
04264         newstrings = (UBYTE **)Malloc1(sizeof(UBYTE *)*(newnum+1), "case strings");
04265         newmodes = (int *)Malloc1(sizeof(int)*(newnum+1), "case strings");
04266         for ( i = 0; i < AP.NumPreSwitchStrings; i++ )
04267             newstrings[i] = AP.PreSwitchStrings[i];
04268         M_free(AP.PreSwitchStrings, "AP.PreSwitchStrings");
04269         for ( i = 0; i <= AP.NumPreSwitchStrings; i++ )
04270             newmodes[i] = AP.PreSwitchModes[i];
04271         M_free(AP.PreSwitchModes, "AP.PreSwitchModes");
04272         AP.PreSwitchStrings = newstrings;
04273         AP.PreSwitchModes = newmodes;
04274         AP.NumPreSwitchStrings = newnum;
04275     }
04276     AP.PreSwitchStrings[++AP.PreSwitchLevel] = switchstring;
04277     if ( ( AP.PreSwitchLevel > 1 )
04278         && ( AP.PreSwitchModes[AP.PreSwitchLevel-1] != EXECUTINGPRESWITCH ) )
04279         AP.PreSwitchModes[AP.PreSwitchLevel] = SEARCHINGPREENDSWITCH;
04280     else
04281         AP.PreSwitchModes[AP.PreSwitchLevel] = SEARCHINGPRECASE;
04282     AddToPreTypes(PRETYPESWITCH);
04283     return(0);
04284 }
04285
04286 /*
04287     #] DoPreSwitch :
04288     #[ DoPreShow :
04289
04290     Print the contents of the preprocessor variables
04291 */
04292
04293 int DoPreShow(UBYTE *s)
04294 {
04295     int i;
04296     UBYTE *name, c;
04297     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
04298     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);

```

```

04299     while ( *s == ' ' || *s == '\t' ) s++;
04300     if ( *s == 0 ) {
04301         MesPrint("%#The preprocessor variables:");
04302         for ( i = 0; i < NumPre; i++ ) {
04303             MesPrint("%d: %s = \"%s\"",i,PreVar[i].name,PreVar[i].value);
04304         }
04305     }
04306     else {
04307         while ( *s ) {
04308             name = s; while ( *s && *s != ' ' && *s != '\t' && *s != ',' ) s++;
04309             c = *s; *s = 0;
04310             for ( i = 0; i < NumPre; i++ ) {
04311                 if ( StrCmp(PreVar[i].name,name) == 0 )
04312                     MesPrint("%d: %s = \"%s\"",i,PreVar[i].name,PreVar[i].value);
04313             }
04314             *s = c;
04315             while ( *s == ' ' || *s == '\t' ) s++;
04316         }
04317     }
04318     return(0);
04319 }
04320
04321 /*
04322     #] DoPreShow :
04323     #[ DoSystem :
04324 */
04325
04326 /*
04327  * A macro for translating the contents of `x' into a string after expanding.
04328 */
04329 #define STRINGIFY(x)  STRINGIFY__(x)
04330 #define STRINGIFY__(x) #x
04331
04332 int DoSystem(UBYTE *s)
04333 {
04334     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
04335     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
04336     if ( AP.preError ) return(0);
04337     #ifndef WITHSYSTEM
04338     FLUSHCONSOLE;
04339     while ( *s == ' ' || *s == '\t' ) s++;
04340     if ( *s == '-' && s[1] == 'e' ) {
04341         LONG err;
04342         UBYTE str[24];
04343         s += 2;
04344         if ( *s != ' ' ) {
04345             MesPrint("@Syntax error in #system command.");
04346             return(-1);
04347         }
04348         while ( *s == ' ' || *s == '\t' ) s++;
04349         err = system((char *)s);
04350         NumToStr(str,err);
04351         PutPreVar((UBYTE *)"SYSTEMERROR_",str,0,1);
04352     }
04353     else if ( system((char *)s) ) {
04354         MesPrint("@System call returned with error condition");
04355         Terminate(-1);
04356     }
04357     return(0);
04358     #else
04359     Error0("External programs not implemented on this computer/system");
04360     return(-1);
04361     #endif
04362 }
04363
04364 /*
04365     #] DoSystem :
04366     #[ PreLoad :
04367
04368     Loads a loop or procedure into a special buffer.
04369     Note: The current instruction is already in the preStart buffer
04370 */
04371
04372 int PreLoad(PRELOAD *p, UBYTE *start, UBYTE *stop, int mode, char *message)
04373 {
04374     UBYTE *s, *t, *top, *newbuffer, c;
04375     LONG i, ppsize, linenum = AC.CurrentStream->linenum;
04376     int size1, size2, level, com=0, last=1, strng = 0;
04377     p->size = AP.pSize;
04378     p->buffer = (UBYTE *)Malloc1(p->size+1,message);
04379     top = p->buffer + p->size - 2;
04380     t = p->buffer; *t++ = '#';
04381     s = start; size1 = size2 = 0;
04382     while ( *s ) { s++; size1++; }
04383     s = stop; while ( *s ) { s++; size2++; }
04384     s = AP.preStart; while ( *s ) *t++ = *s++; *t++ = LINEFEED;
04385     level = 1;

```

```

04386     i = 100;
04387     for (;;) {
04388         c = GetInput();
04389         if ( c == EOFINPUT ) {
04390             MesPrint("@Missing %#s, Should match line %1",stop,linenum);
04391             return(-1);
04392         }
04393         if ( c == AP.ComChar && last == 1 ) com = 1;
04394         if ( c == LINEFEED ) { last = 1; com = 0; }
04395         else last = 0;
04396
04397         if ( ( c == '"' ) && ( com == 0 ) ) { strng ^= 1; }
04398
04399         if ( ( c == '#' ) && ( com == 0 ) ) i = 0;
04400         else i++;
04401
04402         if ( t >= top ) {
04403             ppsize = t - p->buffer;
04404             p->size *= 2;
04405             newbuffer = (UBYTE *)Malloc1(p->size,message);
04406             t = newbuffer; s = p->buffer;
04407             while ( --ppsize >= 0 ) *t++ = *s++;
04408             M_free(p->buffer,"loading do loop");
04409             p->buffer = newbuffer;
04410             top = p->buffer + p->size - 2;
04411         }
04412         *t++ = c;
04413         if ( strng == 0 ) {
04414             if ( ( i == size2 ) && ( com == 0 ) ) {
04415                 *t = 0;
04416                 if ( StrICmp(t-size2,(UBYTE *) (stop)) == 0 ) {
04417                     while ( ( c = GetInput() ) != LINEFEED && c != EOFINPUT ) {}
04418                     level--;
04419                     if ( level <= 0 ) break;
04420                     if ( c == EOFINPUT ) Error1("Missing #",stop);
04421                     *t++ = LINEFEED; *t = 0; last = 1;
04422                 }
04423             }
04424             if ( ( i == size1 ) && mode && ( com == 0 ) ) {
04425                 *t = 0;
04426                 if ( StrICmp(t-size1,(UBYTE *) (start)) == 0 ) {
04427                     /*
04428                         while ( ( c = GetInput() ) != LINEFEED && c != EOFINPUT ) {}
04429                         if ( c == EOFINPUT ) Error1("Missing #",stop);
04430                     */
04431                     level++;
04432                 }
04433             }
04434             if ( i == 1 && t[-2] == LINEFEED ) {
04435                 if ( c == '-' ) AC.NoShowInput = 1;
04436                 else if ( c == '+' ) AC.NoShowInput = 0;
04437             }
04438         }
04439     }
04440     *t++ = LINEFEED;
04441     *t = 0;
04442     return(0);
04443 }
04444
04445 /*
04446     #] PreLoad :
04447     #[ PreSkip :
04448
04449     Skips a loop or procedure.
04450     Note: The current instruction is already in the preStart buffer
04451 */
04452
04453 #define SKIPBUFSIZE 20
04454
04455 int PreSkip(UBYTE *start, UBYTE *stop, int mode)
04456 {
04457     UBYTE *s, *t, buffer[SKIPBUFSIZE+2], c;
04458     LONG i, linenum = AC.CurrentStream->linenum;
04459     int size1, size2, level, com=0, last=1;
04460
04461     t = buffer; *t++ = '#';
04462     s = start; size1 = size2 = 0;
04463     while ( *s ) { s++; size1++; }
04464     s = stop; while ( *s ) { s++; size2++; }
04465     level = 1;
04466     i = 0;
04467     for (;;) {
04468         c = GetInput();
04469         if ( c == EOFINPUT ) {
04470             MesPrint("@Missing %#s, Should match line %1",stop,linenum);
04471             return(-1);
04472         }

```

```

04473         if ( c == AP.ComChar && last == 1 ) com = 1;
04474         if ( c == LINEFEED ) { last = 1; com = 0; i = 0; t = buffer; }
04475         else last = 0;
04476         if ( ( c == '#' ) && ( com == 0 ) ) { i = 0; t = buffer; }
04477         else i++;
04478
04479         if ( i < SKIPBUFSIZE ) *t++ = c;
04480         if ( ( i == size2 ) && ( com == 0 ) ) {
04481             *t = 0;
04482             if ( StrICmp(t-size2, (UBYTE *) (stop)) == 0 ) {
04483                 while ( ( c = GetInput() ) != LINEFEED && c != ENDOFINPUT ) {}
04484                 level--;
04485                 if ( level <= 0 ) {
04486                     pushbackchar = LINEFEED;
04487                     break;
04488                 }
04489                 if ( c == ENDOFINPUT ) Error1("Missing #", stop);
04490                 i = 0; t = buffer;
04491             }
04492         }
04493         if ( ( i == size1 ) && mode && ( com == 0 ) ) {
04494             *t = 0;
04495             if ( StrICmp(t-size1, (UBYTE *) (start)) == 0 ) {
04496                 while ( ( c = GetInput() ) != LINEFEED && c != ENDOFINPUT ) {}
04497                 level++;
04498                 i = 0; t = buffer;
04499             }
04500         }
04501     }
04502     return(0);
04503 }
04504
04505 /*
04506     #] PreSkip :
04507     #[ StartPrepro :
04508 */
04509
04510 void StartPrepro(void)
04511 {
04512     int **ppp;
04513     AP.MaxPreIfLevel = 2;
04514     ppp = &AP.PreIfStack;
04515     if ( DoubleList((void ***) ppp, &AP.MaxPreIfLevel, sizeof(int),
04516         "PreIfLevels") ) Terminate(-1);
04517     AP.PreIfLevel = 0; AP.PreIfStack[0] = EXECUTINGIF;
04518
04519     AP.NumPreSwitchStrings = 10;
04520     AP.PreSwitchStrings = (UBYTE **) Malloc1(sizeof(UBYTE *) *
04521         (AP.NumPreSwitchStrings+1), "case strings");
04522     AP.PreSwitchModes = (int *) Malloc1(sizeof(int) *
04523         (AP.NumPreSwitchStrings+1), "case strings");
04524     AP.PreSwitchModes[0] = EXECUTINGPRESWITCH;
04525     AP.PreSwitchLevel = 0;
04526 }
04527
04528 /*
04529     #] StartPrepro :
04530     #[ EvalPreIf :
04531
04532     Evaluates the condition in an if instruction.
04533     The return value is EXECUTINGIF if the condition is true.
04534     If it is false the returnvalue is LOOKINGFORELSE.
04535     An error gives a return value of -1
04536 */
04537
04538 int EvalPreIf(UBYTE *s)
04539 {
04540     UBYTE *t, *u;
04541     int val;
04542     t = s;
04543     while ( *t ) t++;
04544     *t++ = ')';
04545     *t = 0;
04546     if ( ( u = PreIfEval(s, &val) ) == 0 ) return(-1);
04547     if ( u < t ) {
04548         MesPrint("@Unmatched parentheses in condition");
04549         return(-1);
04550     }
04551     if ( val ) return(EXECUTINGIF);
04552     else return(LOOKINGFORELSE);
04553 }
04554
04555 /*
04556     #] EvalPreIf :
04557     #[ PreIfEval :
04558
04559     Used for recursions in the evaluation of a preprocessor if-condition.

```

```

04560         It determines whether the contents of () is true or false
04561         (or in error).
04562         The return value is the address of the first character after the
04563         closing parenthesis or null if there is an error.
04564         In value we find true(1) or false(0)
04565         We enter after the opening parenthesis.
04566         There are levels:
04567             0: orlevel:  a || b
04568             1: andlevel: a && b
04569             2: eqlevel:  a == b or a != b or a = b
04570             3: cmplevel: a > b or a >= b or a < b or a <= b or a >~ b etc
04571 */
04572
04573 UBYTE *PreIfEval(UBYTE *s, int *value)
04574 {
04575     int orlevel = 0, andlevel = 0, eqlevel = 0, cmplevel = 0;
04576     int type, val;
04577     LONG val2;
04578     int orte, orval, cmptype, cmpval, eqtype, eqval, andtype, andval;
04579     UBYTE *t, *eqt, *cmpt, c;
04580     int eqop, cmpop;
04581     orte = orval = cmptype = cmpval = eqtype = eqval = andtype = andval = 0;
04582     eqop = cmpop = 0;
04583     eqt = cmpt = 0;
04584     *value = 0;
04585     while ( *s != ')' ) {
04586         while ( *s == ' ' || *s == '\t' || *s == '\n' || *s == '\r' ) s++;
04587         t = s;
04588         s = pParseObject(s, &type, &val2);
04589         if ( s == 0 ) return(0);
04590         val = val2;
04591         c = *s;
04592         *s++ = 0; /* in case the object is a string without " */
04593         while ( c == ' ' || c == '\t' || c == '\n' || c == '\r' ) {
04594             c = *s; *s++ = 0;
04595         }
04596         if ( *t == '"' ) t++;
04597         switch(c) {
04598             case '|':
04599                 if ( *s != ' ' ) goto illoper;
04600                 s++;
04601                 /* fall through */
04602             case ')':
04603                 if ( cmplevel ) {
04604                     if ( type == 0 || cmptype == 0 ) goto illobject;
04605                     val = PreCmp(type, val, t, cmptype, cmpval, cmpt, cmpop);
04606                     type = 0;
04607                     cmplevel = 0;
04608                 }
04609                 if ( eqlevel ) {
04610                     val = PreEq(type, val, t, eqtype, eqval, eqt, eqop);
04611                     type = 0;
04612                     eqlevel = 0;
04613                 }
04614                 if ( andlevel ) {
04615                     if ( andtype != 0 || type != 0 ) goto illobject;
04616                     val &= andval;
04617                     andlevel = 0;
04618                 }
04619                 if ( orlevel ) {
04620                     if ( orte != 0 || type != 0 ) goto illobject;
04621                     val |= orval;
04622                 }
04623                 if ( c == ')' ) {
04624                     *value = val;
04625                     return(s);
04626                 }
04627                 orlevel = 1;
04628                 orval = val;
04629                 orte = type;
04630                 break;
04631             case '&':
04632                 if ( *s != ' ' ) goto illoper;
04633                 s++;
04634                 if ( cmplevel ) {
04635                     if ( type == 0 || cmptype == 0 ) goto illobject;
04636                     val = PreCmp(type, val, t, cmptype, cmpval, cmpt, cmpop);
04637                     type = 0;
04638                     cmplevel = 0;
04639                 }
04640                 if ( eqlevel ) {
04641                     val = PreEq(type, val, t, eqtype, eqval, eqt, eqop);
04642                     type = 0;
04643                     eqlevel = 0;
04644                 }
04645                 if ( andlevel ) {
04646                     if ( andtype != 0 || type != 0 ) goto illobject;

```

```

04647         val &= andval;
04648     }
04649     andlevel = 1;
04650     andval = val;
04651     andtype = type;
04652     break;
04653 case '!':
04654 case '=':
04655     if ( eqlevel ) goto illorder;
04656     if ( cmplevel ) {
04657         if ( type == 0 || cmptype == 0 ) goto illobject;
04658         val = PreCmp(type,val,t,cmptype,cmpval,cmpt,cmpop);
04659         type = 0;
04660         cmplevel = 0;
04661     }
04662     if ( c == '!' && *s != '=' ) goto illoper;
04663     if ( *s == '=' ) s++;
04664     if ( c == '!' ) eqop = 1;
04665     else eqop = 0;
04666     eqlevel = 1; eqt = t; eqval = val; eqtype = type;
04667     break;
04668 case '>':
04669 case '<':
04670     if ( cmplevel ) goto illorder;
04671     if ( c == '<' ) cmpop = -1;
04672     else cmpop = 1;
04673     cmplevel = 1; cmpt = t; cmpval = val; cmptype = type;
04674     if ( *s == '=' ) {
04675         s++;
04676         if ( *s == '~' ) { s++; cmpop *= 4; }
04677         else cmpop *= 2;
04678     }
04679     else if ( *s == '~' ) { s++; cmpop *= 3; }
04680     break;
04681 default:
04682     goto illoper;
04683 }
04684 }
04685 return(s);
04686 illorder:
04687     MesPrint("@illegal order of operators");
04688     return(0);
04689 illobject:
04690     MesPrint("@illegal object for this operator");
04691     return(0);
04692 illoper:
04693     MesPrint("@illegal operator");
04694     return(0);
04695 }
04696
04697 /*
04698     #] PreIfEval :
04699     #[ PreCmp :
04700 */
04701
04702 int PreCmp(int type, int val, UBYTE *t, int type2, int val2, UBYTE *t2, int cmpop)
04703 {
04704     if ( type == 2 || type2 == 2 || cmpop < -2 || cmpop > 2 ) {
04705         if ( cmpop < 0 && cmpop > -3 ) cmpop -= 2;
04706         if ( cmpop > 0 && cmpop < 3 ) cmpop += 2;
04707         if ( cmpop == 3 ) val = StrCmp(t2,t) > 0;
04708         else if ( cmpop == 4 ) val = StrCmp(t2,t) >= 0;
04709         else if ( cmpop == -3 ) val = StrCmp(t2,t) < 0;
04710         else if ( cmpop == -4 ) val = StrCmp(t2,t) <= 0;
04711     }
04712     else {
04713         if ( cmpop == 1 ) val = ( val2 > val );
04714         else if ( cmpop == 2 ) val = ( val2 >= val );
04715         else if ( cmpop == -1 ) val = ( val2 < val );
04716         else if ( cmpop == -2 ) val = ( val2 <= val );
04717     }
04718     return(val);
04719 }
04720
04721 /*
04722     #] PreCmp :
04723     #[ PreEq :
04724 */
04725
04726 int PreEq(int type, int val, UBYTE *t, int type2, int val2, UBYTE *t2, int eqop)
04727 {
04728     UBYTE str[20];
04729     if ( type == 2 || type2 == 2 ) {
04730         if ( type != 2 ) { NumToStr(str,val); t = str; }
04731         if ( type2 != 2 ) { NumToStr(str,val2); t2 = str; }
04732         if ( eqop == 1 ) val = StrCmp(t,t2) != 0;
04733         else val = StrCmp(t,t2) == 0;

```

```

04734     }
04735     else {
04736         if ( eqop ) val = val != val2;
04737         else      val = val == val2;
04738     }
04739     return(val);
04740 }
04741
04742 /*
04743     #] PreEq :
04744     #[ pParseObject :
04745
04746     Parses a preprocessor object. We can have:
04747     1: a number (type = 1)
04748     2: a string (type = 2)
04749     3: an expression between parentheses (type = 0)
04750     4: a special function (type = 3)
04751     If the object is not a number, an expression or a special operator
04752     we try to interpret it as a string.
04753 */
04754
04755 UBYTE *pParseObject(UBYTE *s, int *type, LONG *val2)
04756 {
04757     UBYTE *t, c;
04758     int sign, val = 0;
04759     LONG x;
04760     while ( *s == ' ' || *s == '\t' ) s++;
04761     if ( *s == '(' ) {
04762         s++;
04763         while ( *s == ' ' || *s == '\t' || *s == '\n' || *s == '\r' ) s++;
04764         s = PreIfEval(s,&val);
04765         *type = 0;
04766         *val2 = val;
04767         return(s);
04768     }
04769     else if ( *s == '$' && s[1] == '(' ) {
04770         s += 2;
04771         while ( *s == ' ' || *s == '\t' || *s == '\n' || *s == '\r' ) s++;
04772         s = PreIfDollarEval(s,&val);
04773         *type = 0; *val2 = val;
04774         return(s);
04775     }
04776     if ( *s == 0 ) {
04777 illend:
04778         MesPrint("@illegal end of condition");
04779         return(0);
04780     }
04781     if ( *s == '"' ) {
04782         s++;
04783         while ( *s && *s != '"' ) {
04784             if ( *s == '\\' ) s++;
04785             s++;
04786         }
04787         if ( *s == 0 ) goto illend;
04788         else *s = 0;
04789         *type = 2;
04790         s++;
04791
04792         while ( *s == ' ' || *s == '\t' || *s == '\n' || *s == '\r' ) s++;
04793
04794         return(s);
04795     }
04796     t = s; sign = 1; x = 0;
04797     if ( chartype[*t] == 0 ) { /* Special operators and strings without "" */
04798         do { t++; } while ( chartype[*t] <= 1 );
04799         if ( *t == '(' ) {
04800             WORD ttype;
04801             c = *t; *t = 0;
04802             if ( StrICmp(s, (UBYTE *) "termsin") == 0 ) {
04803                 UBYTE *tt;
04804                 WORD numdol, numexp;
04805                 ttype = 0;
04806 together:
04807                 *t++ = c;
04808                 while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
04809                 if ( *t == '$' ) {
04810                     t++; tt = t; while ( chartype[*tt] <= 1 ) tt++;
04811                     c = *tt; *tt = 0;
04812                     if ( ( numdol = GetDollar(t) ) > 0 ) {
04813                         *tt = c;
04814                         if ( ttype == 1 ) {
04815                             x = SizeOfDollar(numdol);
04816                         }
04817                         else {
04818                             x = TermsInDollar(numdol);
04819                         }
04820                     }

```

```

04821         else {
04822             MesPrint("@%s has not (yet) been defined",t);
04823             *tt = c;
04824             Terminate(-1);
04825         }
04826     }
04827     else {
04828         tt = SkipAName(t);
04829         c = *tt; *tt = 0;
04830         if ( GetName(AC.exprnames,t,&numexp,NOAUTO) == NAMENOTFOUND ) {
04831             MesPrint("@%s has not (yet) been defined",t);
04832             *tt = c;
04833             Terminate(-1);
04834         }
04835         else {
04836             *tt = c;
04837             if ( ttype == 1 ) {
04838                 x = SizeOfExpression(numexp);
04839             }
04840             else {
04841                 x = TermsInExpression(numexp);
04842             }
04843         }
04844     }
04845     while ( *tt == ' ' || *tt == '\t'
04846             || *tt == '\n' || *tt == '\r' ) tt++;
04847     if ( *tt != ')' ) {
04848         MesPrint("@Improper use of terms($var) or terms(expr)");
04849         Terminate(-1);
04850     }
04851     *type = 3;
04852     s = tt+1;
04853     *val2 = x;
04854     return(s);
04855 }
04856 else if ( StrICmp(s,(UBYTE *)"sizeof") == 0 ) {
04857     ttype = 1;
04858     goto together;
04859 }
04860 else if ( StrICmp(s,(UBYTE *)"exists") == 0 ) {
04861     UBYTE *tt;
04862     WORD numdol, numexp;
04863     *tt++ = c;
04864     while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
04865     if ( *t == '$' ) {
04866         t++; tt = t; while ( chartype[*tt] <= 1 ) tt++;
04867         c = *tt; *tt = 0;
04868         if ( ( numdol = GetDollar(t) ) >= 0 ) { x = 1; }
04869         else { x = 0; }
04870         *tt = c;
04871     }
04872     else if ( *t == '"' ) { /* see whether a file exists */
04873         /* UBYTE *name, *oldname; */
04874         t++; tt = t;
04875         for (;;) {
04876             if ( *tt == '\\\ ' ) tt++;
04877             else if ( *tt == '"' ) break;
04878             tt++;
04879         }
04880         c = *tt; *tt = 0;
04881         /*
04882         Try to open the file. If possible, return 1.
04883         Afterwards close it.
04884         We do have to run through the FORMPATH. Hence we use LocateFile.
04885         This routine may change the name to the full name.
04886
04887         oldname = name = strDup1(t,"name in exists");
04888         x = LocateFile(&name,-1);
04889         */
04890         x = OpenFile((char *)t);
04891         if ( x >= 0 ) {
04892             CloseFile(x);
04893             x = 1;
04894             /*
04895             if ( name != oldname ) M_free(name,"name from LocateFile");
04896             */
04897         }
04898         else x = 0;
04899         /*
04900         M_free(oldname,"name in exists");
04901         */
04902         *tt++ = c;
04903     }
04904     else {
04905         tt = SkipAName(t);
04906         c = *tt; *tt = 0;
04907         if ( GetName(AC.exprnames,t,&numexp,NOAUTO) == NAMENOTFOUND ) { x = 0; }

```



```

04908         else { x = 1; }
04909         *tt = c;
04910     }
04911     while ( *tt == ' ' || *tt == '\t'
04912             || *tt == '\n' || *tt == '\r' ) tt++;
04913     if ( *tt != ')' ) {
04914         MesPrint("@Improper use of exists($var) or exists(expr)");
04915         Terminate(-1);
04916     }
04917     *type = 3;
04918     s = tt+1;
04919     *val2 = x;
04920     return(s);
04921 }
04922 else if ( StrICmp(s, (UBYTE *) "isnumerical") == 0 ) {
04923     GETIDENTITY
04924     UBYTE *tt;
04925     WORD numdol, numexp;
04926     *tt++ = c;
04927     while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
04928     if ( *t == '$' ) {
04929         tt = t; while ( chartype[*tt] <= 1 ) tt++;
04930         c = *tt; *tt = 0;
04931         if ( ( numdol = GetDollar(t) ) < 0 ) {
04932             MesPrint("@$ variable in isnumerical(%s) does not exist",t);
04933             Terminate(-1);
04934         }
04935         x = DolToLong(BHEAD numdol);
04936         if ( AN.ErrorInDollar ) {
04937             DOLLARS d = Dollars + numdol;
04938             x = 0;
04939             if ( d->type == DOLNUMBER || d->type == DOLTERMS ) {
04940                 if ( d->where[0] == 0 ) x = 1;
04941                 else if ( d->where[d->where[0]] == 0 ) {
04942                     if ( ABS(d->where[d->where[0]-1]) == d->where[0]-1 )
04943                         x = 1;
04944                 }
04945             }
04946         }
04947         else x = 1;
04948         *tt = c;
04949     }
04950     else {
04951         tt = SkipAName(t);
04952         c = *tt; *tt = 0;
04953         if ( GetName(AC.exprnames,t,&numexp,NOAUTO) == NAMENOTFOUND ) {
04954             MesPrint("@expression in isnumerical(%s) does not exist",t);
04955             Terminate(-1);
04956         }
04957         x = TermsInExpression(numexp);
04958         if ( x != 1 ) x = 0;
04959         else {
04960             WORD *term = AT.WorkPointer;
04961             if ( GetFirstTerm(term,numexp,1) < 0 ) {
04962                 MesPrint("@error reading expression in isnumerical(%s)",t);
04963                 Terminate(-1);
04964             }
04965             if ( *term == ABS(term[*term-1])+1 ) x = 1;
04966             else x = 0;
04967         }
04968         *tt = c;
04969     }
04970     while ( *tt == ' ' || *tt == '\t'
04971             || *tt == '\n' || *tt == '\r' ) tt++;
04972     if ( *tt != ')' ) {
04973         MesPrint("@Improper use of isnumerical($var) or numerical(expr)");
04974         Terminate(-1);
04975     }
04976     *type = 3;
04977     s = tt+1;
04978     *val2 = x;
04979     return(s);
04980 }
04981 else if ( StrICmp(s, (UBYTE *) "maxpowerof") == 0 ) {
04982     UBYTE *tt;
04983     WORD numsym;
04984     int stype;
04985     *tt++ = c;
04986     while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
04987     tt = SkipAName(t);
04988     c = *tt; *tt = 0;
04989     if ( ( stype = GetName(AC.varnames,t,&numsym,NOAUTO) ) == NAMENOTFOUND ) {
04990         MesPrint("@%s has not (yet) been defined",t);
04991         *tt = c;
04992         Terminate(-1);
04993     }
04994     else if ( stype != CSYMBOL ) {

```

```

04995         MesPrint("@%s should be a symbol",t);
04996         *tt = c;
04997         Terminate(-1);
04998     }
04999     else {
05000         *tt = c;
05001         x = symbols[numsym].maxpower;
05002     }
05003     while ( *tt == ' ' || *tt == '\t'
05004             || *tt == '\n' || *tt == '\r' ) tt++;
05005     if ( *tt != ')' ) {
05006         MesPrint("@Improper use of maxpowerof(symbol)");
05007         Terminate(-1);
05008     }
05009     *type = 3;
05010     s = tt+1;
05011     *val2 = x;
05012     return(s);
05013 }
05014 else if ( StrICmp(s,(UBYTE *)("minpowerof")) == 0 ) {
05015     UBYTE *tt;
05016     WORD numsym;
05017     int stype;
05018     *tt++ = c;
05019     while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
05020     tt = SkipAName(t);
05021     c = *tt; *tt = 0;
05022     if ( ( stype = GetName(AC.varnames,t,&numsym,NOAUTO) ) == NAMENOTFOUND ) {
05023         MesPrint("@%s has not (yet) been defined",t);
05024         *tt = c;
05025         Terminate(-1);
05026     }
05027     else if ( stype != CSYMBOL ) {
05028         MesPrint("@%s should be a symbol",t);
05029         *tt = c;
05030         Terminate(-1);
05031     }
05032     else {
05033         *tt = c;
05034         x = symbols[numsym].minpower;
05035     }
05036     while ( *tt == ' ' || *tt == '\t'
05037             || *tt == '\n' || *tt == '\r' ) tt++;
05038     if ( *tt != ')' ) {
05039         MesPrint("@Improper use of minpowerof(symbol)");
05040         Terminate(-1);
05041     }
05042     *type = 3;
05043     s = tt+1;
05044     *val2 = x;
05045     return(s);
05046 }
05047 else if ( StrICmp(s,(UBYTE *)("isfactorized")) == 0 ) {
05048     UBYTE *tt;
05049     WORD numdol, numexp;
05050     *tt++ = c;
05051     while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
05052     if ( *t == '$' ) {
05053         t++; tt = t; while (chartype[*tt] <= 1 ) tt++;
05054         c = *tt; *tt = 0;
05055         if ( ( numdol = GetDollar(t) ) > 0 ) {
05056             if ( Dollars[numdol].factors != 0 ) x = 1;
05057             else x = 0;
05058         }
05059         else {
05060             MesPrint("@ %s should be the name of an expression or a $ variable",t-1);
05061             Terminate(-1);
05062         }
05063         *tt = c;
05064     }
05065     else {
05066         tt = SkipAName(t);
05067         c = *tt; *tt = 0;
05068         if ( GetName(AC.exprnames,t,&numexp,NOAUTO) == NAMENOTFOUND ) {
05069             MesPrint("@ %s should be the name of an expression or a $ variable",t);
05070             Terminate(-1);
05071         }
05072         else {
05073             if ( ( Expressions[numexp].vflags & ISFACTORIZED ) != 0 ) x = 1;
05074             else x = 0;
05075         }
05076         *tt = c;
05077     }
05078     while ( *tt == ' ' || *tt == '\t'
05079             || *tt == '\n' || *tt == '\r' ) tt++;
05080     if ( *tt != ')' ) {
05081         MesPrint("@Improper use of isfactorized($var) or isfactorized(expr)");

```

```

05082         Terminate(-1);
05083     }
05084     *type = 3;
05085     s = tt+1;
05086     *val2 = x;
05087     return(s);
05088 }
05089 else if ( StrICmp(s, (UBYTE *) "isdefined") == 0 ) {
05090     UBYTE *tt;
05091     *t++ = c;
05092     while ( *t == ' ' || *t == '\t' || *t == '\n' || *t == '\r' ) t++;
05093     tt = SkipAName(t);
05094     c = *tt; *tt = 0;
05095     if ( GetPreVar(t, WITHOUTERROR) != 0 ) x = 1;
05096     else x = 0;
05097     *tt = c;
05098     while ( *tt == ' ' || *tt == '\t'
05099             || *tt == '\n' || *tt == '\r' ) tt++;
05100     if ( *tt != ')' ) {
05101         MesPrint("@Improper use of isdefined(var)");
05102         Terminate(-1);
05103     }
05104     *type = 3;
05105     s = tt+1;
05106     *val2 = x;
05107     return(s);
05108 }
05109 else if ( StrICmp(s, (UBYTE *) "flag") == 0 ) {
05110     UBYTE *tt;
05111     WORD x = 0, numexp;
05112     {
05113         *t++ = c;
05114         while ( *t == ' ' || *t == '\t' ) t++;
05115         if ( FG.cTable[*t] != 1 ) goto flagerror;
05116         while ( FG.cTable[*t] == 1 ) x = 10*x + (*t++ - '0');
05117         if ( x < 1 || x > BITSINWORD ) {
05118             MesPrint("@Illegal number %d for flag in flag condition", x);
05119             goto flagerror;
05120         }
05121         while ( *t == ' ' || *t == '\t' ) t++;
05122         if ( *t != ',' ) goto flagerror;
05123         t++;
05124         while ( *t == ' ' || *t == '\t' ) t++;
05125         tt = SkipAName(t);
05126         c = *tt; *tt = 0;
05127         if ( GetName(AC.exprnames, t, &numexp, NOAUTO) == NAMENOTFOUND ) {
05128             MesPrint("@ %s should be the name of an expression", t);
05129             goto flagerror;
05130         }
05131         *tt = c;
05132         while ( *t == ' ' || *t == '\t' ) t++;
05133         if ( *tt != ')' ) {
05134 flagerror:
05135             MesPrint("@Improper use of flag(num,expr)");
05136             Terminate(-1);
05137             return(0);
05138         }
05139         if ( ( Expressions[numexp].uflags & ( 1 « (x-1) ) ) != 0 )
05140             *val2 = 1;
05141         else *val2 = 0;
05142         *type = 3;
05143         s = tt+1;
05144         return(s);
05145     }
05146 }
05147 else *t = c;
05148 }
05149 else if ( *t == '=' || *t == '<' || *t == '>' || *t == '!'
05150 || *t == ')' || *t == ' ' || *t == '\t' || *t == 0 || *t == '\n' ) {
05151     *val2 = 0;
05152     *type = 2;
05153     return(t);
05154 }
05155 else {
05156     MesPrint("@Illegal use of string in preprocessor condition: %s", s);
05157     Terminate(-1);
05158 }
05159 }
05160 while ( *t == '-' || *t == '+' || *t == ' ' || *t == '\t' ) {
05161     if ( *t == '-' ) sign = -sign;
05162     t++;
05163 }
05164 while ( chartype[*t] == 1 ) { x = 10*x + *t++ - '0'; }
05165 while ( *t == ' ' || *t == '\t' ) t++;
05166 if ( chartype[*t] == 8 || *t == ')' || *t == '=' || *t == 0 ) {
05167     *val2 = sign > 0 ? x: -x;
05168     *type = 1;

```

```

05169         return(t);
05170     }
05171     while ( chartype[*t] != 8 && *t != ')' && *t != '=' && *t ) t++;
05172     while ( ( t > s ) && ( t[-1] == ' ' || t[-1] == '\t' ) ) t--;
05173     *type = 2;
05174     *val2 = val;
05175     return(t);
05176 }
05177
05178 /*
05179     #] pParseObject :
05180     #[ PreCalc :
05181
05182     To be called when a { is encountered.
05183     Action: read first till matching }. This is to be stored.
05184     Next we look whether this is a set or whether it can be
05185     evaluated. If it is a set we consider it as a new stream.
05186     The stream will have to be deallocated when read completely.
05187     If it is to be evaluated we do that and put the result in
05188     a stream.
05189 */
05190
05191 UBYTE *PreCalc(void)
05192 {
05193     UBYTE *buff, *s = 0, *t, *newb, c;
05194     int size, i, n, parlevel = 0, bralevel = 0;
05195     LONG answer;
05196     ULONG uanswer;
05197     size = n = 0;
05198     buff = 0; c = '{';
05199     for (;;) {
05200         if ( n >= size ) {
05201             if ( size == 0 ) size = 72;
05202             else size *= 2;
05203             if ( ( newb = (UBYTE *)Malloc1(size+2, "{}") ) == 0 ) return(0);
05204             s = newb;
05205             if ( buff ) {
05206                 i = n;
05207                 t = buff;
05208                 NCOPYB(s,t,i);
05209                 M_free(buff,"pre calc buffer");
05210             }
05211             else s = newb;
05212             buff = newb;
05213         }
05214         *s++ = c; n++;
05215         c = GetChar(0);
05216         if ( c == 0 ) {
05217             Error0("Unmatched {}");
05218             M_free(buff,"precalc buffer");
05219             return(0);
05220         }
05221         else if ( c == '{' ) { bralevel++; }
05222         else if ( c == '}' ) {
05223             if ( --bralevel < 0 ) { *s++ = c; *s = 0; break; }
05224         }
05225         else if ( c == '(' ) { parlevel++; }
05226         else if ( c == ')' ) {
05227             if ( --parlevel < 0 ) { *s++ = c; *s = 0; goto setstring; }
05228         }
05229         else if ( chartype[c] != 1 && chartype[c] != 5
05230             && chartype[c] != 6 && c != '!' && c != '&'
05231             && c != '|' && c != '\\') { *s++ = c; *s = 0; goto setstring; }
05232     }
05233     if ( parlevel > 0 ) goto setstring;
05234 /*
05235     Try now to evaluate the string.
05236     If it works, copy the resulting value back into buff as a string.
05237 */
05238     answer = 0;
05239     if ( PreEval(buff+1,&answer) == 0 ) goto setstring;
05240     t = buff + size;
05241     s = buff;
05242     if ( answer < 0 ) { *s++ = '-'; }
05243     uanswer = LongAbs(answer);
05244     n = 0;
05245     do {
05246         *--t = ( uanswer % 10 ) + '0';
05247         uanswer /= 10;
05248         n++;
05249     } while ( uanswer > 0 );
05250     NCOPYB(s,t,n);
05251     *s = 0;
05252     setstring;;
05253 /*
05254     Open a stream that contains the current string.
05255     Mark it to be removed after termination.

```

```

05256 */
05257     if ( OpenStream(buff,PRECALCSTREAM,0,PRENOACTION) == 0 ) return(0);
05258     return(buff);
05259 }
05260
05261 /*
05262     #] PreCalc :
05263     #[ PreEval :
05264
05265     Operations are:
05266     +, -, *, /, %, &, |, ^, !,  ^% (postfix 2log), ^/ (postfix sqrt)
05267 */
05268
05269 UBYTE *PreEval(UBYTE *s, LONG *x)
05270 {
05271     LONG y, z, a;
05272     int tobemultiplied, tobeadded = 1, expsign, i;
05273     UBYTE *t;
05274     *x = 0; a = 1;
05275     while ( *s == ' ' || *s == '\t' ) s++;
05276     for(;;){
05277         if ( *s == '+' || *s == '-' ) {
05278             if ( *s == '-' ) tobeadded = -1;
05279             else tobeadded = 1;
05280             s++;
05281             while ( *s == '-' || *s == '+' || *s == ' ' || *s == '\t' ) {
05282                 if ( *s == '-' ) tobeadded = -tobeadded;
05283                 s++;
05284             }
05285         }
05286         tobemultiplied = 0;
05287         for(;;){
05288             while ( *s == ' ' || *s == '\t' ) s++;
05289             if ( *s <= '9' && *s >= '0' ) {
05290                 ULONG uy;
05291                 ParseNumber(uy,s)
05292                 y = uy; /* may cause an implementation-defined behaviour */
05293             }
05294             else if ( *s == '(' || *s == '{' ) {
05295                 if ( ( t = PreEval(s+1,&y) ) == 0 ) return(0);
05296                 s = t;
05297             }
05298             else return(0);
05299             while ( *s == ' ' || *s == '\t' ) s++;
05300             expsign = 1;
05301             while ( *s == '^' || *s == '!' ) {
05302                 s++;
05303                 if ( s[-1] == '!' ) { /* factorial of course */
05304                     while ( *s == ' ' || *s == '\t' ) s++;
05305                     if ( y < 0 ) {
05306                         MesPrint("@Negative value in preprocessor factorial: %1",y);
05307                         return(0);
05308                     }
05309                     else if ( y == 0 ) y = 1;
05310                     else if ( y > 1 ) {
05311                         z = y-1;
05312                         while ( z > 0 ) { y = y*z; z--; }
05313                     }
05314                     continue;
05315                 }
05316                 else if ( *s == '%' ) { /* ^% is postfix 2log */
05317                     s++;
05318                     while ( *s == ' ' || *s == '\t' ) s++;
05319                     z = y;
05320                     if ( z <= 0 ) {
05321                         MesPrint("@Illegal value in preprocessor logarithm: %1",z);
05322                         return(0);
05323                     }
05324                     y = 0; z >= 1;
05325                     while ( z ) { y++; z >= 1; }
05326                     continue;
05327                 }
05328                 else if ( *s == '/' ) { /* ^/ is postfix sqrt */
05329                     LONG yy, zz;
05330                     s++;
05331                     while ( *s == ' ' || *s == '\t' ) s++;
05332                     z = y;
05333                     if ( z <= 0 ) {
05334                         MesPrint("@Illegal value in preprocessor square root: %1",z);
05335                         return(0);
05336                     }
05337                     if ( z > 8 ) { /* Very crude integer square root */
05338                         zz = z;
05339                         yy = 0; zz >= 1;
05340                         while ( zz ) { yy++; zz >= 1; }
05341                         zz = z > (yy/2); i = 10; y = 0;
05342                         do {

```

```

05343         yy = zz/2 + z/(2*zz); i--;
05344         if ( y == yy ) break;
05345         y = zz; zz = yy;
05346     } while ( y != yy && i > 0 );
05347     while ( y*y < z ) y++;
05348     while ( y*y > z ) y--;
05349 }
05350 else if ( z >= 4 ) y = 2;
05351 else if ( z == 0 ) y = 0;
05352 else y = 1;
05353 continue;
05354 }
05355 while ( *s == ' ' || *s == '\t' ) s++;
05356 while ( *s == '-' || *s == '+' || *s == ' ' || *s == '\t' ) {
05357     if ( *s == '-' ) expsign = -expsign;
05358 }
05359 if ( *s <= '9' && *s >= '0' ) {
05360     ParseNumber(z,s)
05361 }
05362 else if ( *s == '(' || *s == '{' ) {
05363     if ( ( t = PreEval(s+1,&z) ) == 0 ) return(0);
05364     s = t;
05365 }
05366 else return(0);
05367 while ( *s == ' ' || *s == '\t' ) s++;
05368 y = iexp(y,(int)z);
05369 }
05370 if ( tobemultiplied == 0 ) {
05371     if ( expsign < 0 ) a = 1/y;
05372     else a = y;
05373 }
05374 else {
05375     if ( tobemultiplied > 2 && expsign != 1 ) {
05376         MesPrint("&Incorrect use of ^ with & or |. Use brackets!");
05377         Terminate(-1);
05378     }
05379     tobemultiplied *= expsign;
05380     if ( tobemultiplied == 1 ) a *= y;
05381     else if ( tobemultiplied == 3 ) a &= y;
05382     else if ( tobemultiplied == 4 ) a |= y;
05383     else {
05384         if ( y == 0 || tobemultiplied == -2 ) {
05385             MesPrint("@Division by zero in preprocessor calculator");
05386             Terminate(-1);
05387         }
05388         if ( tobemultiplied == 2 ) a %= y;
05389         else a /= y;
05390     }
05391 }
05392 if ( *s == '%' ) tobemultiplied = 2;
05393 else if ( *s == '*' ) tobemultiplied = 1;
05394 else if ( *s == '/' ) tobemultiplied = -1;
05395 else if ( *s == '&' ) tobemultiplied = 3;
05396 else if ( *s == '|' ) tobemultiplied = 4;
05397 else {
05398     ULONG ux, ua;
05399     ux = *x;
05400     ua = a;
05401     if ( tobeadded >= 0 ) ux += ua;
05402     else ux -= ua;
05403     *x = ULONGToLong(ux);
05404     if ( *s == ')' || *s == '}' ) return(s+1);
05405     else if ( *s == '-' || *s == '+' ) { tobeadded = 1; break; }
05406     else return(0);
05407 }
05408 s++;
05409 }
05410 }
05411 /* return(0); */
05412 }
05413
05414 /*
05415     #] PreEval :
05416     #[ AddToPreTypes :
05417 */
05418
05419 void AddToPreTypes(int type)
05420 {
05421     if ( AP.NumPreTypes >= AP.MaxPreTypes ) {
05422         int i, *newlist = (int *)Malloc1(sizeof(int)*(2*AP.MaxPreTypes+1)
05423             ,"preprocessor type lists");
05424         for ( i = 0; i <= AP.MaxPreTypes; i++ ) newlist[i] = AP.PreTypes[i];
05425         M_free(AP.PreTypes,"preprocessor type lists");
05426         AP.PreTypes = newlist;
05427         AP.MaxPreTypes = 2*AP.MaxPreTypes;
05428     }
05429     AP.PreTypes[++AP.NumPreTypes] = type;

```

```

05430 }
05431
05432 /*
05433     #] AddToPreTypes :
05434     #[ MessPreNesting :
05435 */
05436
05437 void MessPreNesting(int par)
05438 {
05439     MesPrint("@(%d)Illegal nesting of %sif, %sdo, %sprocedure and/or %sswitch",par);
05440 }
05441
05442 /*
05443     #] MessPreNesting :
05444     #[ DoPreAddSeparator :
05445
05446     Preprocessor directives "addseparator" and "rmseparator" add/remove
05447     separator characters used to separate function arguments.
05448     Example:
05449
05450         #define QQ "a|g|a"
05451         #addseparator %
05452         *Comma must be quoted!:
05453         #rmseparator ",", "
05454         #rmseparator |
05455         #call H(a,a%`QQ`)
05456
05457     Characters ' ', '\t' and '"' are ignored!
05458 */
05459
05460 int DoPreAddSeparator(UBYTE *s)
05461 {
05462     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
05463     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
05464     for(;*s != '\0';s++){
05465         while ( *s == ' ' || *s == '\t' || *s == '"' ) s++;
05466         /* Todo:
05467         if ( set_in(*s,invalidseparators) ) {
05468             MesPrint("@Invalid separator specified");
05469             return(-1);
05470         }
05471         */
05472         set_set(*s,AC.separators);
05473     }
05474     return(0);
05475 }
05476
05477 /*
05478     #] DoPreAddSeparator :
05479     #[ DoPreRmSeparator :
05480
05481     See commentary with DoPreAddSeparator
05482
05483     Characters ' ', '\t' and '"' are ignored!
05484 */
05485 int DoPreRmSeparator(UBYTE *s)
05486 {
05487     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
05488     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
05489     for(;*s != '\0';s++){
05490         while ( *s == ' ' || *s == '\t' || *s == '"' ) s++;
05491         set_del(*s,AC.separators);
05492     }
05493     return(0);
05494 }
05495
05496 /*
05497     #] DoPreRmSeparator :
05498     #[ DoExternal:
05499
05500     #external ["prevar"] command
05501 */
05502 int DoExternal(UBYTE *s)
05503 {
05504     #ifdef WITHEXTERNALCHANNEL
05505         UBYTE *prevar=0;
05506         int externalD= 0;
05507     #else
05508         DUMMYUSE(s);
05509     #endif
05510     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
05511     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
05512     if ( AP.preError ) return(0);
05513
05514     #ifdef WITHEXTERNALCHANNEL
05515         while ( *s == ' ' || *s == '\t' ) s++;
05516         if(*s == '"'){/*prevar to store the descriptor is defined*/

```

```

05517     prevar++;
05518
05519     if ( chartype[*s] == 0 )for( ;*s != '\0'; s++)switch(chartype[*s]){
05520         case 10:/*'\0' fits here*/
05521             MesPrint("@Can't finde closing \"");
05522             Terminate(-1);
05523         case 0:case 1: continue;
05524         default:
05525             break;
05526     }
05527     if(*s != '\0'){
05528         MesPrint("@Illegal name of preprocessor variable to store external channel");
05529         return(-1);
05530     }
05531     *s='\0';
05532     for(s++; *s == ' ' || *s == '\t'; s++);
05533 }
05534
05535 if(*s == '\0'){
05536     MesPrint("@Illegal external command");
05537     return(-1);
05538 }
05539 /*here s is a command*/
05540 /*See the file extcmd.c*/
05541 /*[08may2006 mt]:*/
05542 externalD=openExternalChannel(
05543     s,
05544     AX.daemonize,
05545     AX.shellname,
05546     AX.stderrname);
05547 /*:[08may2006 mt]*/
05548 if(externalD<1){/*error?*/
05549     /*Not quite correct - terminate the program on error:*/
05550     Error1("Can't start external program",s);
05551     return(-1);
05552 }
05553 /*Now external command runs.*/
05554
05555 if(prevar){/*Store the external channel descriptor in the provided variable:*/
05556     UBYTE buf[21];/* 64/Log_2[10] = 19.3, so this is enough forever...*/
05557     NumToStr(buf,externalD);
05558     if ( PutPreVar(prevar,buf,0,1) < 0 ) return(-1);
05559 }
05560
05561 AX.currentExternalChannel=externalD;
05562 /*[08may2006 mt]:*/
05563 if(AX.currentPrompt!=0){/*Change default terminator*/
05564     if(setTerminatorForExternalChannel( (char *)AX.currentPrompt)){
05565         MesPrint("@Prompt is too long");
05566         return(-1);
05567     }
05568 }
05569 setKillModeForExternalChannel(AX.killSignal,AX.killWholeGroup);
05570 /*:[08may2006 mt]*/
05571 return(0);
05572 #else /*ifdef WITHEXTERNALCHANNEL*/
05573 Error0("External channel: not implemented on this computer/system");
05574 return(-1);
05575 #endif /*ifdef WITHEXTERNALCHANNEL ... else*/
05576 }
05577
05578 /*
05579     #] DoExternal:
05580     #[ DoPrompt:
05581         #prompt string
05582 */
05583
05584 int DoPrompt(UBYTE *s)
05585 {
05586     #ifndef WITHEXTERNALCHANNEL
05587         DUMMYUSE(s);
05588     #endif
05589     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
05590     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
05591
05592     #ifdef WITHEXTERNALCHANNEL
05593         while ( *s == ' ' || *s == '\t' ) s++;
05594         if ( AX.currentPrompt )
05595             M_free(AX.currentPrompt,"external channel prompt");
05596         if ( *s == '\0' )
05597             AX.currentPrompt = (UBYTE *)strDupl((UBYTE *)"", "external channel prompt");
05598         else
05599             AX.currentPrompt = strDupl(s,"external channel prompt");
05600         if( setTerminatorForExternalChannel( (char *)AX.currentPrompt) > 0 ){
05601             MesPrint("@Prompt is too long");
05602             return(-1);
05603         }
05604     }

```



```

05604     /*else: if 0, ok; if -1, there is no current channel-ok, just prompt is stored.*/
05605     return(0);
05606 #else /*ifdef WITHEXTERNALCHANNEL*/
05607     Error0("External channel: not implemented on this computer/system");
05608     return(-1);
05609 #endif /*ifdef WITHEXTERNALCHANNEL ... else*/
05610 }
05611 /*
05612     #] DoPrompt:
05613     #[ DoSetExternal:
05614         #setexternal n
05615 */
05616
05617 int DoSetExternal(UBYTE *s)
05618 {
05619 #ifdef WITHEXTERNALCHANNEL
05620     int n=0;
05621 #else
05622     DUMMYUSE(s);
05623 #endif
05624     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
05625     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
05626     if ( AP.preError ) return(0);
05627
05628 #ifdef WITHEXTERNALCHANNEL
05629     while ( *s == ' ' || *s == '\t' ) s++;
05630     while ( charType[*s] == 1 ) { n = 10*n + *s++ - '0'; }
05631     while ( *s == ' ' || *s == '\t' ) s++;
05632     if (*s!='\0'){
05633         MesPrint("@setexternal: number expected");
05634         return(-1);
05635     }
05636     if(selectExternalChannel(n)<0){
05637         MesPrint("@setexternal: invalid number");
05638         return(-1);
05639     }
05640     AX.currentExternalChannel=n;
05641     return(0);
05642 #else /*ifdef WITHEXTERNALCHANNEL*/
05643     Error0("External channel: not implemented on this computer/system");
05644     return(-1);
05645 #endif /*ifdef WITHEXTERNALCHANNEL ... else*/
05646 }
05647 /*
05648     #] DoSetExternal:
05649     #[ DoSetExternalAttr:
05650 */
05651
05652 static FORM_INLINE UBYTE *pickupword(UBYTE *s)
05653 {
05654
05655     for(;*s>' ';s++)switch(*s){
05656         case '=':
05657         case ',':
05658         case ';':
05659             return(s);
05660     }/*for(;*s>' ';s++)switch(*s)*/
05661     return(s);
05662 }
05663 /*Returns 0 if the first string (case insensitively) equal to
05664 the beginning of the second string (of length n):
05665 */
05666 static inline int strINCmp(UBYTE *a, UBYTE *b, int n)
05667 {
05668     for(;n>0;n--)if(tolower(*a++)!=tolower(*b++))
05669         return(1);
05670     return(*a != '\0');
05671 }
05672
05673 #define KILL "kill"
05674 #define KILLALL "killall"
05675 #define DAEMON "daemon"
05676 #define SHELL "shell"
05677 #define STDERR "stderr"
05678
05679 #define TRUE_EXPR "true"
05680 #define FALSE_EXPR "false"
05681 #define NOSHELL "noshell"
05682 #define TERMINAL "terminal"
05683
05684 /*
05685     Expects comma-separated list of pairs name=value
05686 */
05687 int DoSetExternalAttr(UBYTE *s)
05688 {
05689 #ifdef WITHEXTERNALCHANNEL
05690     int lnam,lval;

```

```

05691     UBYTE *nam,*val;
05692 #else
05693     DUMMYUSE(s);
05694 #endif
05695     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
05696     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
05697     if ( AP.preError ) return(0);
05698
05699 #ifndef WITHEXTERNALCHANNEL
05700     do{
05701         /*Read the name:*/
05702         while ( *s == ' ' || *s == '\t' ) s++;
05703         s=pickupword(nam=s);
05704         lnam=s-nam;
05705         while ( *s == ' ' || *s == '\t' ) s++;
05706         if(*s++!=''){
05707             MesPrint("@External channel:' ' expected instead of %s",s-1);
05708             return(-1);
05709         }
05710         /*Read the value:*/
05711         while ( *s == ' ' || *s == '\t' ) s++;
05712         val=s;
05713
05714         for(;;){
05715             UBYTE *m;
05716             s=pickupword(s);
05717             m=s;
05718             while ( *s == ' ' || *s == '\t' ) s++;
05719             if ( (*s == ',') || (*s == '\n') || (*s == ';') || (*s == '\0') ) {
05720                 s=m;
05721                 break;
05722             }
05723         } /*for(;;)*/
05724
05725         lval=s-val;
05726         while ( *s == ' ' || *s == '\t' ) s++;
05727
05728         if(strINcmp((UBYTE *)SHELL,nam,lnam)==0){
05729             if(AX.shellname!=NULL)
05730                 M_free(AX.shellname,"external channel shellname");
05731             if(strINcmp((UBYTE *)NOSHELL,val,lval)==0)
05732                 AX.shellname=NULL;
05733             else{
05734                 UBYTE *ch,*b;
05735                 b=ch=AX.shellname=Malloc1(lval+1,"external channel shellname");
05736                 while(ch-b<lval)
05737                     *ch++=*val++;
05738                 *ch='\0';
05739             }
05740         }else if(strINcmp((UBYTE *)DAEMON,nam,lnam)==0){
05741             if(strINcmp((UBYTE *)TRUE_EXPR,val,lval)==0)
05742                 AX.daemonize = 1;
05743             else if(strINcmp((UBYTE *)FALSE_EXPR,val,lval)==0)
05744                 AX.daemonize = 0;
05745             else{
05746                 MesPrint("@External channel:true or false expected for %s",DAEMON);
05747                 return(-1);
05748             }
05749         }else if(strINcmp((UBYTE *)KILLALL,nam,lnam)==0){
05750             if(strINcmp((UBYTE *)TRUE_EXPR,val,lval)==0)
05751                 AX.killWholeGroup = 1;
05752             else if(strINcmp((UBYTE *)FALSE_EXPR,val,lval)==0)
05753                 AX.killWholeGroup = 0;
05754             else{
05755                 MesPrint("@External channel: true or false expected for %s",KILLALL);
05756                 return(-1);
05757             }
05758         }else if(strINcmp((UBYTE *)KILL,nam,lnam)==0){
05759             int i,n=0;
05760             for(i=0;i<lval;i++){
05761                 if( *val>='0' && *val<= '9' )
05762                     n = 10*n + *val++ - '0';
05763                 else{
05764                     MesPrint("@External channel: number expected for %s",KILL);
05765                     return(-1);
05766                 }
05767             }
05768             AX.killSignal=n;
05769         }else if(strINcmp((UBYTE *)STDERR,nam,lnam)==0){
05770             if( AX.stdername != NULL ) {
05771                 M_free(AX.stdername,"external channel stderrname");
05772             }
05773             if(strINcmp((UBYTE *)TERMINAL,val,lval)==0)
05774                 AX.stdername = NULL;
05775             else{
05776                 UBYTE *ch,*b;
05777                 b=ch=AX.stdername=Malloc1(lval+1,"external channel stderrname");

```

```

05778         while(ch-b<lval)
05779             *ch++=*val++;
05780         *ch='\0';
05781     }
05782 }else{
05783     nam[lname+1]='\0';
05784     MesPrint("@External channel: unrecognized attribute",nam);
05785     return(-1);
05786 }
05787 }while(*s++ == ' ');
05788 if( (*s-1)>' ')&&(*s-1)!=';') {
05789     MesPrint("@External channel: syntax error: %s",s-1);
05790     return(-1);
05791 }
05792 return(0);
05793 #else /*ifdef WITHEXTERNALCHANNEL*/
05794     Error0("External channel: not implemented on this computer/system");
05795     return(-1);
05796 #endif /*ifdef WITHEXTERNALCHANNEL ... else*/
05797 }
05798 /*
05799     #] DoSetExternalAttr:
05800     #[ DoRmExternal:
05801         #rmexternal [n] (if 0, close all)
05802 */
05803
05804 int DoRmExternal(UBYTE *s)
05805 {
05806     #ifdef WITHEXTERNALCHANNEL
05807         int n = -1;
05808     #else
05809         DUMMYUSE(s);
05810     #endif
05811     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
05812     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
05813     if ( AP.preError ) return(0);
05814
05815     #ifdef WITHEXTERNALCHANNEL
05816         while ( *s == ' ' || *s == '\t' ) s++;
05817         if( chartype[*s] == 1 ){
05818             for(n=0; chartype[*s] == 1 ; s++) { n = 10*n + *s - '0'; }
05819             while ( *s == ' ' || *s == '\t' ) s++;
05820         }
05821         if(*s!='\0'){
05822             MesPrint("@rmexternal: invalid number");
05823             return(-1);
05824         }
05825         switch(n){
05826             case 0:/*Close all opened channels*/
05827                 closeAllExternalChannels();
05828                 AX.currentExternalChannel=0;
05829                 /*Do not clean AX.currentPrompt!*/
05830                 return(0);
05831             case -1:/*number is not specified - try current*/
05832                 n=AX.currentExternalChannel;
05833                 /* fall through */
05834             default:
05835                 closeExternalChannel(n);/*No reaction for possible error*/
05836         }
05837         if (n == AX.currentExternalChannel)/*cleaned up by closeExternalChannel()*/
05838             AX.currentExternalChannel=0;
05839         return(0);
05840     #else /*ifdef WITHEXTERNALCHANNEL*/
05841         Error0("External channel: not implemented on this computer/system");
05842         return(-1);
05843     #endif /*ifdef WITHEXTERNALCHANNEL ... else*/
05844 }
05845
05846 /*
05847     #] DoRmExternal:
05848     #[ DoFromExternal :
05849         #fromexternal
05850             is used to read the text from the running external
05851             program, the syntax is similar to the #include
05852             directive.
05853         #fromexternal "varname"
05854             is used to read the text from the running external
05855             program into the preprocessor variable varname.
05856             directive.
05857         #fromexternal "varname" maxlength
05858             is used to read the text from the running external
05859             program into the preprocessor variable varname.
05860             directive. Only first maxlength characters are
05861             stored.
05862
05863             FORM continues to read the running external
05864             program output until the external program outputs a

```

```

05865             prompt.
05866
05867 */
05868
05869 int DoFromExternal(UBYTE *s)
05870 {
05871     #ifdef WITHEXTERNALCHANNEL
05872         UBYTE *prevar=0;
05873         int lbuf=-1;
05874         int withNoList=AC.NoShowInput;
05875         int oldpreassignflag;
05876     #else
05877         DUMMYUSE(s);
05878     #endif
05879     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
05880     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
05881     if ( AP.preError ) return(0);
05882     #ifdef WITHEXTERNALCHANNEL
05883
05884         FLUSHCONSOLE;
05885
05886         while ( *s == ' ' || *s == '\t' ) s++;
05887         /*[17may2006 mt]:*/
05888         if ( *s == '-' || *s == '+' ) {
05889             if ( *s == '-' )
05890                 withNoList = 1;
05891             else
05892                 withNoList = 0;
05893             s++;
05894             while ( *s == ' ' || *s == '\t' ) s++;
05895         }/*if ( *s == '-' || *s == '+' )*/
05896         /*:[17may2006 mt]*/
05897         /*[02feb2006 mt]:*/
05898         if(*s == '"')/*prevar to store the output is defined*/
05899             prevar=++s;
05900
05901         if ( *s=='$' || chartype[*s] == 0 )for(;*s != '"'; s++)switch(chartype[*s]){
05902             case 10:/*'\0' fits here*/
05903                 MesPrint("@Can't finde closing \"");
05904                 Terminate(-1);
05905             case 0:case 1: continue;
05906             default:
05907                 break;
05908         }
05909         if(*s != '"'){
05910             MesPrint("@Illegal name to store output of external channel");
05911             return(-1);
05912         }
05913         *s='\0';
05914         for(s++; *s == ' ' || *s == '\t'; s++);
05915     }/*if(*s == '"')*/
05916
05917     if(*s != '\0'){
05918         if( chartype[*s] == 1 ){
05919             for(lbuf=0; chartype[*s] == 1; s++) { lbuf = 10*lbuf + *s - '0'; }
05920             while ( *s == ' ' || *s == '\t' ) s++;
05921         }
05922         if( (*s!='\0')||(lbuf<0) ){
05923             MesPrint("@Illegal buffer length in fromexternal");
05924             return(-1);
05925         }
05926     }/*if(*s != '\0')*/
05927     /*:[02feb2006 mt]*/
05928     if(getCurrentExternalChannel() != AX.currentExternalChannel)
05929         /*[08may20006 mt]:*/
05930         /*selectExternalChannel(AX.currentExternalChannel);*/
05931         if(selectExternalChannel(AX.currentExternalChannel)){
05932             MesPrint("@No current external channel");
05933             return(-1);
05934         }
05935         /*:[08may20006 mt]*/
05936
05937     /*[02feb2006 mt]:*/
05938     if(prevar!=0){/*The result must be stored into preprovar*/
05939         UBYTE *buf;
05940         int cc = 0;
05941         if(lbuf == -1){/*Unlimited buffer, everything must be stored*/
05942             int i;
05943             buf=Malloc1( (lbuf=255)+1,"Fromexternal");
05944             /*[18may20006 mt]:*/
05945             /*for(i=0; (cc=getcFromExtChannel()) != EOF; i++) {*/
05946             /* May 2006: now getcFromExtChannelOk returns EOF while
05947             getcFromExtChannelFailure returns -2 (see comments in
05948             exctcmd.c):*/
05949             for(i=0; (cc=getcFromExtChannel())>0; i++) {
05950                 /*:[18may20006 mt]*/
05951                 if(i==lbuf){

```

```

05952         int j;
05953         UBYTE *tmp=Malloc1( (lbuf*=2)+1,"Fromexternal");
05954         for (j=0;j<i;j++)tmp[j]=buf[j];
05955         M_free(buf,"Fromexternal");
05956         buf=tmp;
05957     }
05958     buf[i]=(UBYTE)(cc);
05959     }/*for(i=0;(cc=getcFromExtChannel())>0;i++)*/
05960     /*[18may20006 mt]*/
05961     if(cc == -2){
05962         MesPrint("@No current external channel");
05963         return(-1);
05964     }
05965     lbuf=i;
05966     /*:[18may20006 mt]*/
05967     buf[i]='\0';
05968 }else{/*Fixed buffer, only lbuf chars must be stored*/
05969     int i;
05970     buf=Malloc1(lbuf+1,"Fromexternal");
05971     for(i=0; i<lbuf;i++){
05972         /*[18may20006 mt]*/
05973         /*if( (cc=getcFromExtChannel())==EOF )*/
05974         /* May 2006: now getcFromExtChannelOk returns EOF while
05975         getcFromExtChannelFailure returns -2 (see comments in
05976         exctcmd.c)*/
05977         if( (cc=getcFromExtChannel())<1 )
05978         /*:[18may20006 mt]*/
05979             break;
05980         buf[i]=(UBYTE)(cc);
05981     }
05982     buf[i]='\0';
05983     /*[18may20006 mt]*/
05984     /*if(cc!=EOF)
05985         while(getcFromExtChannel()!=EOF);**/*Eat the rest*/
05986     /* May 2006: now getcFromExtChannelOk returns EOF while
05987     getcFromExtChannelFailure returns -2 (see comments in
05988     exctcmd.c)*/
05989     if(cc>0)
05990         while(getcFromExtChannel()>0);/*Eat the rest*/
05991     else if(cc == -2){
05992         MesPrint("@No current external channel");
05993         return(-1);
05994     }
05995     /*:[18may20006 mt]*/
05996 }
05997 /*[18may20006 mt]*/
05998 if(*prevar == '$'){/*Put the answer to the dollar variable*/
05999     int oldNumPotModdollars = NumPotModdollars;
06000 #ifdef WITHMPI
06001     WORD oldRhsExprInModuleFlag = AC.RhsExprInModuleFlag;
06002     AC.RhsExprInModuleFlag = 0;
06003 #endif
06004     /*Here lbuf is the actual length of buf!*/
06005     /*"prevar=buf'\0'":*/
06006     UBYTE *pbuf=Malloc1(StrLen(prevar)+1+lbuf+1,"Fromexternal to dollar");
06007     UBYTE *c=pbuf;
06008     UBYTE *b=prevar;
06009     while(*b!='\0'){*c++ = *b++;}
06010     *c++='=';
06011     b=buf;
06012     while( (*c++=*b++)!='\0' );
06013     oldpreassignflag = AP.PreAssignFlag;
06014     AP.PreAssignFlag = 1;
06015     if ( ( cc = CompileStatement(pbuf) ) || ( cc = CatchDollar(0) ) ) {
06016         Error1("External channel: can't assign output to dollar variable ",prevar);
06017     }
06018     AP.PreAssignFlag = oldpreassignflag;
06019     NumPotModdollars = oldNumPotModdollars;
06020 #ifdef WITHMPI
06021     AC.RhsExprInModuleFlag = oldRhsExprInModuleFlag;
06022 #endif
06023     M_free(pbuf,"Fromexternal to dollar");
06024 }else{
06025     cc = PutPreVar(prevar, buf, 0, 1) < 0;
06026 }
06027 /*:[18may20006 mt]*/
06028 M_free(buf,"Fromexternal");
06029 if ( cc ) return(-1);
06030 return(0);
06031 }
06032 /*:[02feb2006 mt]*/
06033 if ( OpenStream(s,EXTERNALCHANNELSTREAM,0,PREENOACTION) == 0 ) return(-1);
06034 /*[17may2006 mt]*/
06035 AC.NoShowInput = withNoList;
06036 /*:[17may2006 mt]*/
06037 return(0);
06038 #else

```

```

06039     Error0("External channel: not implemented on this computer/system");
06040     return(-1);
06041 #endif
06042 }
06043
06044 /*
06045     #] DoFromExternal :
06046     #[ DoToExternal :
06047         #toexetrnal
06048 */
06049
06050 #ifdef WITHEXTERNALCHANNEL
06051
06052 /*A wrapper to writeBufToExtChannel, see the file extcmd.c:*/
06053 LONG WriteToExternalChannel(int handle, UBYTE *buffer, LONG size)
06054 {
06055     /*ATT! handle is not used! Actual output is performed to
06056        the current external channel, see extcmd.c!*/
06057     DUMMYUSE(handle);
06058     if(writeBufToExtChannel((char*)buffer,size))
06059         return(-1);
06060     return(size);
06061 }
06062 #endif /*ifdef WITHEXTERNALCHANNEL*/
06063
06064 int DoToExternal(UBYTE *s)
06065 {
06066     #ifdef WITHEXTERNALCHANNEL
06067         HANDLERS h;
06068         LONG (*OldWrite)(int handle, UBYTE *buffer, LONG size) = WriteFile;
06069         int ret=-1;
06070     #else
06071         DUMMYUSE(s);
06072     #endif
06073     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
06074     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
06075     if ( AP.preError ) return(0);
06076     #ifdef WITHEXTERNALCHANNEL
06077
06078     h.oldsilent=AM.silent;
06079     h.newlogonly = h.oldlogonly = AM.FileOnlyFlag;
06080     h.newhandle = h.oldhandle = AC.LogHandle;
06081     h.oldprinttype = AO.PrintType;
06082
06083     WriteFile=&WriteToExternalChannel;
06084
06085     while ( *s == ' ' || *s == '\t' ) s++;
06086
06087     if(AX.currentExternalChannel==0){
06088         MesPrint("@No current external channel");
06089         goto DoToExternalReady;
06090     }
06091
06092     if(getCurrentExternalChannel()!=AX.currentExternalChannel)
06093         selectExternalChannel(AX.currentExternalChannel);
06094
06095     ret=writeToChannel(EXTERNALCHANNELOUT,s,&h);
06096     DoToExternalReady:
06097         WriteFile=OldWrite;
06098         return(ret);
06099     #else /*ifdef WITHEXTERNALCHANNEL*/
06100     Error0("External channel: not implemented on this computer/system");
06101     return(-1);
06102 #endif /*ifdef WITHEXTERNALCHANNEL ... else*/
06103
06104 }
06105
06106 /*
06107     #] DoToExternal :
06108     #[ defineChannel :
06109 */
06110
06111 UBYTE *defineChannel(UBYTE *s, HANDLERS *h)
06112 {
06113     UBYTE *name,*to;
06114
06115     if ( *s != '<' )
06116         return(s);
06117
06118     s++;
06119     name = to = s;
06120     while ( *s && *s != '>' ) {
06121         if ( *s == '\\\ ' ) s++;
06122         *to++ = *s++;
06123     }
06124     if ( *s == 0 ) {
06125         MesPrint("@Improper termination of filename");

```

```

06126         return(0);
06127     }
06128     s++;
06129     *to = 0;
06130     if ( *name ) {
06131         h->newhandle = GetChannel((char *)name,0);
06132         h->newlogonly = 1;
06133     }
06134     else if ( AC.LogHandle >= 0 ) {
06135         h->newhandle = AC.LogHandle;
06136         h->newlogonly = 1;
06137     }
06138     return(s);
06139 }
06140
06141 /*
06142     #] defineChannel :
06143     #[ writeToChannel :
06144 */
06145
06146 int writeToChannel(int wtype, UBYTE *s, HANDLERS *h)
06147 {
06148     UBYTE *to, *fstring, *ss, *sss, *sl, c, cl;
06149     WORD num, number, nfac;
06150     WORD oldOptimizationLevel;
06151     UBYTE Out[MAXLINELENGTH+14], *stopper;
06152     int nosemi, i;
06153     int plus = 0;
06154
06155     /*
06156     Now determine the format string
06157     */
06158     while ( *s == ',' || *s == ' ' ) s++;
06159     if ( *s != '"' ) {
06160         MesPrint("@No format string present");
06161         return(-1);
06162     }
06163     s++; fstring = to = s;
06164     while ( *s ) {
06165         if ( *s == '\\' ) {
06166             s++;
06167             if ( *s == '\\' ) {
06168                 *to++ = *s++;
06169                 if ( *s == '\\' ) *to++ = *s++;
06170             }
06171             else if ( *s == '"' ) *to++ = *s++;
06172             else { *to++ = '\\'; *to++ = *s++; }
06173         }
06174         else if ( *s == '"' ) break;
06175         else *to++ = *s++;
06176     }
06177     if ( *s != '"' ) {
06178         MesPrint("@No closing \" in format string");
06179         return(-1);
06180     }
06181     *to = 0; s++;
06182     if ( AC.LineLength > 20 && AC.LineLength <= MAXLINELENGTH ) stopper = Out + AC.LineLength;
06183     else stopper = Out + MAXLINELENGTH;
06184     to = Out;
06185
06186     /*
06187     s points now at the list of objects (if any)
06188     we can start executing the format string.
06189     */
06189     AM.silent = 0;
06190     AC.LogHandle = h->newhandle;
06191     AM.FileOnlyFlag = h->newlogonly;
06192     if ( h->newhandle >= 0 ) {
06193         AO.PrintType |= PRINTLFILE;
06194     }
06195     while ( *fstring ) {
06196         if ( to >= stopper ) {
06197             if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
06198                 *to++ = '&';
06199             }
06200             num = to - Out;
06201             WriteString(wtype,Out,num);
06202             to = Out;
06203             if ( AC.OutputMode == FORTRANMODE
06204                 || AC.OutputMode == PFORTRANMODE ) {
06205                 number = 7;
06206                 for ( i = 0; i < number; i++ ) *to++ = ' ';
06207                 to[-2] = '&';
06208             }
06209         }
06210         if ( *fstring == '\\' ) {
06211             fstring++;
06212             if ( *fstring == 'n' ) {

```

```

06213         num = to - Out;
06214         WriteString(wtype,Out,num);
06215         to = Out;
06216         fstring++;
06217     }
06218     else if ( *fstring == 't' ) { *to++ = '\\t'; fstring++; }
06219     else if ( *fstring == 'b' ) { *to++ = '\\b'; fstring++; }
06220     else *to++ = *fstring++;
06221 }
06222 else if ( *fstring == '%' ) {
06223     plus = 0;
06224 retry:
06225     fstring++;
06226     if ( *fstring == 'd' ) {
06227         int sign,dig;
06228         number = -1;
06229 donumber:
06230         while ( *s == ',' || *s == ' ' || *s == '\\t' ) s++;
06231         sign = 1;
06232         while ( *s == '+' || *s == '-' ) {
06233             if ( *s == '-' ) sign = -sign;
06234             s++;
06235         }
06236         dig = 0; ss = s; if ( sign < 0 ) { ss--; *ss = '-'; dig++; }
06237         while ( *s >= '0' && *s <= '9' ) { s++; dig++; }
06238         if ( number < 0 ) {
06239             while ( ss < s ) {
06240                 if ( to >= stopper ) {
06241                     num = to - Out;
06242                     WriteString(wtype,Out,num);
06243                     to = Out;
06244                 }
06245                 if ( *ss == '\\\\' ) ss++;
06246                 *to++ = *ss++;
06247             }
06248         }
06249         else {
06250             if ( number < dig ) { dig = number; ss = s - dig; }
06251             while ( number > dig ) {
06252                 if ( to >= stopper ) {
06253                     num = to - Out;
06254                     WriteString(wtype,Out,num);
06255                     to = Out;
06256                 }
06257                 *to++ = ' '; number--;
06258             }
06259             while ( ss < s ) {
06260                 if ( to >= stopper ) {
06261                     num = to - Out;
06262                     WriteString(wtype,Out,num);
06263                     to = Out;
06264                 }
06265                 if ( *ss == '\\\\' ) ss++;
06266                 *to++ = *ss++;
06267             }
06268         }
06269         fstring++;
06270     }
06271     else if ( *fstring == '$' ) {
06272         UBYTE *dolalloc;
06273         number = AO.OutSkip;
06274 dodollar:
06275         while ( *s == ',' || *s == ' ' || *s == '\\t' ) s++;
06276         if ( AC.OutputMode == FORTRANMODE
06277             || AC.OutputMode == PFORTRANMODE ) {
06278             number = 7;
06279         }
06280         if ( *s != '$' ) {
06281 nodollar:
06282             MesPrint("@$-variable expected in #write instruction");
06283             AM.FileOnlyFlag = h->oldlogonly;
06284             AC.LogHandle = h->oldhandle;
06285             AO.PrintType = h->oldprinttype;
06286             AM.silent = h->oldsilent;
06287             return(-1);
06288         }
06289         s++; ss = s;
06290         while ( chartype[*s] <= 1 ) s++;
06291         if ( s == ss ) goto nodollar;
06292         c = *s; *s = 0;
06293         num = GetDollar(ss);
06294         if ( num < 0 ) {
06295             MesPrint("@#write instruction: %s has not been defined",ss);
06296             AM.FileOnlyFlag = h->oldlogonly;
06297             AC.LogHandle = h->oldhandle;
06298             AO.PrintType = h->oldprinttype;
06299             AM.silent = h->oldsilent;
06300             return(-1);

```



```

06300     }
06301     *s = c;
06302     if ( *s == '[' ) {
06303         if ( Dollars[num].nfactors <= 0 ) {
06304             *s = 0;
06305             MesPrint("@#write instruction: $%s has not been factorized",ss);
06306             AM.FileOnlyFlag = h->oldlogonly;
06307             AC.LogHandle = h->oldhandle;
06308             AO.PrintType = h->oldprinttype;
06309             AM.silent = h->oldsilent;
06310             return(-1);
06311         }
06312         /*
06313         Now get the number between the []
06314         */
06315         nfac = GetDollarNumber(&s,Dollars+num);
06316
06317         if ( Dollars[num].nfactors == 1 && nfac == 1 ) goto writewhole;
06318
06319         if ( ( dolalloc = WriteDollarFactorToBuffer(num,nfac,0) ) == 0 ) {
06320             AM.FileOnlyFlag = h->oldlogonly;
06321             AC.LogHandle = h->oldhandle;
06322             AO.PrintType = h->oldprinttype;
06323             AM.silent = h->oldsilent;
06324             return(-1);
06325         }
06326         goto writealloc;
06327     }
06328     else if ( *s && *s != ' ' && *s != ',' && *s != '\t' ) {
06329         MesPrint("@#write instruction: illegal characters after $-variable");
06330         AM.FileOnlyFlag = h->oldlogonly;
06331         AC.LogHandle = h->oldhandle;
06332         AO.PrintType = h->oldprinttype;
06333         AM.silent = h->oldsilent;
06334         return(-1);
06335     }
06336     else {
06337 writewhole:
06338         if ( ( dolalloc = WriteDollarToBuffer(num,0) ) == 0 ) {
06339             AM.FileOnlyFlag = h->oldlogonly;
06340             AC.LogHandle = h->oldhandle;
06341             AO.PrintType = h->oldprinttype;
06342             AM.silent = h->oldsilent;
06343             return(-1);
06344         }
06345         else {
06346 writealloc:
06347             ss = dolalloc;
06348             while ( *ss ) {
06349                 if ( to >= stopper ) {
06350                     if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
06351                         *to++ = '&';
06352                     }
06353                     num = to - Out;
06354                     WriteString(wtype,Out,num);
06355                     to = Out;
06356                     for ( i = 0; i < number; i++ ) *to++ = ' ';
06357                     if ( AC.OutputMode == FORTRANMODE
06358                         || AC.OutputMode == PFORTRANMODE ) to[-2] = '&';
06359                 }
06360                 if ( chartype[*ss] > 3 ) { *to++ = *ss++; }
06361                 else {
06362                     sss = ss; while ( chartype[*ss] <= 3 ) ss++;
06363                     if ( ( to + (ss-sss) ) >= stopper ) {
06364                         if ( (ss-sss) >= (stopper-Out) ) {
06365                             if ( ( to - stopper ) < 10 ) {
06366                                 if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 ==
ISFORTRAN90 ) {
06367                                     *to++ = '&';
06368                                 }
06369                                 num = to - Out;
06370                                 WriteString(wtype,Out,num);
06371                                 to = Out;
06372                                 for ( i = 0; i < number; i++ ) *to++ = ' ';
06373                                 if ( AC.OutputMode == FORTRANMODE
06374                                     || AC.OutputMode == PFORTRANMODE ) to[-2] = '&';
06375                             }
06376                             while ( (ss-sss) >= (stopper-Out) ) {
06377                                 while ( to < stopper-1 ) {
06378                                     *to++ = *sss++;
06379                                 }
06380                                 if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 ==
ISFORTRAN90 ) {
06381                                     *to++ = '&';
06382                                 }
06383                                 else {
06384                                     *to++ = '\\';

```

```

06385         }
06386         num = to - Out;
06387         WriteString(wtype,Out,num);
06388         to = Out;
06389         if ( AC.OutputMode == FORTRANMODE
06390             || AC.OutputMode == PFORTRANMODE ) {
06391             for ( i = 0; i < number; i++ ) *to++ = ' ';
06392             to[-2] = '&';
06393         }
06394     }
06395 }
06396 else {
06397     if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90
06398 ) {
06399         *to++ = '&';
06400     }
06401     num = to - Out;
06402     WriteString(wtype,Out,num);
06403     to = Out;
06404     for ( i = 0; i < number; i++ ) *to++ = ' ';
06405     if ( AC.OutputMode == FORTRANMODE
06406         || AC.OutputMode == PFORTRANMODE ) to[-2] = '&';
06407 }
06408 while ( sss < ss ) *to++ = *sss++;
06409 }
06410 }
06411 }
06412 M_free(dolalloc,"written dollar");
06413 fstring++;
06414 }
06415 }
06416 else if ( *fstring == 's' ) {
06417     fstring++;
06418     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
06419     if ( *s == '"' ) {
06420         s++; ss = s;
06421         while ( *s ) {
06422             if ( *s == '\\' ) s++;
06423             else if ( *s == '"' ) break;
06424             s++;
06425         }
06426         if ( *s == 0 ) {
06427             MesPrint("@#write instruction: Missing \" in string");
06428             AM.FileOnlyFlag = h->oldlogonly;
06429             AC.LogHandle = h->oldhandle;
06430             AO.PrintType = h->oldprinttype;
06431             AM.silent = h->oldsilent;
06432             return(-1);
06433         }
06434         while ( ss < s ) {
06435             if ( to >= stopper ) {
06436                 num = to - Out;
06437                 WriteString(wtype,Out,num);
06438                 to = Out;
06439             }
06440             if ( *ss == '\\' ) ss++;
06441             *to++ = *ss++;
06442         }
06443         s++;
06444     }
06445     else {
06446         sss = ss = s;
06447         while ( *s && *s != ',' ) {
06448             if ( *s == '\\' ) { s++; sss = s+1; }
06449             s++;
06450         }
06451         while ( s > sss+1 && ( s[-1] == ' ' || s[-1] == '\t' ) ) s--;
06452         while ( ss < s ) {
06453             if ( to >= stopper ) {
06454                 num = to - Out;
06455                 WriteString(wtype,Out,num);
06456                 to = Out;
06457             }
06458             if ( *ss == '\\' ) ss++;
06459             *to++ = *ss++;
06460         }
06461     }
06462 }
06463 else if ( *fstring == 'X' ) {
06464     fstring++;
06465     if ( cbuf[AM.sbufnum].numrhs > 0 ) {
06466         /*
06467         This should be only to the value of AM.oldnumextrasyms
06468         */
06469         UBYTE *s = GetPreVar(AM.oldnumextrasyms,0);
06470         WORD x = 0;

```

```

06471         while ( *s >= '0' && *s <= '9' ) x = 10*x + *s++ - '0';
06472         if ( x > 0 )
06473             PrintSubtermList(1,x);
06474         else
06475             PrintSubtermList(1,cbuf[AM.sbufnum].numrhs);
06476     }
06477 }
06478 else if ( *fstring == 'O' ) {
06479     number = AO.OutSkip;
06480 dooptim:
06481     fstring++;
06482 /*
06483     First test whether there is an optimization buffer
06484 */
06485     if ( AO.OptimizeResult.code == NULL && AO.OptimizationLevel != 0 ) {
06486         MesPrint("@In #write instruction: no optimization results available!");
06487         return(-1);
06488     }
06489     num = to - Out;
06490     WriteString(wtype,Out,num);
06491     to = Out;
06492     if ( AO.OptimizationLevel != 0 ) {
06493         WORD oldoutskip = AO.OutSkip;
06494         AO.OutSkip = number;
06495         optimize_print_code(0);
06496         AO.OutSkip = oldoutskip;
06497     }
06498 }
06499 else if ( *fstring == 'e' || *fstring == 'E' ) {
06500     if ( *fstring == 'E'
06501         || AC.OutputMode == FORTRANMODE
06502         || AC.OutputMode == PFORTRANMODE ) nosemi = 1;
06503     else nosemi = 0;
06504     fstring++;
06505     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
06506     if ( chartype[*s] != 0 && *s != '[' ) {
06507 noexpr:
06508         MesPrint("@expression name expected in #write instruction");
06509         AM.FileOnlyFlag = h->oldlogonly;
06510         AC.LogHandle = h->oldhandle;
06511         AO.PrintType = h->oldprinttype;
06512         AM.silent = h->oldsilent;
06513         return(-1);
06514     }
06515     ss = s;
06516     if ( ( s = SkipAName(ss) ) == 0 || s[-1] == '_' ) goto noexpr;
06517     s1 = s; c = c1 = *s1;
06518     if ( c1 == '(' ) {
06519         SKIPBRA3(s)
06520         if ( *s == ')' ) {
06521             AO.CurBufWrt = s1+1;
06522             c = *s; *s = 0;
06523         }
06524         else {
06525             MesPrint("@Illegal () specifier in expression name in #write");
06526             AM.FileOnlyFlag = h->oldlogonly;
06527             AC.LogHandle = h->oldhandle;
06528             AO.PrintType = h->oldprinttype;
06529             AM.silent = h->oldsilent;
06530             return(-1);
06531         }
06532     }
06533     else AO.CurBufWrt = (UBYTE *)underscore;
06534     *s1 = 0;
06535     num = to - Out;
06536     if ( num > 0 ) WriteUnfinString(wtype,Out,num);
06537     to = Out;
06538     oldOptimizationLevel = AO.OptimizationLevel;
06539     AO.OptimizationLevel = 0;
06540     if ( WriteOne(ss,(int)num,nosemi,plus) < 0 ) {
06541         AM.FileOnlyFlag = h->oldlogonly;
06542         AC.LogHandle = h->oldhandle;
06543         AO.PrintType = h->oldprinttype;
06544         AM.silent = h->oldsilent;
06545         return(-1);
06546     }
06547     AO.OptimizationLevel = oldOptimizationLevel;
06548     *s1 = c1;
06549     if ( s > s1 ) *s++ = c;
06550 }
06551 /*
06552 File content
06553 */
06554 else if ( ( *fstring == 'f' ) || ( *fstring == 'F' ) ) {
06555     LONG n;
06556     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
06557     ss = s;
06558     while ( *s && *s != ',' ) {

```

```

06558         if ( *s == '\\\\' ) s++;
06559         s++;
06560     }
06561     c = *s; *s = 0;
06562     s1 = LoadInputFile(ss,HEADERFILE);
06563     *s = c;
06564     /*
06565     There should have been a way to pass the file size.
06566     Also there should be conversions for \r\n etc.
06567     */
06568     if ( s1 ) {
06569         ss = s1; while ( *ss ) ss++;
06570         n = ss-s1;
06571         WriteString(wtype,s1,n);
06572         M_free(s1,"copy file");
06573     }
06574     else if ( *fstring == 'F' ) {
06575         *s = 0;
06576         MesPrint("@Error in #write: could not open file %s",ss);
06577         *s = c;
06578         goto ReturnWithError;
06579     }
06580     fstring++;
06581 }
06582 else if ( *fstring == '%' ) {
06583     *to++ = *fstring++;
06584 }
06585 else if ( FG.cTable[*fstring] == 1 ) { /* %#S */
06586     number = 0;
06587     while ( FG.cTable[*fstring] == 1 ) {
06588         number = 10*number + *fstring++ - '0';
06589     }
06590     if ( *fstring == 'O' ) goto dooptim;
06591     else if ( *fstring == 'd' ) goto donumber;
06592     else if ( *fstring == '$' ) goto dodollar;
06593     else if ( *fstring == 't' ) { /* 'tab' position */
06594         if ( number < (WORD)(stopper-Out) ) {
06595             while ( (WORD)(to-Out) < number ) *to++ = ' ';
06596         }
06597         fstring++;
06598     }
06599     else if ( *fstring == 'X' || *fstring == 'x' ) {
06600         if ( number > 0 && number <= cbuf[AM.sbufnum].numrhs ) {
06601             UBYTE buffer[80], *out, *old1, *old2, *old3;
06602             WORD *term, first;
06603             if ( *fstring == 'X' ) {
06604                 out = StrCopy((UBYTE *)AC.extrasym,buffer);
06605                 if ( AC.extrasymbols == 0 ) {
06606                     out = NumCopy(number,out);
06607                     out = StrCopy((UBYTE *)"_",out);
06608                 }
06609                 else if ( AC.extrasymbols == 1 ) {
06610                     if ( AC.OutputMode == CMODE ) {
06611                         out = StrCopy((UBYTE *)"[",out);
06612                         out = NumCopy(number,out);
06613                         out = StrCopy((UBYTE *)"]",out);
06614                     }
06615                     else {
06616                         out = StrCopy((UBYTE *)"(",out);
06617                         out = NumCopy(number,out);
06618                         out = StrCopy((UBYTE *)")",out);
06619                     }
06620                 }
06621                 out = StrCopy((UBYTE *)"=",out);
06622                 ss = buffer;
06623                 while ( ss < out ) {
06624                     if ( to >= stopper ) {
06625                         num = to - Out;
06626                         WriteString(wtype,Out,num);
06627                         to = Out;
06628                     }
06629                     *to++ = *ss++;
06630                 }
06631             }
06632             term = cbuf[AM.sbufnum].rhs[number];
06633             first = 1;
06634             if ( *term == 0 ) {
06635                 *to++ = '0';
06636             }
06637             else {
06638                 old1 = AO.OutFill;
06639                 old2 = AO.OutputLine;
06640                 old3 = AO.OutStop;
06641                 AO.OutFill = to;
06642                 AO.OutputLine = Out;
06643                 AO.OutStop = Out + AC.LineLength;
06644                 while ( *term ) {

```

```

06645             if ( WriteInnerTerm(term,first) ) Terminate(-1);
06646             term += *term;
06647             first = 0;
06648         }
06649         to = Out + (AO.OutFill-AO.OutputLine);
06650         AO.OutFill = old1;
06651         AO.OutputLine = old2;
06652         AO.OutStop = old3;
06653     }
06654 }
06655 fstring++;
06656 }
06657 else {
06658     goto IllegControlSequence;
06659 }
06660 }
06661 else if ( *fstring == '+' ) {
06662     plus = 1; goto retry;
06663 }
06664 else if ( *fstring == 0 ) {
06665     *to++ = 0;
06666 }
06667 else {
06668     IllegControlSequence:
06669     MesPrint("@Illegal control sequence in format string in #write instruction");
06670     ReturnWithError:
06671     AM.FileOnlyFlag = h->oldlogonly;
06672     AC.LogHandle = h->oldhandle;
06673     AO.PrintType = h->oldprinttype;
06674     AM.silent = h->oldsilent;
06675     return(-1);
06676 }
06677 }
06678 else {
06679     *to++ = *fstring++;
06680 }
06681 }
06682 /*
06683     Now flush the output
06684 */
06685 num = to - Out;
06686 /*[15Apr2004 mt]:*/
06687 if(wtype==EXTERNALCHANNELOUT){
06688     if(num!=0)
06689         WriteUnfinString(wtype,Out,num);
06690 }else
06691 /*:[15Apr2004 mt]*/
06692 WriteString(wtype,Out,num);
06693 /*
06694     and restore original parameters
06695 */
06696 AM.FileOnlyFlag = h->oldlogonly;
06697 AC.LogHandle = h->oldhandle;
06698 AO.PrintType = h->oldprinttype;
06699 AM.silent = h->oldsilent;
06700 return(0);
06701 }
06702
06703 /*
06704     #] writeToChannel :
06705     #[ DoFactDollar :
06706
06707     Executes the #factdollar $var
06708     instruction
06709 */
06710
06711 int DoFactDollar(UBYTE *s)
06712 {
06713     GETIDENTITY
06714     WORD numdollar, *oldworkpointer;
06715
06716     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
06717     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
06718     while ( *s == ' ' || *s == '\t' ) s++;
06719     if ( *s == '$' ) {
06720         if ( GetName(AC.dollarnames,s+1,&numdollar,NOAUTO) != CDOLLAR ) {
06721             MesPrint("@%s is undefined",s);
06722             return(-1);
06723         }
06724         s = SkipAName(s+1);
06725         if ( *s != 0 ) {
06726             MesPrint("@#FactDollar should have a single $variable for its argument");
06727             return(-1);
06728         }
06729         NewSort(BHEAD0);
06730         oldworkpointer = AT.WorkPointer;
06731         if ( DollarFactorize(BHEAD numdollar) ) return(-1);

```

```

06732     AT.WorkPointer = oldworkpointer;
06733     LowerSortLevel();
06734     return(0);
06735 }
06736 else if ( ParenthesesTest(s) ) return(-1);
06737 else {
06738     MesPrint("@#FactDollar should have a single $variable for its argument");
06739     return -1;
06740 }
06741 }
06742
06743 /*
06744     #] DoFactDollar :
06745     #[ GetDollarNumber :
06746 */
06747
06748 WORD GetDollarNumber(UBYTE **inp, DOLLARS d)
06749 {
06750     UBYTE *s = *inp, c, *name;
06751     WORD number, nfac, *w;
06752     DOLLARS dd;
06753     s++;
06754     if ( *s == '$' ) {
06755         s++; name = s;
06756         while ( FG.cTable[*s] < 2 ) s++;
06757         c = *s; *s = 0;
06758         if ( GetName(AC.dollarnames,name,&number,NOAUTO) == NAMENOTFOUND ) {
06759             MesPrint("@dollar in #write should have been defined previously");
06760             Terminate(-1);
06761         }
06762         *s = c;
06763         dd = Dollars + number;
06764         if ( c == '[' ) {
06765             *inp = s;
06766             nfac = GetDollarNumber(inp,dd);
06767             s = *inp;
06768             if ( *s != ']' ) {
06769                 MesPrint("@Illegal factor for dollar variable");
06770                 Terminate(-1);
06771             }
06772             *inp = s+1;
06773             if ( nfac == 0 ) {
06774                 if ( dd->nfactors > d->nfactors ) {
06775 TooBig:
06776                     MesPrint("@Factor number for dollar variable too large");
06777                     Terminate(-1);
06778                 }
06779                 return(dd->nfactors);
06780             }
06781             w = dd->factors[nfac-1].where;
06782             if ( w == 0 ) {
06783                 if ( dd->factors[nfac-1].value > d->nfactors ||
06784                     dd->factors[nfac-1].value < 0 ) goto TooBig;
06785                 return(dd->factors[nfac-1].value);
06786             }
06787             if ( *w == 4 && w[4] == 0 && w[3] == 3 && w[2] == 1
06788                 && w[1] <= d->nfactors ) return(w[1]);
06789             if ( w[*w] == 0 && w[*w-1] == *w-1 ) goto TooBig;
06790 IllNum:
06791             MesPrint("@Illegal factor number for dollar variable");
06792             Terminate(-1);
06793         }
06794     } else { /* The dollar should be a number */
06795         if ( dd->type == DOLZERO ) {
06796             return(0);
06797         }
06798         else if ( dd->type == DOLTERMS || dd->type == DOLNUMBER ) {
06799             w = dd->where;
06800             if ( *w == 4 && w[4] == 0 && w[3] == 3 && w[2] == 1
06801                 && w[1] <= d->nfactors ) return(w[1]);
06802             if ( w[*w] == 0 && w[*w-1] == *w-1 ) goto TooBig;
06803             goto IllNum;
06804         }
06805         else goto IllNum;
06806     }
06807 }
06808 else if ( FG.cTable[*s] == 1 ) {
06809     WORD x = *s++ - '0';
06810     while ( FG.cTable[*s] == 1 ) {
06811         x = 10*x + *s++ - '0';
06812         if ( x > d->nfactors ) {
06813             MesPrint("@Factor number %d for dollar variable too large",x);
06814             Terminate(-1);
06815         }
06816     }
06817     if ( *s != ']' ) {
06818         MesPrint("@Illegal factor number for dollar variable");

```

```

06819         Terminate(-1);
06820     }
06821     s++; *inp = s;
06822     return(x);
06823 }
06824 else {
06825     MesPrint("@Illegal factor indicator for dollar variable");
06826     Terminate(-1);
06827 }
06828 return(-1);
06829 }
06830
06831 /*
06832     #] GetDollarNumber :
06833     #[ DoSetRandom :
06834
06835     Executes the #SetRandom number
06836 */
06837
06838 int DoSetRandom(UBYTE *s)
06839 {
06840     ULONG x;
06841     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
06842     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
06843     while ( *s == ' ' || *s == '\t' ) s++;
06844     x = 0;
06845     while ( FG.cTable[*s] == 1 ) {
06846         x = 10*x + (*s++-'0');
06847     }
06848     while ( *s == ' ' || *s == '\t' ) s++;
06849     if ( *s == 0 ) {
06850 #ifdef WITHPTHREADS
06851 #ifdef WITHSORTBOTS
06852         int id, totnum = MaX(2*AM.totalnumberofthreads-3,AM.totalnumberofthreads);
06853 #else
06854         int id, totnum = AM.totalnumberofthreads;
06855 #endif
06856         for ( id = 0; id < totnum; id++ ) {
06857             AB[id]->R.wranfseed = x;
06858             if ( AB[id]->R.wranfia ) M_free(AB[id]->R.wranfia,"wranf");
06859             AB[id]->R.wranfia = 0;
06860         }
06861 #else
06862         AR.wranfseed = x;
06863         if ( AR.wranfia ) M_free(AR.wranfia,"wranf");
06864         AR.wranfia = 0;
06865 #endif
06866         return(0);
06867     }
06868     else {
06869         MesPrint("@proper syntax is #SetRandom number");
06870         return(-1);
06871     }
06872 }
06873
06874 /*
06875     #] DoSetRandom :
06876     #[ DoOptimize :
06877
06878     Executes the #Optimize(expr) instruction.
06879 */
06880
06881 int DoOptimize(UBYTE *s)
06882 {
06883     GETIDENTITY
06884     UBYTE *exprname;
06885     WORD numexpr;
06886     int error = 0, i;
06887     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
06888     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
06889     DUMMYUSE(*s)
06890     exprname = s; s = SkipAName(s);
06891     if ( *s != 0 && *s != ';' ) {
06892         MesPrint("@proper syntax is #Optimize,expression");
06893         return(-1);
06894     }
06895     *s = 0;
06896     if ( GetName(AC.exprnames,exprname,&numexpr,NOAUTO) != CEXPRESSION ) {
06897         MesPrint("@%s is not an expression",exprname);
06898         error = 1;
06899     }
06900     else if ( AP.preError == 0 ) {
06901         EXPRESSIONS e = Expressions + numexpr;
06902         POSITION position;
06903         int firstterm;
06904         WORD *term = AT.WorkPointer;
06905         ClearOptimize();

```

```

06906         if ( AO.OptimizationLevel == 0 ) return(0);
06907         switch ( e->status ) {
06908             case LOCALEXPRESSION:
06909             case GLOBALEXPRESSION:
06910                 break;
06911             default:
06912                 MesPrint("@Expression %s is not an active unhidden local or global
expression.",exprname);
06913                 Terminate(-1);
06914                 break;
06915         }
06916 #ifdef WITHMPI
06917         if ( PF.me == MASTER )
06918 #endif
06919             RevertScratch();
06920         for ( i = NumExpressions-1; i >= 0; i-- ) {
06921             AS.OldOnFile[i] = Expressions[i].onfile;
06922             AS.OldNumFactors[i] = Expressions[i].numfactors;
06923             AS.Oldvflags[i] = Expressions[i].vflags;
06924             Expressions[i].vflags &= ~(ISUNMODIFIED|ISZERO);
06925         }
06926         for ( i = 0; i < NumExpressions; i++ ) {
06927             if ( i == numexpr ) {
06928                 PutPreVar(AM.oldnumextrasymbols,
06929                     GetPreVar((UBYTE *)"EXTRASYMBOLS_",0),0,1);
06930                 Optimize(numexpr, 0);
06931                 AO.OptimizeResult.nameofexpr = strDup1(exprname,"optimize expression name");
06932                 continue;
06933             }
06934 #ifdef WITHMPI
06935             if ( PF.me == MASTER ) {
06936 #endif
06937                 e = Expressions + i;
06938                 switch ( e->status ) {
06939                     case LOCALEXPRESSION:
06940                     case SKIPLEXPRESSION:
06941                     case DROPLEXPRESSION:
06942                     case DROPEDEXPRESSION:
06943                     case GLOBALEXPRESSION:
06944                     case SKIPGEXPRESSION:
06945                     case DROPGEXPRESSION:
06946                     case HIDELEXPRESSION:
06947                     case HIDEGEXPRESSION:
06948                     case DROPHLEXPRESSION:
06949                     case DROPHGEXPRESSION:
06950                     case INTOHIDELEXPRESSION:
06951                     case INTOHIDEGEXPRESSION:
06952                         break;
06953                     default:
06954                         continue;
06955                 }
06956                 AR.GetFile = 0;
06957                 SetScratch(AR.infile,&(e->onfile));
06958                 if ( GetTerm(BHEAD term) <= 0 ) {
06959                     MesPrint("@Expression %d has problems reading from scratchfile",i);
06960                     Terminate(-1);
06961                 }
06962                 term[3] = i;
06963                 AR.DeferFlag = 0;
06964                 SeekScratch(AR.outfile,&position);
06965                 e->onfile = position;
06966                 *AM.S0->sBuffer = 0; firstterm = -1;
06967                 do {
06968                     WORD *oldipointer = AR.CompressPointer;
06969                     WORD *comprtop = AR.ComprTop;
06970                     AR.ComprTop = AM.S0->sTop;
06971                     AR.CompressPointer = AM.S0->sBuffer;
06972                     if ( firstterm > 0 ) {
06973                         if ( PutOut(BHEAD term,&position,AR.outfile,1) < 0 ) goto DoSerr;
06974                     }
06975                     else if ( firstterm < 0 ) {
06976                         if ( PutOut(BHEAD term,&position,AR.outfile,0) < 0 ) goto DoSerr;
06977                         firstterm++;
06978                     }
06979                     else {
06980                         if ( PutOut(BHEAD term,&position,AR.outfile,-1) < 0 ) goto DoSerr;
06981                         firstterm++;
06982                     }
06983                     AR.CompressPointer = oldipointer;
06984                     AR.ComprTop = comprtop;
06985                 } while ( GetTerm(BHEAD term) );
06986                 if ( FlushOut(&position,AR.outfile,1) ) {
06987 DoSerr:
06988                     MesPrint("@Expression %d has problems writing to scratchfile",i);
06989                     Terminate(-1);
06990                 }
06991 #ifdef WITHMPI

```



```

06992         }
06993     #endif
06994     }
06995     /*
06996         Now some administration and we are done
06997     */
06998     UpdateMaxSize();
06999     }
07000     else {
07001         ClearOptimize();
07002     }
07003     return(error);
07004 }
07005 }
07006
07007 /*
07008     #] DoOptimize :
07009     #[ DoClearOptimize :
07010
07011     Clears all relevant buffers of the output optimization
07012 */
07013
07014 int DoClearOptimize(UBYTE *s)
07015 {
07016     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07017     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07018     DUMMYUSE(*s);
07019     return(ClearOptimize());
07020 }
07021
07022 /*
07023     #] DoClearOptimize :
07024     #[ DoSkipExtraSymbols :
07025
07026     Adds the intermediate variables of the previous optimization
07027     to the list of extra symbols, provided it has not yet been erased
07028     by a #clearoptimize
07029     To remove them again one needs to use the 'delete extrasymbols;'
07030     or the 'delete extrasymbols>num;' statement in which num is the
07031     old number of extra symbols.
07032 */
07033
07034 int DoSkipExtraSymbols(UBYTE *s)
07035 {
07036     CBUF *C = cbuf + AM.sbufnum;
07037     WORD tt = 0, j = 0, oldval = AO.OptimizeResult.minvar;
07038     if ( AO.OptimizeResult.code == NULL ) return(0);
07039     if ( AO.OptimizationLevel == 0 ) return(0);
07040     while ( *s == ',' ) s++;
07041     if ( *s == 0 ) {
07042         AO.OptimizeResult.minvar = AO.OptimizeResult.maxvar+1;
07043     }
07044     else {
07045         while ( *s <= '9' && *s >= '0' ) j = 10*j + *s++ - '0';
07046         if ( *s ) {
07047             MesPrint("@Illegal use of #SkipExtraSymbols instruction");
07048             Terminate(-1);
07049         }
07050         AO.OptimizeResult.minvar += j;
07051         if ( AO.OptimizeResult.minvar > AO.OptimizeResult.maxvar )
07052             AO.OptimizeResult.minvar = AO.OptimizeResult.maxvar+1;
07053     }
07054     j = AO.OptimizeResult.minvar - oldval;
07055     while ( j > 0 ) {
07056         AddRHS(AM.sbufnum,1);
07057         AddNtoC(AM.sbufnum,1,&tt,16);
07058         AddToCB(C,0)
07059         InsTree(AM.sbufnum,C->numrhs);
07060         j--;
07061     }
07062     return(0);
07063 }
07064
07065 /*
07066     #] DoSkipExtraSymbols :
07067     #[ DoPreReset :
07068
07069     Does a reset of variables.
07070     Currently only the timer (stopwatch) of 'timer_'
07071 */
07072
07073 int DoPreReset(UBYTE *s)
07074 {
07075     UBYTE *ss, c;
07076     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07077     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07078     while ( *s == ' ' || *s == '\t' ) s++;

```

```

07079     if ( *s == 0 ) {
07080         MesPrint("@proper syntax is #Reset variable");
07081         return(-1);
07082     }
07083     ss = s;
07084     while ( FG.cTable[*s] == 0 ) s++;
07085     c = *s; *s = 0;
07086     if ( ( StrICmp(ss, (UBYTE *) "timer") == 0 )
07087         || ( StrICmp(ss, (UBYTE *) "stopwatch") == 0 ) ) {
07088         *s = c;
07089         AP.StopWatchZero = GetRunningTime();
07090         return(0);
07091     }
07092     else {
07093         *s = c;
07094         MesPrint("@proper syntax is #Reset variable");
07095         return(-1);
07096     }
07097 }
07098
07099 /*
07100     #] DoPreReset :
07101     #[ DoPreAppendPath :
07102 */
07103
07104 static int DoAddPath(UBYTE *s, int bPrepend)
07105 {
07106     /* NOTE: this doesn't support some file systems, e.g., 0x5c with CP932. */
07107
07108     UBYTE *path, *path_end, *current_dir, *current_dir_end, *NewPath, *t;
07109     int bRelative, n;
07110
07111     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07112     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07113
07114     /* Parse the path in the input. */
07115     while ( *s == ' ' || *s == '\t' ) s++; /* skip spaces */
07116     if ( *s == '"' ) { /* the path is given by "..." */
07117         path = ++s;
07118         while ( *s && *s != '"' ) {
07119             if ( SEPARATOR != '\\ ' && *s == '\\ ' ) { /* escape character, e.g., "\\\" */
07120                 if ( !s[1] ) goto ImproperPath;
07121                 s++;
07122             }
07123             s++;
07124         }
07125         if ( *s != '"' ) goto ImproperPath;
07126         path_end = s++;
07127     }
07128     else {
07129         path = s;
07130         while ( *s && *s != ' ' && *s != '\t' ) {
07131             if ( SEPARATOR != '\\ ' && *s == '\\ ' ) { /* escape character, e.g., "\" */
07132                 if ( !s[1] ) goto ImproperPath;
07133                 s++;
07134             }
07135             s++;
07136         }
07137         path_end = s;
07138     }
07139     if ( path == path_end ) goto ImproperPath; /* empty path */
07140     while ( *s == ' ' || *s == '\t' ) s++; /* skip spaces */
07141     if ( *s ) goto ImproperPath; /* extra tokens found */
07142
07143     /* Check if the path is an absolute path. */
07144     bRelative = 1;
07145     if ( path[0] == SEPARATOR ) { /* starts with the directory separator */
07146         bRelative = 0;
07147     }
07148 #ifdef WINDOWS
07149     else if ( chartype[path[0]] == 0 && path[1] == ':' ) { /* starts with (drive letter): */
07150         bRelative = 0;
07151     }
07152 #endif
07153
07154     /* Get the current file directory when a relative path is given. */
07155     if ( bRelative ) {
07156         if ( !AC.CurrentStream ) goto FileNameUnavailable;
07157         if ( AC.CurrentStream->type != FILESTREAM && AC.CurrentStream->type != REVERSEFILESTREAM )
07158             goto FileNameUnavailable;
07159         if ( !AC.CurrentStream->name ) goto FileNameUnavailable;
07160         s = current_dir = current_dir_end = AC.CurrentStream->name;
07161         while ( *s ) {
07162             if ( SEPARATOR != '\\ ' && *s == '\\ ' && s[1] ) { /* escape character, e.g., "\\\" */
07163                 s += 2;
07164                 continue;
07165             }

```

```

07165         if ( *s == SEPARATOR ) {
07166             current_dir_end = s;
07167         }
07168         s++;
07169     }
07170 }
07171 else {
07172     current_dir = current_dir_end = NULL;
07173 }
07174
07175 /* Allocate a buffer for new AM.Path. */
07176 n = path_end - path;
07177 if ( AM.Path ) n += StrLen(AM.Path) + 1;
07178 if ( current_dir != current_dir_end ) n+= current_dir_end - current_dir + 1;
07179 s = NewPath = (UBYTE *)Malloca1(n + 1,"add path");
07180
07181 /* Construct new FORM path. */
07182 if ( bPrepend ) {
07183     if ( current_dir != current_dir_end ) {
07184         t = current_dir;
07185         while ( t != current_dir_end ) *s++ = *t++;
07186         *s++ = SEPARATOR;
07187     }
07188     t = path;
07189     while ( t != path_end ) *s++ = *t++;
07190     if ( AM.Path ) *s++ = PATHSEPARATOR;
07191 }
07192 if ( AM.Path ) {
07193     t = AM.Path;
07194     while ( *t ) *s++ = *t++;
07195 }
07196 if ( !bPrepend ) {
07197     if ( AM.Path ) *s++ = PATHSEPARATOR;
07198     if ( current_dir != current_dir_end ) {
07199         t = current_dir;
07200         while ( t != current_dir_end ) *s++ = *t++;
07201         *s++ = SEPARATOR;
07202     }
07203     t = path;
07204     while ( t != path_end ) *s++ = *t++;
07205 }
07206 *s = '\0';
07207
07208 /* Update AM.Path. */
07209 if ( AM.Path ) M_free(AM.Path,"add path");
07210 AM.Path = NewPath;
07211
07212 return(0);
07213
07214 ImproperPath:
07215 MesPrint("@Improper syntax for %%%sPath", bPrepend ? "Prepend" : "Append");
07216 return(-1);
07217
07218 FileNameUnavailable:
07219 /* This may be improved in future. */
07220 MesPrint("@Sorry, %%%sPath can't resolve the current file name from here", bPrepend ? "Prepend" :
"Append");
07221 return(-1);
07222 }
07223
07231 int DoPreAppendPath(UBYTE *s)
07232 {
07233     return DoAddPath(s, 0);
07234 }
07235
07236 /*
07237     #] DoPreAppendPath :
07238     #[ DoPrePrependPath :
07239 */
07240
07248 int DoPrePrependPath(UBYTE *s)
07249 {
07250     return DoAddPath(s, 1);
07251 }
07252
07253 /*
07254     #] DoPrePrependPath :
07255     #[ DoTimeOutAfter :
07256
07257     Executes the #timeoutafter number
07258 */
07259
07260 int DoTimeOutAfter(UBYTE *s)
07261 {
07262     #ifdef WITH_ALARM
07263         ULONG x;
07264     #else

```

```

07265     DUMMYUSE(s);
07266 #endif
07267     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07268     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07269 #ifndef WITH_ALARM
07270     while ( *s == ' ' || *s == '\t' ) s++;
07271     x = 0;
07272     while ( FG.cTable[*s] == 1 ) {
07273         x = 10*x + (*s++-'0');
07274     }
07275     while ( *s == ' ' || *s == '\t' ) s++;
07276     if ( *s == 0 ) {
07277         alarm(x);
07278         return(0);
07279     }
07280     else {
07281         MesPrint("@proper syntax is #TimeoutAfter number");
07282         return(-1);
07283     }
07284 #else
07285     Error0("#timeoutafter not implemented on this computer/system");
07286     return(-1);
07287 #endif
07288 }
07289
07290 /*
07291     #] DoTimeOutAfter :
07292     #[ DoNamespace :
07293
07294     Syntax:
07295         #Namespace name
07296         .....
07297         #use variables
07298         .....
07299         #EndNamespace
07300     Effect:
07301         All variables/expressions defined inside the range of the
07302         namespace get name_ prepended.
07303         This holds also for $-variables, names of procedures and
07304         names of files.
07305     Namespaces can be used in a nested way. cf this_is_deep_x
07306     A leading _ takes the role of what is super:: in some other languages.
07307     Remarks:
07308         Names of preprocessor variables are excluded!
07309         Names of built in objects are excluded! (like sum_, d_ etc.)
07310 */
07311
07312 int DoNamespace(UBYTE *s)
07313 {
07314     UBYTE *s1, *s2, c;
07315     NAMESPACE *namespace;
07316     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07317     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07318     while ( *s == ' ' || *s == '\t' ) s++;
07319     if ( FG.cTable[*s] != 0 ) {
07320         MesPrint("@Illegal name in #namespace instruction: %s",s);
07321         return(-1);
07322     }
07323     s1 = s;
07324     while ( FG.cTable[*s1] <= 1 ) s1++;
07325     s2 = s1;
07326     while ( *s2 == ' ' || *s2 == ',' || *s2 == '\t' ) s2++;
07327     if ( *s2 != 0 ) {
07328         MesPrint("@A #namespace instruction can only have one name with only alphanumeric
07329         characters.");
07330         return(-1);
07331     }
07332     c = *s1; *s1 = 0;
07333
07334     /*
07335         Now we have the name and the statement is legal.
07336         We can proceed creating the namespace and its use tree.
07337     */
07338     namespace = (NAMESPACE *)Malloc1(sizeof(NAMESPACE),"namespace");
07339     namespace->name = strdup(s,"namespace_name");
07340     namespace->usenames = MakeNameTree();
07341     if ( AP.firstnamespace == 0 ) {
07342         namespace->previous = 0;
07343         namespace->next = 0;
07344         AP.firstnamespace = namespace;
07345         AP.lastnamespace = namespace;
07346     }
07347     else {
07348         AP.lastnamespace->next = namespace;
07349         namespace->next = 0;
07350         namespace->previous = AP.lastnamespace;
07351         AP.lastnamespace = namespace;
07352     }
07353 }

```

```

07351     *s1 = c;
07352     return(0);
07353 }
07354
07355 /*
07356     #] DoNamespace :
07357     #[ DoEndNamespace :
07358 */
07359
07360 int DoEndNamespace(UBYTE *s)
07361 {
07362     NAMESPACE *namespace;
07363     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07364     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07365     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
07366     if ( *s != 0 ) {
07367         MesPrint("@Illegal #endnamespace instruction");
07368         return(-1);
07369     }
07370     namespace = AP.lastnamespace;
07371     AP.lastnamespace = namespace->previous;
07372     M_free(namespace->name, "namespace_name");
07373     FreeNameTree(namespace->usenames);
07374     M_free(namespace, "namespace");
07375     return(0);
07376 }
07377
07378 /*
07379     #] DoEndNamespace :
07380     #[ SkipName :
07381 */
07382
07383 UBYTE *SkipName(UBYTE *s)
07384 {
07385     UBYTE *t = s, *s1, c;
07386     int num = 0, block = 0;
07387     if ( *s == '[' ) {
07388         straight:
07389         SKIPBRAL(s)
07390         if ( *s == 0 ) {
07391             MesPrint("&Illegal name: '%s'",t);
07392             return(0);
07393         }
07394         s++; s1 = s;
07395         while ( FG.cTable[*s] <= 1 || *s == '_' ) s++;
07396         if ( s1 != s ) goto witherror;
07397     }
07398     else if ( *s == '$' ) {
07399         s++;
07400         while ( *s == '_' ) { s++; num++; }
07401         block = 1;
07402         while ( FG.cTable[*s] <= 1 || *s == '_' ) {
07403             if ( FG.cTable[*s] != 0 && block == 1 ) {
07404                 blocked:
07405                 MesPrint("&Illegally formed name: %s",t);
07406                 return(0);
07407             }
07408             if ( *s == '_' ) { num++; block = 1; }
07409             else block = 0;
07410             s++;
07411         }
07412         if ( s[-1] == '_' && num > 1 ) goto built;
07413     }
07414     else if ( FG.cTable[*s] == 0 ) {
07415         regular:
07416         while ( FG.cTable[*s] <= 1 || *s == '_' ) {
07417             if ( FG.cTable[*s] != 0 && block == 1 ) goto blocked;
07418             if ( *s == '_' ) { block = 1; num++; }
07419             else block = 0;
07420             s++;
07421         }
07422         if ( *s == '[' ) goto straight;
07423         if ( s[-1] == '_' && num > 1 ) {
07424             built:
07425             c = *s; *s = 0;
07426             MesPrint("&Built in objects cannot be part of namespaces: %s",t);
07427             *s = c;
07428             return(0);
07429         }
07430     }
07431     else if ( *s == '_' ) {
07432         while ( *s == '_' ) { s++; num++; }
07433         if ( FG.cTable[*s] == 0 ) { block = 0; goto regular; }
07434         goto witherror;
07435     }
07436     else if ( *s == '@' ) {
07437         s++; block = 1;

```

```

07438         if ( *s == '_' || FG.cTable[*s] == 1 ) {
07439             MesPrint("@Illegally formed name: %s",s-1);
07440         }
07441         goto regular;
07442     }
07443     else if ( *s == '#' ) { /* name of a procedure */
07444         s++;
07445         while ( *s == '_' ) { s++; num++; }
07446         block = 1;
07447         while ( FG.cTable[*s] <= 1 || *s == '_' ) {
07448             if ( FG.cTable[*s] != 0 && block == 1 ) goto blocked;
07449             if ( *s == '_' ) { num++; block = 1; }
07450             else block = 0;
07451             s++;
07452         }
07453         if ( s[-1] == '_' ) {
07454 witherror:
07455             c = *s; *s = 0;
07456             MesPrint("&Illegally formed name: %s",t);
07457             *s = c;
07458             return(0);
07459         }
07460     }
07461     else if ( *s == '<' ) { /* name of a file. Can be anything (more or less) */
07462         s++;
07463         while ( *s && *s != '>' ) s++;
07464         if ( *s != '>' ) goto witherror;
07465         s++;
07466     }
07467     return(s);
07468 }
07469
07470 /*
07471     #] SkipName :
07472     #[ ConstructName :
07473
07474     Routine gets a 'raw' name and modifies it if the namespace
07475     settings ask for it. It puts the new name in a buffer that
07476     may be expanded if the names become rather long.
07477     Note that eventually that name needs to be copied, because
07478     we do not allocate new buffers for each name.
07479
07480     type tells what kind of name we look for
07481 */
07482
07483 UBYTE *ConstructName(UBYTE *s,UBYTE type)
07484 {
07485     int len;
07486     UBYTE *t, *u;
07487     WORD number;
07488     NAMESPACE *namespace;
07489     if ( AP.lastnamespace == 0 ) return(s);
07490     if ( *s == '@' ) return(s+1);
07491     if ( GetName(AP.lastnamespace->usenames,s,&number,NOAUTO) !=
07492         NAMENOTFOUND ) return(s);
07493 /*
07494     Now the real stuff
07495     First we have to compute the size of the new name.
07496 */
07497     len = StrLen(s) + 1;
07498     namespace = AP.firstnamespace;
07499     while ( namespace ) {
07500         len += StrLen(namespace->name)+1;
07501         namespace = namespace->previous;
07502     }
07503     if ( len > AP.fullnamesize ) {
07504         while ( len > AP.fullnamesize ) AP.fullnamesize *= 2;
07505         M_free(AP.fullname,"AP.fullname");
07506         AP.fullname = (UBYTE *)Malloc1(AP.fullnamesize*sizeof(UBYTE *),"AP.fullname");
07507     }
07508     namespace = AP.firstnamespace;
07509     t = AP.fullname;
07510     switch ( type ) {
07511         case ' ':
07512             case 0:
07513                 while ( namespace ) {
07514                     u = namespace->name;
07515                     while ( *u ) *t++ = *u++;
07516                     *t++ = '_';
07517                     namespace = namespace->previous;
07518                 }
07519                 while ( *s ) *t++ = *s++;
07520                 *t = 0;
07521                 break;
07522         case '$':
07523         case '#':
07524             *t++ = type;

```

```

07525         while ( namespace ) {
07526             u = namespace->name;
07527             while ( *u ) *t++ = *u++;
07528             *t++ = '_';
07529             namespace = namespace->previous;
07530         }
07531         if ( type == '$' ) s++;
07532         while ( *s ) *t++ = *s++;
07533         *t = 0;
07534         break;
07535     case '<':
07536         while ( namespace ) {
07537             u = namespace->name;
07538             while ( *u ) *t++ = *u++;
07539             *t++ = '_';
07540             namespace = namespace->previous;
07541         }
07542         s++;
07543         while ( *s ) *t++ = *s++;
07544         t--; /* strip the '>' */
07545         *t = 0;
07546         break;
07547     default:
07548         MesPrint("&Unrecognized datatype in ConstructName");
07549         *t = 0;
07550         break;
07551 }
07552 return(AP.fullname);
07553 }
07554
07555 /*
07556     #] ConstructName :
07557     #[ DoUse :
07558
07559     Routine makes (inside the confines of the current namespace)
07560     a list of variables that are excluded from the namespace.
07561     Once the namespace is ended, the list is removed.
07562     The list can include names of variables, dollars and procedures.
07563     Preprocessor variables are excluded from the namespace. Their
07564     inclusion would be too complicated for the input streams.
07565     Note that the preprocessor variables that are arguments in a #do
07566     or in a procedure are on a stack and should not cause problems.
07567     Names of variables can be just that.
07568     Names of $-variables are also straightforward.
07569     Names of procedures should be preceded by a # character.
07570     Names of files (like in #include) should be enclosed by <>.
07571     The names are stored in a balanced tree. Each namespace may have
07572     its own tree. The toplevel (no namespace) does not allow a #use.
07573 */
07574
07575 int DoUse(UBYTE *s)
07576 {
07577     NAMESPACE *namespace;
07578     UBYTE *t, c;
07579     int number;
07580     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07581     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07582     if ( AP.lastnamespace == 0 ) {
07583         MesPrint("@It is not allowed to use #use outside the scope of a namespace.");
07584         return(-1);
07585     }
07586     namespace = AP.lastnamespace;
07587     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
07588     while ( *s ) {
07589         t = s;
07590         if ( ( s = SkipName(t) ) == 0 ) return(-1);
07591         if ( s == t ) {
07592             MesPrint("@Unrecognized object in #use instruction: %s",t);
07593             return(-1);
07594         }
07595         c = *s; *s = 0;
07596     }
07597     /*
07598     In usenames we only need the names to know whether they are 'protected'.
07599     We need to keep the $, # and <> to avoid potential double names.
07600 */
07601     AddName(namespace->usenames,t,0,0,&number);
07602     *s = c;
07603     while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
07604 }
07605 return(0);
07606 }
07607
07608 #] DoUse :
07609 #[ UserFlags :
07610
07611 Syntax:

```

```

07612         #ClearFlag number(s),expression(s)
07613         #ClearFlag number(s)
07614         #ClearFlag expression(s)
07615         #ClearFlag
07616         #SetFlag number(s),expression(s)
07617         #SetFlag number(s)
07618         #SetFlag expression(s)
07619         #SetFlag
07620         par == 0: Clear, par == 1: Set.
07621 */
07622
07623 int UserFlags(UBYTE *s,int par)
07624 {
07625     int mask = 0, error = 0, i;
07626     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07627     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07628     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
07629     if ( *s == 0 ) { /* Treat all flags in all active expressions */
07630         allexpr:
07631         for ( i = 0; i < NumExpressions; i++ ) {
07632             switch ( Expressions[i].status ) {
07633                 case UNHIDELEXPRESSION:
07634                 case UNHIDEGEXPRESSION:
07635                 case INTOHIDELEXPRESSION:
07636                 case INTOHIDEGEXPRESSION:
07637                 case LOCALEXPRESSION:
07638                 case GLOBALEXPRESSION:
07639                 case SKIPLEXPRESSION:
07640                 case SKIPGEXPRESSION:
07641                 case HIDELEXPRESSION:
07642                 case HIDEGEXPRESSION:
07643                     if ( par == 1 ) Expressions[i].uflags |= ~mask;
07644                     else Expressions[i].uflags &= mask;
07645                     break;
07646                 case DROPEDEXPRESSION:
07647                 case DROPLEXPRESSION:
07648                 case DROPGEXPRESSION:
07649                 case DROPHLEXPRESSION:
07650                 case DROPHGEXPRESSION:
07651                 case STOREDEXPRESSION:
07652                 case HIDDENLEXPRESSION:
07653                 case HIDDENGEXPRESSION:
07654                 case SPECTATOREXPRESSION:
07655                 default:
07656                     break;
07657             }
07658         }
07659     }
07660     else if ( FG.cTable[*s] == 1 ) {
07661         mask = (int)WORDMASK;
07662         while ( FG.cTable[*s] == 1 ) {
07663             int x = 0;
07664             while ( FG.cTable[*s] == 1 ) { x = 10*x + (*s++-'0'); }
07665             if ( x < 1 || x > BITSINWORD ) {
07666                 MesPrint("@Illegal number %d for flag in #...Flag instruction",x);
07667                 return(1);
07668             }
07669             mask ^= (1<<(x-1));
07670             while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
07671         }
07672         if ( *s == 0 ) goto allexpr;
07673     }
07674     else { /* Clear all flags in all expressions that are specified */
07675         mask = (int)WORDMASK;
07676     }
07677     while ( *s ) { /* now read the expressions */
07678         UBYTE *s1, c;
07679         WORD num1;
07680         if ( FG.cTable[*s] != 0 && *s != '[' ) goto syntax;
07681         s1 = s; s = SkipAName(s);
07682         c = *s; *s = 0;
07683         if ( GetName(AC.exprnames,s1,&num1,NOAUTO) != CEXPRESSION ) {
07684             MesPrint("@%s is not an active expression",s1);
07685             error = 1;
07686             return(error);
07687         }
07688         switch ( Expressions[num1].status ) {
07689             case UNHIDELEXPRESSION:
07690             case UNHIDEGEXPRESSION:
07691             case INTOHIDELEXPRESSION:
07692             case INTOHIDEGEXPRESSION:
07693             case LOCALEXPRESSION:
07694             case GLOBALEXPRESSION:
07695             case SKIPLEXPRESSION:
07696             case SKIPGEXPRESSION:
07697             case HIDELEXPRESSION:
07698             case HIDEGEXPRESSION:

```



```

07699         if ( par == 1 ) Expressions[num1].uflags |= ~mask;
07700         else             Expressions[num1].uflags &= mask;
07701         break;
07702     case DROPEDEXPRESSION:
07703     case DROPEXPRESSION:
07704     case DROPGEXPRESSION:
07705     case DROPHLEXPRESSION:
07706     case DROPHGEXPRESSION:
07707     case STOREDEXPRESSION:
07708     case HIDDENLEXPRESSION:
07709     case HIDDENGEXPRESSION:
07710     case SPECTATOREXPRESSION:
07711     default:
07712         MesPrint("@%s is not an active expression",s1);
07713         error = 1;
07714         break;
07715     }
07716     *s = c;
07717     while ( *s == ',' || *s == ' ' || *s == '\\t' ) s++;
07718 }
07719 return(error);
07720 syntax:
07721 MesPrint("@Illegal name in #...Flag instruction.");
07722 return(1);
07723 }
07724
07725 /*
07726     #] UserFlags :
07727     #[ DoClearUserFlag :
07728 */
07729
07730 int DoClearUserFlag(UBYTE *s)
07731 {
07732     return(UserFlags(s,0));
07733 }
07734
07735 /*
07736     #] DoClearUserFlag :
07737     #[ DoSetUserFlag :
07738 */
07739
07740 int DoSetUserFlag(UBYTE *s)
07741 {
07742     return(UserFlags(s,1));
07743 }
07744
07745 /*
07746     #] DoSetUserFlag :
07747     #[ DoStartFloat :
07748
07749     If there is a number following, it will be the new default precision.
07750     If float has been started before, the old one will be removed first.
07751     If there are two numbers, the second one is the maximum weight for
07752     MZV's.
07753 */
07754 #ifdef WITHFLOAT
07755
07756 int DoStartFloat(UBYTE *s)
07757 {
07758     GETIDENTITY
07759     int error = 0;
07760     LONG x;
07761     UBYTE *ss;
07762     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07763     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07764     if ( AR.PolyFun != 0 ) {
07765         MesPrint("@Simultaneous use of Poly(Rat)Fun and float_ is not allowed.");
07766         error = 1;
07767     }
07768     if ( AC.ncmod != 0 ) {
07769         MesPrint("@Simultaneous use of floating point and modulus arithmetic makes no sense.");
07770         error = 1;
07771     }
07772     if ( AT.aux_ ) { // First, we clean up any previous floating point system.
07773         ClearfFloat();
07774         ClearMZVTables();
07775     }
07776     while ( *s == ',' || *s == ' ' || *s == '\\t' ) s++;
07777 /*
07778     The first parameter is the float precision
07779 */
07780     ss = s;
07781     if ( *s >= '0' && *s <= '9' ) {
07782         x = 0;
07783         do {
07784             x = 10*x + (*s++-'0');
07785         } while ( *s >= '0' && *s <= '9' );

```

```

07786 /*
07787     The precision can either be in digits or bits.
07788     AC.DefaultPrecision is always in bits.
07789 */
07790     if ( tolower(*s) == 'd' ) { AC.tDefaultPrecision = (LONG)ceil(x*log2(10.0)); s++; }
07791     else if ( tolower(*s) == 'b' ) { AC.tDefaultPrecision = x; s++; }
07792     else goto IllPar;
07793     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
07794 /*
07795     The second parameter is either absent, which implies zero MZV weight,
07796     or of the form MZV = <weight>
07797 */
07798     if ( tolower(*s) == 'm' && tolower(s[1]) == 'z' && tolower(s[2]) == 'v' ) {
07799         s+=3;
07800         while ( *s == ' ' || *s == '\t' ) s++;
07801         if ( *s != '=' ) goto IllPar;
07802         s++;
07803         while ( *s == ' ' || *s == '\t' ) s++;
07804         if ( *s >= '0' && *s <= '9' ) {
07805             x = 0;
07806             do {
07807                 x = 10*x + (*s++ - '0');
07808             } while ( *s >= '0' && *s <= '9' );
07809             AC.tMaxWeight = x;
07810             while ( *s == ' ' || *s == '\t' ) s++;
07811         }
07812         else goto IllPar;
07813     }
07814     else {
07815         AC.tMaxWeight = 0;
07816     }
07817     if ( *s ) goto IllPar;
07818 }
07819 else if ( *s != 0 ) {
07820 IllPar:
07821     MesPrint("@Illegal parameter in %#StartFloat: %s ",ss);
07822     error = 1;
07823 }
07824 if ( error == 0 ) {
07825     if ( AC.tDefaultPrecision && ( AC.tDefaultPrecision != AC.DefaultPrecision
07826     || AT.aux_ == 0 ) ) {
07827         AC.DefaultPrecision = AC.tDefaultPrecision;
07828         AC.tDefaultPrecision = 0;
07829     }
07830     if ( AC.tMaxWeight && ( AC.tMaxWeight != AC.MaxWeight
07831     || AT.aux_ == 0 ) ) {
07832         AC.MaxWeight = AC.tMaxWeight;
07833         AC.tMaxWeight = 0;
07834     }
07835     SetFloatPrecision(AC.DefaultPrecision+AC.MaxWeight+1);
07836     SetupMPFTables();
07837     if ( AC.MaxWeight > 0 ) SetupMZVTables();
07838     SetfFloatPrecision(AC.DefaultPrecision);
07839 }
07840 else {
07841     AC.tDefaultPrecision = 0;
07842     AC.tMaxWeight = 0;
07843 }
07844 return(error);
07845 }
07846
07847 #endif
07848 /*
07849     #] DoStartFloat :
07850     #[ DoEndFloat :
07851 */
07852 #ifdef WITHFLOAT
07853
07854 int DoEndFloat(UBYTE *s)
07855 {
07856     int error = 0;
07857     if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
07858     if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
07859     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
07860     if ( *s != 0 ) {
07861         MesPrint("@Illegal parameter in %#EndFloat instruction: %s ",s);
07862         error = 1;
07863     }
07864     if ( error == 0 ) {
07865         ClearfFloat();
07866         ClearMZVTables();
07867     }
07868     return(error);
07869 }
07870
07871 #endif
07872 /*

```

```

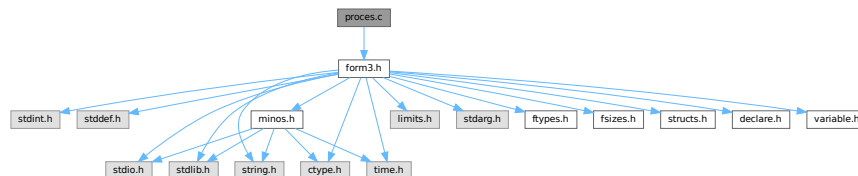
07873         #] DoEndFloat :
07874     # ] PreProcessor :
07875 */

```

4.115 proces.c File Reference

```
#include "form3.h"
```

Include dependency graph for proces.c:



Macros

- `#define DONE(x) { retvalue = x; goto Done; }`

Functions

- `int Processor (void)`
- `WORD TestSub (PHEAD WORD *term, WORD level)`
- `int InFunction (PHEAD WORD *term, WORD *termout)`
- `int InsertTerm (PHEAD WORD *term, WORD replac, WORD extractbuff, WORD *position, WORD *termout, WORD tepos)`
- `LONG PasteFile (PHEAD WORD number, WORD *accum, POSITION *position, WORD **accfill, RENUMBER renumber, WORD *freeze, WORD nexpr)`
- `WORD * PasteTerm (PHEAD WORD number, WORD *accum, WORD *position, WORD times, WORD divby)`
- `int FiniTerm (PHEAD WORD *term, WORD *accum, WORD *termout, WORD number, WORD tepos)`
- `int Generator (PHEAD WORD *term, WORD level)`
- `int DoOnePow (PHEAD WORD *term, WORD power, WORD nexpr, WORD *accum, WORD *aa, WORD level, WORD *freeze)`
- `int Deferred (PHEAD WORD *term, WORD level)`
- `int PrepPoly (PHEAD WORD *term, WORD par)`
- `int PolyFunMul (PHEAD WORD *term)`

Variables

- `WORD printscratch [2]`

4.115.1 Detailed Description

Contains the central terms processor routines. This is the core of the virtual machine. All other files are to help these routines.

Definition in file [proces.c](#).

4.115.2 Macro Definition Documentation

4.115.2.1 DONE

```
#define DONE(
    x ) { retvalue = x; goto Done; }
```

TestSub hunts for subexpression pointers. If one is found its power is given in AN.TeSuOut. and the returnvalue is 'expressionnumber'. If the expression number is negative it is an expression on disk.

In addition this routine tries to locate subexpression pointers in functions. It also notices that action must be taken with any of the special functions.

Parameters

<i>term</i>	The term in which TestSub hunts for potential action
<i>level</i>	The number of the 'level' in the compiler buffer.

Returns

The number of the (sub)expression that was encountered.

Other values that are returned are in AN.TeSuOut, AR.TePos, AT.TMbuff, AN.TelFun, AN.Frozen, AT.TMaddr

The level in the compiler buffer is more or less the number of the statement in the module. Hence it refers to the element in the lhs array.

This routine is one of the most important routines in FORM.

Definition at line [685](#) of file [proces.c](#).

4.115.3 Function Documentation

4.115.3.1 Processor()

```
int Processor (
    void )
```

This is the central processor. It accepts a stream of Expressions which is accessed by calls to GetTerm. The expressions reside either in AR.infile or AR.hidefile The definitions of an expression are seen as an id-statement, so the primary Expressions should be written to the system of scratch files as single terms with an expression pointer. Each expression is terminated with a zero and the whole is terminated by two zeroes.

The routine DoExecute should determine whether results are to be printed, should revert the scratch I/O directions etc. In principle it is DoExecute that calls Processor.

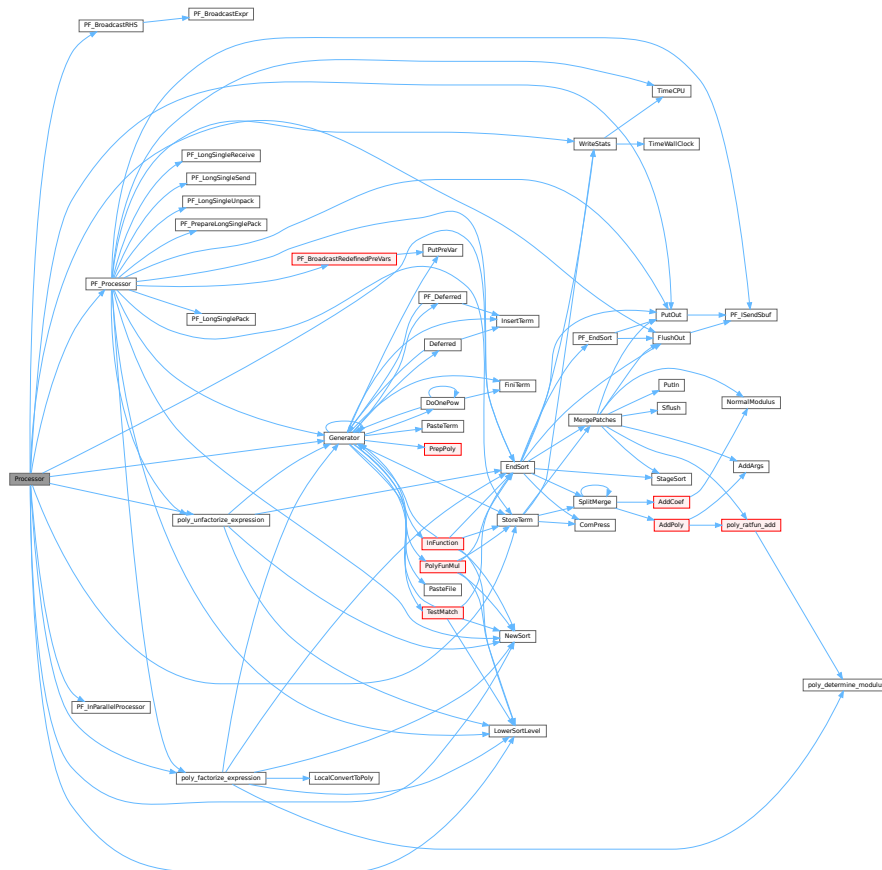
Returns

if everything OK: 0. Otherwise error. The preprocessor may continue with compilation though. Really fatal errors should return on the spot by calling `Terminate`.

Definition at line 64 of file [proces.c](#).

References [CbUf::Buffer](#), [EndSort\(\)](#), [FlushOut\(\)](#), [Generator\(\)](#), [CbUf::lhs](#), [LowerSortLevel\(\)](#), [NewSort\(\)](#), [PF_BroadcastRHS\(\)](#), [PF_InParallelProcessor\(\)](#), [PF_Processor\(\)](#), [CbUf::Pointer](#), [poly_factorize_expression\(\)](#), [poly_unfactorize_expression\(\)](#), [PutOut\(\)](#), [CbUf::rhs](#), and [StoreTerm\(\)](#).

Here is the call graph for this function:



4.115.3.2 TestSub()

```
WORD TestSub (
    PHEAD WORD * term,
    WORD level )
```

Definition at line 687 of file [proces.c](#).

4.115.3.3 InFunction()

```
int InFunction (
    PHEAD WORD * term,
    WORD * termout )
```

Makes the replacement of the subexpression with the number 'replac' in a function argument. Additional information is passed in some of the AR, AN, AT variables.

Parameters

<i>term</i>	The input term
<i>termout</i>	The output term

Returns

0: everything is fine, Negative: fatal, Positive: error.

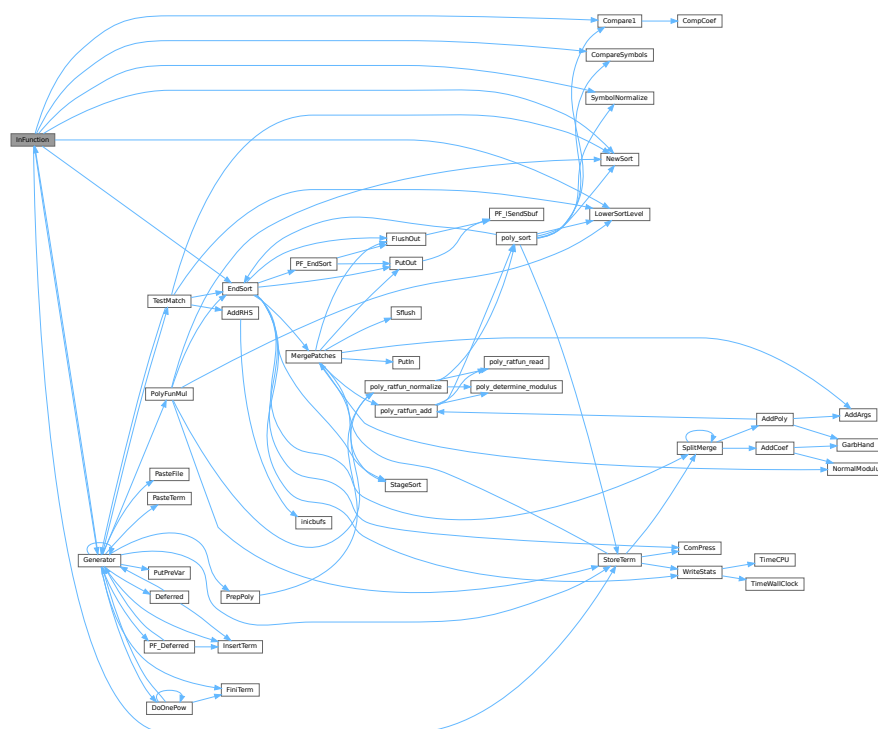
Special attention should be given to nested functions!

Definition at line 2160 of file [proces.c](#).

References [Compare1\(\)](#), [CompareSymbols\(\)](#), [EndSort\(\)](#), [Generator\(\)](#), [LowerSortLevel\(\)](#), [NewSort\(\)](#), [StoreTerm\(\)](#), and [SymbolNormalize\(\)](#).

Referenced by [Generator\(\)](#).

Here is the call graph for this function:



4.115.3.4 InsertTerm()

```

int InsertTerm (
    PHEAD WORD * term,
    WORD replac,
    WORD extractbuff,
    WORD * position,

```

```
WORD * termout,  
WORD tepos )
```

Puts the terms 'term' and 'position' together into a single legal term in termout. replac is the number of the subexpression that should be replaced. It must be a positive term. When action is needed in the argument of a function all terms in that argument are dealt with recursively. The subexpression is sorted. Only one subexpression is done at a time this way.

Parameters

<i>term</i>	the input term
<i>replac</i>	number of the subexpression pointer to replace
<i>extractbuff</i>	number of the compiler buffer <i>replac</i> refers to
<i>position</i>	position from where to take the term in the compiler buffer
<i>termout</i>	the output term
<i>tepos</i>	offset in term where the subexpression is.

Returns

Normal conventions (OK = 0).

Definition at line 2730 of file [proces.c](#).

Referenced by [Deferred\(\)](#), [Generator\(\)](#), and [PF_Deferred\(\)](#).

4.115.3.5 PasteFile()

```

LONG PasteFile (
    PHEAD WORD number,
    WORD * accum,
    POSITION * position,
    WORD ** accfill,
    RENUMBER renumber,
    WORD * freeze,
    WORD nexpr )

```

Gets a term from stored expression *expr* and puts it in the accumulator at position *number*. It returns the length of the term that came from file.

Parameters

<i>number</i>	number of partial terms to skip in <i>accum</i>
<i>accum</i>	the accumulator
<i>position</i>	file position from where to get the stored term
<i>accfill</i>	returns tail position in <i>accum</i>
<i>renumber</i>	the renumber struct for the variables in the stored expression
<i>freeze</i>	information about if we need only the contents of a bracket
<i>nexpr</i>	the number of the stored expression

Returns

Normal conventions (OK = 0).

Definition at line 2866 of file [proces.c](#).

Referenced by [Generator\(\)](#).

4.115.3.6 PasteTerm()

```
WORD * PasteTerm (
    PHEAD WORD number,
    WORD * accum,
    WORD * position,
    WORD times,
    WORD divby )
```

Puts the term at position in the accumulator *accum* at position '*number*+1'. if *times* > 0 the coefficient of this term is multiplied by *times*/*divby*.

Parameters

<i>number</i>	The number of term fragments in <i>accum</i> that should be skipped
<i>accum</i>	The accumulator of term fragments
<i>position</i>	A position in (typically) a compiler buffer from where a (piece of a) term comes.
<i>times</i>	Multiply the result by this
<i>divby</i>	Divide the result by this.

This routine is typically used when we have to replace a (sub)expression pointer by a power of a (sub)expression. This uses mostly a binomial expansion and the new term is the old term multiplied one by one by terms of the new expression. The factors *times* and *divby* keep track of the binomial coefficient. Once this is complete, the routine *FiniTerm* will make the contents of the accumulator into a proper term that still needs to be normalized.

Definition at line [2988](#) of file [proces.c](#).

Referenced by [Generator\(\)](#).

4.115.3.7 FiniTerm()

```
int FiniTerm (
    PHEAD WORD * term,
    WORD * accum,
    WORD * termout,
    WORD number,
    WORD tepos )
```

Concatenates the contents of the accumulator into a single legal term, which replaces the subexpression pointer

Parameters

<i>term</i>	the input term with the (sub)expression subterm
<i>accum</i>	the accumulator with the term fragments
<i>termout</i>	the location where the output should be written
<i>number</i>	the number of term fragments in the accumulator
<i>tepos</i>	the position of the subterm in term to be replaced

Definition at line [3053](#) of file [proces.c](#).

Referenced by [DoOnePow\(\)](#), and [Generator\(\)](#).

4.115.3.8 Generator()

```
int Generator (
    PHEAD WORD * term,
    WORD level )
```

The heart of the program. Here the expansion tree is set up in one giant recursion

Parameters

<i>term</i>	the input term. may be overwritten
<i>level</i>	the level in the compiler buffer (number of statement)

Returns

Normal conventions (OK = 0).

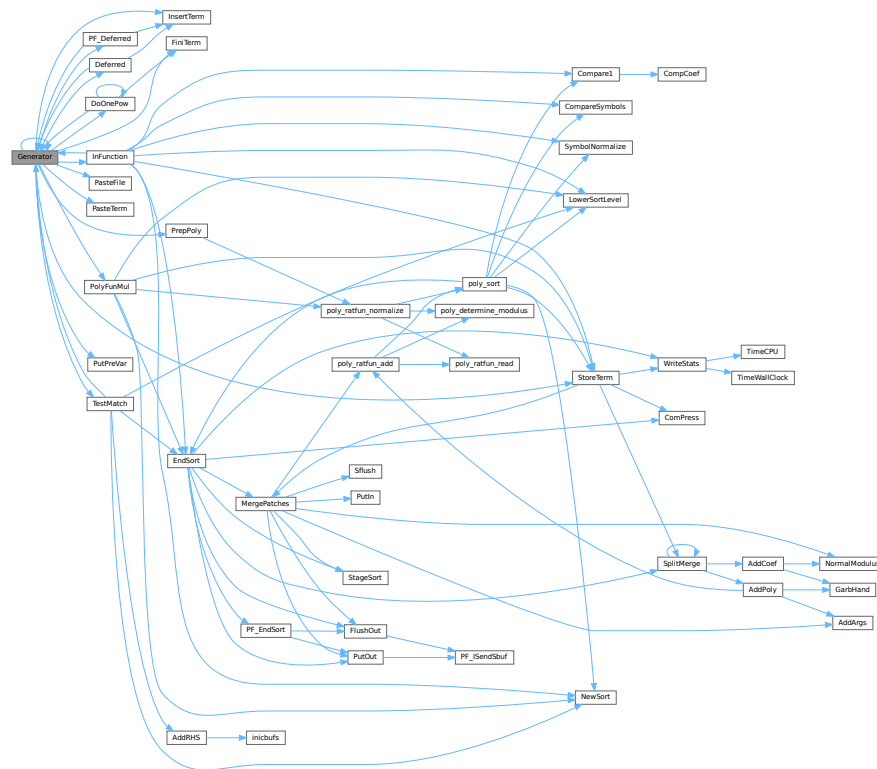
The routine looks first whether there are unsubstituted (sub)expressions. If so, one of them gets inserted term by term and the new term is used in a renewed call to Generator. If there are no (sub)expressions, the term is normalized, the compiler level is raised (next statement) and the program looks what type of statement this is. If this is a special statement it is either treated on the spot or the appropriate routine is called. If it is a substitution, the pattern matcher is called (TestMatch) which tells whether there was a match. If so we need to call TestSub again to test for (sub)expressions. If we run out of levels, the term receives a final treatment for modulus calculus and/or brackets and is then sent off to the sorting routines.

Definition at line 3252 of file [proces.c](#).

References [Deferred\(\)](#), [DoOnePow\(\)](#), [FiniTerm\(\)](#), [Generator\(\)](#), [InFunction\(\)](#), [InsertTerm\(\)](#), [CbUf::lhs](#), [VaRrEnUm::lo](#), [PasteFile\(\)](#), [PasteTerm\(\)](#), [PF_Deferred\(\)](#), [CbUf::Pointer](#), [PolyFunMul\(\)](#), [PrepPoly\(\)](#), [PutPreVar\(\)](#), [CbUf::rhs](#), [StoreTerm\(\)](#), [ReNuMbEr::symb](#), and [TestMatch\(\)](#).

Referenced by [Deferred\(\)](#), [DoOnePow\(\)](#), [generate_expression\(\)](#), [Generator\(\)](#), [InFunction\(\)](#), [MakeDollarInteger\(\)](#), [MakeDollarMod\(\)](#), [PF_CollectModifiedDollars\(\)](#), [PF_Deferred\(\)](#), [PF_Processor\(\)](#), [poly_factorize_expression\(\)](#), [poly_unfactorize_expression\(\)](#), [Processor\(\)](#), and [TestMatch\(\)](#).

Here is the call graph for this function:



4.115.3.9 DoOnePow()

```

int DoOnePow (
    PHEAD WORD * term,
    WORD power,
    WORD nex,
    WORD * accum,
    WORD * aa,
    WORD level,
    WORD * freeze )

```

Routine gets one power of an expression in the scratch system. If there are more powers needed there will be a recursion.

No attempt is made to use binomials because we have no information about commuting properties.

There is a searching for the contents of brackets if needed. This searching may be rather slow because of the single links.

Parameters

<i>term</i>	is the term we are adding to.
<i>power</i>	is the power of the expression that we need.
<i>nex</i>	is the number of the expression.
<i>accum</i>	is the accumulator of terms. It accepts the termfragments that are made into a proper term in FiniTerm

Parameters

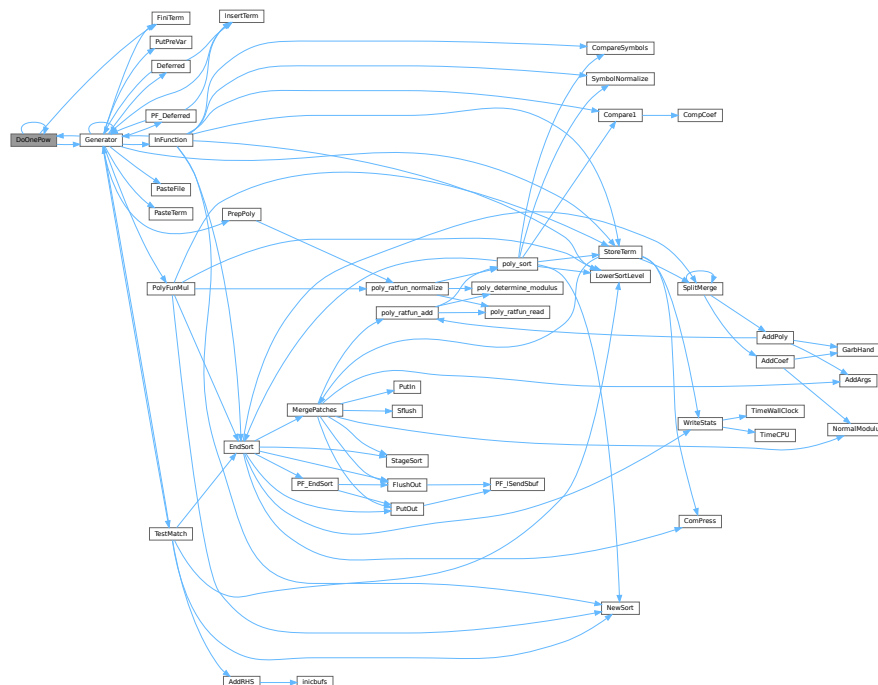
<i>aa</i>	points to the start of the entire accumulator. In *aa we store the number of term fragments that are in the accumulator.
<i>level</i>	is the current depth in the tree of statements. It is needed to continue to the next operation/substitution with each generated term
<i>freeze</i>	is the pointer to the bracket information that should be matched.

Definition at line 4619 of file [proces.c](#).

References [DoOnePow\(\)](#), [FiniTerm\(\)](#), [Generator\(\)](#), and [FiLe::handle](#).

Referenced by [DoOnePow\(\)](#), and [Generator\(\)](#).

Here is the call graph for this function:



4.115.3.10 Deferred()

```

int Deferred (
    PHEAD WORD * term,
    WORD level )

```

Picks up the deferred brackets. These are the bracket contents of which we postpone the reading when we use the 'Keep Brackets' statement. These contents are multiplying the terms just before they are sent to the sorting system. Special attention goes to having it thread-safe We have to lock positioning the file and reading it in a thread specific buffer.

Parameters

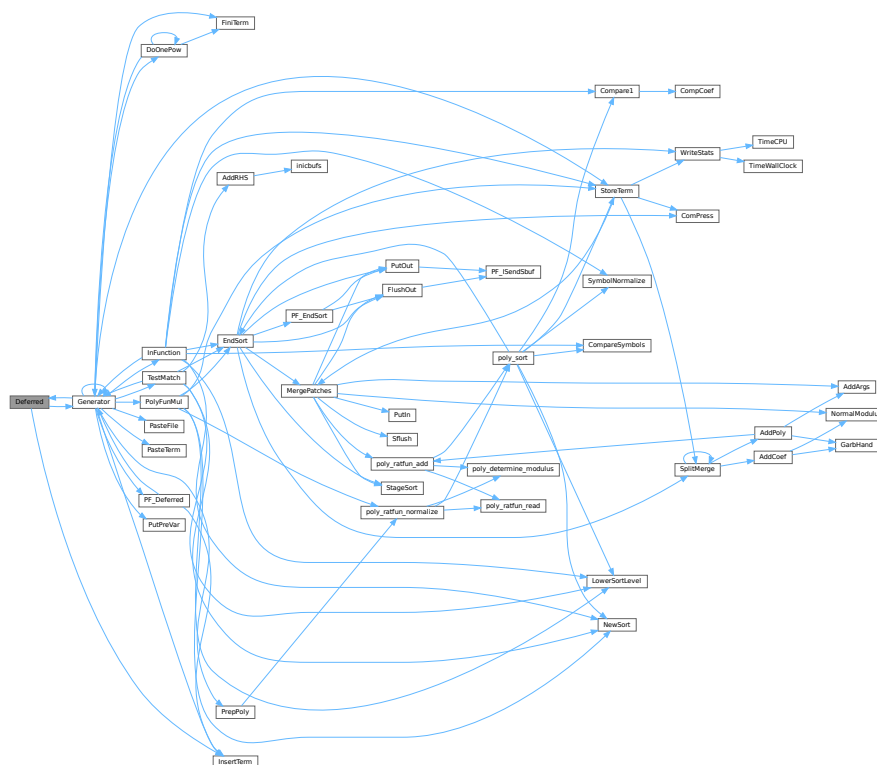
<i>term</i>	The term that must be multiplied by the contents of the current bracket
<i>level</i>	The compiler level. This is needed because after multiplying term by term we call Generator again.

Definition at line 4840 of file proces.c.

References [Generator\(\)](#), and [InsertTerm\(\)](#).

Referenced by [Generator\(\)](#).

Here is the call graph for this function:



4.115.3.11 PrepPoly()

```
int PrepPoly (
    PHEAD WORD * term,
    WORD par )
```

Routine checks whether the count of function AR.PolyFun is zero or one. If it is one and it has one scalarlike argument the coefficient of the term is pulled inside the argument. If the count is zero a new function is made with the coefficient as its only argument. The function should be placed at its proper position.

When this function is active it places the PolyFun as last object before the coefficient. This is needed because otherwise the compress algorithm has problems in MergePatches.

The bracket routine should also place the PolyFun at a comparable spot. The compression should then stop at the PolyFun. It doesn't really have to stop when writing the final result but this may be too complicated.

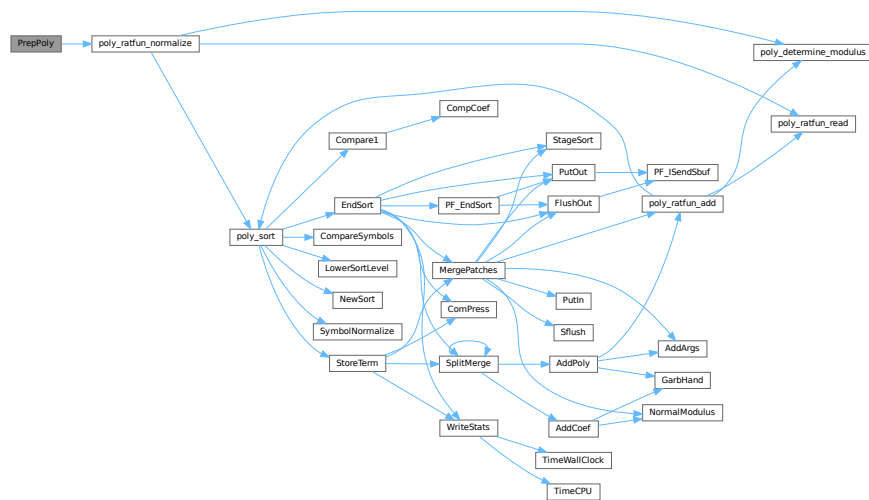
The parameter `par` tells whether we are at groundlevel or inside a function or dollar variable.

Definition at line 4968 of file `proces.c`.

References `poly_ratfun_normalize()`.

Referenced by `Generator()`.

Here is the call graph for this function:



4.115.3.12 PolyFunMul()

```
int PolyFunMul (
    PHEAD WORD * term )
```

Multiplies the arguments of multiple occurrences of the polyfun. In this routine we do the original PolyFun with one argument only. The PolyRatFun (PolyFunType = 2) is done in a dedicated routine in the file `polywrap.cc`. The new result is written over the old result.

Parameters

<code>term</code>	It contains the input term and later the output.

Returns

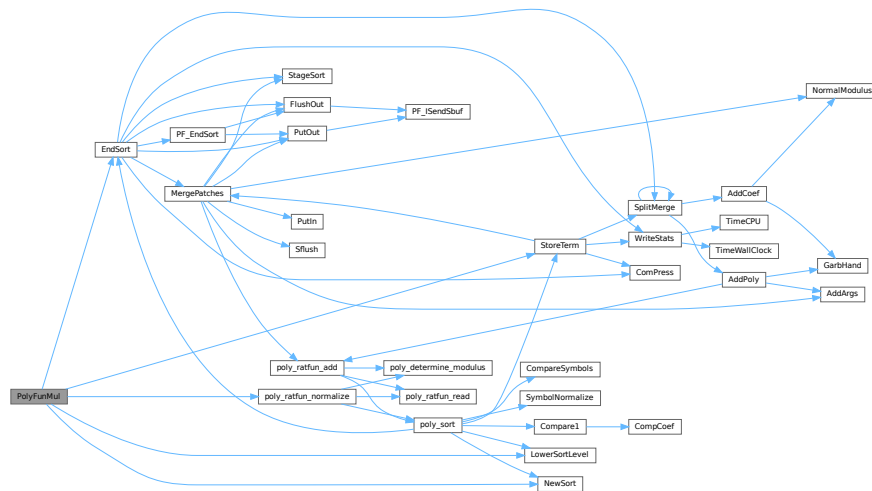
Normal conventions (OK = 0).

Definition at line 5356 of file `proces.c`.

References `EndSort()`, `LowerSortLevel()`, `NewSort()`, `poly_ratfun_normalize()`, and `StoreTerm()`.

Referenced by [Generator\(\)](#).

Here is the call graph for this function:



4.115.4 Variable Documentation

4.115.4.1 printscratch

WORD printscratch[2]

Definition at line 39 of file [proces.c](#).

4.116 proces.c

[Go to the documentation of this file.](#)

```

00001
00006 /* # [ License : */
00007 /*
00008 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00009 * When using this file you are requested to refer to the publication
00010 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011 * This is considered a matter of courtesy as the development was paid
00012 * for by FOM the Dutch physics granting agency and we would like to
00013 * be able to track its scientific use to convince FOM of its value
00014 * for the community.
00015 *
00016 * This file is part of FORM.
00017 *
00018 * FORM is free software: you can redistribute it and/or modify it under the
00019 * terms of the GNU General Public License as published by the Free Software
00020 * Foundation, either version 3 of the License, or (at your option) any later
00021 * version.
00022 *
00023 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00026 * details.
00027 *
00028 * You should have received a copy of the GNU General Public License along
00029 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00030 */
00031 /* # [ License : */
00032 /*
00033 #define HIDEDEBUG

```

```

00034     # [ Includes : proces.c
00035 */
00036
00037 #include "form3.h"
00038
00039 WORD printscratch[2];
00040
00041 /*
00042     # ] Includes :
00043     # [ Processor :
00044         # [ Processor :          WORD Processor()
00045 */
00064 int Processor(void)
00065 {
00066     GETIDENTITY
00067     WORD *term, *t, i, size;
00068     int retval = 0;
00069     EXPRESSIONS e;
00070     POSITION position;
00071     WORD last, LastExpression, fromspectator;
00072     LONG dd = 0;
00073     CBUF *C = cbuf+AC.cbunum;
00074     int firstterm;
00075     CBUF *CC = cbuf+AT.ebunum;
00076     WORD **w, *cpo, *cbo;
00077     FILEHANDLE *curfile, *oldoutfile = AR.outfile;
00078     WORD oldBracketOn = AR.BracketOn;
00079     WORD *oldBrackBuf = AT.BrackBuf;
00080     WORD oldbracketindexflag = AT.bracketindexflag;
00081 #ifdef WITHPTHREADS
00082     int OldMultiThreaded = AS.MultiThreaded, Oldmparallelflag = AC.mparallelflag;
00083 #endif
00084     if ( CC->numrhs > 0 || CC->numlhs > 0 ) {
00085         if ( CC->rhs ) {
00086             w = CC->rhs; i = CC->numrhs;
00087             do { *w++ = 0; } while ( --i > 0 );
00088         }
00089         if ( CC->lhs ) {
00090             w = CC->lhs; i = CC->numlhs;
00091             do { *w++ = 0; } while ( --i > 0 );
00092         }
00093         CC->numlhs = CC->numrhs = 0;
00094         ClearTree(AT.ebunum);
00095         CC->Pointer = CC->Buffer;
00096     }
00097
00098     if ( NumExpressions == 0 ) return(0);
00099     AR.expflags = 0;
00100     AR.CompressPointer = AR.CompressBuffer;
00101     AR.NoCompress = AC.NoCompress;
00102     term = AT.WorkPointer;
00103     if ( ( (WORD *)(((UBYTE *) (AT.WorkPointer)) + AM.MaxTer) ) > AT.WorkTop ) {
00104         MesWork();
00105     }
00106     UpdatePositions();
00107     C->rhs[C->numrhs+1] = C->Pointer;
00108     AR.KeptInHold = 0;
00109     if ( AC.CollectFun ) AR.DeferFlag = 0;
00110     AR.outtohide = 0;
00111     AN.PolyFunTodo = 0;
00112 #ifdef HIDEDEBUG
00113     MesPrint("Status at the start of Processor (HideLevel = %d)",AC.HideLevel);
00114     for ( i = 0; i < NumExpressions; i++ ) {
00115         e = Expressions+i;
00116         ExprStatus(e);
00117     }
00118 #endif
00119 /*
00120     Next determine the last expression. This is used for removing the
00121     input file when the final stage of the sort of this expression is
00122     reached. That can save up to 1/3 in disk space.
00123 */
00124     for ( i = NumExpressions-1; i >= 0; i-- ) {
00125         e = Expressions+i;
00126         if ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION
00127             || e->status == HIDELEXPRESSION || e->status == HIDEGEXPRESSION
00128             || e->status == SKIPEXPRESSION || e->status == SKIPGEXPRESSION
00129             || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION
00130             || e->status == INTOHIDELEXPRESSION || e->status == INTOHIDEGEXPRESSION
00131         ) break;
00132     }
00133     last = i;
00134     for ( i = NumExpressions-1; i >= 0; i-- ) {
00135         AS.OldOnFile[i] = Expressions[i].onfile;
00136         AS.OldNumFactors[i] = Expressions[i].numfactors;
00137 /*
00138         AS.Oldvflags[i] = e[i].vflags; */
00138         AS.Oldvflags[i] = Expressions[i].vflags;

```



```

00139     AS.Olduflags[i] = Expressions[i].uflags;
00140     Expressions[i].vflags &= ~(ISUNMODIFIED|ISZERO);
00141 }
00142 #ifdef WITHPTHREADS
00143 /*
00144     When we run with threads we have to make sure that all local input
00145     buffers are pointed correctly. Of course this isn't needed if we
00146     run on a single thread only.
00147 */
00148 if ( AC.partodoflag && AM.totalnumberofthreads > 1 ) {
00149     AS.MultiThreaded = 1; AC.mparallelflag = PARALLELFLAG;
00150 }
00151 if ( AS.MultiThreaded && AC.mparallelflag == PARALLELFLAG ) {
00152     SetWorkerFiles();
00153 }
00154 /*
00155     We start with running the expressions with expr->partodo in parallel.
00156     The current model is: give each worker an expression. Wait for
00157     workers to finish and tell them where to write.
00158     Then give them a new expression. Workers may have to wait for each
00159     other. This is also the case with the last one.
00160 */
00161 if ( AS.MultiThreaded && AC.mparallelflag == PARALLELFLAG ) {
00162     if ( InParallelProcessor() ) {
00163         retval = 1;
00164     }
00165     AS.MultiThreaded = OldMultiThreaded;
00166     AC.mparallelflag = Oldmparallelflag;
00167 }
00168 #endif
00169 #ifdef WITHMPI
00170 if ( AC.RhsExprInModuleFlag && PF.rhsInParallel && (AC.mparallelflag == PARALLELFLAG ||
AC.partodoflag) ) {
00171     if ( PF_BroadcastRHS() ) {
00172         retval = -1;
00173     }
00174 }
00175 PF.exprtodo = -1; /* This means, the slave does not perform inparallel */
00176 if ( AC.partodoflag > 0 ) {
00177     if ( PF_InParallelProcessor() ) {
00178         retval = -1;
00179     }
00180 }
00181 #endif
00182 for ( i = 0; i < NumExpressions; i++ ) {
00183     #ifdef INNERTEST
00184     if ( AC.InnerTest ) {
00185         if ( StrCmp(AC.TestValue, (UBYTE *)INNERTEST) == 0 ) {
00186             MesPrint("Testing(Processor): value = %s",AC.TestValue);
00187         }
00188     }
00189     #endif
00190     e = Expressions+i;
00191     #ifdef WITHPTHREADS
00192     if ( AC.partodoflag > 0 && e->partodo > 0 && AM.totalnumberofthreads > 2 ) {
00193         e->partodo = 0;
00194         continue;
00195     }
00196     #endif
00197     #ifdef WITHMPI
00198     if ( AC.partodoflag > 0 && e->partodo > 0 && PF.numtasks > 2 ) {
00199         e->partodo = 0;
00200         continue;
00201     }
00202     #endif
00203     AS.CollectOverFlag = 0;
00204     AR.expchanged = 0;
00205     if ( i == last ) LastExpression = 1;
00206     else LastExpression = 0;
00207     if ( e->inmem ) {
00208     /*
00209         #[ in memory : Memory allocated by poly.c only thusfar.
00210         Here GetTerm cannot work.
00211         For the moment we ignore this for parallelization.
00212     */
00213         WORD j;
00214
00215         AR.GetFile = 0;
00216         SetScratch(AR.infile,&(e->onfile));
00217         if ( GetTerm(BHEAD term) <= 0 ) {
00218             MesPrint("(1) Expression %d has problems in scratchfile",i);
00219             retval = -1;
00220             break;
00221         }
00222         term[3] = i;
00223         AR.CurExpr = i;
00224         SeekScratch(AR.outfile,&position);

```

```

00225     e->onfile = position;
00226     if ( PutOut (BHEAD term,&position,AR.outfile,0) < 0 ) goto ProcErr;
00227     AR.DeferFlag = AC.ComDefer;
00228     NewSort (BHEAD0);
00229     AN.ninterms = 0;
00230     t = e->inmem;
00231     while ( *t ) {
00232         for ( j = 0; j < *t; j++ ) term[j] = t[j];
00233         t += *t;
00234         AN.ninterms++; dd = AN.deferskipped;
00235         if ( AC.CollectFun && *term <= (AM.MaxTer/(2*(LONG)(sizeof(WORD)))) ) {
00236             if ( GetMoreFromMem(term,&t) ) {
00237                 LowerSortLevel(); goto ProcErr;
00238             }
00239         }
00240         AT.WorkPointer = term + *term;
00241         AN.RepPoint = AT.RepCount + 1;
00242         AN.IndDum = AM.IndDum;
00243         AR.CurDum = ReNumber(BHEAD term);
00244         if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
00245         if ( AN.ncmod ) {
00246             if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
00247             else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
00248         }
00249         else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
00250         if ( Generator(BHEAD term,0) ) {
00251             LowerSortLevel(); goto ProcErr;
00252         }
00253         AN.ninterms += dd;
00254     }
00255     AN.ninterms += dd;
00256     if ( EndSort(BHEAD AM.S0->sBuffer,0) < 0 ) goto ProcErr;
00257     if ( AM.S0->TermsLeft ) e->vflags &= ~ISZERO;
00258     else e->vflags |= ISZERO;
00259     if ( AR.expchanged == 0 ) e->vflags |= ISUNMODIFIED;
00260     if ( AM.S0->TermsLeft ) AR.expflags |= ISZERO;
00261     if ( AR.expchanged ) AR.expflags |= ISUNMODIFIED;
00262     AR.GetFile = 0;
00263     /*
00264     #] in memory :
00265     */
00266     }
00267     else {
00268         AR.CurExpr = i;
00269         switch ( e->status ) {
00270             case UNHIDELEXPRESSION:
00271             case UNHIDEGEXPRESSION:
00272                 AR.GetFile = 2;
00273             #ifdef WITHMPI
00274                 if ( PF.me == MASTER ) SetScratch(AR.hidefile,&(e->onfile));
00275             #else
00276                 SetScratch(AR.hidefile,&(e->onfile));
00277                 AR.InHiBuf = AR.hidefile->POfull-AR.hidefile->POfill;
00278             #ifdef HIDEDEBUG
00279                 MesPrint("Hidefile: onfile: %15p, PPosition: %15p, filesize: %15p",&(e->onfile)
00280                 ,&(AR.hidefile->POposition),&(AR.hidefile->filesize));
00281                 MesPrint("Set hidefile to buffer position %1/%1; AR.InHiBuf = %1"
00282                 , (AR.hidefile->POfill-AR.hidefile->PObuffer)*sizeof(WORD)
00283                 , (AR.hidefile->POfull-AR.hidefile->PObuffer)*sizeof(WORD)
00284                 ,AR.InHiBuf
00285                 );
00286             #endif
00287             #endif
00288                 curfile = AR.hidefile;
00289                 goto commonread;
00290             case INTOHIDELEXPRESSION:
00291             case INTOHIDEGEXPRESSION:
00292                 AR.outthide = 1;
00293             /*
00294                 BugFix 12-feb-2016
00295                 This may not work when the file is open and we move around.
00296                 AR.hidefile->POfill = AR.hidefile->POfull;
00297             */
00298                 SetEndHScratch(AR.hidefile,&position);
00299                 /* fall through */
00300             case LOCALEXPRESSION:
00301             case GLOBALEXPRESSION:
00302                 AR.GetFile = 0;
00303             /*[20oct2009 mt]:*/
00304             #ifdef WITHMPI
00305                 if( ( PF.me == MASTER ) || (PF.mkSlaveInfile) )
00306             #endif
00307                 SetScratch(AR.infile,&(e->onfile));
00308             /*:[20oct2009 mt]:*/
00309                 curfile = AR.infile;
00310             commonread:;
00311             #ifdef WITHMPI

```

```

00312         if ( PF_Processor(e,i,LastExpression) ) {
00313             MesPrint("Error in PF_Processor");
00314             goto ProcErr;
00315         }
00316         /*[20oct2009 mt]:*/
00317         if ( AC.mparallelflag != PARALLELFLAG ) {
00318             if(PF.me != MASTER)
00319                 break;
00320         #endif
00321         /*:[20oct2009 mt]:*/
00322         if ( GetTerm(BHEAD term) <= 0 ) {
00323             #ifdef HIDEDEBUG
00324                 MesPrint("Error condition 1a");
00325                 ExprStatus(e);
00326             #endif
00327             MesPrint("(2) Expression %d has problems in scratchfile(process)",i);
00328             retval = -1;
00329             break;
00330         }
00331         term[3] = i;
00332         if ( term[5] < 0 ) { /* Fill with spectator */
00333             fromspectator = -term[5];
00334             PUTZERO(AM.SpectatorFiles[fromspectator-1].readpos);
00335             term[5] = AC.cbufnum;
00336         }
00337         else fromspectator = 0;
00338         if ( AR.outtohide ) {
00339             SeekScratch(AR.hidefile,&position);
00340             e->onfile = position;
00341             if ( PutOut(BHEAD term,&position,AR.hidefile,0) < 0 ) goto ProcErr;
00342         }
00343         else {
00344             SeekScratch(AR.outfile,&position);
00345             e->onfile = position;
00346             if ( PutOut(BHEAD term,&position,AR.outfile,0) < 0 ) goto ProcErr;
00347         }
00348         AR.DeferFlag = AC.ComDefer;
00349         AR.Eside = RHSIDE;
00350         if ( ( e->vflags & ISFACTORIZED ) != 0 ) {
00351             AR.BraceOn = 1;
00352             AT.BrackBuf = AM.BracketFactors;
00353             AT.bracketindexflag = 1;
00354         }
00355         if ( AT.bracketindexflag > 0 ) OpenBracketIndex(i);
00356         #ifdef WITHPTHREADS
00357         if ( AS.MultiThreaded && AC.mparallelflag == PARALLELFLAG ) {
00358             if ( ThreadsProcessor(e,LastExpression,fromspectator) ) {
00359                 MesPrint("Error in ThreadsProcessor");
00360                 goto ProcErr;
00361             }
00362             if ( AR.outtohide ) {
00363                 AR.outfile = oldoutfile;
00364                 AR.hidefile->POfull = AR.hidefile->POfill;
00365             }
00366         }
00367         else
00368         #endif
00369         {
00370             NewSort(BHEAD0);
00371             AR.MaxDum = AM.IndDum;
00372             AN.ninterms = 0;
00373             for(;;) {
00374                 if ( fromspectator ) size = GetFromSpectator(term,fromspectator-1);
00375                 else size = GetTerm(BHEAD term);
00376                 if ( size <= 0 ) break;
00377                 SeekScratch(curfile,&position);
00378                 if ( ( e->vflags & ISFACTORIZED ) != 0 && term[1] == HAAKJE ) {
00379                     StoreTerm(BHEAD term);
00380                 }
00381                 else {
00382                     AN.ninterms++; dd = AN.deferskipped;
00383                     if ( AC.CollectFun && *term <= (AM.MaxTer/(2*(LONG)(sizeof(WORD)))) ) {
00384                         if ( GetMoreTerms(term) < 0 ) {
00385                             LowerSortLevel(); goto ProcErr;
00386                         }
00387                         SeekScratch(curfile,&position);
00388                     }
00389                     AT.WorkPointer = term + *term;
00390                     AN.RepPoint = AT.RepCount + 1;
00391                     if ( AR.DeferFlag ) {
00392                         AN.IndDum = Expressions[AR.CurExpr].numdummies + AM.IndDum;
00393                         AR.CurDum = AN.IndDum;
00394                     }
00395                     else {
00396                         AN.IndDum = AM.IndDum;
00397                         AR.CurDum = ReNumber(BHEAD term);
00398                     }
00399                 }
00400             }
00401         }

```

```

00399         if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
00400         if ( AN.ncmod ) {
00401             if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
00402             else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
00403         }
00404         else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
00405         if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
00406             && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
00407             PolyFunClean(BHEAD term);
00408         }
00409         if ( Generator(BHEAD term,0) ) {
00410             LowerSortLevel(); goto ProcErr;
00411         }
00412         AN.ninterms += dd;
00413     }
00414     SetScratch(curfile,&position);
00415     if ( AR.GetFile == 2 ) {
00416         AR.InHiBuf = (curfile->POfull-curfile->PObuffer)
00417             -DIFBASE(position,curfile->POposition)/sizeof(WORD);
00418     }
00419     else {
00420         AR.InInBuf = (curfile->POfull-curfile->PObuffer)
00421             -DIFBASE(position,curfile->POposition)/sizeof(WORD);
00422     }
00423 }
00424 AN.ninterms += dd;
00425 if ( LastExpression ) {
00426     UpdateMaxSize();
00427     if ( AR.infile->handle >= 0 ) {
00428         CloseFile(AR.infile->handle);
00429         AR.infile->handle = -1;
00430         remove(AR.infile->name);
00431         PUTZERO(AR.infile->POposition);
00432     }
00433     AR.infile->POfill = AR.infile->POfull = AR.infile->PObuffer;
00434 }
00435 if ( AR.outtohide ) AR.outfile = AR.hidefile;
00436 if ( EndSort(BHEAD AM.S0->sBuffer,0) < 0 ) goto ProcErr;
00437 if ( AR.outtohide ) {
00438     AR.outfile = oldoutfile;
00439     AR.hidefile->POfull = AR.hidefile->POfill;
00440 }
00441 e->numdummies = AR.MaxDum - AM.IndDum;
00442 UpdateMaxSize();
00443 }
00444 AR.BracketOn = oldBracketOn;
00445 AT.BrackBuf = oldBrackBuf;
00446 if ( ( e->vflags & TOBEFACTORED ) != 0 ) {
00447     poly_factorize_expression(e);
00448 }
00449 else if ( ( ( e->vflags & TOBEUNFACTORED ) != 0 )
00450     && ( ( e->vflags & ISFACTORIZED ) != 0 ) ) {
00451     poly_unfactorize_expression(e);
00452 }
00453 AT.bracketindexflag = oldbracketindexflag;
00454 if ( AM.S0->TermsLeft ) e->vflags &= ~ISZERO;
00455 else e->vflags |= ISZERO;
00456 if ( AR.expchanged == 0 ) e->vflags |= ISUNMODIFIED;
00457 if ( AM.S0->TermsLeft ) AR.expflags |= ISZERO;
00458 if ( AR.expchanged ) AR.expflags |= ISUNMODIFIED;
00459 AR.GetFile = 0;
00460 AR.outtohide = 0;
00461 /*[20oct2009 mt]:*/
00462 #ifdef WITHMPI
00463     }
00464 #endif
00465 #ifdef WITHPTHREADS
00466     if ( e->status == INTOHIDELEXPRESSION ||
00467         e->status == INTOHIDEGEXPRESSION ) {
00468         SetHideFiles();
00469     }
00470 #endif
00471     break;
00472 case SKIPLEXPRESSION:
00473 case SKIPGEXPRESSION:
00474     /*
00475         This can be greatly improved of course by file-to-file copy.
00476     */
00477     #ifdef WITHMPI
00478         if ( PF.me != MASTER ) break;
00479     #endif
00480     AR.GetFile = 0;
00481     SetScratch(AR.infile,&(e->onfile));
00482     if ( GetTerm(BHEAD term) <= 0 ) {
00483         #ifdef HIDEDEBUG
00484             MesPrint("Error condition 1b");
00485             ExprStatus(e);

```

```

00486 #endif
00487         MesPrint("(3) Expression %d has problems in scratchfile",i);
00488         retval = -1;
00489         break;
00490     }
00491     term[3] = i;
00492     AR.DeferFlag = 0;
00493     SeekScratch(AR.outfile,&position);
00494     e->onfile = position;
00495     *AM.S0->sBuffer = 0; firstterm = -1;
00496     do {
00497         WORD *oldipointer = AR.CompressPointer;
00498         WORD *comprtop = AR.ComprTop;
00499         AR.ComprTop = AM.S0->sTop;
00500         AR.CompressPointer = AM.S0->sBuffer;
00501         if ( firstterm > 0 ) {
00502             if ( PutOut(BHEAD term,&position,AR.outfile,1) < 0 ) goto ProcErr;
00503         }
00504         else if ( firstterm < 0 ) {
00505             if ( PutOut(BHEAD term,&position,AR.outfile,0) < 0 ) goto ProcErr;
00506             firstterm++;
00507         }
00508         else {
00509             if ( PutOut(BHEAD term,&position,AR.outfile,-1) < 0 ) goto ProcErr;
00510             firstterm++;
00511         }
00512         AR.CompressPointer = oldipointer;
00513         AR.ComprTop = comprtop;
00514     } while ( GetTerm(BHEAD term) );
00515     if ( FlushOut(&position,AR.outfile,1) ) goto ProcErr;
00516     UpdateMaxSize();
00517     break;
00518 case HIDELEXPRESSION:
00519 case HIDEGEXPRESSION:
00520 #ifdef WITHMPI
00521     if ( PF.me != MASTER ) break;
00522 #endif
00523     AR.GetFile = 0;
00524     SetScratch(AR.infile,&(e->onfile));
00525     if ( GetTerm(BHEAD term) <= 0 ) {
00526 #ifdef HIDEDEBUG
00527         MesPrint("Error condition 1c");
00528         ExprStatus(e);
00529 #endif
00530         MesPrint("(4) Expression %d has problems in scratchfile",i);
00531         retval = -1;
00532         break;
00533     }
00534     term[3] = i;
00535     AR.DeferFlag = 0;
00536     SetEndHScratch(AR.hidefile,&position);
00537     e->onfile = position;
00538 #ifdef HIDEDEBUG
00539     if ( AR.hidefile->handle >= 0 ) {
00540         POSITION possize,pos;
00541         PUTZERO(possiz);
00542         PUTZERO(pos);
00543         SeekFile(AR.hidefile->handle,&pos,SEEK_CUR);
00544         SeekFile(AR.hidefile->handle,&possiz,SEEK_END);
00545         SeekFile(AR.hidefile->handle,&pos,SEEK_SET);
00546         MesPrint("Processor Hidel: filesize(th) = %12p, filesize(ex) = %12p",&(position),
00547                 &(possiz));
00548         MesPrint("      in buffer:
00549 %1", (AR.hidefile->POfill-AR.hidefile->PObuffer)*sizeof(WORD));
00550     }
00551 #endif
00552     *AM.S0->sBuffer = 0; firstterm = -1;
00553     cbo = cpo = AM.S0->sBuffer;
00554     do {
00555         WORD *oldipointer = AR.CompressPointer;
00556         WORD *oldibuffer = AR.CompressBuffer;
00557         WORD *comprtop = AR.ComprTop;
00558         AR.ComprTop = AM.S0->sTop;
00559         AR.CompressPointer = cpo;
00560         AR.CompressBuffer = cbo;
00561         if ( firstterm > 0 ) {
00562             if ( PutOut(BHEAD term,&position,AR.hidefile,1) < 0 ) goto ProcErr;
00563         }
00564         else if ( firstterm < 0 ) {
00565             if ( PutOut(BHEAD term,&position,AR.hidefile,0) < 0 ) goto ProcErr;
00566             firstterm++;
00567         }
00568         else {
00569             if ( PutOut(BHEAD term,&position,AR.hidefile,-1) < 0 ) goto ProcErr;
00570             firstterm++;
00571         }
00572     }
00573     cpo = AR.CompressPointer;

```

```

00572         cbo = AR.CompressBuffer;
00573         AR.CompressPointer = oldipointer;
00574         AR.CompressBuffer = oldibuffer;
00575         AR.ComprTop = comprtop;
00576     } while ( GetTerm(BHEAD term) );
00577 #ifdef HIIDEDEBUG
00578     if ( AR.hidefile->handle >= 0 ) {
00579         POSITION posize,pos;
00580         PUTZERO(posize);
00581         PUTZERO(pos);
00582         SeekFile(AR.hidefile->handle,&pos,SEEK_CUR);
00583         SeekFile(AR.hidefile->handle,&posize,SEEK_END);
00584         SeekFile(AR.hidefile->handle,&pos,SEEK_SET);
00585         MesPrint("Processor Hide2: filesize(th) = %12p, filesize(ex) = %12p",&(position),
00586                 &(posize));
00587         MesPrint("    in buffer:
%1", (AR.hidefile->POfill-AR.hidefile->PObuffer)*sizeof(WORD));
00588     }
00589 #endif
00590     if ( FlushOut(&position,AR.hidefile,1) ) goto ProcErr;
00591     AR.hidefile->POfull = AR.hidefile->POfill;
00592 #ifdef HIIDEDEBUG
00593     if ( AR.hidefile->handle >= 0 ) {
00594         POSITION posize,pos;
00595         PUTZERO(posize);
00596         PUTZERO(pos);
00597         SeekFile(AR.hidefile->handle,&pos,SEEK_CUR);
00598         SeekFile(AR.hidefile->handle,&posize,SEEK_END);
00599         SeekFile(AR.hidefile->handle,&pos,SEEK_SET);
00600         MesPrint("Processor Hide3: filesize(th) = %12p, filesize(ex) = %12p",&(position),
00601                 &(posize));
00602         MesPrint("    in buffer:
%1", (AR.hidefile->POfill-AR.hidefile->PObuffer)*sizeof(WORD));
00603     }
00604 #endif
00605 /*
00606     Because we direct the e->onfile already to the hide file, we
00607     need to change the status of the expression. Otherwise the use
00608     of parts (or the whole) of the expression looks in the infile
00609     while the position is that of the hide file.
00610     We choose to get everything from the hide file. On average that
00611     should give least file activity.
00612 */
00613     if ( e->status == HIDELEXPRESSION ) {
00614         e->status = HIDDENLEXPRESSION;
00615         AS.OldOnFile[i] = e->onfile;
00616         AS.OldNumFactors[i] = Expressions[i].numfactors;
00617     }
00618     if ( e->status == HIDEGEXPRESSION ) {
00619         e->status = HIDDENGEXPRESSION;
00620         AS.OldOnFile[i] = e->onfile;
00621         AS.OldNumFactors[i] = Expressions[i].numfactors;
00622     }
00623 #ifdef WITHPTHREADS
00624     SetHideFiles();
00625 #endif
00626     UpdateMaxSize();
00627     break;
00628 case DROPEXPRESSION:
00629 case DROPLEXPRESSION:
00630 case DROPGEXPRESSION:
00631 case DROPHLEXPRESSION:
00632 case DROPHGEXPRESSION:
00633 case STOREDEXPRESSION:
00634 case HIDDENLEXPRESSION:
00635 case HIDDENGEXPRESSION:
00636 case SPECTATOREXPRESSION:
00637 default:
00638     break;
00639 }
00640 }
00641 AR.KeptInHold = 0;
00642 }
00643 AR.DeferFlag = 0;
00644 AT.WorkPointer = term;
00645 #ifdef HIIDEDEBUG
00646 MesPrint("Status at the end of Processor (HideLevel = %d)",AC.HideLevel);
00647 for ( i = 0; i < NumExpressions; i++ ) {
00648     e = Expressions+i;
00649     ExprStatus(e);
00650 }
00651 #endif
00652 return(retval);
00653 ProcErr:
00654 AT.WorkPointer = term;
00655 if ( AM.tracebackflag ) MesCall("Processor");
00656 return(-1);

```

```

00657 }
00658 /*
00659      #] Processor :
00660      #[ TestSub :      WORD TestSub(term,level)
00661 */
00685 #define DONE(x) { retvalue = x; goto Done; }
00686
00687 WORD TestSub(PHEAD WORD *term, WORD level)
00688 {
00689     GETBIDENTITY
00690     WORD *m, *t, *r, retvalue = 0, funflag, j, oldncmod, nexpr, *Tpattern = 0;
00691     WORD *stop, *t1, *t2, funnum, wilds, tbufnum, stilldirty = 0;
00692     NESTING n;
00693     CBUF *C = cbuf+AT.ebufnum;
00694     LONG isp, i;
00695     TABLES T;
00696     COMPARE oldcompareroutine = (COMPARE)(AR.CompareRoutine);
00697     WORD oldsorttype = AR.SortType;
00698 ReStart:
00699     tbufnum = 0; i = 0; retvalue = 0;
00700     AT.TMbuff = AM.rbufnum;
00701     funflag = 0;
00702     t = term;
00703     r = t + *t - 1;
00704     m = r - ABS(*r) + 1;
00705     t++;
00706     if ( t < m ) do {
00707         if ( *t == SUBEXPRESSION ) {
00708             /*
00709              Subexpression encountered
00710              There may be more than one.
00711              The old strategy was to take the last.
00712              A newer strategy was to take the lowest power first.
00713              The current strategy is that we compute the number of terms
00714              generated by this subexpression and take the minimum of that.
00715             */
00716
00717 #ifdef WHICHSUBEXPRESSION
00718             WORD *tmin = t, AN.nbino;
00719             /*
00720              LONG minval = MAXLONG; */
00721             LONG minval = -1;
00722             LONG mm, mnum1 = 1;
00723             if ( AN.BinoScrat == 0 ) {
00724                 AN.BinoScrat = (UWORD *)Malloc1((AM.MaxTal+2)*sizeof(UWORD),"GetBinoScrat");
00725             }
00726 #endif
00727             if ( t[3] ) {
00728                 r = t + t[1];
00729                 while ( AN.subsubveto == 0 &&
00730                     *r == SUBEXPRESSION && r < m && r[3] ) {
00731 #ifdef WHICHSUBEXPRESSION
00732                     mnum1++;
00733 #endif
00734                     if ( r[1] == t[1] && r[2] == t[2] && r[4] == t[4] ) {
00735                         j = t[1] - SUBEXPSIZE;
00736                         t1 = t + SUBEXPSIZE;
00737                         t2 = r + SUBEXPSIZE;
00738                         while ( j > 0 && *t1++ == *t2++ ) j--;
00739                         if ( j <= 0 ) {
00740                             t[3] += r[3];
00741                             if ( t[3] == 0 ) {
00742                                 t1 = r + r[1];
00743                                 t2 = term + *term;
00744                                 *term -= r[1]+t[1];
00745                                 r = t;
00746                                 while ( t1 < t2 ) *r++ = *t1++;
00747                                 goto ReStart;
00748                             }
00749                             else {
00750                                 t1 = r + r[1];
00751                                 t2 = term + *term;
00752                                 *term -= r[1];
00753                                 m -= r[1];
00754                                 while ( t1 < t2 ) *r++ = *t1++;
00755                                 r = t;
00756                             }
00757                         }
00758                     }
00759 #ifdef WHICHSUBEXPRESSION
00760                     else if ( t[2] >= 0 ) {
00761                         /*
00762                          Compute Binom(numterms+power-1,power-1)
00763                          We need potentially long arithmetic.
00764                          That is why we had to allocate AN.BinoScrat
00765                         */

```

```

00767         if ( AN.last1 == t[3] && AN.last2 == cbuf[t[4]].NumTerms[t[2]] + t[3] - 1 ) {
00768             if ( AN.last3 > minval ) {
00769                 minval = AN.last3; tmin = t;
00770             }
00771         }
00772     else {
00773         AN.last1 = t[3]; mm = AN.last2 = cbuf[t[4]].NumTerms[t[2]] + t[3] - 1;
00774         if ( t[3] == 1 ) {
00775             if ( mm > minval ) {
00776                 minval = mm; tmin = t;
00777             }
00778         }
00779     else if ( t[3] > 0 ) {
00780         if ( mm > MAXPOSITIVE ) goto TooMuch;
00781         GetBinom(AN.BinoScrat,&AN.nbino,(WORD)mm,t[3]);
00782         if ( AN.nbino > 2 ) goto TooMuch;
00783         if ( AN.nbino == 2 ) {
00784             mm = AN.BinoScrat[1];
00785             mm = ( mm << BITSINWORD ) + AN.BinoScrat[0];
00786         }
00787         else if ( AN.nbino == 1 ) mm = AN.BinoScrat[0];
00788         else mm = 0;
00789         if ( mm > minval ) {
00790             minval = mm; tmin = t;
00791         }
00792     }
00793     AN.last3 = mm;
00794 }
00795 #endif
00796
00797     t = r;
00798     r += r[1];
00799 }
00800 #ifdef WHICHSUBEXPRESSION
00801     if ( mnum1 > 1 && t[2] >= 0 ) {
00802         /*
00803             To keep the flowcontrol simple we duplicate some code here
00804         */
00805         if ( AN.last1 == t[3] && AN.last2 == cbuf[t[4]].NumTerms[t[2]] + t[3] - 1 ) {
00806             if ( AN.last3 > minval ) {
00807                 minval = AN.last3; tmin = t;
00808             }
00809         }
00810     else {
00811         AN.last1 = t[3]; mm = AN.last2 = cbuf[t[4]].NumTerms[t[2]] + t[3] - 1;
00812         if ( t[3] == 1 ) {
00813             if ( mm > minval ) {
00814                 minval = mm; tmin = t;
00815             }
00816         }
00817     else if ( t[3] > 0 ) {
00818         if ( mm > MAXPOSITIVE ) {
00819             /*
00820                 We will generate more terms than we can count
00821             */
00822             TooMuch;;
00823             MLOCK(ErrorMessageLock);
00824             MesPrint("Attempt to generate more terms than FORM can count");
00825             MUNLOCK(ErrorMessageLock);
00826             Terminate(-1);
00827         }
00828         GetBinom(AN.BinoScrat,&AN.nbino,(WORD)mm,t[3]);
00829         if ( AN.nbino > 2 ) goto TooMuch;
00830         if ( AN.nbino == 2 ) {
00831             mm = AN.BinoScrat[1];
00832             mm = ( mm << BITSINWORD ) + AN.BinoScrat[0];
00833         }
00834         else if ( AN.nbino == 1 ) mm = AN.BinoScrat[0];
00835         else mm = 0;
00836         if ( mm > minval ) {
00837             minval = mm; tmin = t;
00838         }
00839     }
00840     AN.last3 = mm;
00841 }
00842 }
00843     t = tmin;
00844 #endif
00845 /*
00846     AR.TePos = 0;
00847     AR.TePos = WORDDIFF(t,term);
00848     AT.TMbuff = t[4];
00849     if ( t[4] == AM.dbufnum && (t+t[1]) < m && t[t[1]] == DOLLAREXP2 ) {
00850         if ( t[t[1]+2] < 0 ) AT.TMdolfac = -t[t[1]+2];
00851         else { /* resolve the element number */
00852             AT.TMdolfac = GetDolNum(BHEAD t+t[1],m)+1;
00853         }
00854     }

```



```

00854         else AT.TMdolfac = 0;
00855         if ( t[3] < 0 ) {
00856             AN.TeInFun = 1;
00857             AR.TePos = WORDDIF(t,term);
00858             DONE(t[2])
00859         }
00860         else {
00861             AN.TeInFun = 0;
00862             AN.TeSuOut = t[3];
00863         }
00864         if ( t[2] < 0 ) {
00865             AN.TeSuOut = -t[3];
00866             DONE(-t[2])
00867         }
00868         DONE(t[2])
00869     }
00870 }
00871 else if ( *t == EXPRESSION ) {
00872     WORD *toTMaddr;
00873     i = -t[2] - 1;
00874     if ( t[3] < 0 ) {
00875         AN.TeInFun = 1;
00876         AR.TePos = WORDDIF(t,term);
00877         DONE(i)
00878     }
00879     nexpr = t[3];
00880     toTMaddr = m = AT.WorkPointer;
00881     AN.Frozen = 0;
00882 /*
00883     We have to be very careful with respect to setting variables
00884     like AN.TeInFun, because we may still call Generator and that
00885     may change those variables. That is why we set them at the
00886     last moment only.
00887 */
00888     j = t[1];
00889     AT.WorkPointer += j;
00890     r = t;
00891     NCOPY(m,r,j);
00892     r = t + t[1];
00893     t += SUBEXPSIZE;
00894     while ( t < r ) {
00895         if ( *t == FROMBRAC ) {
00896             WORD *ttstop,*tttstop;
00897 /*
00898             Note: Convention is that wildcards are done
00899             after the expression has been picked up. So
00900             no wildcard substitutions are needed here.
00901 */
00902             t += 2;
00903             AN.Frozen = m = AT.WorkPointer;
00904 /*
00905             We should check now for subexpressions and if necessary
00906             we substitute them. Keep in mind: only one term allowed!
00907
00908             In retrospect (26-jan-2010): take also functions that
00909             have a dirty flag on
00910 */
00911             j = *t; tttstop = t + j;
00912             GETSTOP(t,tttstop);
00913             *m++ = j; t++;
00914             while ( t < tttstop ) {
00915                 if ( *t == SUBEXPRESSION ) break;
00916                 if ( *t >= FUNCTION && ( ( t[2] & DIRTYFLAG ) == DIRTYFLAG ) ) break;
00917                 j = t[1]; NCOPY(m,t,j);
00918             }
00919             if ( t < tttstop ) {
00920 /*
00921                 We ran into a subexpression or a function with a
00922                 'dirty' argument. It could also be a $ or
00923                 just e[(a^2)*b]. In all cases we should evaluate
00924 */
00925                 while ( t < tttstop ) *m++ = *t++;
00926                 *AT.WorkPointer = m-AT.WorkPointer;
00927                 m = AT.WorkPointer;
00928                 AT.WorkPointer = m + *m;
00929                 NewSort(BHEAD0);
00930                 if ( Generator(BHEAD m,AR.Cnumlhs) ) {
00931                     LowerSortLevel(); goto EndTest;
00932                 }
00933                 if ( EndSort(BHEAD m,0) < 0 ) goto EndTest;
00934                 AN.Frozen = m;
00935                 if ( *m == 0 ) {
00936                     *m++ = 4; *m++ = 1; *m++ = 1; *m++ = 3;
00937                 }
00938                 else if ( m[*m] != 0 ) {
00939                     MLOCK(ErrorMessageLock);
00940                     MesPrint("Bracket specification in expression should be one single term");

```

```

00941             MUNLOCK(ErrorMessageLock);
00942             Terminate(-1);
00943         }
00944         else {
00945             m += *m;
00946             m -= ABS(m[-1]);
00947             *m++ = 1; *m++ = 1; *m++ = 3;
00948             *AN.Frozen = m - AN.Frozen;
00949         }
00950     }
00951     else {
00952         while ( t < tttstop ) *m++ = *t++;
00953         *AT.WorkPointer = m-AT.WorkPointer;
00954         m = AT.WorkPointer;
00955         AT.WorkPointer = m + *m;
00956         if ( Normalize(BHEAD m) ) {
00957             MLOCK(ErrorMessageLock);
00958             MesPrint("Error while picking up contents of bracket");
00959             MUNLOCK(ErrorMessageLock);
00960             Terminate(-1);
00961         }
00962         if ( !*m ) {
00963             *m++ = 4; *m++ = 1; *m++ = 1; *m++ = 3;
00964         }
00965         else m += *m;
00966     }
00967     AT.WorkPointer = m;
00968     break;
00969 }
00970 t += t[1];
00971 }
00972 AN.TeInFun = 0;
00973 AR.TePos = 0;
00974 AN.TeSuOut = nexpr;
00975 AT.TMaddr = toTMaddr;
00976 DONE(i)
00977 }
00978 else if ( *t >= FUNCTION ) {
00979     if ( t[0] == EXPONENT ) {
00980         if ( t[1] == FUNHEAD+4 && t[FUNHEAD] == -SYMBOL &&
00981             t[FUNHEAD+2] == -SNUMBER && t[FUNHEAD+3] < MAXPOWER
00982             && t[FUNHEAD+3] > -MAXPOWER ) {
00983             t[0] = SYMBOL;
00984             t[1] = 4;
00985             t[2] = t[FUNHEAD+1];
00986             t[3] = t[FUNHEAD+3];
00987             r = term + *term;
00988             m = t + FUNHEAD+4;
00989             t += 4;
00990             while ( m < r ) *t++ = *m++;
00991             *term = WORDDIFF(t,term);
00992             goto ReStart;
00993         }
00994         else if ( t[1] == FUNHEAD+ARGHEAD+11 && t[FUNHEAD] == ARGHEAD+9
00995             && t[FUNHEAD+ARGHEAD] == 9 && t[FUNHEAD+ARGHEAD+1] == DOTPRODUCT
00996             && t[FUNHEAD+ARGHEAD+8] == 3
00997             && t[FUNHEAD+ARGHEAD+7] == 1
00998             && t[FUNHEAD+ARGHEAD+6] == 1
00999             && t[FUNHEAD+ARGHEAD+5] == 1
01000             && t[FUNHEAD+ARGHEAD+9] == -SNUMBER
01001             && t[FUNHEAD+ARGHEAD+10] < MAXPOWER
01002             && t[FUNHEAD+ARGHEAD+10] > -MAXPOWER ) {
01003             t[0] = DOTPRODUCT;
01004             t[1] = 5;
01005             t[2] = t[FUNHEAD+ARGHEAD+3];
01006             t[3] = t[FUNHEAD+ARGHEAD+4];
01007             t[4] = t[FUNHEAD+ARGHEAD+10];
01008             r = term + *term;
01009             m = t + FUNHEAD+ARGHEAD+11;
01010             t += 5;
01011             while ( m < r ) *t++ = *m++;
01012             *term = WORDDIFF(t,term);
01013             goto ReStart;
01014         }
01015     }
01016     funnum = *t;
01017     if ( *t >= FUNCTION + WILDOFFSET ) funnum -= WILDOFFSET;
01018     if ( *t == EXPONENT ) {
01019         /*
01020             Test whether the second argument is an integer
01021         */
01022         r = t+FUNHEAD;
01023         NEXTARG(r)
01024         if ( *r == -SNUMBER && r[1] < MAXPOWER && r+2 == t+t[1] &&
01025             t[FUNHEAD] > -FUNCTION && ( t[FUNHEAD] != -SNUMBER
01026             || t[FUNHEAD+1] != 0 ) && t[FUNHEAD] != ARGHEAD ) {
01027             if ( r[1] == 0 ) {

```

```

01028         if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] == 0 ) {
01029             MLOCK(ErrorMessageLock);
01030             MesPrint("Encountered 0^0. Fatal error.");
01031             MUNLOCK(ErrorMessageLock);
01032             SETERROR(-1);
01033         }
01034         *t = DUMMYFUN;
01035     /*
01036         Now mark it clean to avoid further interference.
01037         Normalize will remove this object.
01038     */
01039     t[2] = 0;
01040 }
01041 else {
01042     /* Note that the case 0^ is treated in Normalize */
01043     t1 = AddRHS(AT.ebufnum,1);
01044     m = t + FUNHEAD;
01045     if ( *m > 0 ) {
01046         m += ARGHEAD;
01047         i = t[FUNHEAD] - ARGHEAD;
01048         while ( (t1 + i + 10) > C->Top )
01049             t1 = DoubleCbuffer(AT.ebufnum,t1,9);
01050         while ( --i >= 0 ) *t1++ = *m++;
01051     }
01052     else {
01053         if ( (t1 + 20) > C->Top )
01054             t1 = DoubleCbuffer(AT.ebufnum,t1,10);
01055         ToGeneral(m,t1,1);
01056         t1 += *t1;
01057     }
01058     *t1++ = 0;
01059     C->rhs[C->numrhs+1] = t1;
01060     C->Pointer = t1;
01061
01062     /* No provisions yet for commuting objects */
01063
01064     C->CanCommu[C->numrhs] = 1;
01065     *t++ = SUBEXPRESSION;
01066     *t++ = SUBEXPSIZE;
01067     *t++ = C->numrhs;
01068     *t++ = r[1];
01069     *t++ = AT.ebufnum;
01070
01071     #if SUBEXPSIZE > 5
01072     Important: we may not have enough spots here
01073     #endif
01074     FILLSUB(t) /* Important: We have maybe only 5 spots! */
01075     r += 2;
01076     m = term + *term;
01077     do { *t++ = *r++; } while ( r < m );
01078     *term -= WORDDIF(r,t);
01079     goto ReStart;
01080 }
01081 }
01082 }
01083 else if ( *t == SUMF1 || *t == SUMF2 ) {
01084     /*
01085         What we are looking for is:
01086         1-st argument: Single symbol or index.
01087         2-nd argument: Number.
01088         3-rd argument: Number.
01089         (4-th argument):Number.
01090         One more argument.
01091         This would activate the summation procedure.
01092         Note that the initiated recursion here can be done
01093         without upsetting the regular procedures.
01094     */
01095     WORD *tstop, lcounter, lccmin, lccmax, lccinc;
01096     tstop = t + t[1];
01097     r = t+FUNHEAD;
01098     if ( r+6 < tstop && r[2] == -SNUMBER && r[4] == -SNUMBER
01099         && ( r[0] == -SYMBOL )
01100         || ( r[0] == -INDEX && r[1] >= AM.OffsetIndex
01101             && r[3] >= 0 && r[3] < AM.OffsetIndex
01102             && r[5] >= 0 && r[5] < AM.OffsetIndex ) ) {
01103         lcounter = r[0] == -INDEX ? -r[1]: r[1]; /* The loop counter */
01104         lccmin = r[3];
01105         lccmax = r[5];
01106         r += 6;
01107         if ( *r == -SNUMBER && r+2 < tstop ) {
01108             lccinc = r[1];
01109             r += 2;
01110         }
01111         else lccinc = 1;
01112         if ( r < tstop && ( (*r > 0 && (r++*r) == tstop )
01113             || ( *r <= -FUNCTION && r+1 == tstop )
01114             || ( *r > -FUNCTION && *r < 0 && r+2 == tstop ) ) ) {

```

```

01115         m = AddRHS(AT.ebufnum,1);
01116         if ( *r > 0 ) {
01117             i = *r - ARGHEAD;
01118             r += ARGHEAD;
01119             while ( (m + i + 10) > C->Top )
01120                 m = DoubleCbuffer(AT.ebufnum,m,11);
01121             while ( --i >= 0 ) *m++ = *r++;
01122         }
01123         else {
01124             while ( (m + 20) > C->Top )
01125                 m = DoubleCbuffer(AT.ebufnum,m,12);
01126             ToGeneral(r,m,1);
01127             m += *m;
01128         }
01129         *m++ = 0;
01130         C->rhs[C->numrhs+1] = m;
01131         C->Pointer = m;
01132         m = AT.TMout;
01133         *m++ = 6;
01134         if ( *t == SUMF1 ) *m++ = SUMNUM1;
01135         else *m++ = SUMNUM2;
01136         *m++ = lcounter;
01137         *m++ = lcmin;
01138         *m++ = lcmax;
01139         *m++ = lcinc;
01140         m = t + t[1];
01141         r = C->rhs[C->numrhs];
01142     /*
01143
01144     Test now if the argument was already evaluated.
01145     In that case it needs a new subexpression prototype.
01146     In either case we replace the function now by a
01147     subexpression prototype.
01148     */
01149     if ( *r >= (SUBEXPSIZE+4)
01150         && ABS(*(r+r-1)) < (*r - 1)
01151         && r[1] == SUBEXPRESSION ) {
01152         r++;
01153         i = r[1] - 5;
01154         *t++ = *r++; *t++ = *r++; *t++ = C->numrhs;
01155         r++; *t++ = *r++; *t++ = AT.ebufnum; r++;
01156         while ( --i >= 0 ) *t++ = *r++;
01157     }
01158     else {
01159         *t++ = SUBEXPRESSION;
01160         *t++ = 4+SUBEXPSIZE;
01161         *t++ = C->numrhs;
01162         *t++ = 1;
01163         *t++ = AT.ebufnum;
01164         FILLSUB(t)
01165         if ( lcounter < 0 ) {
01166             *t++ = INDTOIND;
01167             *t++ = 4;
01168             *t++ = -lcounter;
01169         }
01170         else {
01171             *t++ = SYMTONUM;
01172             *t++ = 4;
01173             *t++ = lcounter;
01174         }
01175         *t++ = lcmin;
01176     }
01177     t2 = term + *term;
01178     while ( m < t2 ) *t++ = *m++;
01179     *term = WORDDIF(t,term);
01180     AN.TeInFun = -C->numrhs;
01181     AR.TePos = 0;
01182     AN.TeSuOut = 0;
01183     AT.TMbuff = AT.ebufnum;
01184     DONE(C->numrhs)
01185 }
01186 }
01187 else if ( *t == TOPOLOGIES ) {
01188     MesPrint("&The topologies_ function was removed in FORM 5.0.");
01189     MesPrint("&See the TopologiesOnly_ option of diagrams_.");
01190     Terminate(-1);
01191 }
01192 else if ( *t == DIAGRAMS ) {
01193     /*
01194     Syntax:
01195     diagrams_(model,setinparticles,setoutparticles,
01196             setextmomenta,setintmomenta,couplings or loops)
01197     */
01198     if ( AC.nummodels > 0 ) { /* No model, no diagrams */
01199         t1 = t+FUNHEAD; t2 = t+t[1];
01200         if (
01201             t1[0] == -SETSET && Sets[t1[1]].type == CMODEL &&

```

```

01202         t1[2] == -SETSET && ( Sets[t1[3]].type == CFUNCTION
01203         || ( Sets[t1[3]].type == ANYTYPE && ( Sets[t1[3]].first == Sets[t1[3]].last ) ) )
01204         &&
01204         t1[4] == -SETSET && ( Sets[t1[5]].type == CFUNCTION
01205         || ( Sets[t1[5]].type == ANYTYPE && ( Sets[t1[5]].first == Sets[t1[5]].last ) ) )
01206         &&
01206         t1[6] == -SETSET && ( Sets[t1[7]].type == CVECTOR
01207         || ( Sets[t1[7]].type == ANYTYPE && ( Sets[t1[7]].first == Sets[t1[7]].last ) ) )
01208         &&
01208         t1[8] == -SETSET && ( Sets[t1[9]].type == CVECTOR
01209         || ( Sets[t1[9]].type == ANYTYPE && ( Sets[t1[9]].first == Sets[t1[9]].last ) ) )
01210         &&
01210         t1+12 <= t2 ) {
01211     /*
01212         Test that the sets of particles correspond to particles
01213         of the set model.
01214     */
01215     MODEL *m = AC.models[SetElements[Sets[t1[1]].first]];
01216     int nn0,nn1,nn2;
01217     for ( nn0 = 3; nn0 <= 5; nn0 += 2 ) {
01218         for ( nn1 = Sets[t1[nn0]].first; nn1 < Sets[t1[nn0]].last; nn1++ ) {
01219             for ( nn2 = 0; nn2 < m->nparticles; nn2++ ) {
01220                 if ( m->vertices[nn2]->particles[0].number == SetElements[nn1]
01221                     || m->vertices[nn2]->particles[1].number == SetElements[nn1] ) break;
01222             }
01223             if ( nn2 >= m->nparticles ) goto doesnetwork;
01224         }
01225     }
01226     /*
01227         Now test for a single argument indicating the order
01228         in perturbation theory.
01229     */
01230     if ( ( t1[10] == -SNUMBER && t1[11] >= 0 && t1+12 == t2 )
01231         || ( t1[10] == -SYMBOL && t1+12 == t2 )
01232         || ( t1+10+t1[10] == t2 && t1+10+ARGHEAD+t1[10+ARGHEAD] == t2
01233             && t1[11+ARGHEAD] == SYMBOL ) ) {
01234     /*
01235         Now test that all symbols are valid coupling constants.
01236     */
01237         if ( t1+12 > t2 ) {
01238             WORD *ttl = t1+13+ARGHEAD, im;
01239             t2 -= ABS(t2[-1]);
01240             while ( ttl < t2 ) {
01241                 for ( im = 0; im < m->ncouplings; im++ ) {
01242                     if ( *ttl == m->couplings[im] ) break;
01243                 }
01244                 if ( im >= m->ncouplings ) goto doesnetwork;
01245                 ttl += 2;
01246             }
01247         }
01248         AN.TeInFun = -15;
01249         AN.TeSuOut = 0;
01250         AR.TePos = -1;
01251         AR.funoffset = t - term;
01252         DONE(1)
01253     }
01254     else if ( ( ( t1[10] == -SNUMBER && t1[11] >= 0 && t1+12 == t2-2 )
01255         || ( t1[10] == -SYMBOL && t1+12 == t2-2 )
01256         || ( t1+10+t1[10] == t2-2 && t1+10+ARGHEAD+t1[10+ARGHEAD] == t2-2
01257             && t1[11+ARGHEAD] == SYMBOL ) ) && t2[-2] == -SNUMBER ) {
01258     /*
01259         With options at t2[-2],t2[-1]
01260         Now test that all symbols are valid coupling constants.
01261     */
01262         t2 -= 2;
01263         if ( t1+12 > t2 ) {
01264             WORD *ttl = t1+13+ARGHEAD, im;
01265             t2 -= ABS(t2[-1]);
01266             while ( ttl < t2 ) {
01267                 for ( im = 0; im < m->ncouplings; im++ ) {
01268                     if ( *ttl == m->couplings[im] ) break;
01269                 }
01270                 if ( im >= m->ncouplings ) goto doesnetwork;
01271                 ttl += 2;
01272             }
01273         }
01274         AN.TeInFun = -15;
01275         AN.TeSuOut = 0;
01276         AR.TePos = -1;
01277         AR.funoffset = t - term;
01278         DONE(1)
01279     }
01280     doesnetwork;;
01281     }
01282     }
01283     }
01284     if ( functions[funnum-FUNCTION].spec <= 0

```

```

01285         || ( t[2] & (DIRTYFLAG|MUSTCLEANPRF) ) != 0 ) {
01286             funflag = 1;
01287         }
01288     if ( *t <= MAXBUILTINFUNCTION ) {
01289         if ( *t <= DELTAP && *t >= THETA ) { /* Speeds up by 2 or 3 compares */
01290             if ( *t == THETA || *t == THETA2 ) {
01291                 WORD *tstop, *tt2, kk;
01292                 tstop = t + t[1];
01293                 tt2 = t + FUNHEAD;
01294                 while ( tt2 < tstop ) {
01295                     if ( *tt2 > 0 && tt2[1] != 0 ) goto DoSpec;
01296                     NEXTARG(tt2)
01297                 }
01298                 if ( !AT.RecFlag ) {
01299                     if ( ( kk = DoTheta(BHEAD t) ) == 0 ) {
01300                         *term = 0;
01301                         DONE(0)
01302                     }
01303                     else if ( kk > 0 ) {
01304                         m = t + t[1];
01305                         r = term + *term;
01306                         while ( m < r ) *t++ = *m++;
01307                         *term = WORDDIFF(t,term);
01308                         goto ReStart;
01309                     }
01310                 }
01311             }
01312             else if ( *t == DELTA2 || *t == DELTAP ) {
01313                 WORD *tstop, *tt2, kk;
01314                 tstop = t + t[1];
01315                 tt2 = t + FUNHEAD;
01316                 while ( tt2 < tstop ) {
01317                     if ( *tt2 > 0 && tt2[1] != 0 ) goto DoSpec;
01318                     NEXTARG(tt2)
01319                 }
01320                 if ( !AT.RecFlag ) {
01321                     if ( ( kk = DoDelta(t) ) == 0 ) {
01322                         *term = 0;
01323                         DONE(0)
01324                     }
01325                     else if ( kk > 0 ) {
01326                         m = t + t[1];
01327                         r = term + *term;
01328                         while ( m < r ) *t++ = *m++;
01329                         *term = WORDDIFF(t,term);
01330                         goto ReStart;
01331                     }
01332                 }
01333             }
01334             else if ( *t == DISTRIBUTION && t[FUNHEAD] == -SNUMBER
01335                 && t[FUNHEAD+1] >= -2 && t[FUNHEAD+1] <= 2
01336                 && t[FUNHEAD+2] == -SNUMBER
01337                 && t[FUNHEAD+4] <= -FUNCTION
01338                 && t[FUNHEAD+5] <= -FUNCTION ) {
01339                 WORD *ttt = t+FUNHEAD+6, *tttstop = t+t[1];
01340                 while ( ttt < tttstop ) {
01341                     if ( *ttt == -DOLLAREXPRESSION ) break;
01342                     NEXTARG(ttt);
01343                 }
01344                 if ( ttt >= tttstop ) {
01345                     AN.TeInFun = -1;
01346                     AN.TeSuOut = 0;
01347                     AR.TePos = -1;
01348                     DONE(1)
01349                 }
01350             }
01351             else if ( *t == DELTA3 && ((t[1]-FUNHEAD) & 1) == 0 ) {
01352                 AN.TeInFun = -2;
01353                 AN.TeSuOut = 0;
01354                 AR.TePos = -1;
01355                 DONE(1)
01356             }
01357             else if ( ( *t == TABLEFUNCTION ) && ( t[FUNHEAD] <= -FUNCTION )
01358                 && ( T = functions[-t[FUNHEAD]-FUNCTION].tabl ) != 0
01359                 && ( t[1] >= FUNHEAD+1+2*ABS(T->numind) )
01360                 && ( t[FUNHEAD+1] == -SYMBOL ) ) {
01361                 /*
01362                 The case of table_(tab,sym1,...,symn)
01363                 */
01364                 for ( isp = 0; isp < ABS(T->numind); isp++ ) {
01365                     if ( t[FUNHEAD+1+2*isp] != -SYMBOL ) break;
01366                 }
01367                 if ( isp >= ABS(T->numind) ) {
01368                     AN.TeInFun = -3;
01369                     AN.TeSuOut = 0;
01370                     AR.TePos = -1;
01371                     DONE(1)

```

```

01372     }
01373 }
01374 else if ( *t == TABLEFUNCTION && t[FUNHEAD] <= -FUNCTION
01375 && ( T = functions[-t[FUNHEAD]-FUNCTION].tabl ) != 0
01376 && ( t[1] == FUNHEAD+2 )
01377 && ( t[FUNHEAD+1] <= -FUNCTION ) ) {
01378 /*
01379     The case of table_(tab,fun)
01380 */
01381     AN.TeInFun = -3;
01382     AN.TeSuOut = 0;
01383     AR.TePos = -1;
01384     DONE(1)
01385 }
01386 else if ( *t == FACTORIN ) {
01387     if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -DOLLAREXPRESSION ) {
01388         AN.TeInFun = -4;
01389         AN.TeSuOut = 0;
01390         AR.TePos = -1;
01391         DONE(1)
01392     }
01393     else if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -EXPRESSION ) {
01394         AN.TeInFun = -5;
01395         AN.TeSuOut = 0;
01396         AR.TePos = -1;
01397         DONE(1)
01398     }
01399 }
01400 else if ( *t == TERMSINBRACKET ) {
01401     if ( t[1] == FUNHEAD || (
01402         t[1] == FUNHEAD+2
01403         && t[FUNHEAD] == -SNUMBER
01404         && t[FUNHEAD+1] == 0
01405     ) ) {
01406         AN.TeInFun = -6;
01407         AN.TeSuOut = 0;
01408         AR.TePos = -1;
01409         DONE(1)
01410     }
01411 /*
01412     The other cases have not yet been implemented
01413     We still have to add the case of short arguments
01414     First the different bracket in same expression
01415
01416     else if ( t[1] > FUNHEAD+ARGHEAD
01417         && t[FUNHEAD] == t[1]-FUNHEAD
01418         && t[FUNHEAD+ARGHEAD] == t[1]-FUNHEAD-ARGHEAD
01419         && t[t[1]-1] == 3
01420         && t[t[1]-2] == 1
01421         && t[t[1]-3] == 1 ) {
01422         AN.TeInFun = -6;
01423         AN.TeSuOut = 0;
01424         AR.TePos = -1;
01425         DONE(1)
01426     }
01427
01428     Next the bracket in an other expression
01429
01430     else if ( t[1] > FUNHEAD+ARGHEAD+2
01431         && t[FUNHEAD] == -EXPRESSION
01432         && t[FUNHEAD+2] == t[1]-FUNHEAD-2
01433         && t[FUNHEAD+ARGHEAD+2] == t[1]-FUNHEAD-ARGHEAD-2
01434         && t[t[1]-1] == 3
01435         && t[t[1]-2] == 1
01436         && t[t[1]-3] == 1 ) {
01437         AN.TeInFun = -6;
01438         AN.TeSuOut = 0;
01439         AR.TePos = -1;
01440         DONE(1)
01441     }
01442 */
01443 }
01444 else if ( *t == EXTRASYMFUN ) {
01445     if ( t[1] == FUNHEAD+2 && (
01446         ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] <= cbuf[AM.sbufnum].numrhs
01447           && t[FUNHEAD+1] > 0 ) ||
01448         ( t[FUNHEAD] == -SYMBOL && t[FUNHEAD+1] < MAXVARIABLES
01449           && t[FUNHEAD+1] >= MAXVARIABLES-cbuf[AM.sbufnum].numrhs ) ) ) {
01450         AN.TeInFun = -7;
01451         AN.TeSuOut = 0;
01452         AR.TePos = -1;
01453         DONE(1)
01454     }
01455     else if ( t[1] == FUNHEAD ) {
01456         AN.TeInFun = -7;
01457         AN.TeSuOut = 0;
01458         AR.TePos = -1;

```

```

01459         DONE(1)
01460     }
01461 }
01462 else if ( *t == DIVFUNCTION || *t == REMFUNCTION
01463         || *t == INVERSEFUNCTION || *t == MULFUNCTION
01464         || *t == GCDFUNCTION ) {
01465     WORD *tf;
01466     int todo = 1, numargs = 0;
01467     tf = t + FUNHEAD;
01468     while ( tf < t + t[1] ) {
01469         DOLLARS d;
01470         if ( *tf == -DOLLAREXPRESSION ) {
01471             d = Dollars + tf[1];
01472             if ( d->type == DOLWILDARGS ) {
01473                 WORD *tterm = AT.WorkPointer, *tw;
01474                 WORD *ta = term, *tb = tterm, *tc, *td = term + *term;
01475                 while ( ta < t ) *tb++ = *ta++;
01476                 tc = tb;
01477                 while ( ta < tf ) *tb++ = *ta++;
01478                 tw = d->where+1;
01479                 while ( *tw ) {
01480                     if ( *tw < 0 ) {
01481                         if ( *tw > -FUNCTION ) *tb++ = *tw++;
01482                         *tb++ = *tw++;
01483                     }
01484                     else {
01485                         int ia;
01486                         for ( ia = 0; ia < *tw; ia++ ) *tb++ = *tw++;
01487                     }
01488                 }
01489                 NEXTARG(ta)
01490                 while ( ta < t+t[1] ) *tb++ = *ta++;
01491                 tc[1] = tb-tc;
01492                 while ( ta < td ) *tb++ = *ta++;
01493                 *tterm = tb - tterm;
01494                 {
01495                     int ia, na = *tterm;
01496                     ta = tterm; tb = term;
01497                     for ( ia = 0; ia < na; ia++ ) *tb++ = *ta++;
01498                 }
01499                 if ( tb > AT.WorkTop ) {
01500                     MLOCK(ErrorMessageLock);
01501                     MesWork();
01502                     goto EndTest2;
01503                 }
01504                 AT.WorkPointer = tb;
01505                 goto ReStart;
01506             }
01507         }
01508         NEXTARG(tf);
01509     }
01510     tf = t + FUNHEAD;
01511     while ( tf < t + t[1] ) {
01512         numargs++;
01513         if ( *tf > 0 && tf[1] != 0 ) todo = 0;
01514         NEXTARG(tf);
01515     }
01516     if ( todo && numargs == 2 ) {
01517         if ( *t == DIVFUNCTION ) AN.TeInFun = -9;
01518         else if ( *t == REMFUNCTION ) AN.TeInFun = -10;
01519         else if ( *t == INVERSEFUNCTION ) AN.TeInFun = -11;
01520         else if ( *t == MULFUNCTION ) AN.TeInFun = -14;
01521         else if ( *t == GCDFUNCTION ) AN.TeInFun = -8;
01522         AN.TeSuOut = 0;
01523         AR.TePos = -1;
01524         DONE(1)
01525     }
01526     else if ( todo && numargs == 3 ) {
01527         if ( *t == DIVFUNCTION ) AN.TeInFun = -9;
01528         else if ( *t == REMFUNCTION ) AN.TeInFun = -10;
01529         else if ( *t == GCDFUNCTION ) AN.TeInFun = -8;
01530         AN.TeSuOut = 0;
01531         AR.TePos = -1;
01532         DONE(1)
01533     }
01534     else if ( todo && *t == GCDFUNCTION ) {
01535         AN.TeInFun = -8;
01536         AN.TeSuOut = 0;
01537         AR.TePos = -1;
01538         DONE(1)
01539     }
01540 }
01541 else if ( *t == PERMUTATIONS && ( ( t[1] >= FUNHEAD+1
01542     && t[FUNHEAD] <= -FUNCTION ) || ( t[1] >= FUNHEAD+3
01543     && t[FUNHEAD] == -SNUMBER && t[FUNHEAD+2] <= -FUNCTION ) ) ) {
01544     AN.TeInFun = -12;
01545     AN.TeSuOut = 0;

```



```

01546         AR.TePos = -1;
01547         DONE(1)
01548     }
01549     else if ( *t == PARTITIONS ) {
01550         if ( TestPartitions(t,&(AT.partitions)) ) {
01551             AT.partitions.where = t-term;
01552             AN.TeInFun = -13;
01553             AN.TeSuOut = 0;
01554             AR.TePos = -1;
01555             DONE(1)
01556         }
01557     }
01558 }
01559 }
01560 t += t[1];
01561 } while ( t < m );
01562 if ( funflag ) { /* Search in functions */
01563 DoSpec:
01564     t = term;
01565     AT.NestPoin->termsize = t;
01566     if ( AT.NestPoin == AT.Nest ) AN.EndNest = t + *t;
01567     t++;
01568     oldncmod = AN.ncmod;
01569     if ( t < m ) do {
01570         if ( *t < FUNCTION ) {
01571             t += t[1]; continue;
01572         }
01573         if ( AN.ncmod && ( ( AC.modmode & ALSOFUNARGS ) == 0 ) ) {
01574             if ( *t != AR.PolyFun ) AN.ncmod = 0;
01575             else AN.ncmod = oldncmod;
01576         }
01577         r = t + t[1];
01578         funnum = *t;
01579         if ( *t >= FUNCTION + WILDOFFSET ) funnum -= WILDOFFSET;
01580         if ( ( *t == NUMFACTORS || *t == FIRSTTERM || *t == CONTENTTERM )
01581             && t[1] == FUNHEAD+2 &&
01582             ( t[FUNHEAD] == -EXPRESSION || t[FUNHEAD] == -DOLLAREXPRESSION ) ) {
01583             /*
01584                 if ( *t == NUMFACTORS ) {
01585                     This we leave for Normalize
01586                 }
01587             */
01588         }
01589         else if ( functions[funnum-FUNCTION].spec <= 0 ) {
01590             AT.NestPoin->funsize = t + 1;
01591             t1 = t;
01592             t += FUNHEAD;
01593             while ( t < r ) { /* Sum over arguments */
01594                 if ( *t > 0 && t[1] ) { /* Argument is dirty */
01595                     AT.NestPoin->argsize = t;
01596                     AT.NestPoin++;
01597                     stop = t + *t; /*
01598                 */
01599                     t2 = t;
01600                     t += ARGHEAD;
01601                     while ( t < AT.NestPoin[-1].argsize+*(AT.NestPoin[-1].argsize) ) {
01602                         /* Sum over terms */
01603                         AT.RecFlag++;
01604                         i = *t;
01605                         AN.subsubveto = 1;
01606                     /*
01607                         AN.subsubveto repairs a bug that became apparent
01608                         in an example by York Schroeder:
01609                         f(k1.k1)*replace_(k1,2*k2)
01610                         Is it possible to repair the counting of the various
01611                         length indicators? (JV 1-jun-2010)
01612                     */
01613                     if ( ( retvalue = TestSub(BHEAD t,level) ) != 0 ) {
01614                         Possible size changes:
01615                         Note defs at 471,467,460,400,425,328
01616                     */
01617                     redosize:
01618                         if ( i > *t ) {
01619                             /*
01620                                 i -= *t;
01621                                 *t2 -= i;
01622                                 t1[1] -= i;
01623                                 t += *t;
01624                                 r = t + i;
01625                                 m = term + *term;
01626                                 while ( r < m ) *t++ = *r++;
01627                                 *term -= i;
01628                             */
01629                             i -= *t;
01630                             t += *t;
01631                             r = t + i;
01632                             m = AN.EndNest;

```

```

01633         while ( r < m ) *t++ = *r++;
01634         t = AT.NestPoin[-1].argsize + ARGHEAD;
01635         n = AT.Nest;
01636         while ( n < AT.NestPoin ) {
01637             *(n->argsize) -= i;
01638             *(n->funsize) -= i;
01639             *(n->termsize) -= i;
01640             n++;
01641         }
01642         AN.EndNest -= i;
01643     }
01644     AN.subsubveto = 0;
01645     t1[2] = 1;
01646     if ( *t1 == AR.PolyFun && AR.PolyFunType == 2 )
01647         t1[2] |= MUSTCLEANPRF;
01648     AT.RecFlag--;
01649     AT.NestPoin--;
01650     AN.TeInFun++;
01651     AR.TePos = 0;
01652     AN.ncmod = oldncmod;
01653     DONE(retvalue)
01654 }
01655 else {
01656     /*
01657     * Somehow the next line fixes Issue #106.
01658     */
01659     i = *t;
01660     Normalize(BHEAD t);
01661     /* if ( i > *t ) { retvalue = 1; goto redosize; } */
01662     /*
01663     * Experimentally, the next line fixes Issue #105.
01664     */
01665     if ( *t == 0 ) { retvalue = 1; goto redosize; }
01666     {
01667         WORD *tend = t + *t, *tt = t+1;
01668         stilldirty = 0;
01669         tend -= ABS(tend[-1]);
01670         while ( tt < tend ) {
01671             if ( *tt == SUBEXPRESSION || *tt == EXPRESSION ) {
01672                 stilldirty = 1; break;
01673             }
01674             tt += tt[1];
01675         }
01676     }
01677     if ( i > *t ) {
01678         /*
01679         We should not forget to correct the Nest
01680         stack. That caused trouble in the past.
01681         */
01682         retvalue = 1;
01683         i -= *t;
01684         t += *t;
01685         r = t + i;
01686         m = AN.EndNest;
01687         while ( r < m ) *t++ = *r++;
01688         t = AT.NestPoin[-1].argsize + ARGHEAD;
01689         n = AT.Nest;
01690         while ( n < AT.NestPoin ) {
01691             *(n->argsize) -= i;
01692             *(n->funsize) -= i;
01693             *(n->termsize) -= i;
01694             n++;
01695         }
01696         AN.EndNest -= i;
01697     }
01698 }
01699 AN.subsubveto = 0;
01700 AT.RecFlag--;
01701 t += *t;
01702 }
01703 AT.NestPoin--;
01704 /*
01705 Argument contains no subexpressions.
01706 It should be normalized and sorted.
01707 The main problem is the storage.
01708 */
01709 t = AT.NestPoin->argsize;
01710 j = *t;
01711 t += ARGHEAD;
01712 NewSort(BHEAD0);
01713 if ( *t1 == AR.PolyFun && AR.PolyFunType == 2 ) {
01714     AR.CompareRoutine = (COMPAREDUMMY)(&CompareSymbols);
01715     AR.SortType = SORTHIGHFIRST;
01716 }
01717 if ( AT.WorkPointer < term + *term )
01718     AT.WorkPointer = term + *term;
01719

```

```

01720 while ( t < AT.NestPoin->argsize+(AT.NestPoin->argsize) ) {
01721     m = AT.WorkPointer;
01722     r = t + *t;
01723     do { *m++ = *t++; } while ( t < r );
01724     r = AT.WorkPointer;
01725     AT.WorkPointer = r + *r;
01726     if ( Normalize(BHEAD r) ) {
01727         if ( *t1 == AR.PolyFun && AR.PolyFunType == 2 ) {
01728             AR.SortType = oldsorttype;
01729             AR.CompareRoutine = (COMPAREDUMMY)oldcompareroutine;
01730             t1[2] |= MUSTCLEANPRF;
01731         }
01732         LowerSortLevel(); goto EndTest;
01733     }
01734     if ( AN.ncmod != 0 ) {
01735         if ( *r ) {
01736             if ( Modulus(r) ) {
01737                 LowerSortLevel();
01738                 AT.WorkPointer = r;
01739                 if ( *t1 == AR.PolyFun && AR.PolyFunType == 2 ) {
01740                     AR.SortType = oldsorttype;
01741                     AR.CompareRoutine = (COMPAREDUMMY)oldcompareroutine;
01742                     t1[2] |= MUSTCLEANPRF;
01743                 }
01744                 goto EndTest;
01745             }
01746         }
01747     }
01748     if ( AR.PolyFun > 0 ) {
01749         if ( PrepPoly(BHEAD r,1) != 0 ) goto EndTest;
01750     }
01751     if ( *r ) StoreTerm(BHEAD r);
01752     AT.WorkPointer = r;
01753 }
01754 if ( EndSort(BHEAD AT.WorkPointer+ARGHEAD,1) < 0 ) goto EndTest;
01755 m = AT.WorkPointer+ARGHEAD;
01756 if ( *t1 == AR.PolyFun && AR.PolyFunType == 2 ) {
01757     AR.SortType = oldsorttype;
01758     AR.CompareRoutine = (COMPAREDUMMY)oldcompareroutine;
01759     t1[2] |= MUSTCLEANPRF;
01760 }
01761 while ( *m ) m += *m;
01762 i = WORDDIF(m,AT.WorkPointer);
01763 *AT.WorkPointer = i;
01764 AT.WorkPointer[1] = stilldirty;
01765 if ( ToFast(AT.WorkPointer,AT.WorkPointer) ) {
01766     m = AT.WorkPointer;
01767     if ( *m <= -FUNCTION ) { m++; i = 1; }
01768     else { m += 2; i = 2; }
01769 }
01770 j = i - j;
01771 if ( j > 0 ) {
01772     r = m + j;
01773     if ( r > AT.WorkTop ) {
01774         MLOCK(ErrorMessageLock);
01775         MesWork();
01776         goto EndTest2;
01777     }
01778     do { *--r = *--m; } while ( m > AT.WorkPointer );
01779     AT.WorkPointer = r;
01780     m = AN.EndNest;
01781     r = m + j;
01782     stop = AT.NestPoin->argsize+(AT.NestPoin->argsize);
01783     do { *--r = *--m; } while ( m >= stop );
01784 }
01785 else if ( j < 0 ) {
01786     m = AT.NestPoin->argsize+(AT.NestPoin->argsize);
01787     r = m + j;
01788     do { *r++ = *m++; } while ( m < AN.EndNest );
01789 }
01790 m = AT.NestPoin->argsize;
01791 r = AT.WorkPointer;
01792 while ( --i >= 0 ) *m++ = *r++;
01793 n = AT.Nest;
01794 while ( n <= AT.NestPoin ) {
01795     if ( *(n->argsize) > 0 && n != AT.NestPoin )
01796         *(n->argsize) += j;
01797     *(n->funsize) += j;
01798     *(n->termsize) += j;
01799     n++;
01800 }
01801 AN.EndNest += j;
01802 /* (AT.NestPoin->argsize)[1] = 0; */
01803 if ( funnum == DENOMINATOR || funnum == EXPONENT ) {
01804     if ( Normalize(BHEAD term) ) {
01805         /*
01806             In this case something has been substituted

```

```

01807                                     Either a $ or a replace_????
01808                                     Originally we had here:
01809
01810                                     goto EndTest;
01811
01812                                     It seems better to restart.
01813 */
01814                                     AN.ncmod = oldncmod;
01815                                     goto ReStart;
01816                                 }
01817 /*
01818                                     And size changes here????
01819 */
01820                                 }
01821                                 AN.ncmod = oldncmod;
01822                                 goto ReStart;
01823                             }
01824                             else if ( *t == -DOLLAREXPRESSION ) {
01825                                 if ( ( *t1 == TERMSINEXPR || *t1 == SIZEOFFUNCTION )
01826                                     && t1[1] == FUNHEAD+2 ) {}
01827                                 else {
01828                                     if ( AR.Eside != LHSIDE ) {
01829                                         AN.TeInFun = 1; AR.TePos = 0;
01830                                         AT.Tmbuff = AM.dbufnum; t1[2] |= DIRTYFLAG;
01831                                         AN.ncmod = oldncmod;
01832                                         DONE(1)
01833                                     }
01834                                     AC.lhdollarflag = 1;
01835                                 }
01836                             }
01837                             else if ( *t == -TERMSINBRACKET ) {
01838                                 if ( AR.Eside != LHSIDE ) {
01839                                     AN.TeInFun = 1; AR.TePos = 0;
01840                                     t1[2] |= DIRTYFLAG;
01841                                     AN.ncmod = oldncmod;
01842                                     DONE(1)
01843                                 }
01844                             }
01845                             else if ( AN.ncmod != 0 && *t == -SNUMBER ) {
01846                                 if ( AN.ncmod == 1 || AN.ncmod == -1 ) {
01847                                     isp = (UWORD)(AC.cmod[0]);
01848                                     isp = t[1] % isp;
01849                                     if ( ( AC.modmode & POSNEG ) != 0 ) {
01850                                         if ( isp > (UWORD)(AC.cmod[0])/2 ) isp = isp - (UWORD)(AC.cmod[0]);
01851                                         else if ( -isp > (UWORD)(AC.cmod[0])/2 ) isp = isp +
(UWORD)(AC.cmod[0]);
01852                                     }
01853                                     else {
01854                                         if ( isp < 0 ) isp += (UWORD)(AC.cmod[0]);
01855                                     }
01856                                     if ( isp <= MAXPOSITIVE && isp >= -MAXPOSITIVE ) {
01857                                         t[1] = isp;
01858                                     }
01859                                 }
01860                             }
01861                             NEXTARG(t)
01862                         }
01863                     if ( funnum >= FUNCTION && functions[funnum-FUNCTION].tabl ) {
01864 /*
01865                                     Test whether the table catches
01866                                     Test 1: index arguments and range. i will be the number
01867                                     of the element in the table.
01868 */
01869                                     WORD rhsnumber, *oldwork = AT.WorkPointer;
01870                                     WORD ii, *p, *pp, *ppstop;
01871                                     MINMAX *mm;
01872                                     T = functions[funnum-FUNCTION].tabl;
01873 /*
01874                                     Because of tables with a variable number of indices
01875                                     we need to make a copy of the pattern.
01876                                     If we do this in the Workspace we get problems with EndNest.
01877                                     This is why we use TermMalloc.
01878                                     Now Tpattern is a copy that can be modified.
01879 */
01880 #ifdef WITHPTHREADS
01881                                     pp = T->pattern[AT.identity];
01882 #else
01883                                     pp = T->pattern;
01884 #endif
01885                                     if ( Tpattern == 0 ) Tpattern = TermMalloc("Tpattern");
01886                                     p = Tpattern;
01887                                     ii = pp[1];
01888                                     for ( i = 0; i < ii; i++ ) *p++ = *pp++;
01889
01890                                     p = Tpattern + FUNHEAD+1;
01891                                     mm = T->mm;
01892                                     if ( T->sparse ) {

```

```

01893      WORD xx;
01894      t = t1+FUNHEAD;
01895      if ( T->numind == 0 ) { isp = 0; xx = 0; }
01896      else {
01897          if ( T->numind < 0 ) {
01898              if ( *t != -SNUMBER && t[2] != -SNUMBER ) {
01899                  xx = ABS(T->numind);
01900                  goto teststrict;
01901              }
01902              xx = t[1]+1;
01903              if ( xx < 2 || xx > -T->numind ) goto teststrict;
01904          }
01905          else xx = T->numind;
01906          for ( i = 0; i < xx; i++, t += 2 ) {
01907              if ( *t != -SNUMBER ) break;
01908          }
01909          if ( i < xx ) goto teststrict;
01910          isp = FindTableTree(T,t1+FUNHEAD,2);
01911      }
01912      if ( isp < 0 ) {
01913          if ( T->strict == -2 ) {
01914              rhsnumber = AM.zerorhs;
01915              tbufnum = AM.zbufnum;
01916          }
01917          else if ( T->strict == -3 ) {
01918              rhsnumber = AM.onerhs;
01919              tbufnum = AM.zbufnum;
01920          }
01921          else if ( T->strict < 0 ) goto NextFun;
01922          else {
01923              MLOCK(ErrorMessageLock);
01924              MesPrint("Element in table is undefined");
01925              if ( Tpattern ) {
01926                  TermFree(Tpattern,"Tpattern");
01927                  Tpattern = 0;
01928              }
01929              goto showtable;
01930          }
01931          /*
01932          Copy the indices;
01933          */
01934          t = t1+FUNHEAD+1;
01935          for ( i = 0; i < xx; i++ ) {
01936              *p = *t; p+=2; t+=2;
01937          }
01938      }
01939      else {
01940          rhsnumber = T->tablepointers[isp+ABS(T->numind)];
01941          #if ( TABLEEXTENSION == 2 )
01942          tbufnum = T->bufnum;
01943          #else
01944          tbufnum = T->tablepointers[isp+ABS(T->numind)+1];
01945          #endif
01946          t = t1+FUNHEAD+1;
01947          ii = xx;
01948          while ( --ii >= 0 ) {
01949              *p = *t; t += 2; p += 2;
01950          }
01951      }
01952      if ( xx < ABS(T->numind) ) {
01953          p--; ppstop = Tpattern+Tpattern[1];
01954          pp = p+2*(-T->numind-xx);
01955          while ( pp < ppstop ) *p++ = *pp++;
01956          Tpattern[1] = p - Tpattern;
01957      }
01958      goto caughttable;
01959  }
01960  else {
01961      i = 0;
01962      t = t1 + FUNHEAD;
01963      j = T->numind;
01964      while ( --j >= 0 ) {
01965          if ( *t != -SNUMBER ) goto NextFun;
01966          t++;
01967          if ( *t < mm->mini || *t > mm->maxi ) {
01968              if ( T->bounds ) {
01969                  MLOCK(ErrorMessageLock);
01970                  MesPrint("Table boundary check. Argument %d",
01971                      T->numind-j);
01972                  AO.OutFill = AO.OutputLine = (UBYTE *)m;
01973                  AO.OutSkip = 8;
01974                  IniLine(0);
01975                  WriteSubTerm(t1,1);
01976                  FiniLine();
01977                  MUNLOCK(ErrorMessageLock);
01978                  if ( Tpattern ) {
01979                      TermFree(Tpattern,"Tpattern");

```

```

01980             Tpattern = 0;
01981         }
01982         SETERROR(-1)
01983     }
01984     if ( Tpattern ) {
01985         TermFree(Tpattern,"Tpattern");
01986         Tpattern = 0;
01987     }
01988     goto NextFun;
01989 }
01990 i += ( *t - mm->mini ) * (LONG) (mm->size);
01991 *p = *t++;
01992 p += 2;
01993 mm++;
01994 }
01995 /*
01996     Test now whether the entry exists.
01997 */
01998 i *= TABLEEXTENSION;
01999 if ( T->tablepointers[i] == -1 ) {
02000     if ( T->strict == -2 ) {
02001         rhsnumber = AM.zerorhs;
02002         tbufnum = AM.zbufnum;
02003     }
02004     else if ( T->strict == -3 ) {
02005         rhsnumber = AM.onerhs;
02006         tbufnum = AM.zbufnum;
02007     }
02008     else if ( T->strict < 0 ) {
02009         if ( Tpattern ) {
02010             TermFree(Tpattern,"Tpattern");
02011             Tpattern = 0;
02012         }
02013         goto NextFun;
02014     }
02015     else {
02016         MLOCK(ErrorMessageLock);
02017         MesPrint("Element in table is undefined");
02018         if ( Tpattern ) {
02019             TermFree(Tpattern,"Tpattern");
02020             Tpattern = 0;
02021         }
02022         goto showtable;
02023     }
02024 }
02025 else {
02026     rhsnumber = T->tablepointers[i];
02027     #if ( TABLEEXTENSION == 2 )
02028     tbufnum = T->bbufnum;
02029 #else
02030     tbufnum = T->tablepointers[i+1];
02031 #endif
02032 }
02033 }
02034 /*
02035     If there are more arguments we have to do some
02036     pattern matching. This should be easy. We adapted the
02037     pattern, so that the array indices match already.
02038     Note that if there is no match the program will become
02039     very slow.
02040 */
02041 caughttable:
02042 #ifdef WITHPTHREADS
02043     AN.FullProto = T->prototype[AT.identity];
02044 #else
02045     AN.FullProto = T->prototype;
02046 #endif
02047 AN.WildValue = AN.FullProto + SUBEXPSIZE;
02048 AN.WildStop = AN.FullProto+AN.FullProto[1];
02049 ClearWild(BHEAD0);
02050 AN.RepFunNum = 0;
02051 AN.RepFunList = AN.EndNest;
02052 AT.WorkPointer = (WORD *) (((UBYTE *) (AN.EndNest)) + AM.MaxTer/2);
02053 if ( AT.WorkPointer >= AT.WorkTop ) {
02054     MLOCK(ErrorMessageLock);
02055     MesWork();
02056     MUNLOCK(ErrorMessageLock);
02057 }
02058 wilds = 0;
02059 if ( MatchFunction(BHEAD Tpattern,t1,&wilds) > 0 ) {
02060     AT.WorkPointer = oldwork;
02061     if ( AT.NestPoin != AT.Nest ) {
02062         AN.ncmod = oldncmod;
02063         if ( Tpattern ) {
02064             TermFree(Tpattern,"Tpattern");
02065             Tpattern = 0;
02066         }
02067     }

```

```

02067         DONE(1)
02068     }
02069
02070     m = AN.FullProto;
02071     retvalue = m[2] = rhsnumber;
02072     m[4] = tbufnum;
02073     t = t1;
02074     j = t[1];
02075     i = m[1];
02076     if ( j > i ) {
02077         j = i - j;
02078         NCOPY(t,m,i);
02079         m = term + *term;
02080         while ( r < m ) *t++ = *r++;
02081         *term += j;
02082     }
02083     else if ( j < i ) {
02084         j = i - j;
02085         t = term + *term;
02086         while ( t >= r ) { t[j] = *t; t--; }
02087         t = t1;
02088         NCOPY(t,m,i);
02089         *term += j;
02090     }
02091     else {
02092         NCOPY(t,m,j);
02093     }
02094     AN.TeInFun = 0;
02095     AR.TePos = 0;
02096     AN.TeSuOut = -1;
02097     if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
02098     AT.TMbuff = tbufnum;
02099     AN.ncmod = oldncmod;
02100     DONE(retvalue);
02101 }
02102 AT.WorkPointer = oldwork;
02103 if ( Tpattern ) {
02104     TermFree(Tpattern,"Tpattern");
02105     Tpattern = 0;
02106 }
02107 }
02108 NextFun;;
02109 }
02110 else if ( ( t[2] & DIRTYFLAG ) != 0 ) {
02111     t += FUNHEAD;
02112     while ( t < r ) {
02113         if ( *t == FUNNYDOLLAR ) {
02114             if ( AR.Eside != LHSIDE ) {
02115                 AN.TeInFun = 1;
02116                 AR.TePos = 0;
02117                 AT.TMbuff = AM.dbufnum;
02118                 AN.ncmod = oldncmod;
02119                 DONE(1)
02120             }
02121             AC.lhdollarflag = 1;
02122         }
02123         t++;
02124     }
02125 }
02126 t = r;
02127 AN.ncmod = oldncmod;
02128 } while ( t < m );
02129 }
02130 Done:
02131 if ( Tpattern ) {
02132     TermFree(Tpattern,"Tpattern");
02133     Tpattern = 0;
02134 }
02135 return(retvalue);
02136 EndTest;;
02137 MLOCK(ErrorMessageLock);
02138 EndTest2;;
02139 MesCall("TestSub");
02140 MUNLOCK(ErrorMessageLock);
02141 SETERROR(-1)
02142 }
02143
02144 /*
02145     #] TestSub :
02146     #[ InFunction :      WORD InFunction(term,termout)
02147 */
02160 int InFunction(PHEAD WORD *term, WORD *termout)
02161 {
02162     GETBIDENTITY
02163     WORD *m, *t, *r, *rr, sign = 1, oldncmod;
02164     WORD *u, *v, *w, *from, *to,
02165     ipp, olddefer = AR.DeferFlag, oldPolyFun = AR.PolyFun, i, j;

```

```

02166     LONG numterms;
02167     from = t = term;
02168     r = t + *t - 1;
02169     m = r - ABS(*r) + 1;
02170     t++;
02171     while ( t < m ) {
02172         if ( *t >= FUNCTION+WILDOFFSET ) ipp = *t - WILDOFFSET;
02173         else ipp = *t;
02174         if ( AR.TePos ) {
02175             if ( ( term + AR.TePos ) == t ) {
02176                 m = termout;
02177                 while ( from < t ) *m++ = *from++;
02178                 *m++ = DENOMINATOR;
02179                 *m++ = t[1] + 4 + FUNHEAD + ARGHEAD;
02180                 *m++ = DIRTYFLAG;
02181                 FILLFUN3(m);
02182                 *m++ = t[1] + 4 + ARGHEAD;
02183                 *m++ = 1;
02184                 FILLARG(m);
02185                 *m++ = t[1] + 4;
02186                 t[3] = -t[3];
02187                 v = t + t[1];
02188                 while ( t < v ) *m++ = *t++;
02189                 from[3] = -from[3];
02190                 *m++ = 1;
02191                 *m++ = 1;
02192                 *m++ = 3;
02193                 r = term + *term;
02194                 while ( t < r ) *m++ = *t++;
02195                 if ( (m-termout) > (LONG)(AM.MaxTer/sizeof(WORD)) ) {
02196                     MLOCK(ErrorMessageLock);
02197                     MesPrint("Output term too large (%d words) (MaxTermSize: %d words)", m-termout,
AM.MaxTer/sizeof(WORD));
02198                     MUNLOCK(ErrorMessageLock);
02199                     goto TooLarge;
02200                 }
02201                 *termout = WORDDIF(m,termout);
02202                 return(0);
02203             }
02204         }
02205     } else if ( ( *t >= FUNCTION && functions[ipp-FUNCTION].spec <= 0 )
&& ( t[2] & DIRTYFLAG ) == DIRTYFLAG ) {
02206         m = termout;
02207         r = t + t[1];
02208         u = t;
02209         t += FUNHEAD;
02210         oldncmod = AN.ncmod;
02211         while ( t < r ) { /* t points at an argument */
02212             if ( *t > 0 && t[1] ) { /* Argument has been modified */
02213                 WORD oldsorctype = AR.SortType;
02214                 /* This whole argument must be redone */
02215
02216                 if ( ( AN.ncmod != 0 )
&& ( ( AC.modmode & ALSOFUNARGS ) == 0 )
&& ( *u != AR.PolyFun ) ) { AN.ncmod = 0; }
02217                 AR.DeferFlag = 0;
02218                 v = t + *t;
02219                 t += ARGHEAD; /* First term */
02220                 w = 0; /* to appease the compilers warning devices */
02221                 while ( from < t ) {
02222                     if ( from == u ) w = m;
02223                     *m++ = *from++;
02224                 }
02225                 to = m;
02226                 NewSort(BHEAD0);
02227                 if ( *u == AR.PolyFun && AR.PolyFunType == 2 ) {
02228                     AR.CompareRoutine = (COMPAREDUMMY)(&CompareSymbols);
02229                     AR.SortType = SORTHIGHFIRST;
02230                 }
02231                 AR.PolyFun = 0;
02232
02233                 while ( t < v ) {
02234                     i = *t;
02235                     NCOPY(m,t,i);
02236                     m = to;
02237                     if ( AT.WorkPointer < m+m ) AT.WorkPointer = m + *m;
02238                     if ( Generator(BHEAD m,AR.Cnumlhs) ) {
02239                         AN.ncmod = oldncmod;
02240                         LowerSortLevel(); goto InFunc;
02241                     }
02242                 }
02243                 /* w = the function */
02244                 /* v = the next argument */
02245                 /* u = the function */
02246                 /* to is new argument */
02247             }
02248         }
02249     }
02250 }
02251

```



```

02252         to -= ARGHEAD;
02253         if ( EndSort(BHEAD m,1) < 0 ) {
02254             AN.ncmod = oldncmod;
02255             goto InFunc;
02256         }
02257         AR.PolyFun = oldPolyFun;
02258         if ( *u == AR.PolyFun && AR.PolyFunType == 2 ) {
02259             AR.CompareRoutine = (COMPAREDDUMMY) (&Compare1);
02260             AR.SortType = oldsorttype;
02261         }
02262         while ( *m ) m += *m;
02263         *to = WORDDIF(m,to);
02264         to[1] = 1; /* ?????? or rather 0?. 24-mar-2006 JV */
02265         if ( ToFast(to,to) ) {
02266             if ( *to <= -FUNCTION ) m = to+1;
02267             else m = to+2;
02268         }
02269         w[1] = WORDDIF(m,w) + WORDDIF(r,v);
02270         r = term + *term;
02271         t = v;
02272         while ( t < r ) *m++ = *t++;
02273         if ( (m-termout) > (LONG)(AM.MaxTer/sizeof(WORD)) ) {
02274             MLOCK(ErrorMessageLock);
02275             MesPrint("Output term too large (%d words) (MaxTermSize: %d words)",
m-termout, AM.MaxTer/sizeof(WORD));
02276             MUNLOCK(ErrorMessageLock);
02277             goto TooLarge;
02278         }
02279         *termout = WORDDIF(m,termout);
02280         AR.DeferFlag = olddefer;
02281         AN.ncmod = oldncmod;
02282         return(0);
02283     }
02284     else if ( *t == -DOLLAREXPRESSION ) {
02285         if ( AR.Eside == LHSIDE ) {
02286             NEXTARG(t)
02287             AC.lhdollarflag = 1;
02288         }
02289         else {
02290             /*
02291              This whole argument must be redone
02292             */
02293             DOLLARS d = Dollars + t[1];
02294             #ifdef WITHPTHREADS
02295             int nummodopt, dtype = -1;
02296             if ( AS.MultiThreaded ) {
02297                 for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
02298                     if ( t[1] == ModOptdollars[nummodopt].number ) break;
02299                 }
02300                 if ( nummodopt < NumModOptdollars ) {
02301                     dtype = ModOptdollars[nummodopt].type;
02302                     if ( dtype == MODLOCAL ) {
02303                         d = ModOptdollars[nummodopt].dstruct+AT.identity;
02304                     }
02305                     else {
02306                         LOCK(d->pthreadlockread);
02307                     }
02308                 }
02309             }
02310             #endif
02311             oldncmod = AN.ncmod;
02312             if ( ( AN.ncmod != 0 )
02313                 && ( ( AC.modmode & ALSOFUNARGS ) == 0 )
02314                 && ( *u != AR.PolyFun ) ) { AN.ncmod = 0; }
02315             AR.DeferFlag = 0;
02316             v = t + 2;
02317             w = 0; /* to appease the compilers warning devices */
02318             while ( from < t ) {
02319                 if ( from == u ) w = m;
02320                 *m++ = *from++;
02321             }
02322             to = m;
02323             switch ( d->type ) {
02324                 case DOLINDEX:
02325                     if ( d->index >= 0 && d->index < AM.OffsetIndex ) {
02326                         *m++ = -SNUMBER; *m++ = d->index;
02327                     }
02328                     else { *m++ = -INDEX; *m++ = d->index; }
02329                     break;
02330                 case DOLZERO:
02331                     *m++ = -SNUMBER; *m++ = 0; break;
02332                 case DOLNUMBER:
02333                     if ( d->where[0] == 4 &&
02334                         ( d->where[1] & MAXPOSITIVE ) == d->where[1] ) {
02335                         *m++ = -SNUMBER;
02336                         if ( d->where[3] >= 0 ) *m++ = d->where[1];
02337                         else *m++ = -d->where[1];

```

```

02338             break;
02339         }
02340         /* fall through */
02341     case DOLTERMS:
02342         /*
02343             Here we have the special case of the PolyRatFun
02344             That function may have a different sort of the
02345             terms in the argument.
02346         */
02347         to = m; r = d->where;
02348         *m++ = 0; *m++ = 1;
02349         FILLARG(m)
02350         while ( *r ) {
02351             i = *r; NCOPY(m,r,i)
02352         }
02353         *to = m-to;
02354         if ( ToFast(to,to) ) {
02355             if ( *to <= -FUNCTION ) m = to+1;
02356             else m = to+2;
02357         }
02358         else if ( *u == AR.PolyFun && AR.PolyFunType == 2 ) {
02359             AR.PolyFun = 0;
02360             NewSort(BHEAD0);
02361             AR.CompareRoutine = (COMPAREDUMMY)(&CompareSymbols);
02362             r = to + ARGHEAD;
02363             while ( r < m ) {
02364                 rr = r; r += *r;
02365                 if ( SymbolNormalize(rr) ) goto InFunc;
02366                 if ( StoreTerm(BHEAD rr) ) {
02367                     AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
02368                     LowerSortLevel();
02369                     Terminate(-1);
02370                 }
02371             }
02372             if ( EndSort(BHEAD to+ARGHEAD,1) < 0 ) goto InFunc;
02373             AR.PolyFun = oldPolyFun;
02374             AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
02375             m = to+ARGHEAD;
02376             if ( *m == 0 ) {
02377                 *to = -SNUMBER;
02378                 to[1] = 0;
02379                 m = to + 2;
02380             }
02381             else {
02382                 while ( *m ) m += *m;
02383                 *t = m - to;
02384                 if ( ToFast(to,to) ) {
02385                     if ( *to <= -FUNCTION ) m = to+1;
02386                     else m = to+2;
02387                 }
02388             }
02389         }
02390         w[1] = w[1] - 2 + (m-to);
02391         break;
02392     case DOLSUBTERM:
02393         to = m; r = d->where;
02394         i = r[1];
02395         *m++ = i+4+ARGHEAD; *m++ = 1;
02396         FILLARG(m)
02397         *m++ = i+4;
02398         while ( --i >= 0 ) *m++ = *r++;
02399         *m++ = 1; *m++ = 1; *m++ = 3;
02400         if ( ToFast(to,to) ) {
02401             if ( *to <= -FUNCTION ) m = to+1;
02402             else m = to+2;
02403         }
02404         w[1] = w[1] - 2 + (m-to);
02405         break;
02406     case DOLARGUMENT:
02407         to = m; r = d->where;
02408         if ( *r > 0 ) {
02409             i = *r - 2;
02410             *m++ = *r++; *m++ = 1; r++;
02411             while ( --i >= 0 ) *m++ = *r++;
02412         }
02413         else if ( *r <= -FUNCTION ) *m++ = *r++;
02414         else { *m++ = *r++; *m++ = *r++; }
02415         w[1] = w[1] - 2 + (m-to);
02416         break;
02417     case DOLWILDARGS:
02418         to = m; r = d->where;
02419         if ( *r > 0 ) { /* Tensor arguments */
02420             i = *r++;
02421             while ( --i >= 0 ) {
02422                 if ( *r < 0 ) {
02423                     *m++ = -VECTOR; *m++ = *r++;
02424                 }

```

```

02425         else if ( *r >= AM.OffsetIndex ) {
02426             *m++ = -INDEX; *m++ = *r++;
02427         }
02428         else { *m++ = -SNUMBER; *m++ = *r++; }
02429     }
02430 }
02431 else { /* Regular arguments */
02432     r++;
02433     while ( *r ) {
02434         if ( *r > 0 ) {
02435             i = *r - 2;
02436             *m++ = *r++; *m++ = 1; r++;
02437             while ( --i >= 0 ) *m++ = *r++;
02438         }
02439         else if ( *r <= -FUNCTION ) *m++ = *r++;
02440         else { *m++ = *r++; *m++ = *r++; }
02441     }
02442 }
02443 w[1] = w[1] - 2 + (m-to);
02444 break;
02445 case DOLUNDEFINED:
02446 default:
02447     MLOCK(ErrorMessageLock);
02448     MesPrint("!!!Undefined $-variable: %s!!!",
02449         AC.dollarnames->namebuffer+d->name);
02450     MUNLOCK(ErrorMessageLock);
02451 #ifdef WITHPTHREADS
02452     if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
02453 #endif
02454     Terminate(-1);
02455 }
02456 #ifdef WITHPTHREADS
02457 if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
02458 #endif
02459 r = term + *term;
02460 t = v;
02461 while ( t < r ) *m++ = *t++;
02462 if ( (m-termout) > (LONG)(AM.MaxTer/sizeof(WORD)) ) {
02463     MLOCK(ErrorMessageLock);
02464     MesPrint("Output term too large (%d words) (MaxTermSize: %d words)",
02465         m-termout, AM.MaxTer/sizeof(WORD));
02465     MUNLOCK(ErrorMessageLock);
02466     goto TooLarge;
02467 }
02468 *termout = WORDDIFF(m,termout);
02469 AR.DeferFlag = olddefer;
02470 AN.ncmod = oldncmod;
02471 return(0);
02472 }
02473 }
02474 else if ( *t == -TERMSINBRACKET ) {
02475     if ( AC.ComDefer ) numterms = CountTerms1(BHEAD0);
02476     else numterms = 1;
02477 /*
02478     Compose the output term
02479     First copy the part till this function argument
02480     m points at the output term space
02481     u points at the start of the function
02482     t points at the start of the argument
02483 */
02484     w = 0;
02485     while ( from < t ) {
02486         if ( from == u ) w = m;
02487         *m++ = *from++;
02488     }
02489     if ( ( numterms & MAXPOSITIVE ) == numterms ) {
02490         *m++ = -SNUMBER; *m++ = numterms & MAXPOSITIVE;
02491         w[1] += 1;
02492     }
02493     else if ( ( i = numterms » BITSINWORD ) == 0 ) {
02494         *m++ = ARGHEAD+4;
02495         for ( j = 1; j < ARGHEAD; j++ ) *m++ = 0;
02496         *m++ = 4; *m++ = numterms & WORDMASK; *m++ = 1; *m++ = 3;
02497         w[1] += ARGHEAD+3;
02498     }
02499     else {
02500         *m++ = ARGHEAD+6;
02501         for ( j = 1; j < ARGHEAD; j++ ) *m++ = 0;
02502         *m++ = 6; *m++ = numterms & WORDMASK;
02503         *m++ = i; *m++ = 1; *m++ = 0; *m++ = 5;
02504         w[1] += ARGHEAD+5;
02505     }
02506     from++; /* Skip our function */
02507     r = term + *term;
02508     while ( from < r ) *m++ = *from++;
02509     if ( (m-termout) > (LONG)(AM.MaxTer/sizeof(WORD)) ) {
02510         MLOCK(ErrorMessageLock);

```

```

02511             MesPrint("Output term too large (%d words) (MaxTermSize: %d words)",
m-termout, AM.MaxTer/sizeof(WORD));
02512             MUNLOCK(ErrorMessageLock);
02513             goto TooLarge;
02514         }
02515         *termout = WORDDIFF(m,termout);
02516         return(0);
02517     }
02518     else { NEXTARG(t) }
02519 }
02520 t = u;
02521 }
02522 else if ( ( *t >= FUNCTION && functions[ipp-FUNCTION].spec > 0 )
&& ( t[2] & DIRTYFLAG ) == DIRTYFLAG ) { /* Could be FUNNYDOLLAR */
02523     u = t; v = t + t[1];
02524     t += FUNHEAD;
02525     while ( t < v ) {
02526         if ( *t == FUNNYDOLLAR ) {
02527             if ( AR.Eside != LHSIDE ) {
02528                 DOLLARS d = Dollars + t[1];
02529 #ifdef WITHPTHREADS
02530                 int nummodopt, dtype = -1;
02531                 if ( AS.MultiThreaded ) {
02532                     for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
02533                         if ( t[1] == ModOptdollars[nummodopt].number ) break;
02534                     }
02535                     if ( nummodopt < NumModOptdollars ) {
02536                         dtype = ModOptdollars[nummodopt].type;
02537                         if ( dtype == MODLOCAL ) {
02538                             d = ModOptdollars[nummodopt].dstruct+AT.identity;
02539                         }
02540                         else {
02541                             LOCK(d->pthreadlockread);
02542                         }
02543                     }
02544                 }
02545             }
02546 #endif
02547             oldncmod = AN.ncmod;
02548             if ( ( AN.ncmod != 0 )
&& ( ( AC.modmode & ALSOFUNARGS ) == 0 )
&& ( *u != AR.PolyFun ) ) { AN.ncmod = 0; }
02549             m = termout; w = 0;
02550             while ( from < t ) {
02551                 if ( from == u ) w = m;
02552                 *m++ = *from++;
02553             }
02554             to = m;
02555             switch ( d->type ) {
02556                 case DOLINDEX:
02557                     *m++ = d->index; break;
02558                 case DOLZERO:
02559                     *m++ = 0; break;
02560                 case DOLNUMBER:
02561                 case DOLTERMS:
02562                     if ( d->where[0] == 4 && d->where[4] == 0
&& d->where[3] == 3 && d->where[2] == 1
&& d->where[1] < AM.OffsetIndex ) {
02563                         *m++ = d->where[1];
02564                     }
02565                     else {
02566                         if( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
02567                         MLOCK(ErrorMessageLock);
02568                         MesPrint("$%s has wrong type for tensor substitution",
AC.dollarnames->namebuffer+d->name);
02569                         MUNLOCK(ErrorMessageLock);
02570                         AN.ncmod = oldncmod;
02571                         return(-1);
02572                     }
02573                     break;
02574                 case DOLARGUMENT:
02575                     if ( d->where[0] == -INDEX ) {
02576                         *m++ = d->where[1]; break;
02577                     }
02578                     else if ( d->where[0] == -VECTOR ) {
02579                         *m++ = d->where[1]; break;
02580                     }
02581                     else if ( d->where[0] == -MINVECTOR ) {
02582                         *m++ = d->where[1];
02583                         sign = -sign;
02584                         break;
02585                     }
02586                     else if ( d->where[0] == -SNUMBER ) {
02587                         if ( d->where[1] >= 0
&& d->where[1] < AM.OffsetIndex ) {
02588

```

```

02597         *m++ = d->where[1]; break;
02598     }
02599 }
02600 goto wrongtype;
02601 case DOLWILDARGS:
02602     if ( d->where[0] > 0 ) {
02603         r = d->where; i = *r++;
02604         while ( --i >= 0 ) *m++ = *r++;
02605     }
02606     else {
02607         r = d->where + 1;
02608         while ( *r ) {
02609             if ( *r == -INDEX ) {
02610                 *m++ = r[1]; r += 2; continue;
02611             }
02612             else if ( *r == -VECTOR ) {
02613                 *m++ = r[1]; r += 2; continue;
02614             }
02615             else if ( *r == -MINVECTOR ) {
02616                 *m++ = r[1]; r += 2;
02617                 sign = -sign; continue;
02618             }
02619             else if ( *r == -SNUMBER ) {
02620                 if ( r[1] >= 0
02621                     && r[1] < AM.OffsetIndex ) {
02622                     *m++ = r[1]; r += 2; continue;
02623                 }
02624             }
02625             goto wrongtype;
02626         }
02627     }
02628     break;
02629 case DOLSUBTERM:
02630     r = d->where;
02631     if ( *r == INDEX && r[1] == 3 ) {
02632         *m++ = r[2];
02633     }
02634     else goto wrongtype;
02635     break;
02636 case DOLUNDEFINED:
02637     #ifdef WITHPTHREADS
02638         if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
02639     #endif
02640     MLOCK(ErrorMessageLock);
02641     MesPrint("$%s is undefined in tensor substitution",
02642             AC.dollarnames->namebuffer+d->name);
02643     MUNLOCK(ErrorMessageLock);
02644     AN.ncmod = oldncmod;
02645     return(-1);
02646 }
02647 #ifdef WITHPTHREADS
02648     if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
02649 #endif
02650     w[1] = w[1] - 2 + (m-to);
02651     from += 2;
02652     term += *term;
02653     while ( from < term ) *m++ = *from++;
02654     if ( sign < 0 ) m[-1] = -m[-1];
02655     if ( (m-termout) > (LONG)(AM.MaxTer/sizeof(WORD)) ) {
02656         MLOCK(ErrorMessageLock);
02657         MesPrint("Output term too large (%d words) (MaxTermSize: %d words)",
02658                 m-termout, AM.MaxTer/sizeof(WORD));
02659         MUNLOCK(ErrorMessageLock);
02660         goto TooLarge;
02661     }
02662     *termout = m - termout;
02663     AN.ncmod = oldncmod;
02664     return(0);
02665 }
02666     else {
02667         AC.lhdollarflag = 1;
02668     }
02669 }
02670     t++;
02671     t = u;
02672 }
02673     t += t[1];
02674 }
02675 MLOCK(ErrorMessageLock);
02676 MesPrint("Internal error in InFunction: Function not encountered.");
02677 if ( AM.tracebackflag ) {
02678     MesPrint("%w: AR.TePos = %d",AR.TePos);
02679     MesPrint("%w: AN.TeInFun = %d",AN.TeInFun);
02680     termout = term;
02681     AO.OutFill = AO.OutputLine = (UBYTE *)AT.WorkPointer + AM.MaxTer;
02682     AO.OutSkip = 3;

```

```

02683     FiniLine();
02684     i = *termout;
02685     while ( --i >= 0 ) {
02686         TalToLine((UWORD) (*termout++));
02687         TokenToLine((UBYTE *) " ");
02688     }
02689     AO.OutSkip = 0;
02690     FiniLine();
02691     MesCall("InFunction");
02692 }
02693 MUNLOCK(ErrorMessageLock);
02694 return(1);
02695
02696 InFunc:
02697     MLOCK(ErrorMessageLock);
02698     MesCall("InFunction");
02699     MUNLOCK(ErrorMessageLock);
02700     SETERROR(-1)
02701
02702 TooLarge:
02703     MLOCK(ErrorMessageLock);
02704     MesCall("InFunction");
02705     MUNLOCK(ErrorMessageLock);
02706     SETERROR(-1)
02707 }
02708
02709 /*
02710     #] InFunction :
02711     #[ InsertTerm :          WORD InsertTerm(term,replac,extractbuff,position,termout)
02712 */
02730 int InsertTerm(PHEAD WORD *term, WORD replac, WORD extractbuff, WORD *position, WORD *termout,
02731               WORD tepos)
02732 {
02733     GETBIDENTITY
02734     WORD *m, *t, *r, i, l2, j;
02735     WORD *u, *v, ll, *coef;
02736     coef = AT.WorkPointer;
02737     if ( ( AT.WorkPointer = coef + 2*AM.MaxTal ) > AT.WorkTop ) {
02738         MLOCK(ErrorMessageLock);
02739         MesWork();
02740         MUNLOCK(ErrorMessageLock);
02741         return(-1);
02742     }
02743     t = term;
02744     r = t + *t;
02745     ll = l2 = r[-1];
02746     m = r - ABS(l2);
02747     if ( tepos > 0 ) {
02748         t = term + tepos;
02749         goto foundit;
02750     }
02751     t++;
02752     while ( t < m ) {
02753         if ( *t == SUBEXPRESSION && t[2] == replac && t[3] && t[4] == extractbuff ) {
02754             r = t + t[1];
02755             while ( *r == SUBEXPRESSION && r[2] == replac && r[3] && r < m && r[4] == extractbuff ) {
02756                 t = r; r += r[1];
02757             }
02758 foundit:;
02759             u = m;
02760             r = term;
02761             m = termout;
02762             do { *m++ = *r++; } while ( r < t );
02763             if ( t[1] > SUBEXPSIZE ) {
02764 /*
02765                 if this is a dollar expression there are no wildcards
02766 */
02767                 i = *--m;
02768                 if ( ( l2 = WildFill(BHEAD m,position,t) ) < 0 ) goto InsCall;
02769                 *m = i;
02770                 m += l2-1;
02771                 l2 = *m;
02772                 i = ( j = ABS(l2) ) - 1;
02773                 r = coef + i;
02774                 do { *--r = *--m; } while ( --i > 0 );
02775             }
02776             else {
02777                 v = t;
02778                 t = position;
02779                 r = t + *t;
02780                 l2 = r[-1];
02781                 r -= ( j = ABS(l2) );
02782                 t++;
02783                 if ( t < r ) do { *m++ = *t++; } while ( t < r );
02784                 t = v;
02785             }
02786             t += t[1];

```

```

02787         while ( t < u && *t == DOLLAREXP2 ) t += t[1];
02788 ComAct:    if ( t < u ) do { *m++ = *t++; } while ( t < u );
02789         if ( *r == 1 && r[1] == 1 && j == 3 ) {
02790             if ( l2 < 0 ) l1 = -l1;
02791             i = ABS(l1)-1;
02792             NCOPY(m,t,i);
02793             *m++ = l1;
02794         }
02795         else {
02796             if ( MulRat(BHEAD (UWORD *)u,REDLENG(l1), (UWORD *)r,REDLENG(l2),
02797 (UWORD *)m,&l1) ) goto InsCall;
02798             l2 = l1;
02799             l2 *= 2;
02800             if ( l2 < 0 ) {
02801                 m -= l2;
02802                 *m++ = l2-1;
02803             }
02804             else {
02805                 m += l2;
02806                 *m++ = l2+1;
02807             }
02808         }
02809         *termout = WORDDIF(m,termout);
02810         if ( (*termout)*((LONG)sizeof(WORD)) > AM.MaxTer ) {
02811             MLOCK(ErrorMessageLock);
02812             MesPrint("Term too complex during substitution (%d words). MaxTermSize (%l words) is
too small.", *termout, AM.MaxTer/(LONG)sizeof(WORD) );
02813             goto InsCall2;
02814         }
02815         AT.WorkPointer = coef;
02816         return(0);
02817     }
02818     t += t[1];
02819 }
02820 /*
02821 The next action is for when there is no subexpression pointer.
02822 We append the extra term. Effectively the routine becomes now a
02823 merge routine for two terms.
02824 */
02825 v = t;
02826 u = m;
02827 r = term;
02828 m = termout;
02829 do { *m++ = *r++; } while ( r < t );
02830 t = position;
02831 r = t + *t;
02832 l2 = r[-1];
02833 r -= ( j = ABS(l2) );
02834 t++;
02835 if ( t < r ) do { *m++ = *t++; } while ( t < r );
02836 t = v;
02837 goto ComAct;
02838
02839 InsCall:
02840     MLOCK(ErrorMessageLock);
02841 InsCall2:
02842     MesCall("InsertTerm");
02843     MUNLOCK(ErrorMessageLock);
02844     SETERROR(-1)
02845 }
02846
02847 /*
02848 #] InsertTerm :
02849 #[ PasteFile :          WORD PasteFile(num,acc,pos,accf,renum,freeze,nexpr)
02850 */
02866 LONG PasteFile(PHEAD WORD number, WORD *accum, POSITION *position, WORD **accfill,
02867 RENUMBER renumber, WORD *freeze, WORD nexpr)
02868 {
02869     GETBIDENTITY
02870     WORD *r, l, *m, i;
02871     WORD *stop, *s1, *s2;
02872 /* POSITION AccPos; bug 12-apr-2008 JV */
02873     WORD InCompState;
02874     WORD *oldipointer;
02875     LONG retlength;
02876     stop = (WORD *)(((UBYTE *) (accum)) + 2*AM.MaxTer);
02877     *accum++ = number;
02878     while ( --number >= 0 ) accum += *accum;
02879     if ( freeze ) {
02880 /* AccPos = *position; bug 12-apr-2008 JV */
02881         oldipointer = AR.CompressPointer;
02882         do {
02883             AR.CompressPointer = oldipointer;
02884 /* if ( ( l = GetFromStore(accum,&AccPos,renumber,&InCompState,nexpr) ) < 0 ) bug 12-apr-2008
JV */
02885             if ( ( l = GetFromStore(accum,position,renumber,&InCompState,nexpr) ) < 0 )
02886                 goto PasErr;

```

```

02887         if ( !l ) { *accum = 0; return(0); }
02888         r = accum;
02889         m = r + *r;
02890         m -= ABS(m[-1]);
02891         r++;
02892         while ( r < m && *r != HAAKJE ) r += r[1];
02893         if ( r >= m ) {
02894             if ( *freeze != 4 ) l = -1;
02895         }
02896         else {
02897             /*
02898
02899                 The algorithm for accepting terms with a given (freeze)
02900                 representation outside brackets is rather crude. A refinement
02901                 would be to store the part outside the bracket and skip the
02902                 term when this part doesn't alter (and is unacceptable).
02903                 Once accepting one can keep accepting till the bracket alters
02904                 and then one may stop the generation. It is necessary to
02905                 set up a struct to remember the bracket and the progress
02906                 status.
02907
02908             */
02909             m = AT.WorkPointer;
02910             s2 = r;
02911             r = accum;
02912             *m++ = WORDDIF(s2,r) + 3;
02913             r++;
02914             while ( r < s2 ) *m++ = *r++;
02915             *m++ = 1; *m++ = 1; *m++ = 3;
02916             m = AT.WorkPointer;
02917             if ( Normalize(BHEAD AT.WorkPointer) ) goto PasErr;
02918             r = freeze;
02919             i = *m;
02920             while ( --i >= 0 && *m++ == *r++ ) {}
02921             if ( i > 0 ) {
02922                 l = -1;
02923             }
02924             else { /* Term to be accepted */
02925                 r = accum;
02926                 s1 = r + *r;
02927                 r++;
02928                 m = s2;
02929                 m += m[1];
02930                 do { *r++ = *m++; } while ( m < s1 );
02931                 *accum = 1 = WORDDIF(r,accum);
02932             }
02933         } while ( l < 0 );
02934         retlength = InCompState;
02935         /* retlength = DIFBASE(AccPos,*position) / sizeof(WORD); bug 12-apr-2008 JV */
02936     }
02937     else {
02938         if ( ( l = GetFromStore(accum,position,renumber,&InCompState,nexpr) ) < 0 ) {
02939             MLOCK(ErrorMessageLock);
02940             MesCall("PasteFile");
02941             MUNLOCK(ErrorMessageLock);
02942             SETERROR(-1)
02943         }
02944         if ( l == 0 ) { *accum = 0; return(0); }
02945         retlength = InCompState;
02946     }
02947     accum += l;
02948     if ( accum > stop ) {
02949         MLOCK(ErrorMessageLock);
02950         MesPrint("Buffer too small in PasteFile");
02951         MUNLOCK(ErrorMessageLock);
02952         SETERROR(-1)
02953     }
02954     *accum = 0;
02955     *accfill = accum;
02956     return(retlength);
02957 PasErr:
02958     MLOCK(ErrorMessageLock);
02959     MesCall("PasteFile");
02960     MUNLOCK(ErrorMessageLock);
02961     SETERROR(-1)
02962 }
02963 /*
02964     #] PasteFile :
02965     #[ PasteTerm :          WORD PasteTerm(number,accum,position,times,divby)
02966 */
02988 WORD *PasteTerm(PHEAD WORD number, WORD *accum, WORD *position, WORD times, WORD divby)
02989 {
02990     GETBIDENTITY
02991     WORD *t, *r, x, y, z;
02992     WORD *m, *u, ll, a[2];
02993     m = (WORD *)(((UBYTE *) (accum)) + AM.MaxTer);
02994     /* m = (WORD *)(((UBYTE *) (accum)) + 2*AM.MaxTer); */

```



```

02995     *accum++ = number;
02996     while ( --number >= 0 ) accum += *accum;
02997     if ( times == divby ) {
02998         t = position;
02999         r = t + *t;
03000         if ( t < r ) do { *accum++ = *t++; } while ( t < r );
03001     }
03002     else {
03003         u = accum;
03004         t = position;
03005         r = t + *t - 1;
03006         ll = *r;
03007         r -= ABS(*r) - 1;
03008         if ( t < r ) do { *accum++ = *t++; } while ( t < r );
03009         if ( divby > times ) { x = divby; y = times; }
03010         else { x = times; y = divby; }
03011         z = x*y;
03012         while ( z ) { x = y; y = z; z = x*y; }
03013         if ( y != 1 ) { divby /= y; times /= y; }
03014         a[1] = divby;
03015         a[0] = times;
03016         if ( MulRat(BHEAD (UWORD *)t, REDLENG(ll), (UWORD *)a, 1, (UWORD *)accum, &ll) ) {
03017             MLOCK(ErrorMessageLock);
03018             MesCall("PasteTerm");
03019             MUNLOCK(ErrorMessageLock);
03020             return(0);
03021         }
03022         x = ll;
03023         x *= 2;
03024         if ( x < 0 ) { accum -= x; *accum++ = x - 1; }
03025         else { accum += x; *accum++ = x + 1; }
03026         *u = WORDDIF(accum, u);
03027     }
03028     if ( accum >= m ) {
03029         MLOCK(ErrorMessageLock);
03030         MesPrint("Buffer too small in PasteTerm");
03031         MUNLOCK(ErrorMessageLock);
03032         return(0);
03033     }
03034     *accum = 0;
03035     return(accum);
03036 }
03037
03038 /*
03039     #] PasteTerm :
03040     #[ FiniTerm :          WORD FiniTerm(term, accum, termout, number)
03041 */
03053 int FiniTerm(PHEAD WORD *term, WORD *accum, WORD *termout, WORD number, WORD tepos)
03054 {
03055     GETBIDENTITY
03056     WORD *m, *t, *r, i, numacc, l2, ipp;
03057     WORD *u, *v, ll, *coef = AT.WorkPointer, *oldaccum;
03058     if ( ( AT.WorkPointer = coef + 2*AM.MaxTal ) > AT.WorkTop ) {
03059         MLOCK(ErrorMessageLock);
03060         MesWork();
03061         MUNLOCK(ErrorMessageLock);
03062         return(-1);
03063     }
03064     oldaccum = accum;
03065     t = term;
03066     m = t + *t - 1;
03067     ll = REDLENG(*m);
03068     i = ABS(*m) - 1;
03069     r = coef + i;
03070     do { *--r = *--m; } while ( --i > 0 ); /* Copies coefficient */
03071     if ( tepos > 0 ) {
03072         t = term + tepos;
03073         goto foundit;
03074     }
03075     t++;
03076     if ( t < m ) do {
03077         if ( ( (*t == SUBEXPRESSION && ( *(r=t+t[1]) != SUBEXPRESSION
03078             || r >= m || !r[3] ) ) || *t == EXPRESSION ) && t[2] == number && t[3] ) {
03079 foundit:;
03080             u = m;
03081             r = term;
03082             m = termout;
03083             if ( r < t ) do { *m++ = *r++; } while ( r < t );
03084             numacc = *accum++;
03085             if ( numacc >= 0 ) do {
03086                 if ( *t == EXPRESSION ) {
03087                     v = t + t[1];
03088                     r = t + SUBEXPSIZE;
03089                     while ( r < v ) {
03090                         if ( *r == WILDCARDS ) {
03091                             r += 2;
03092                             i = *--m;

```

```

03093             if ( ( l2 = WildFill(BHEAD m,accum,r) ) < 0 ) goto FiniCall;
03094             goto AllWild;
03095         }
03096         r += r[1];
03097     }
03098     goto NoWild;
03099 }
03100 else if ( t[1] > SUBEXPSIZE && t[SUBEXPSIZE] != FROMBRAC ) {
03101     i = *--m;
03102     if ( ( l2 = WildFill(BHEAD m,accum,t) ) < 0 ) goto FiniCall;
03103 AllWild:
03104     *m = i;
03105     m += l2-1;
03106     l2 = *m;
03107     m -= ABS(l2) - 1;
03108     r = m;
03109 }
03110 else {
03111     r = accum;
03112     v = r + *r - 1;
03113     l2 = *v;
03114     v -= ABS(l2) - 1;
03115     r++;
03116     if ( r < v ) do { *m++ = *r++; } while ( r < v );
03117 }
03118 if ( *r == 1 && r[1] == 1 && ABS(l2) == 3 ) {
03119     if ( l2 < 0 ) l1 = -l1;
03120 }
03121 else {
03122     l2 = REDLENG(l2);
03123     if ( l2 == 0 ) {
03124         t = oldaccum;
03125         numacc = *t++;
03126         AO.OutSkip = 3;
03127         FiniLine();
03128         while ( --numacc >= 0 ) {
03129             i = *t;
03130             while ( --i >= 0 ) {
03131                 TalToLine((UWORD) (*t++));
03132                 TokenToLine((UBYTE *) " ");
03133             }
03134             AO.OutSkip = 0;
03135             FiniLine();
03136             goto FiniCall;
03137         }
03138         if ( MulRat(BHEAD (UWORD *)coef,l1,(UWORD *)r,l2,(UWORD *)coef,&l1) ) goto
03139 FiniCall;
03140         if ( AN.ncmod != 0 && TakeModulus((UWORD
03141 *)coef,&l1,AC.cmod,AN.ncmod,UNPACK|AC.modmode) ) goto FiniCall;
03142     }
03143     accum += *accum;
03144     } while ( --numacc >= 0 );
03145     if ( *t == SUBEXPRESSION ) {
03146         while ( t+t[1] < u && t[t[1]] == DOLLAREXPR2 ) t += t[1];
03147     }
03148     t += t[1];
03149     if ( t < u ) do { *m++ = *t++; } while ( t < u );
03150     l2 = l1;
03151 /*
03152 Code to economize when taking x = (a+b)/2
03153 */
03154 r = termout+1;
03155 while ( r < m ) {
03156     if ( *r == SUBEXPRESSION ) {
03157         t = r + r[1];
03158         l1 = (WORD)(cbuf[r[4]].CanCommu[r[2]]);
03159         while ( t < m ) {
03160             if ( *t == SUBEXPRESSION &&
03161                 t[1] == r[1] && t[2] == r[2] && t[4] == r[4] ) {
03162                 i = t[1] - SUBEXPSIZE;
03163                 u = r + SUBEXPSIZE; v = t + SUBEXPSIZE;
03164                 while ( i > 0 ) {
03165                     if ( *v++ != *u++ ) break;
03166                     i--;
03167                 }
03168                 if ( i <= 0 ) {
03169                     u = r;
03170                     r[3] += t[3];
03171                     r = t + t[1];
03172                     while ( r < m ) *t++ = *r++;
03173                     m = t;
03174                     r = u;
03175                     goto Nexttr;
03176                 }
03177                 if ( l1 && cbuf[t[4]].CanCommu[t[2]] ) break;
03178                 while ( t+t[1] < m && t[t[1]] == DOLLAREXPR2 ) t += t[1];
03179             }
03180         }
03181     }
03182 }

```

```

03178             else if ( l1 ) {
03179                 if ( *t == SUBEXPRESSION && cbuf[t[4]].CanCommu[t[2]] )
03180                     break;
03181                 if ( *t >= FUNCTION+WILDOFFSET )
03182                     ipp = *t - WILDOFFSET;
03183                 else ipp = *t;
03184                 if ( *t >= FUNCTION
03185                     && functions[ipp-FUNCTION].commute && l1 ) break;
03186                 if ( *t == EXPRESSION ) break;
03187             }
03188             t += t[1];
03189         }
03190         r += r[1];
03191     }
03192     else r += r[1];
03193 Nextr;;
03194     }
03195
03196     i = ABS(l2);
03197     i *= 2;
03198     i++;
03199     l2 = ( l2 >= 0 ) ? i: -i;
03200     r = coef;
03201     while ( --i > 0 ) *m++ = *r++;
03202     *m++ = l2;
03203     *termout = WORDDIF(m,termout);
03204     AT.WorkPointer = coef;
03205     return(0);
03206 }
03207 t += t[1];
03208 } while ( t < m );
03209 AT.WorkPointer = coef;
03210 return(1);
03211
03212 FiniCall:
03213     MLOCK(ErrorMessageLock);
03214     MesCall("FiniTerm");
03215     MUNLOCK(ErrorMessageLock);
03216     SETERROR(-1)
03217 }
03218
03219 /*
03220     #] FiniTerm :
03221     #[ Generator :          WORD Generator(BHEAD term,level)
03222 */
03223
03224 static WORD zeroDollar[] = { 0, 0 };
03225 /*
03226 static LONG debugcounter = 0;
03227 */
03228
03252 int Generator(PHEAD WORD *term, WORD level)
03253 {
03254     GETBIDENTITY
03255     WORD replac, *accum, *termout, *t, i, j, tepos, applyflag = 0, *StartBuf;
03256     WORD *a, power, powerl, DumNow = AR.CurDum, oldtoprhs, oldatoprhs, extractbuff;
03257     int ret;
03258     int *RepSto = AN.RepPoint, iscopy = 0;
03259     CBUF *C = cbuf+AM.rbufnum, *CC = cbuf + AT.ebufnum, *CCC = cbuf + AT.aebufnum;
03260     LONG posisub, oldcpointer, oldacpointer;
03261     DOLLARS d = 0;
03262     WORD numfac[5], idfunctionflag;
03263 #ifdef WITHPTHREADS
03264     int nummodopt, dtype = -1, id;
03265 #endif
03266     oldtoprhs = CC->numrhs;
03267     oldcpointer = CC->Pointer - CC->Buffer;
03268     oldatoprhs = CCC->numrhs;
03269     oldacpointer = CCC->Pointer - CCC->Buffer;
03270 ReStart:
03271     if ( ( replac = TestSub(BHEAD term,level) ) == 0 ) {
03272         if ( applyflag ) { TableReset(); applyflag = 0; }
03273     /*
03274         if ( AN.PolyNormFlag > 1 ) {
03275             if ( PolyFunMul(BHEAD term) < 0 ) goto GenCall;
03276             AN.PolyNormFlag = 0;
03277             if ( !*term ) goto Return0;
03278         }
03279     */
03280 Renormalize:
03281     AN.PolyNormFlag = 0;
03282     AN.idfunctionflag = 0;
03283     if ( ( ret = Normalize(BHEAD term) ) != 0 ) {
03284         if ( ret > 0 ) {
03285             if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
03286             goto ReStart;
03287         }

```

```

03288         goto GenCall;
03289     }
03290     idfunctionflag = AN.idfunctionflag;
03291     if ( !*term ) { AN.PolyNormFlag = 0; goto Return0; }
03292
03293     if ( AN.PolyNormFlag ) {
03294         if ( AN.PolyFunTodo == 0 ) {
03295             if ( PolyFunMul(BHEAD term) < 0 ) goto GenCall;
03296             if ( !*term ) { AN.PolyNormFlag = 0; goto Return0; }
03297         }
03298         else {
03299             WORD oldPolyFunExp = AR.PolyFunExp;
03300             AR.PolyFunExp = 0;
03301             if ( PolyFunMul(BHEAD term) < 0 ) goto GenCall;
03302             AT.WorkPointer = term+*term;
03303             AR.PolyFunExp = oldPolyFunExp;
03304             if ( !*term ) { AN.PolyNormFlag = 0; goto Return0; }
03305             if ( Normalize(BHEAD term) < 0 ) goto GenCall;
03306             if ( !*term ) { AN.PolyNormFlag = 0; goto Return0; }
03307             AT.WorkPointer = term+*term;
03308             if ( AN.PolyNormFlag ) {
03309                 if ( PolyFunMul(BHEAD term) < 0 ) goto GenCall;
03310                 if ( !*term ) { AN.PolyNormFlag = 0; goto Return0; }
03311                 AT.WorkPointer = term+*term;
03312             }
03313             AN.PolyFunTodo = 0;
03314         }
03315     }
03316     if ( idfunctionflag > 0 ) {
03317         if ( TakeIDfunction(BHEAD term) ) {
03318             AT.WorkPointer = term + *term;
03319             goto ReStart;
03320         }
03321     }
03322     if ( AT.WorkPointer < (WORD *)(((UBYTE *) (term)) + AM.MaxTer) )
03323         AT.WorkPointer = (WORD *)(((UBYTE *) (term)) + AM.MaxTer);
03324     do {
03325         SkipCount: level++;
03326         if ( level > AR.Cnumlhs ) {
03327             if ( AR.DeferFlag && AR.sLevel <= 0 ) {
03328                 #ifdef WITHMPI
03329                     if ( PF.me != MASTER && AC.mparallelfld == PARALLELFLAG && PF.exprtodo < 0 ) {
03330                         if ( PF_Deferred(term,level) ) goto GenCall;
03331                     }
03332                     else
03333                         #endif
03334                     {
03335                         if ( Deferred(BHEAD term,level) ) goto GenCall;
03336                     }
03337                     goto Return0;
03338                 }
03339                 if ( AN.ncmod != 0 ) {
03340                     if ( Modulus(term) ) goto GenCall;
03341                     if ( !*term ) goto Return0;
03342                 }
03343                 if ( AR.CurDum > AM.IndDum && AR.sLevel <= 0 ) {
03344                     WORD olddummies = AN.IndDum;
03345                     AN.IndDum = AM.IndDum;
03346                     ReNumber(BHEAD term);
03347                     Normalize(BHEAD term);
03348                     AN.IndDum = olddummies;
03349                     if ( !*term ) goto Return0;
03350                     olddummies = DetCurDum(BHEAD term);
03351                     if ( olddummies > AR.MaxDum ) AR.MaxDum = olddummies;
03352                 }
03353                 if ( AR.PolyFun > 0 && ( AR.sLevel <= 0 || AN.FunSorts[AR.sLevel]->PolyFlag > 0 ) ) {
03354                     if ( PrepPoly(BHEAD term,0) != 0 ) goto Return0;
03355                 }
03356                 else if ( AR.PolyFun > 0 ) {
03357                     if ( PrepPoly(BHEAD term,1) != 0 ) goto Return0;
03358                 }
03359                 if ( AR.sLevel <= 0 && AR.BracketOn ) {
03360                     if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
03361                     termout = AT.WorkPointer;
03362                     if ( AT.WorkPointer + *term + 3 > AT.WorkTop ) goto OverWork;
03363                     if ( PutBracket(BHEAD term) ) return(-1);
03364                     AN.RepPoint = RepSto;
03365                     *AT.WorkPointer = 0;
03366                     ret = StoreTerm(BHEAD termout);
03367                     AT.WorkPointer = termout;
03368                     CC->numrhs = oldtoprhs;
03369                     CC->Pointer = CC->Buffer + oldcpointer;
03370                     CCC->numrhs = oldatoprhs;
03371                     CCC->Pointer = CCC->Buffer + oldacpointer;
03372                     return(ret);
03373                 }
03374             }
03375         }
03376     }
03377     else {

```

```

03375         if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
03376         if ( AT.WorkPointer >= AT.WorkTop ) goto OverWork;
03377         *AT.WorkPointer = 0;
03378         AN.RepPoint = RepSto;
03379         ret = StoreTerm(BHEAD term);
03380         CC->numrhs = oldtoprhs;
03381         CC->Pointer = CC->Buffer + oldcpointer;
03382         CCC->numrhs = oldatoprhs;
03383         CCC->Pointer = CCC->Buffer + oldacpointer;
03384         return(ret);
03385     }
03386 }
03387 i = C->lhs[level][0];
03388 if ( i >= TYPECOUNT ) {
03389 /*
03390     #[ Special action :
03391 */
03392     switch ( i ) {
03393     case TYPECOUNT:
03394         if ( CountDo(term,C->lhs[level]) < C->lhs[level][2] ) {
03395             AT.WorkPointer = term + *term;
03396             goto Return0;
03397         }
03398         break;
03399     case TYPEMULT:
03400         if ( MultDo(BHEAD term,C->lhs[level]) ) goto GenCall;
03401         goto ReStart;
03402     case TYPEGOTO:
03403         level = AC.Labels[C->lhs[level][2]];
03404         break;
03405     case TYPEDISCARD:
03406         AT.WorkPointer = term + *term;
03407         goto Return0;
03408     case TYPEIF:
03409 #ifdef WITHPTHREADS
03410     {
03411 /*
03412         We may be writing in the space here when wildcards
03413         are involved in a match(). Hence we have to make
03414         a private copy here!!!!
03415 */
03416         WORD ic, jc, *ifcode, *jfcodes;
03417         jfcodes = C->lhs[level]; jc = jfcodes[1];
03418         ifcode = AT.WorkPointer; AT.WorkPointer += jc;
03419         for ( ic = 0; ic < jc; ic++ ) ifcode[ic] = jfcodes[ic];
03420         while ( !DoIfStatement(BHEAD ifcode,term) ) {
03421             level = C->lhs[level][2];
03422             if ( C->lhs[level][0] != TYPEELIF ) break;
03423         }
03424         AT.WorkPointer = ifcode;
03425     }
03426 #else
03427     while ( !DoIfStatement(BHEAD C->lhs[level],term) ) {
03428         level = C->lhs[level][2];
03429         if ( C->lhs[level][0] != TYPEELIF ) break;
03430     }
03431 #endif
03432     break;
03433     case TYPEELIF:
03434     do {
03435         level = C->lhs[level][2];
03436     } while ( C->lhs[level][0] == TYPEELIF );
03437     break;
03438     case TYPEELSE:
03439     case TYPEENDIF:
03440         level = C->lhs[level][2];
03441         break;
03442     case TYPESUMFIX:
03443     {
03444         WORD *cp = AR.CompressPointer, *op = AR.CompressPointer;
03445         WORD *tlhs = C->lhs[level] + 3, *mlhs, *jlhs;
03446         WORD theindex = C->lhs[level][2];
03447         if ( theindex < 0 ) { /* $-variable */
03448 #ifdef WITHPTHREADS
03449             int ddtype = -1;
03450             theindex = -theindex;
03451             d = Dollars + theindex;
03452             if ( AS.MultiThreaded ) {
03453                 for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
03454                     if ( theindex == ModOptdollars[nummodopt].number ) break;
03455                 }
03456                 if ( nummodopt < NumModOptdollars ) {
03457                     ddtype = ModOptdollars[nummodopt].type;
03458                     if ( ddtype == MODLOCAL ) {
03459                         d = ModOptdollars[nummodopt].dstruct+AT.identity;
03460                     }
03461                     else {

```

```

03462             LOCK(d->pthreadlockread);
03463         }
03464     }
03465 }
03466 #else
03467     theindex = -theindex;
03468     d = Dollars + theindex;
03469 #endif
03470
03471     if ( d->type != DOLINDEX
03472         || d->index < AM.OffsetIndex
03473         || d->index >= AM.OffsetIndex + WILDOffset ) {
03474         MLOCK(ErrorMessageLock);
03475         MesPrint("$%s should have been an index"
03476             ,AC.dollarnames->namebuffer+d->name);
03477         AN.currentTerm = term;
03478         MesPrint("Current term: %t");
03479         AN.listinprint = printscratch;
03480         printscratch[0] = DOLLAREXPRESSION;
03481         printscratch[1] = theindex;
03482         MesPrint("$%s = %s"
03483             ,AC.dollarnames->namebuffer+d->name);
03484         MUNLOCK(ErrorMessageLock);
03485 #ifdef WITHPTHREADS
03486         if ( ddtype > 0 && ddtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
03487 #endif
03488         goto GenCall;
03489     }
03490     theindex = d->index;
03491 #ifdef WITHPTHREADS
03492     if ( ddtype > 0 && ddtype != MODLOCAL ) { UNLOCK(d->pthreadlockread); }
03493 #endif
03494 }
03495 cp[1] = SUBEXPSIZE+4;
03496 cp += SUBEXPSIZE;
03497 *cp++ = INDTOIND;
03498 *cp++ = 4;
03499 *cp++ = theindex;
03500 i = C->lhs[level][1] - 3;
03501 cp++;
03502 AR.CompressPointer = cp;
03503 while ( --i >= 0 ) {
03504     cp[-1] = *tlhs++;
03505     termout = AT.WorkPointer;
03506     if ( ( jlhs = WildFill(BHEAD termout,term,op) < 0 )
03507         goto GenCall;
03508     m = term;
03509     jlhs = *m;
03510     while ( --jlhs >= 0 ) {
03511         if ( *m++ != *termout++ ) break;
03512     }
03513     if ( jlhs >= 0 ) {
03514         termout = AT.WorkPointer;
03515         AT.WorkPointer = termout + *termout;
03516         if ( Generator(BHEAD termout,level) ) goto GenCall;
03517         AT.WorkPointer = termout;
03518     }
03519     else {
03520         AR.CompressPointer = op;
03521         goto SkipCount;
03522     }
03523 }
03524 AR.CompressPointer = op;
03525 goto CommonEnd;
03526 }
03527 case TYPESUM:
03528 {
03529     WORD *wp, *cp = AR.CompressPointer, *op = AR.CompressPointer;
03530     WORD theindex;
03531     WORD *ow;
03532     /*
03533     At this point it is safest to determine CurDum
03534     */
03535     AR.CurDum = DetCurDum(BHEAD term);
03536     i = C->lhs[level][1]-2;
03537     wp = C->lhs[level] + 2;
03538     cp[1] = SUBEXPSIZE+4*i;
03539     cp += SUBEXPSIZE;
03540     while ( --i >= 0 ) {
03541         theindex = *wp++;
03542         if ( theindex < 0 ) { /* $-variable */
03543             int ddtype = -1;
03544             theindex = -theindex;
03545             d = Dollars + theindex;
03546             if ( AS.MultiThreaded ) {
03547                 for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {

```

```

03549             if ( theindex == ModOptdollars[nummodopt].number ) break;
03550         }
03551         if ( nummodopt < NumModOptdollars ) {
03552             ddtype = ModOptdollars[nummodopt].type;
03553             if ( ddtype == MODLOCAL ) {
03554                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
03555             }
03556             else {
03557                 LOCK(d->pthreadslackread);
03558             }
03559         }
03560     }
03561 #else
03562     theindex = -theindex;
03563     d = Dollars + theindex;
03564 #endif
03565     if ( d->type != DOLINDEX
03566         || d->index < AM.OffsetIndex
03567         || d->index >= AM.OffsetIndex + WILDOFFSET ) {
03568         MLOCK(ErrorMessageLock);
03569         MesPrint("%s should have been an index"
03570             ,AC.dollarnames->namebuffer+d->name);
03571         AN.currentTerm = term;
03572         MesPrint("Current term: %t");
03573         AN.listinprint = printscratch;
03574         printscratch[0] = DOLLAREXPRESSION;
03575         printscratch[1] = theindex;
03576         MesPrint("%$s = % $"
03577             ,AC.dollarnames->namebuffer+d->name);
03578         MUNLOCK(ErrorMessageLock);
03579 #ifdef WITHPTHREADS
03580         if ( ddtype > 0 && ddtype != MODLOCAL ) {
03581             UNLOCK(d->pthreadslackread); }
03582 #endif
03583         goto GenCall;
03584     }
03585     theindex = d->index;
03586 #ifdef WITHPTHREADS
03587     if ( ddtype > 0 && ddtype != MODLOCAL ) { UNLOCK(d->pthreadslackread);
03588     }
03589     *cp++ = INDTOIND;
03590     *cp++ = 4;
03591     *cp++ = theindex;
03592     *cp++ = ++AR.CurDum;
03593 }
03594 ow = AT.WorkPointer;
03595 AR.CompressPointer = cp;
03596 if ( WildFill(BHEAD ow,term,op) < 0 ) goto GenCall;
03597 AR.CompressPointer = op;
03598 i = ow[0];
03599 WORD term_changed = 0;
03600 for ( j = 0; j < i; j++ ) {
03601     if ( term[j] != ow[j] ) term_changed = 1;
03602     term[j] = ow[j];
03603 }
03604 // If the term was modified by WildFill, set RepCount.
03605 if ( term_changed ) *AN.RepPoint = 1;
03606 AT.WorkPointer = ow;
03607 // Most other calls to ReNumber reset AN.IndDum to AM.IndDum first. Here it is
03608 // not done, but doing it is one way to fix Issue #710. The fix that is
    actually
03609 // implemented is to change a comparison in FunLevel and then a reset of
03610 // AN.IndDum appears unnecessary here. But maybe one day this comment is
    useful.
03611     ReNumber(BHEAD term);
03612     goto Renormalize;
03613 }
03614 case TYPECHISHOLM:
03615     if ( Chisholm(BHEAD term,level) ) goto GenCall;
03616 CommonEnd:
03617     AT.WorkPointer = term + *term;
03618     goto Return0;
03619 case TYPEARG:
03620     if ( ( i = execarg(BHEAD term,level) ) < 0 ) goto GenCall;
03621     level = C->lhs[level][2];
03622     if ( i > 0 ) goto ReStart;
03623     break;
03624 case TYPENORM:
03625 case TYPENORM2:
03626 case TYPENORM3:
03627 case TYPENORM4:
03628 case TYPESPLITARG:
03629 case TYPESPLITARG2:
03630 case TYPESPLITFIRSTARG:
03631 case TYPESPLITLASTARG:

```

```

03632     case TYPEARGTOEXTRASymbol:
03633         if ( execarg(BHEAD term,level) < 0 ) goto GenCall;
03634         level = C->lhs[level][2];
03635         break;
03636     case TYPEFACTARG:
03637     case TYPEFACTARG2:
03638         { WORD jjj;
03639         if ( ( jjj = execarg(BHEAD term,level) ) < 0 ) goto GenCall;
03640         if ( jjj > 0 ) goto ReStart;
03641         level = C->lhs[level][2];
03642         break; }
03643     case TYPEEXIT:
03644         if ( C->lhs[level][2] > 0 ) {
03645             MLOCK(ErrorMessageLock);
03646             MesPrint("%s",C->lhs[level]+3);
03647             MUNLOCK(ErrorMessageLock);
03648         }
03649         Terminate(-1);
03650         goto GenCall;
03651     case TYPESETEXIT:
03652         AM.exitflag = 1; /* no danger of race conditions */
03653         break;
03654     case TYPEPRINT:
03655         AN.currentTerm = term;
03656         AN.numlistinprint = (C->lhs[level][1] - C->lhs[level][4] - 5)/2;
03657         AN.listinprint = C->lhs[level]+5+C->lhs[level][4];
03658         MLOCK(ErrorMessageLock);
03659         AO.ErrorBlock = 1;
03660         MesPrint((char *) (C->lhs[level]+5));
03661         AO.ErrorBlock = 0;
03662         MUNLOCK(ErrorMessageLock);
03663         break;
03664     case TYPEFPRI:
03665         {
03666             int oldFOflag;
03667             WORD oldPrintType, oldLogHandle = AC.LogHandle;
03668             AC.LogHandle = C->lhs[level][2];
03669             MLOCK(ErrorMessageLock);
03670             oldFOflag = AM.FileOnlyFlag;
03671             oldPrintType = AO.PrintType;
03672             if ( AC.LogHandle >= 0 ) {
03673                 AM.FileOnlyFlag = 1;
03674                 AO.PrintType |= PRINTLFILE;
03675             }
03676             AO.PrintType |= C->lhs[level][3];
03677             AN.currentTerm = term;
03678             AN.numlistinprint = (C->lhs[level][1] - C->lhs[level][4] - 5)/2;
03679             AN.listinprint = C->lhs[level]+5+C->lhs[level][4];
03680             MesPrint((char *) (C->lhs[level]+5));
03681             AO.PrintType = oldPrintType;
03682             AM.FileOnlyFlag = oldFOflag;
03683             MUNLOCK(ErrorMessageLock);
03684             AC.LogHandle = oldLogHandle;
03685         }
03686         break;
03687     case TYPEDEFPRE:
03688         j = C->lhs[level][2];
03689 #ifndef WITHMPI
03690     {
03691         /*
03692         * Regardless of parallel/nonparallel switch, we need to set
03693         * AC.inputnumbers[i], which indicates that the corresponding
03694         * preprocessor variable is redefined and so we need to
03695         * send/broadcast it.
03696         */
03697         int ii;
03698         for ( ii = 0; ii < AC.numpfirstnum; ii++ ) {
03699             if ( AC.pfirstnum[ii] == j ) break;
03700         }
03701         AC.inputnumbers[ii] = AN.ninterms;
03702     }
03703 #endif
03704 #ifndef WITHPTHREADS
03705     if ( AS.MultiThreaded ) {
03706         int ii;
03707         for ( ii = 0; ii < AC.numpfirstnum; ii++ ) {
03708             if ( AC.pfirstnum[ii] == j ) break;
03709         }
03710         if ( AN.inputnumber < AC.inputnumbers[ii] ) break;
03711         LOCK(AP.PreVarLock);
03712         if ( AN.inputnumber >= AC.inputnumbers[ii] ) {
03713             a = C->lhs[level]+4;
03714             if ( a[a[-1]] == 0 )
03715                 PutPreVar(PreVar[j].name, (UBYTE *) (a), 0, 1);
03716             else
03717                 PutPreVar(PreVar[j].name, (UBYTE *) (a)
03718                     , (UBYTE *) (a+a[-1]+1), 1);

```



```

03719 /*
03720                                     PutPreVar(PreVar[j].name, (UBYTE *) (C->lhs[level]+4), 0, 1);
03721 */
03722                                     AC.inputnumbers[ii] = AN.inputnumber;
03723                                     }
03724                                     UNLOCK(AP.PreVarLock);
03725                                     }
03726                                     else
03727 #endif
03728                                     {
03729                                         a = C->lhs[level]+4;
03730                                         LOCK(AP.PreVarLock);
03731                                         if ( a[a[-1]] == 0 )
03732                                             PutPreVar(PreVar[j].name, (UBYTE *) (a), 0, 1);
03733                                         else
03734                                             PutPreVar(PreVar[j].name, (UBYTE *) (a)
03735                                                 , (UBYTE *) (a+a[-1]+1), 1);
03736                                         UNLOCK(AP.PreVarLock);
03737                                     }
03738                                     break;
03739 case TYPERENNUMBER:
03740     AT.WorkPointer = term + *term;
03741     if ( FullRenumber(BHEAD term, C->lhs[level][2]) ) goto GenCall;
03742     AT.WorkPointer = term + *term;
03743     if ( *term == 0 ) goto Return0;
03744     break;
03745 case TYPETRY:
03746     if ( TryDo(BHEAD term, C->lhs[level], level) ) goto GenCall;
03747     AT.WorkPointer = term + *term;
03748     goto Return0;
03749 case TYPEASSIGN:
03750     { WORD onc = AR.NoCompress, oldEside = AR.Eside;
03751       WORD oldrepeat = *AN.RepPoint;
03752     }
03753     /*
03754     */
03755     AR.Eside = RHSIDE;
03756     AR.NoCompress = 1;
03757     AN.cTerm = AN.currentTerm = term;
03758     AT.WorkPointer = term + *term;
03759     *AT.WorkPointer++ = 0;
03760     if ( AssignDollar(BHEAD term, level) ) goto GenCall;
03761     AT.WorkPointer = term + *term;
03762     AN.cTerm = 0;
03763     *AN.RepPoint = oldrepeat;
03764     AR.NoCompress = onc;
03765     AR.Eside = oldEside;
03766     break;
03767 }
03768 case TYPEFINDLOOP:
03769     if ( Lus(term, C->lhs[level][3], C->lhs[level][4],
03770         C->lhs[level][5], C->lhs[level][6], C->lhs[level][2]) ) {
03771         AT.WorkPointer = term + *term;
03772         goto Renormalize;
03773     }
03774     break;
03775 case TYPEINSIDE:
03776     if ( InsideDollar(BHEAD C->lhs[level], level) < 0 ) goto GenCall;
03777     level = C->lhs[level][2];
03778     break;
03779 case TYPETERM:
03780     ret = execterm(BHEAD term, level);
03781     AN.RepPoint = RepSto;
03782     AR.CurDum = DumNow;
03783     CC->numrhs = oldtoprhs;
03784     CC->Pointer = CC->Buffer + oldcpointer;
03785     CCC->numrhs = oldatoprhs;
03786     CCC->Pointer = CCC->Buffer + oldacpointer;
03787     return(ret);
03788 case TYPEDETCURDUM:
03789     AT.WorkPointer = term + *term;
03790     AR.CurDum = DetCurDum(BHEAD term);
03791     break;
03792 case TYPEINEXPRESSION:
03793     {WORD *ll = C->lhs[level];
03794       int numexprs = (int)(ll[1]-3);
03795       ll += 3;
03796       while ( numexprs-- >= 0 ) {
03797         if ( *ll == AR.CurExpr ) break;
03798         ll++;
03799       }
03800       if ( numexprs < 0 ) level = C->lhs[level][2];
03801     }
03802     break;
03803 case TYPEMERGE:
03804     AT.WorkPointer = term + *term;
03805     if ( DoShuffle(term, level, C->lhs[level][2], C->lhs[level][3]) )

```

```

03806         goto GenCall;
03807         AT.WorkPointer = term + *term;
03808         goto Return0;
03809     case TYPESTUFFLE:
03810         AT.WorkPointer = term + *term;
03811         if ( DoStuffle(term,level,C->lhs[level][2],C->lhs[level][3]) )
03812             goto GenCall;
03813         AT.WorkPointer = term + *term;
03814         goto Return0;
03815     case TYPETESTUSE:
03816         AT.WorkPointer = term + *term;
03817         if ( TestUse(term,level) ) goto GenCall;
03818         AT.WorkPointer = term + *term;
03819         break;
03820     case TYPEAPPLY:
03821         AT.WorkPointer = term + *term;
03822         if ( ApplyExec(term,C->lhs[level][2],level) < C->lhs[level][2] ) {
03823             AT.WorkPointer = term + *term;
03824             *AN.RepPoint = 1;
03825             goto ReStart;
03826         }
03827         AT.WorkPointer = term + *term;
03828         break;
03829 /*
03830     case TYPEAPPLYRESET:
03831         AT.WorkPointer = term + *term;
03832         if ( ApplyReset(level) ) goto GenCall;
03833         AT.WorkPointer = term + *term;
03834         break;
03835 */
03836     case TYPECHAININ:
03837         { int lter = *term;
03838           AT.WorkPointer = term + *term;
03839           if ( ChainIn(BHEAD term,C->lhs[level][2]) ) goto GenCall;
03840           AT.WorkPointer = term + *term;
03841           /* Symmetry properties might mean the term has vanished */
03842           if ( *term == 0 ) goto Return0;
03843           if ( *term != lter ) *AN.RepPoint = 1;
03844         }
03845         break;
03846     case TYPECHAINOUT:
03847         { int lter = *term;
03848           AT.WorkPointer = term + *term;
03849           if ( ChainOut(BHEAD term,C->lhs[level][2]) ) goto GenCall;
03850           AT.WorkPointer = term + *term;
03851           if ( *term != lter ) *AN.RepPoint = 1;
03852         }
03853         break;
03854     case TYPEFACTOR:
03855         AT.WorkPointer = term + *term;
03856         if ( DollarFactorize(BHEAD C->lhs[level][2]) ) goto GenCall;
03857         AT.WorkPointer = term + *term;
03858         break;
03859     case TYPEARGIMPLODE:
03860         AT.WorkPointer = term + *term;
03861         if ( ArgumentImplode(BHEAD term,C->lhs[level]) ) goto GenCall;
03862         AT.WorkPointer = term + *term;
03863         break;
03864     case TYPEARGEXPLODE:
03865         AT.WorkPointer = term + *term;
03866         if ( ArgumentExplode(BHEAD term,C->lhs[level]) ) goto GenCall;
03867         AT.WorkPointer = term + *term;
03868         break;
03869     case TYPEDENOMINATORS:
03870         if ( DenToFunction(term,C->lhs[level][2]) ) goto ReStart;
03871         break;
03872     case TYPEDROPCOEFFICIENT:
03873         DropCoefficient(BHEAD term);
03874         break;
03875     case TYPETRANSFORM:
03876         AT.WorkPointer = term + *term;
03877         if ( RunTransform(BHEAD term,C->lhs[level]+2) ) goto GenCall;
03878         AT.WorkPointer = term + *term;
03879         if ( *term == 0 ) goto Return0;
03880         goto ReStart;
03881     case TYPETOPOLYNOMIAL:
03882         AT.WorkPointer = term + *term;
03883         termout = AT.WorkPointer;
03884         if ( ConvertToPoly(BHEAD term,termout,C->lhs[level],0) < 0 ) goto GenCall;
03885         if ( *termout == 0 ) goto Return0;
03886         i = termout[0]; t = term; NCOPY(t,termout,i);
03887         AT.WorkPointer = term + *term;
03888         break;
03889     case TYPEFROMPOLYNOMIAL:
03890         AT.WorkPointer = term + *term;
03891         termout = AT.WorkPointer;
03892         if ( ConvertFromPoly(BHEAD term,termout,0,numsymbol,0,0) < 0 ) goto GenCall;

```

```

03893         if ( *term == 0 ) goto Return0;
03894         i = termout[0]; t = term; NCOPY(t,termout,i);
03895         AT.WorkPointer = term + *term;
03896         goto ReStart;
03897     case TYPEDOLOOP:
03898         level = TestDoLoop(BHEAD C->lhs[level],level);
03899         if ( level < 0 ) goto GenCall;
03900         break;
03901     case TYPEENDDOLOOP:
03902         level = TestEndDoLoop(BHEAD C->lhs[C->lhs[level][2]],C->lhs[level][2]);
03903         if ( level < 0 ) goto GenCall;
03904         break;
03905     case TYPEDROPSYMBOLS:
03906         DropSymbols(BHEAD term);
03907         break;
03908     case TYPEPUTINSIDE:
03909         AT.WorkPointer = term + *term;
03910         if ( PutInside(BHEAD term,C->lhs[level]) < 0 ) goto GenCall;
03911         AT.WorkPointer = term + *term;
03912         /*
03913          * We need to call Generator() to convert slow notation to
03914          * fast notation, which fixes Issue #30.
03915          */
03916         if ( Generator(BHEAD term,level) < 0 ) goto GenCall;
03917         goto Return0;
03918     case TYPEOTOSPECTATOR:
03919         if ( PutInSpectator(term,C->lhs[level][2]) < 0 ) goto GenCall;
03920         goto Return0;
03921     case TYPECANONICALIZE:
03922         AT.WorkPointer = term + *term;
03923         if ( DoCanonicalize(BHEAD term,C->lhs[level]) ) goto GenCall;
03924         AT.WorkPointer = term + *term;
03925         if ( *term == 0 ) goto Return0;
03926         break;
03927     case TYPESWITCH:
03928         AT.WorkPointer = term + *term;
03929         if ( DoSwitch(BHEAD term,C->lhs[level]) ) goto GenCall;
03930         goto Return0;
03931     case TYPEENDSWITCH:
03932         AT.WorkPointer = term + *term;
03933         if ( DoEndSwitch(BHEAD term,C->lhs[level]) ) goto GenCall;
03934         goto Return0;
03935     case TYPESETUSERFLAG:
03936         Expressions[AR.CurExpr].uflags |= 1 « (C->lhs[level][2]);
03937         break;
03938     case TYPECLEARUSERFLAG:
03939         Expressions[AR.CurExpr].uflags &= ~(1 « (C->lhs[level][2]));
03940         break;
03941     case TYPEALLLOOPS:
03942         AT.WorkPointer = term + *term;
03943         if ( AllLoops(BHEAD term,level) ) goto GenCall;
03944         goto Return0;
03945     case TYPEALLPATHS:
03946         AT.WorkPointer = term + *term;
03947         if ( AllPaths(BHEAD term,level) ) goto GenCall;
03948         goto Return0;
03949 #ifdef WITHFLOAT
03950     case TYPEEVALUATE:
03951         AT.WorkPointer = term + *term;
03952         if ( C->lhs[level][2] == MZV
03953             || C->lhs[level][2] == EULER
03954             || C->lhs[level][2] == MZVHALF
03955             || C->lhs[level][2] == ALLMZVFUNCTIONS
03956         ) {
03957             if ( EvaluateEuler(BHEAD term,level,C->lhs[level][2]) ) goto GenCall;
03958         }
03959         else {
03960             if ( EvaluateFun(BHEAD term,level,C->lhs[level]) ) goto GenCall;
03961         }
03962         /*
03963         else if ( C->lhs[level][2] == SQRTFUNCTION ) {
03964             if ( EvaluateSqrt(BHEAD term,level,C->lhs[level][2]) ) goto GenCall;
03965         }
03966         else {
03967             MLOCK(ErrorMessageLock);
03968             MesPrint("Illegal function %d in evaluate statement.",C->lhs[level][2]);
03969             MUNLOCK(ErrorMessageLock);
03970             goto GenCall;
03971         }
03972         */
03973         goto Return0;
03974     case TYPETOFLOAT:
03975         AT.WorkPointer = term + *term;
03976         if ( ToFloat(BHEAD term,level) ) goto GenCall;
03977         goto Return0;
03978     case TYPETORAT:
03979         AT.WorkPointer = term + *term;

```

```

03980             if ( ToRat(BHEAD term,level) ) goto GenCall;
03981             goto Return0;
03982         case TYPESTRICTROUNDING:
03983             AT.WorkPointer = term + *term;
03984             if ( StrictRounding(BHEAD term,level,C->lhs[level][2],C->lhs[level][3]) ) goto
GenCall;
03985             goto Return0;
03986         case TYPECHOP:
03987             AT.WorkPointer = term + *term;
03988             if ( Chop(BHEAD term,level) ) goto GenCall;
03989             goto Return0;
03990     #endif
03991     }
03992     goto SkipCount;
03993 /*
03994     #] Special action :
03995 */
03996     }
03997     } while ( ( i = TestMatch(BHEAD term,&level) ) == 0 );
03998     if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
03999     if ( i > 0 ) replac = TestSub(BHEAD term,level);
04000     else replac = i;
04001     if ( replac >= 0 || AT.TMout[1] != SYMMETRIZE ) {
04002         *AN.RepPoint = 1;
04003         AR.expchanged = 1;
04004     }
04005     if ( replac < 0 ) { /* Terms come from automatic generation */
04006     AutoGen:    i = *AT.TMout;
04007                 t = termout = AT.WorkPointer;
04008                 if ( ( AT.WorkPointer += i ) > AT.WorkTop ) goto OverWork;
04009                 accum = AT.TMout;
04010                 while ( --i >= 0 ) *t++ = *accum++;
04011                 if ( (*FG.Operation[termout[1]])(BHEAD term,termout,replac,level) ) goto GenCall;
04012                 AT.WorkPointer = termout;
04013                 goto Return0;
04014     }
04015     }
04016     if ( applyflag ) { TableReset(); applyflag = 0; }
04017
04018     if ( AN.TeInFun ) { /* Match in function argument */
04019         if ( AN.TeInFun < 0 && !AN.TeSuOut ) {
04020
04021             if ( AR.TePos >= 0 ) goto AutoGen;
04022             switch ( AN.TeInFun ) {
04023             case -1:
04024                 if ( DoDistrib(BHEAD term,level) ) goto GenCall;
04025                 break;
04026             case -2:
04027                 if ( DoDelta3(BHEAD term,level) ) goto GenCall;
04028                 break;
04029             case -3:
04030                 if ( DoTableExpansion(term,level) ) goto GenCall;
04031                 break;
04032             case -4:
04033                 if ( FactorIn(BHEAD term,level) ) goto GenCall;
04034                 break;
04035             case -5:
04036                 if ( FactorInExpr(BHEAD term,level) ) goto GenCall;
04037                 break;
04038             case -6:
04039                 if ( TermsInBracket(BHEAD term,level) < 0 ) goto GenCall;
04040                 break;
04041             case -7:
04042                 if ( ExtraSymFun(BHEAD term,level) < 0 ) goto GenCall;
04043                 break;
04044             case -8:
04045                 if ( GCDfunction(BHEAD term,level) < 0 ) goto GenCall;
04046                 break;
04047             case -9:
04048                 if ( DIVfunction(BHEAD term,level,0) < 0 ) goto GenCall;
04049                 break;
04050             case -10:
04051                 if ( DIVfunction(BHEAD term,level,1) < 0 ) goto GenCall;
04052                 break;
04053             case -11:
04054                 if ( DIVfunction(BHEAD term,level,2) < 0 ) goto GenCall;
04055                 break;
04056             case -12:
04057                 if ( DoPermutations(BHEAD term,level) ) goto GenCall;
04058                 break;
04059             case -13:
04060                 if ( DoPartitions(BHEAD term,level) ) goto GenCall;
04061                 break;
04062             case -14:
04063                 if ( DIVfunction(BHEAD term,level,3) < 0 ) goto GenCall;
04064                 break;
04065             case -15:

```

```

04066             if ( GenDiagrams(BHEAD term,level) < 0 ) goto GenCall;
04067             break;
04068         }
04069     }
04070     else {
04071         termout = AT.WorkPointer;
04072         AT.WorkPointer = (WORD *) (((UBYTE *) (AT.WorkPointer)) + AM.MaxTer);
04073         if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
04074         if ( InFunction(BHEAD term,termout) ) goto GenCall;
04075         AT.WorkPointer = termout + *termout;
04076         *AN.RepPoint = 1;
04077         AR.expchanged = 1;
04078         if ( *termout && Generator(BHEAD termout,level) < 0 ) goto GenCall;
04079         AT.WorkPointer = termout;
04080     }
04081 }
04082 else if ( replac > 0 ) {
04083     power = AN.TeSuOut;
04084     tepos = AR.TePos;
04085     if ( power < 0 ) { /* Table expansion */
04086         power = -power; tepos = 0;
04087     }
04088     extractbuff = AT.TMbuff;
04089     if ( extractbuff == AM.dbufnum ) {
04090         d = DolToTerms(BHEAD replac);
04091         if ( d && d->where != 0 ) {
04092             iscopy = 1;
04093             if ( AT.TMdolfac > 0 ) { /* We need a factor */
04094                 if ( AT.TMdolfac == 1 ) {
04095                     if ( d->nfactors ) {
04096                         numfac[0] = 4;
04097                         numfac[1] = d->nfactors;
04098                         numfac[2] = 1;
04099                         numfac[3] = 3;
04100                         numfac[4] = 0;
04101                     }
04102                     else {
04103                         numfac[0] = 0;
04104                     }
04105                     StartBuf = numfac;
04106                 }
04107                 else {
04108                     if ( (AT.TMdolfac-1) > d->nfactors && d->nfactors > 0 ) {
04109                         MLOCK(ErrorMessageLock);
04110                         MesPrint("Attempt to use an nonexistent factor %d of a
$-variable", (WORD) (AT.TMdolfac-1));
04111                         if ( d->nfactors == 1 )
04112                             MesPrint("There is only one factor");
04113                         else
04114                             MesPrint("There are only %d factors", (WORD) (d->nfactors));
04115                         MUNLOCK(ErrorMessageLock);
04116                         goto GenCall;
04117                     }
04118                     if ( d->nfactors > 1 ) {
04119                         DOLLARS dd;
04120                         LONG dsize;
04121                         WORD *td1, *td2;
04122                         dd = Dollars + replac;
04123 #ifdef WITHPTHREADS
04124                     {
04125                         int nummodopt, dtype = -1;
04126                         if ( AS.MultiThreaded && ( AC.mparallelfalg == PARALLELFALG ) ) {
04127                             for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
04128                                 if ( replac == ModOptdollars[nummodopt].number ) break;
04129                             }
04130                             if ( nummodopt < NumModOptdollars ) {
04131                                 dtype = ModOptdollars[nummodopt].type;
04132                                 if ( dtype == MODLOCAL ) {
04133                                     dd = ModOptdollars[nummodopt].dstruct+AT.identity;
04134                                 }
04135                             }
04136                         }
04137                     }
04138 #endif
04139                     dsize = dd->factors[AT.TMdolfac-2].size;
04140                     /*
04141                     We copy only the factor we need
04142                     */
04143                     if ( dsize == 0 ) {
04144                         numfac[0] = 4;
04145                         numfac[1] = d->factors[AT.TMdolfac-2].value;
04146                         numfac[2] = 1;
04147                         numfac[3] = 3;
04148                         numfac[4] = 0;
04149                         StartBuf = numfac;
04150                         if ( numfac[1] < 0 ) {
04151                             numfac[1] = -numfac[1];

```

```

04152             numfac[3] = -numfac[3];
04153         }
04154     }
04155     else {
04156         d->factors[AT.TMdolfac-2].where = td2 = (WORD *)Malloc1(
04157             (dsize+1)*sizeof(WORD), "Copy of factor");
04158         td1 = dd->factors[AT.TMdolfac-2].where;
04159         StartBuf = td2;
04160         d->size = dsize; d->type = DOLTERMS;
04161         NCOPY(td2,td1,dsize);
04162         *td2 = 0;
04163     }
04164 }
04165 else if ( d->nfactors == 1 ) {
04166     StartBuf = d->where;
04167 }
04168 else {
04169     MLOCK(ErrorMessageLock);
04170     if ( d->nfactors == 0 ) {
04171         MesPrint("Attempt to use factor %d of an unfactored
04172 $-variable", (WORD) (AT.TMdolfac-1));
04173     }
04174     else {
04175         MesPrint("Internal error. Illegal number of factors for $-variable");
04176     }
04177     MUNLOCK(ErrorMessageLock);
04178     goto GenCall;
04179 }
04180 }
04181 else StartBuf = d->where;
04182 }
04183 else {
04184     d = Dollars + replac;
04185     StartBuf = zeroDollar;
04186 }
04187 posisub = 0;
04188 i = DetCommu(d->where);
04189 #ifdef WITHPTHREADS
04190     if ( AS.MultiThreaded ) {
04191         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
04192             if ( replac == ModOptdollars[nummodopt].number ) break;
04193         }
04194         if ( nummodopt < NumModOptdollars ) {
04195             dtype = ModOptdollars[nummodopt].type;
04196             if ( dtype != MODLOCAL && dtype != MODSUM ) {
04197                 if ( StartBuf[0] && StartBuf[StartBuf[0]] ) {
04198                     MLOCK(ErrorMessageLock);
04199                     MesPrint("A dollar variable with modoption max or min can have only one
04200 term");
04201                     MUNLOCK(ErrorMessageLock);
04202                     goto GenCall;
04203                 }
04204                 LOCK(d->pthreadlockread);
04205             }
04206         }
04207     }
04208 #endif
04209 }
04210 else {
04211     StartBuf = cbuf[extractbuff].Buffer;
04212     posisub = cbuf[extractbuff].rhs[replac] - StartBuf;
04213     i = (WORD)cbuf[extractbuff].CanCommu[replac];
04214 }
04215 if ( power == 1 ) { /* Just a single power */
04216     termout = AT.WorkPointer;
04217     AT.WorkPointer = (WORD *)(((UBYTE *) (AT.WorkPointer)) + AM.MaxTer);
04218     if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
04219     while ( StartBuf[posisub] ) {
04220         if ( extractbuff == AT.allbufnum ) WildDollars(BHEAD &(StartBuf[posisub]));
04221         AT.WorkPointer = (WORD *)(((UBYTE *) (termout)) + AM.MaxTer);
04222         if ( InsertTerm(BHEAD term,replac,extractbuff,
04223             &(StartBuf[posisub]),termout,tepos) < 0 ) goto GenCall;
04224         AT.WorkPointer = termout + *termout;
04225         *AN.RepPoint = 1;
04226         AR.expchanged = 1;
04227         posisub += StartBuf[posisub];
04228     }
04229     /*
04230      * For multiple table substitutions it may be better to
04231      * do modulus arithmetic right here
04232      * Turns out to be not very effective.
04233     */
04234     if ( AN.ncmod != 0 ) {
04235         if ( Modulus(termout) ) goto GenCall;
04236         if ( !*termout ) goto Return0;
04237     }
04238 }
04239 */

```

```

04237 #ifdef WITHPTHREADS
04238     if ( dtype > 0 && dtype != MODLOCAL && dtype != MODSUM ) {
        UNLOCK(d->pthreadlockread); }
04239     if ( ( AS.Balancing && CC->numrhs == 0 ) && StartBuf[posisub] ) {
04240         if ( ( id = ConditionalGetAvailableThread() ) >= 0 ) {
04241             if ( BalanceRunThread(BHEAD id,termout,level) < 0 ) goto GenCall;
04242         }
04243     }
04244     else
04245 #endif
04246     if ( Generator(BHEAD termout,level) < 0 ) goto GenCall;
04247 #ifdef WITHPTHREADS
04248     if ( dtype > 0 && dtype != MODLOCAL ) { dtype = 0; break; }
04249 #endif
04250     if ( iscopy == 0 && ( extractbuff != AM.dbufnum ) ) {
04251 /*
04252      * There are cases in which a bigger buffer is created
04253      * on the fly, like with wildcard buffers.
04254      * We play it safe here. Maybe we can be more selective
04255      * in some distant future?
04256 */
04257         StartBuf = cbuf[extractbuff].Buffer;
04258     }
04259 }
04260 if ( extractbuff == AT.allbufnum ) {
04261     CBUF *Ce = cbuf + extractbuff;
04262     Ce->Pointer = Ce->rhs[Ce->numrhs--];
04263 }
04264 #ifdef WITHPTHREADS
04265 if ( dtype > 0 && dtype != MODLOCAL && dtype != MODSUM ) { UNLOCK(d->pthreadlockread);
dtype = 0; }
04266 #endif
04267 if ( iscopy ) {
04268     if ( d->nfactors > 1 ) {
04269         int j;
04270         for ( j = 0; j < d->nfactors; j++ ) {
04271             if ( d->factors[j].where ) M_free(d->factors[j].where,"Copy of factor");
04272         }
04273         M_free(d->factors,"Dollar factors");
04274     }
04275     M_free(d,"Copy of dollar variable");
04276     d = 0; iscopy = 0;
04277 }
04278 AT.WorkPointer = termout;
04279 }
04280 else if ( i <= 1 ) { /* Use binomials */
04281     LONG posit, olw;
04282     WORD *same, *ow = AT.WorkPointer;
04283     LONG olpw = AT.posWorkPointer;
04284     power1 = power+1;
04285     WantAddLongs(power1);
04286     olw = posit = AT.lWorkPointer; AT.lWorkPointer += power1;
04287     same = ++AT.WorkPointer;
04288     a = accum = ( AT.WorkPointer += power1+1 );
04289     AT.WorkPointer = (WORD *)(((UBYTE *) (AT.WorkPointer)) + 2*AM.MaxTer);
04290     if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
04291     AT.lWorkspace[posit] = posisub;
04292     same[-1] = 0;
04293     *same = 1;
04294     *accum = 0;
04295     tepos = AR.TePos;
04296     i = 1;
04297     do {
04298         if ( StartBuf[AT.lWorkspace[posit]] ) {
04299             if ( ( a = PasteTerm(BHEAD i-1,accum,
04300 &(StartBuf[AT.lWorkspace[posit]]),i,*same) ) == 0 )
04301                 goto GenCall;
04302             AT.lWorkspace[posit+1] = AT.lWorkspace[posit];
04303             same[1] = *same + 1;
04304             if ( i > 1 && AT.lWorkspace[posit] < AT.lWorkspace[posit-1] ) *same = 1;
04305             AT.lWorkspace[posit] += StartBuf[AT.lWorkspace[posit]];
04306             i++;
04307             posit++;
04308             same++;
04309         }
04310         else {
04311             i--; posit--; same--;
04312         }
04313     } if ( i > power ) {
04314         termout = AT.WorkPointer = a;
04315         AT.WorkPointer = (WORD *)(((UBYTE *) (AT.WorkPointer)) + 2*AM.MaxTer);
04316         if ( AT.WorkPointer > AT.WorkTop )
04317             goto OverWork;
04318         if ( FiniTerm(BHEAD term,accum,termout,replac,tepos) ) goto GenCall;
04319         AT.WorkPointer = termout + *termout;
04320         *AN.RepPoint = 1;
04321         AR.expchanged = 1;

```

```

04322 #ifdef WITHPTHREADS
04323         if ( dtype > 0 && dtype != MODLOCAL && dtype != MODSUM ) {
04324             UNLOCK(d->pthreadlockread); }
04324         if ( ( AS.Balancing && CC->numrhs == 0 ) && ( i > 0 )
04325             && ( id = ConditionalGetAvailableThread() ) >= 0 ) {
04326             if ( BalanceRunThread(BHEAD id,termout,level) < 0 ) goto GenCall;
04327         }
04328         else
04329 #endif
04330         if ( Generator(BHEAD termout,level) ) goto GenCall;
04331 #ifdef WITHPTHREADS
04332         if ( dtype > 0 && dtype != MODLOCAL ) { dtype = 0; break; }
04333 #endif
04334         if ( iscopy == 0 && ( extractbuff != AM.dbufnum ) )
04335             StartBuf = cbuf[extractbuff].Buffer;
04336         i--; posit--; same--;
04337     }
04338     } while ( i > 0 );
04339 #ifdef WITHPTHREADS
04340     if ( dtype > 0 && dtype != MODLOCAL && dtype != MODSUM ) { UNLOCK(d->pthreadlockread);
dtype = 0; }
04341 #endif
04342     if ( iscopy ) {
04343         if ( d->nfactors > 1 ) {
04344             int j;
04345             for ( j = 0; j < d->nfactors; j++ ) {
04346                 if ( d->factors[j].where ) M_free(d->factors[j].where,"Copy of factor");
04347             }
04348             M_free(d->factors,"Dollar factors");
04349         }
04350         M_free(d,"Copy of dollar variable");
04351         d = 0; iscopy = 0;
04352     }
04353     AT.WorkPointer = ow; AT.lWorkPointer = olw; AT.posWorkPointer = olpw;
04354 }
04355 else { /* No binomials */
04356     LONG posit, olw, olpw = AT.posWorkPointer;
04357     WantAddLongs(power);
04358     posit = olw = AT.lWorkPointer; AT.lWorkPointer += power;
04359     a = accum = AT.WorkPointer;
04360     AT.WorkPointer = (WORD *)(((UBYTE *) (AT.WorkPointer)) + 2*AM.MaxTer);
04361     if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
04362     for ( i = 0; i < power; i++ ) AT.lWorkspace[posit++] = posisub;
04363     posit = olw;
04364     *accum = 0;
04365     tepos = AR.TePos;
04366     i = 0;
04367     while ( i >= 0 ) {
04368         if ( StartBuf[AT.lWorkspace[posit]] ) {
04369             if ( ( a = PasteTerm(BHEAD i,accum,
04370                 &(StartBuf[AT.lWorkspace[posit]]),1,1) ) == 0 ) goto GenCall;
04371             AT.lWorkspace[posit] += StartBuf[AT.lWorkspace[posit]];
04372             i++; posit++;
04373         }
04374         else {
04375             AT.lWorkspace[posit--] = posisub;
04376             i--;
04377         }
04378         if ( i >= power ) {
04379             termout = AT.WorkPointer = a;
04380             AT.WorkPointer = (WORD *)(((UBYTE *) (AT.WorkPointer)) + 2*AM.MaxTer);
04381             if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
04382             if ( FiniTerm(BHEAD term,accum,termout,replac,tepos) ) goto GenCall;
04383             AT.WorkPointer = termout + *termout;
04384             *AN.RepPoint = 1;
04385             AR.expchanged = 1;
04386 #ifdef WITHPTHREADS
04387             if ( dtype > 0 && dtype != MODLOCAL && dtype != MODSUM ) {
04388                 UNLOCK(d->pthreadlockread); }
04388             if ( ( AS.Balancing && CC->numrhs == 0 ) && ( i > 0 ) && ( id =
ConditionalGetAvailableThread() ) >= 0 ) {
04389                 if ( BalanceRunThread(BHEAD id,termout,level) < 0 ) goto GenCall;
04390             }
04391             else
04392 #endif
04393             if ( Generator(BHEAD termout,level) ) goto GenCall;
04394 #ifdef WITHPTHREADS
04395             if ( dtype > 0 && dtype != MODLOCAL && dtype != MODSUM ) { dtype = 0; break; }
04396 #endif
04397             if ( iscopy == 0 && ( extractbuff != AM.dbufnum ) )
04398                 StartBuf = cbuf[extractbuff].Buffer;
04399             i--; posit--;
04400         }
04401     }
04402 #ifdef WITHPTHREADS
04403     if ( dtype > 0 && dtype != MODLOCAL && dtype != MODSUM ) { UNLOCK(d->pthreadlockread);
dtype = 0; }

```



```

04404 #endif
04405         if ( iscopy ) {
04406             if ( d->nfactors > 1 ) {
04407                 int j;
04408                 for ( j = 0; j < d->nfactors; j++ ) {
04409                     if ( d->factors[j].where ) M_free(d->factors[j].where,"Copy of factor");
04410                 }
04411                 M_free(d->factors,"Dollar factors");
04412             }
04413             M_free(d,"Copy of dollar variable");
04414             d = 0; iscopy = 0;
04415         }
04416         AT.WorkPointer = accum;
04417         AT.lWorkPointer = olw;
04418         AT.posWorkPointer = olpw;
04419     }
04420 }
04421 else { /* Expression from disk */
04422     POSITION StartPos;
04423     LONG position, olpw, opw, comprev, extra;
04424     RENUMBER renumber;
04425     WORD *Freeze, *aa, *dummies;
04426     replac = -replac-1;
04427     power = AN.TeSuOut;
04428     Freeze = AN.Frozen;
04429     if ( Expressions[replac].status == STOREDEXPRESSION ) {
04430         POSITION firstpos;
04431         SETSTARTPOS(firstpos);
04432
04433         /* Note that AT.TMaddr is needed for GetTable just once! */
04434         /*
04435         We need space for the previous term in the compression
04436         This is made available in AR.CompressBuffer, although we may get
04437         problems with this sooner or later. Hence we need to keep
04438         a set of pointers in AR.CompressBuffer
04439         Note that after the last call there has been no use made
04440         of AR.CompressPointer, so it points automatically at its original
04441         position!
04442         */
04443         WantAddPointers(power+1);
04444         comprev = opw = AT.pWorkPointer;
04445         AT.pWorkPointer += power+1;
04446         WantAddPositions(power+1);
04447         position = olpw = AT.posWorkPointer;
04448         AT.posWorkPointer += power + 1;
04449
04450         AT.pWorkSpace[comprev++] = AR.CompressPointer;
04451
04452         for ( i = 0; i < power; i++ ) {
04453             PUTZERO(AT.posWorkSpace[position]); position++;
04454         }
04455         position = olpw;
04456         if ( ( renumber = GetTable(replac,&(AT.posWorkSpace[position]),1) ) == 0 ) goto GenCall;
04457         dummies = AT.WorkPointer;
04458         *dummies++ = AR.CurDum;
04459         AT.WorkPointer += power+2;
04460         accum = AT.WorkPointer;
04461         AT.WorkPointer = (WORD *)(((UBYTE *) (AT.WorkPointer)) + 2*AM.MaxTer);
04462         if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
04463         aa = AT.WorkPointer;
04464         *accum = 0;
04465         i = 0; StartPos = AT.posWorkSpace[position];
04466         dummies[i] = AR.CurDum;
04467         while ( i >= 0 ) {
04468             skippedfirst:
04469                 AR.CompressPointer = AT.pWorkSpace[comprev-1];
04470                 if ( ( extra = PasteFile(BHEAD i,accum,&(AT.posWorkSpace[position])
04471                     ,&aa,renumber,Freeze,replac) ) < 0 ) goto GenCall;
04472                 if ( Expressions[replac].numdummies > 0 ) {
04473                     AR.CurDum = dummies[i] + Expressions[replac].numdummies;
04474                 }
04475                 if ( NOTSTARTPOS(firstpos) ) {
04476                     if ( ISMINPOS(firstpos) || ISEQUALPOS(firstpos,AT.posWorkSpace[position]) ) {
04477                         firstpos = AT.posWorkSpace[position];
04478                     /*
04479                     ADDPOS(AT.posWorkSpace[position],extra * sizeof(WORD));
04480                     */
04481                     goto skippedfirst;
04482                 }
04483             }
04484             if ( extra ) {
04485                 /*
04486                 ADDPOS(AT.posWorkSpace[position],extra * sizeof(WORD));
04487                 */
04488                 ++; AT.posWorkSpace[++position] = StartPos;
04489                 AT.pWorkSpace[comprev++] = AR.CompressPointer;
04490                 dummies[i] = AR.CurDum;

```

```

04491         }
04492     else {
04493         PUTZERO(AT.posWorkSpace[position]); position--; i--;
04494         AR.CurDum = dummies[i];
04495         comprev--;
04496     }
04497     if ( i >= power ) {
04498         termout = AT.WorkPointer = a;
04499         AT.WorkPointer = (WORD *)(((UBYTE *) (AT.WorkPointer)) + 2*AM.MaxTer);
04500         if ( AT.WorkPointer > AT.WorkTop ) goto OverWork;
04501         if ( FiniTerm(BHEAD term,accum,termout,replac,0) ) goto GenCall;
04502         if ( *termout ) {
04503             AT.WorkPointer = termout + *termout;
04504             *AN.RepPoint = 1;
04505             AR.expchanged = 1;
04506 #ifdef WITHPTHREADS
04507             if ( ( AS.Balancing && CC->numrhs == 0 ) && ( i > 0 ) && ( id =
ConditionalGetAvailableThread() ) >= 0 ) {
04508                 if ( BalanceRunThread(BHEAD id,termout,level) < 0 ) goto GenCall;
04509             }
04510             else
04511                 if ( Generator(BHEAD termout,level) ) goto GenCall;
04512 #endif
04513             i--; position--;
04514             AR.CurDum = dummies[i];
04515             comprev--;
04516             AT.WorkPointer = aa;
04517         }
04518         AT.WorkPointer = accum;
04519         AT.posWorkPointer = olpw;
04520         AT.pWorkPointer = opw;
04521         /* Bug fix. See also GetTable
04522 #ifdef WITHPTHREADS
04523     M_free(renumber->symb.lo,"VarSpace");
04524     M_free(renumber,"Renumbrer");
04525 #endif
04526 */
04527     if ( renumber->symb.lo != AN.dummyrenumlist )
04528         M_free(renumber->symb.lo,"VarSpace");
04529     M_free(renumber,"Renumbrer");
04530 }
04531 else { /* Active expression */
04532     aa = accum = AT.WorkPointer;
04533     if ( ( (WORD *)(((UBYTE *) (AT.WorkPointer)) + 2 * AM.MaxTer + sizeof(WORD)) ) > AT.WorkTop
04534 )
04535         goto OverWork;
04536     *accum++ = -1; AT.WorkPointer++;
04537     if ( DoOnePow(BHEAD term,power,replac,accum,aa,level,Freeze) ) goto GenCall;
04538     AT.WorkPointer = aa;
04539 }
04540 }
04541 Return0:
04542     AR.CurDum = DumNow;
04543     AN.RepPoint = RepSto;
04544     CC->numrhs = oldtoprhs;
04545     CC->Pointer = CC->Buffer + oldcpointer;
04546     CCC->numrhs = oldatoprhs;
04547     CCC->Pointer = CCC->Buffer + oldacpointer;
04548     return(0);
04549 GenCall:
04550     if ( AM.tracebackflag ) {
04551         termout = term;
04552         MLOCK(ErrorMessageLock);
04553         AO.OutFill = AO.OutputLine = (UBYTE *)AT.WorkPointer;
04554         AO.OutSkip = 3;
04555         FiniLine();
04556         i = *termout;
04557         while ( --i >= 0 ) {
04558             TalToLine((UWORD) (*termout++));
04559             TokenToLine((UBYTE *) " ");
04560         }
04561         AO.OutSkip = 0;
04562         FiniLine();
04563         MesCall("Generator");
04564         MUNLOCK(ErrorMessageLock);
04565     }
04566     CC->numrhs = oldtoprhs;
04567     CC->Pointer = CC->Buffer + oldcpointer;
04568     CCC->numrhs = oldatoprhs;
04569     CCC->Pointer = CCC->Buffer + oldacpointer;
04570     return(-1);

```

```

04576 OverWork:
04577     CC->numrhs = oldtoprhs;
04578     CC->Pointer = CC->Buffer + oldcpointer;
04579     CCC->numrhs = oldatoprhs;
04580     CCC->Pointer = CCC->Buffer + oldacpointer;
04581     MLOCK(ErrorMessageLock);
04582     MesWork();
04583     MUNLOCK(ErrorMessageLock);
04584     return(-1);
04585 }
04586
04587 /*
04588     #] Generator :
04589     #[ DoOnePow :           WORD DoOnePow(term,power,nexp,accum,aa,level,freeze)
04590 */
04615 #ifdef WITHPTHREADS
04616 char freezestring[] = "freeze<-xxxx";
04617 #endif
04618
04619 int DoOnePow(PHEAD WORD *term, WORD power, WORD nexp, WORD * accum,
04620             WORD *aa, WORD level, WORD *freeze)
04621 {
04622     GETBIDENTITY
04623     POSITION oldposition, startposition;
04624     WORD *acc, *termout, fromfreeze = 0;
04625     WORD *oldpointer = AR.CompressPointer;
04626     FILEHANDLE *fi;
04627     WORD type, retval;
04628     WORD oldGetOneFile = AR.GetOneFile;
04629     WORD olddummies = AR.CurDum;
04630     WORD extradummies = Expressions[nexp].numdummies;
04631     /*
04632     The next code is for some tricky debugging. (5-jan-2010 JV)
04633     Normally it should be disabled.
04634     */
04635     /*
04636     #ifdef WITHPTHREADS
04637     if ( freeze ) {
04638         MLOCK(ErrorMessageLock);
04639         if ( AT.identity < 10 ) {
04640             freezestring[8] = '0'+AT.identity;
04641             freezestring[9] = '>';
04642             freezestring[10] = 0;
04643         }
04644         else if ( AT.identity < 100 ) {
04645             freezestring[8] = '0'+AT.identity/10;
04646             freezestring[9] = '0'+AT.identity%10;
04647             freezestring[10] = '>';
04648             freezestring[11] = 0;
04649         }
04650         else {
04651             freezestring[8] = 0;
04652         }
04653         PrintTerm(freeze,freezestring);
04654         MUNLOCK(ErrorMessageLock);
04655     }
04656     #else
04657     if ( freeze ) PrintTerm(freeze,"freeze");
04658     #endif
04659     */
04660     type = Expressions[nexp].status;
04661     if ( type == HIDDENLEXPRESSION || type == HIDDENEXPRESSION
04662         || type == DROPHLEXPRESSION || type == DROPHGEXPRESSION
04663         || type == UNHIDELEXPRESSION || type == UNHIDEGEXPRESSION ) {
04664         AR.GetOneFile = 2; fi = AR.hidefile;
04665     }
04666     else {
04667         AR.GetOneFile = 0; fi = AR.infile;
04668     }
04669     if ( fi->handle >= 0 ) {
04670         PUTZERO(oldposition);
04671     #ifdef WITHSEEK
04672         LOCK(AS.inputslock);
04673         SeekFile(fi->handle,&oldposition,SEEK_CUR);
04674         UNLOCK(AS.inputslock);
04675     #endif
04676     }
04677     else {
04678         SETBASEPOSITION(oldposition,fi->POfill-fi->PObuffer);
04679     }
04680     if ( freeze && ( Expressions[nexp].bracketinfo != 0 ) ) {
04681         POSITION *brapos;
04682     /*
04683     There is a bracket index
04684     AR.CompressPointer = oldpointer;
04685     */
04686     (*aa)++;

```

```

04687         power--;
04688         if ( ( brapos = FindBracket(nexp,freeze) ) == 0 )
04689             goto EndExpr;
04690         startposition = *brapos;
04691         goto doterms;
04692     }
04693     startposition = AS.OldOnFile[nexp];
04694     retval = GetOneTerm(BHEAD accum,fi,&startposition,0);
04695     if ( retval > 0 ) {          /* Skip prototype */
04696         (*aa)++;
04697         power--;
04698 doterms:
04699         AR.CompressPointer = oldipointer;
04700         for (;;) {
04701             retval = GetOneTerm(BHEAD accum,fi,&startposition,0);
04702             if ( retval <= 0 ) break;
04703         /*
04704             Here should come the code to test for [].
04705         */
04706         if ( freeze ) {
04707             WORD *t, *m, *r, *mstop;
04708             WORD *tset;
04709             t = accum;
04710             m = freeze;
04711             m += *m;
04712             m -= ABS(m[-1]);
04713             mstop = m;
04714             m = freeze + 1;
04715             r = t;
04716             r += *t;
04717             r -= ABS(r[-1]);
04718             t++;
04719             tset = t;
04720             while ( t < r && *t != HAAKJE ) t += t[1];
04721             if ( t >= r ) {
04722                 if ( m < mstop ) {
04723                     if ( fromfreeze ) goto EndExpr;
04724                     goto NextTerm;
04725                 }
04726                 t = tset;
04727             }
04728             else {
04729                 r = tset;
04730                 while ( r < t && m < mstop ) {
04731                     if ( *r == *m ) { m++; r++; }
04732                     else {
04733                         if ( fromfreeze ) goto EndExpr;
04734                         goto NextTerm;
04735                     }
04736                 }
04737                 if ( r < t || m < mstop ) {
04738                     if ( fromfreeze ) goto EndExpr;
04739                     goto NextTerm;
04740                 }
04741             }
04742             fromfreeze = 1;
04743             r = tset;
04744             m = accum;
04745             m += *m;
04746             while ( t < m ) *r++ = *t++;
04747             *accum = WORDDIF(r,accum);
04748         }
04749         if ( extradummies > 0 ) {
04750             if ( olddummies > AM.IndDum ) {
04751                 MoveDummies(BHEAD accum,olddummies-AM.IndDum);
04752             }
04753             AR.CurDum = olddummies+extradummies;
04754         }
04755         acc = accum;
04756         acc += *acc;
04757         if ( power <= 0 ) {
04758             termout = acc;
04759             AT.WorkPointer = (WORD *)(((UBYTE *) (acc)) + 2*AM.MaxTer);
04760             if ( AT.WorkPointer > AT.WorkTop ) {
04761                 MLOCK(ErrorMessageLock);
04762                 MesWork();
04763                 MUNLOCK(ErrorMessageLock);
04764                 return(-1);
04765             }
04766             if ( FiniTerm(BHEAD term,aa,termout,nexp,0) ) goto PowCall;
04767             if ( *termout ) {
04768                 MarkPolyRatFunDirty(termout)
04769                 PolyFunDirty(BHEAD termout); /*
04770                 AT.WorkPointer = termout + *termout;
04771                 *AN.RepPoint = 1;
04772                 AR.expchanged = 1;
04773                 if ( Generator(BHEAD termout,level) ) goto PowCall;

```

```

04774         }
04775     }
04776     else {
04777         if ( acc > AT.WorkTop ) {
04778             MLOCK(ErrorMessageLock);
04779             MesWork();
04780             MUNLOCK(ErrorMessageLock);
04781             return(-1);
04782         }
04783         if ( DoOnePow(BHEAD term,power,nexp,acc,aa,level,freeze) ) goto PowCall;
04784     }
04785 NextTerm;;
04786     AR.CompressPointer = oldipointer;
04787 }
04788 EndExpr:
04789     (*aa)--;
04790 }
04791     AR.CompressPointer = oldipointer;
04792     if ( fi->handle >= 0 ) {
04793 #ifdef WITHSEEK
04794         LOCK(AS.inputslock);
04795         SeekFile(fi->handle,&oldposition,SEEK_SET);
04796         UNLOCK(AS.inputslock);
04797         if ( ISNEGPOS(oldposition) ) {
04798             MLOCK(ErrorMessageLock);
04799             MesPrint("File error");
04800             goto PowCall2;
04801         }
04802 #endif
04803     }
04804     else {
04805         fi->POfill = fi->PObuffer + BASEPOSITION(oldposition);
04806     }
04807     AR.GetOneFile = oldGetOneFile;
04808     AR.CurDum = olddummies;
04809     return(0);
04810 PowCall;;
04811     MLOCK(ErrorMessageLock);
04812 #ifdef WITHSEEK
04813 PowCall2;;
04814 #endif
04815     MesCall("DoOnePow");
04816     MUNLOCK(ErrorMessageLock);
04817     SETERROR(-1)
04818 }
04819
04820 /*
04821     #] DoOnePow :
04822     #[ Deferred :          WORD Deferred(term,level)
04823 */
04840 int Deferred(PHEAD WORD *term, WORD level)
04841 {
04842     GETBIDENTITY
04843     POSITION startposition;
04844     WORD *t, *m, *mstop, *tstart, decr, oldb, *termout, i, *oldwork, retval;
04845     WORD *oldipointer = AR.CompressPointer, *oldPOfill = AR.infile->POfill;
04846     WORD oldGetOneFile = AR.GetOneFile;
04847     AR.GetOneFile = 1;
04848     oldwork = AT.WorkPointer;
04849     AT.WorkPointer = (WORD *)(((UBYTE *) (AT.WorkPointer)) + AM.MaxTer);
04850     termout = AT.WorkPointer;
04851     AR.DeferFlag = 0;
04852     startposition = AR.DefPosition;
04853 /*
04854     Store old position
04855 */
04856     if ( AR.infile->handle >= 0 ) {
04857 /*
04858         PUTZERO(oldposition);
04859         SeekFile(AR.infile->handle,&oldposition,SEEK_CUR);
04860 */
04861     }
04862     else {
04863 /*
04864         SETBASEPOSITION(oldposition,AR.infile->POfill-AR.infile->PObuffer);
04865 */
04866         AR.infile->POfill = (WORD *)(((UBYTE *) (AR.infile->PObuffer)
04867             +BASEPOSITION(startposition)));
04868     }
04869 /*
04870     Look in the CompressBuffer where the bracket contents start
04871 */
04872     t = m = AR.CompressBuffer;
04873     t += *t;
04874     mstop = t - ABS(t[-1]);
04875     m++;
04876     while ( *m != HAAKJE && m < mstop ) m += m[1];

```

```

04877     if ( m >= mstop ) { /* No deferred action! */
04878         AT.WorkPointer = term + *term;
04879         if ( Generator(BHEAD term,level) ) goto DefCall;
04880         AR.DeferFlag = 1;
04881         AT.WorkPointer = oldwork;
04882         AR.GetOneFile = oldGetOneFile;
04883         return(0);
04884     }
04885     mstop = m + m[1];
04886     decr = WORDDIF(mstop,AR.CompressBuffer)-1;
04887     tstart = AR.CompressPointer + decr;
04888
04889     m = AR.CompressBuffer;
04890     t = AR.CompressPointer;
04891     i = *m;
04892     NCOPY(t,m,i);
04893     oldb = *tstart;
04894     AR.TePos = 0;
04895     AN.TeSuOut = 0;
04896 /*
04897     Status:
04898     First bracket content starts at mstop.
04899     Next term starts at startposition.
04900     Decompression information is in AR.CompressPointer.
04901     The outside of the bracket runs from AR.CompressBuffer+1 to mstop.
04902 */
04903     for(;;) {
04904         *tstart = *(AR.CompressPointer)-decr;
04905         AR.CompressPointer = AR.CompressPointer+AR.CompressPointer[0];
04906         if ( InsertTerm(BHEAD term,0,AM.rbufnum,tstart,termout,0) < 0 ) {
04907             goto DefCall;
04908         }
04909         *tstart = oldb;
04910         AT.WorkPointer = termout + *termout;
04911         if ( Generator(BHEAD termout,level) ) goto DefCall;
04912         AR.CompressPointer = oldipointer;
04913         AT.WorkPointer = termout;
04914         retval = GetOneTerm(BHEAD AT.WorkPointer,AR.infile,&startposition,0);
04915         if ( retval >= 0 ) AR.CompressPointer = oldipointer;
04916         if ( retval <= 0 ) break;
04917         t = AR.CompressPointer;
04918         if ( *t < (1 + decr + ABS(*(t+*t-1))) ) break;
04919         t++;
04920         m = AR.CompressBuffer+1;
04921         while ( m < mstop ) {
04922             if ( *m != *t ) goto Thatsit;
04923             m++; t++;
04924         }
04925     }
04926 Thatsit:;
04927 /*
04928     Finished. Reposition the file, restore information and return.
04929 */
04930     if ( AR.infile->handle < 0 ) AR.infile->POfill = oldPOfill;
04931     AR.DeferFlag = 1;
04932     AR.GetOneFile = oldGetOneFile;
04933     AT.WorkPointer = oldwork;
04934     return(0);
04935 DefCall:;
04936     MLOCK(ErrorMessageLock);
04937     MesCall("Deferred");
04938     MUNLOCK(ErrorMessageLock);
04939     SETERROR(-1)
04940 }
04941
04942 /*
04943     #] Deferred :
04944     #[ PrepPoly :          WORD PrepPoly(term,par)
04945 */
04968 int PrepPoly(PHEAD WORD *term,WORD par)
04969 {
04970     GETBIDENTITY
04971     WORD count = 0, i, jcoef, ncoef;
04972     WORD *t, *m, *r, *tstop, *poly = 0, *v, *w, *vv, *ww;
04973     WORD *oldworkpointer = AT.WorkPointer;
04974 /*
04975     The problem here is that the function will be forced into 'long'
04976     notation. After this -SNUMBER,1 becomes 6,0,4,1,1,3 and the
04977     pattern matcher cannot match a short 1 with a long 1.
04978     But because this is an undocumented feature for very special
04979     purposes, we don't do anything about it. (30-aug-2011)
04980 */
04981     if ( AR.PolyFunType == 2 && AR.PolyFunExp != 2 ) {
04982         WORD oldtype = AR.SortType;
04983         AR.SortType = SORTHIGHFIRST;
04984         if ( poly_ratfun_normalize(BHEAD term) != 0 ) Terminate(-1);
04985 /*

```

```

04986         oldworkpointer = AT.WorkPointer;
04987         AR.SortType = oldtype;
04988     }
04989     AT.PolyAct = 0;
04990     t = term;
04991     GETSTOP(t,tstop);
04992     t++;
04993     while ( t < tstop ) {
04994         if ( *t == AR.PolyFun ) {
04995             if ( count > 0 ) return(0);
04996             poly = t;
04997             count++;
04998         }
04999         t += t[1];
05000     }
05001     r = m = term + *term;
05002     i = ABS(m[-1]);
05003     if ( par > 0 ) {
05004         if ( count == 0 ) return(0);
05005         else if ( AR.PolyFunType == 1 || (AR.PolyFunType == 2 && AR.PolyFunExp == 2) )
05006             goto DoOne;
05007         else if ( AR.PolyFunType == 2 )
05008             goto DoTwo;
05009         else
05010             goto DoError;
05011     }
05012     else if ( count == 0 ) {
05013         /*
05014             #! Create a PolyFun :
05015         */
05016         poly = t = tstop;
05017         if ( i == 3 && m[-2] == 1 && (m[-3]&MAXPOSITIVE) == m[-3] ) {
05018             *m++ = AR.PolyFun;
05019             if ( AR.PolyFunType == 1 || (AR.PolyFunType == 2 && AR.PolyFunExp == 2) ) {
05020                 *m++ = FUNHEAD+2;
05021                 FILLFUN(m)
05022                 *m++ = -SNUMBER;
05023                 *m = m[-2-FUNHEAD] < 0 ? -m[-4-FUNHEAD]: m[-4-FUNHEAD];
05024                 m++;
05025             }
05026             else if ( AR.PolyFunType == 2 ) {
05027                 *m++ = FUNHEAD+4;
05028                 FILLFUN(m)
05029                 *m++ = -SNUMBER;
05030                 *m = m[-2-FUNHEAD] < 0 ? -m[-4-FUNHEAD]: m[-4-FUNHEAD];
05031                 m++;
05032                 *m++ = -SNUMBER;
05033                 *m++ = 1;
05034             }
05035         }
05036         else {
05037             WORD *vm;
05038             r = tstop;
05039             if ( AR.PolyFunType == 1 || (AR.PolyFunType == 2 && AR.PolyFunExp == 2) ) {
05040                 *m++ = AR.PolyFun;
05041                 *m++ = FUNHEAD+ARGHEAD+i+1;
05042                 FILLFUN(m)
05043                 *m++ = ARGHEAD+i+1;
05044                 *m++ = 0;
05045                 FILLARG(m)
05046                 *m++ = i+1;
05047                 NCOPY(m,r,i);
05048             }
05049             else if ( AR.PolyFunType == 2 ) {
05050                 WORD *num, *den, size, sign, sizenum, sizeden;
05051                 if ( m[-1] < 0 ) { sign = -1; size = -m[-1]; }
05052                 else { sign = 1; size = m[-1]; }
05053                 num = m - size; size = (size-1)/2; den = num + size;
05054                 sizenum = size; while ( num[sizenum-1] == 0 ) sizenum--;
05055                 sizeden = size; while ( den[sizeden-1] == 0 ) sizeden--;
05056                 v = m;
05057                 AT.PolyAct = WORDDIF(v,term);
05058                 *v++ = AR.PolyFun;
05059                 v++;
05060                 FILLFUN(v);
05061                 vm = v;
05062                 *v++ = ARGHEAD+2*sizenum+2;
05063                 *v++ = 0;
05064                 FILLARG(v);
05065                 *v++ = 2*sizenum+2;
05066                 for ( i = 0; i < sizenum; i++ ) *v++ = num[i];
05067                 *v++ = 1;
05068                 for ( i = 1; i < sizenum; i++ ) *v++ = 0;
05069                 *v++ = sign*(2*sizenum+1);
05070                 if ( ToFast(vm,vm) ) v = vm+2;
05071                 vm = v;
05072                 *v++ = ARGHEAD+2*sizeden+2;

```

```

05073         *v++ = 0;
05074         FILLARG(v);
05075         *v++ = 2*sizeden+2;
05076         for ( i = 0; i < sizeden; i++ ) *v++ = den[i];
05077         *v++ = 1;
05078         for ( i = 1; i < sizeden; i++ ) *v++ = 0;
05079         *v++ = 2*sizeden+1;
05080         if ( ToFast(v,m,v) ) v = vm+2;
05081         i = v-m;
05082         m[1] = i;
05083         w = num;
05084         NCOPY(w,m,i);
05085         *w++ = 1; *w++ = 1; *w++ = 3; *term = w - term;
05086         return(0);
05087     }
05088 }
05089 /*
05090     #] Create a PolyFun :
05091 */
05092 }
05093 else if ( AR.PolyFunType == 1 || (AR.PolyFunType == 2 && AR.PolyFunExp == 2) ) {
05094     DoOne;;
05095 /*
05096     #[ One argument :
05097 */
05098     m = term + *term;
05099     r = poly + poly[1];
05100     if ( ( poly[1] == FUNHEAD+2 && poly[FUNHEAD+1] == 0
05101         && poly[FUNHEAD] == -SNUMBER ) || poly[1] == FUNHEAD ) return(1);
05102     t = poly + FUNHEAD;
05103     if ( t >= r ) return(0);
05104     if ( m[-1] == 3 && *tstop == 1 && tstop[1] == 1 ) {
05105         i = poly[1];
05106         t = poly;
05107         NCOPY(m,t,i);
05108     }
05109     else if ( *t <= -FUNCTION ) {
05110         if ( t+1 < r ) return(0); /* More than one argument */
05111         r = tstop;
05112         *m++ = AR.PolyFun;
05113         *m++ = FUNHEAD*2+ARGHEAD+i+1;
05114         FILLFUN(m)
05115         *m++ = FUNHEAD+ARGHEAD+i+1;
05116         *m++ = 0;
05117         FILLARG(m)
05118         *m++ = FUNHEAD+i+1;
05119         *m++ = -t++;
05120         *m++ = FUNHEAD;
05121         FILLFUN(m)
05122         NCOPY(m,r,i);
05123     }
05124     else if ( *t < 0 ) {
05125         if ( t+2 < r ) return(0); /* More than one argument */
05126         r = tstop;
05127         if ( *t == -SNUMBER ) {
05128             if ( t[1] == 0 ) return(1); /* Term should be zero now */
05129             *m = AR.PolyFun;
05130             w = m+1;
05131             m += FUNHEAD+ARGHEAD;
05132             v = m;
05133             *m++ = 5+i;
05134             *m++ = SNUMBER;
05135             *m++ = 4;
05136             *m++ = t[1];
05137             *m++ = 1;
05138             NCOPY(m,r,i);
05139             if ( m >= AT.WorkSpace && m < AT.WorkTop )
05140                 AT.WorkPointer = m;
05141             if ( Normalize(BHEAD v) ) Terminate(-1);
05142             AT.WorkPointer = oldworkpointer;
05143             m = w;
05144             if ( *v == 4 && v[2] == 1 && (v[1]&MAXPOSITIVE) == v[1] ) {
05145                 *m++ = FUNHEAD+2;
05146                 FILLFUN(m)
05147                 *m++ = -SNUMBER;
05148                 *m++ = v[3] < 0 ? -v[1] : v[1];
05149             }
05150             else if ( *v == 0 ) return(1);
05151             else {
05152                 *m++ = FUNHEAD+ARGHEAD+*v;
05153                 FILLFUN(m)
05154                 *m++ = ARGHEAD+*v;
05155                 *m++ = 0;
05156                 FILLARG(m)
05157                 m = v + *v;
05158             }
05159         }

```



```

05160         else if ( *t == -SYMBOL ) {
05161             *m++ = AR.PolyFun;
05162             *m++ = FUNHEAD+ARGHEAD+5+i;
05163             FILLFUN(m)
05164             *m++ = ARGHEAD+5+i;
05165             *m++ = 0;
05166             FILLARG(m)
05167             *m++ = 5+i;
05168             *m++ = SYMBOL;
05169             *m++ = 4;
05170             *m++ = t[1];
05171             *m++ = 1;
05172             NCOPY(m,r,i);
05173         }
05174         else return(0);          /* Not symbol-like */
05175     }
05176     else {
05177         if ( t + *t < r ) return(0); /* More than one argument */
05178         i = m[-1];
05179         *m++ = AR.PolyFun;
05180         w = m;
05181         m += ARGHEAD+FUNHEAD-1;
05182         t += ARGHEAD;
05183         jcoef = i < 0 ? (i+1)»1:(i-1)»1;
05184         v = t;
05185     /*
05186         Test now the scalar nature of the argument.
05187         No indices allowed.
05188     */
05189         while ( t < r ) {
05190             WORD *vstop;
05191             vv = t + *t;
05192             vstop = vv - ABS(vv[-1]);
05193             t++;
05194             while( t < vstop ) {
05195                 if ( *t == INDEX ) return(0);
05196                 t += t[1];
05197             }
05198             t = vv;
05199         }
05200     /*
05201         Now multiply each term by the coefficient.
05202     */
05203         t = v;
05204         while ( t < r ) {
05205             ww = m;
05206             v = t + *t;
05207             ncoef = v[-1];
05208             vv = v - ABS(ncoef);
05209             if ( ncoef < 0 ) ncoef++;
05210             else ncoef--;
05211             ncoef »= 1;
05212             while ( t < vv ) *m++ = *t++;
05213             if ( MulRat(BHEAD (UWORD *)vv,ncoef, (UWORD *)tstop,jcoef,
05214                 (UWORD *)m,&ncoef) ) Terminate(-1);
05215             ncoef *= 2;
05216             m += ABS(ncoef);
05217             if ( ncoef < 0 ) ncoef--;
05218             else ncoef++;
05219             *m++ = ncoef;
05220             *ww = WORDDIF(m,ww);
05221             if ( AN.ncmod != 0 ) {
05222                 if ( Modulus(ww) ) Terminate(-1);
05223                 if ( *ww == 0 ) return(1);
05224                 m = ww + *ww;
05225             }
05226             t = v;
05227         }
05228         *w = (WORDDIF(m,w))+1;
05229         w[FUNHEAD-1] = w[0] - FUNHEAD;
05230         w[FUNHEAD] = 0;
05231         w[1] = 0; /* omission survived for years. 23-mar-2006 JV */
05232         w += FUNHEAD-1;
05233         if ( ToFast(w,w) ) {
05234             if ( *w <= -FUNCTION ) { w[-FUNHEAD+1] = FUNHEAD+1; m = w+1; }
05235             else { w[-FUNHEAD+1] = FUNHEAD+2; m = w+2; }
05236         }
05237     }
05238     }
05239     t = poly + poly[1];
05240     while ( t < tstop ) *poly++ = *t++;
05241 /*
05242     #] One argument :
05243 */
05244 }
05245 else if ( AR.PolyFunType == 2 ) {
05246     DoTwo;

```

```

05247 /*
05248      #] Two arguments :
05249 */
05250      WORD *num, *den, size, sign, sizenum, sizeden;
05251 /*
05252      First make sure that the PolyFun is last
05253 */
05254      m = term + *term;
05255      if ( poly + poly[1] < tstop ) {
05256          for ( i = 0; i < poly[1]; i++ ) m[i] = poly[i];
05257          t = poly; v = poly + poly[1];
05258          while ( v < tstop ) *t++ = *v++;
05259          poly = t;
05260          for ( i = 0; i < m[1]; i++ ) t[i] = m[i];
05261          t += m[1];
05262      }
05263      AT.PolyAct = WORDDIFF(poly,term);
05264 /*
05265      If needed we convert the coefficient into a PolyRatFun and then
05266      we call poly_ratfun_normalize
05267 */
05268      if ( m[-1] == 3 && m[-2] == 1 && m[-3] == 1 ) return(0);
05269      if ( AR.PolyFunExp != 1 ) {
05270          if ( m[-1] < 0 ) { sign = -1; size = -m[-1]; } else { sign = 1; size = m[-1]; }
05271          num = m - size; size = (size-1)/2; den = num + size;
05272          sizenum = size; while ( num[sizenum-1] == 0 ) sizenum--;
05273          sizeden = size; while ( den[sizeden-1] == 0 ) sizeden--;
05274          v = m;
05275          *v++ = AR.PolyFun;
05276          *v++ = FUNHEAD + 2*(ARGHEAD+sizenum+sizeden+2);
05277 /*
05278          *v++ = MUSTCLEANPRF; */
05279          *v++ = 0;
05280          FILLFUN3(v);
05281          *v++ = ARGHEAD+2*sizenum+2;
05282          *v++ = 0;
05283          FILLARG(v);
05284          *v++ = 2*sizenum+2;
05285          for ( i = 0; i < sizenum; i++ ) *v++ = num[i];
05286          *v++ = 1;
05287          for ( i = 1; i < sizenum; i++ ) *v++ = 0;
05288          *v++ = sign*(2*sizenum+1);
05289          *v++ = ARGHEAD+2*sizeden+2;
05290          *v++ = 0;
05291          FILLARG(v);
05292          *v++ = 2*sizeden+2;
05293          for ( i = 0; i < sizeden; i++ ) *v++ = den[i];
05294          *v++ = 1;
05295          for ( i = 1; i < sizeden; i++ ) *v++ = 0;
05296          *v++ = 2*sizeden+1;
05297          w = num;
05298          i = v - m;
05299          NCOPY(w,m,i);
05300      }
05301      else {
05302          w = m-ABS(m[-1]);
05303      }
05304      *w++ = 1; *w++ = 1; *w++ = 3; *term = w - term;
05305      {
05306          WORD oldtype = AR.SortType;
05307          AR.SortType = SORTHIGHFIRST;
05308      }
05309      if ( count > 0 )
05310          poly_ratfun_normalize(BHEAD term);
05311      else
05312          ReadPolyRatFun(BHEAD term);
05313      poly_ratfun_normalize(BHEAD term);
05314
05315      oldworkpointer = AT.WorkPointer; */
05316      AR.SortType = oldtype;
05317      }
05318      goto endofit;
05319 /*
05320      #] Two arguments :
05321 */
05322      }
05323      else {
05324          DoError;;
05325          MLOCK(ErrorMessageLock);
05326          MesPrint("Illegal value for PolyFunType in PrepPoly");
05327          MUNLOCK(ErrorMessageLock);
05328          Terminate(-1);
05329      }
05330      r = term + *term;
05331      AT.PolyAct = WORDDIFF(poly,term);
05332      while ( r < m ) *poly++ = *r++;
05333      *poly++ = 1;

```

```

05334     *poly++ = 1;
05335     *poly++ = 3;
05336     *term = WORDDIF(poly,term);
05337 endofit;;
05338     return(0);
05339 }
05340
05341 /*
05342     #] PrepPoly :
05343     #[ PolyFunMul :          WORD PolyFunMul(term)
05344 */
05356 int PolyFunMul(PHEAD WORD *term)
05357 {
05358     GETBIDENTITY
05359     WORD *t, *fun1, *fun2, *t1, *t2, *m, *w, *ww, *tt1, *tt2, *tt4, *arg1, *arg2;
05360     WORD *tstop, i, dirty = 0, OldPolyFunPow = AR.PolyFunPow, minp1, minp2;
05361     WORD n1, n2, i1, i2, l1, l2, l3, l4, action = 0, noac = 0;
05362     int retval = 0;
05363     if ( AR.PolyFunType == 2 && AR.PolyFunExp == 1 ) {
05364         WORD pow = 0, pow1;
05365         t = term + 1; t1 = term + *term; t1 -= ABS(t1[-1]);
05366         w = t;
05367         while ( t < t1 ) {
05368             if ( *t != AR.PolyFun ) {
05369 SkipFun:
05370                 if ( t == w ) { t += t[1]; w = t; }
05371                 else { i = t[1]; NCOPY(w,t,i) }
05372                 continue;
05373             }
05374             pow1 = 0;
05375             t2 = t + t[1]; t += FUNHEAD;
05376             if ( *t < 0 ) {
05377                 if ( *t == -SYMBOL && t[1] == AR.PolyFunVar ) pow1++;
05378                 else if ( *t != -SNUMBER ) goto NoLegal;
05379                 t += 2;
05380             }
05381             else if ( t[0] == ARGHEAD+8 && t[ARGHEAD] == 8
05382                     && t[ARGHEAD+1] == SYMBOL && t[ARGHEAD+3] == AR.PolyFunVar
05383                     && t[ARGHEAD+5] == 1 && t[ARGHEAD+6] == 1 && t[ARGHEAD+7] == 3 ) {
05384                 pow1 += t[ARGHEAD+4];
05385                 t += *t;
05386             }
05387             else {
05388 NoLegal:
05389                 MLOCK(ErrorMessageLock);
05390                 MesPrint("Illegal term with divergence in PolyRatFun");
05391                 MesCall("PolyFunMul");
05392                 MUNLOCK(ErrorMessageLock);
05393                 Terminate(-1);
05394             }
05395             if ( *t < 0 ) {
05396                 if ( *t == -SYMBOL && t[1] == AR.PolyFunVar ) pow1--;
05397                 else if ( *t != -SNUMBER ) goto NoLegal;
05398                 t += 2;
05399             }
05400             else if ( t[0] == ARGHEAD+8 && t[ARGHEAD] == 8
05401                     && t[ARGHEAD+1] == SYMBOL && t[ARGHEAD+3] == AR.PolyFunVar
05402                     && t[ARGHEAD+5] == 1 && t[ARGHEAD+6] == 1 && t[ARGHEAD+7] == 3 ) {
05403                 pow1 -= t[ARGHEAD+4];
05404                 t += *t;
05405             }
05406             else goto NoLegal;
05407             if ( t == t2 ) pow += pow1;
05408             else goto SkipFun;
05409         }
05410         m = w;
05411         *w++ = AR.PolyFun; *w++ = 0; FILLFUN(w);
05412         if ( pow > 1 ) {
05413             *w++ = 8+ARGHEAD; *w++ = 0; FILLARG(w);
05414             *w++ = 8; *w++ = SYMBOL; *w++ = 4; *w++ = AR.PolyFunVar; *w++ = pow;
05415             *w++ = 1; *w++ = 1; *w++ = 3; *w++ = -SNUMBER; *w++ = 1;
05416         }
05417         else if ( pow == 1 ) {
05418             *w++ = -SYMBOL; *w++ = AR.PolyFunVar; *w++ = -SNUMBER; *w++ = 1;
05419         }
05420         else if ( pow < -1 ) {
05421             *w++ = -SNUMBER; *w++ = 1; *w++ = 8+ARGHEAD; *w++ = 0; FILLARG(w);
05422             *w++ = 8; *w++ = SYMBOL; *w++ = 4; *w++ = AR.PolyFunVar; *w++ = -pow;
05423             *w++ = 1; *w++ = 1; *w++ = 3;
05424         }
05425         else if ( pow == -1 ) {
05426             *w++ = -SNUMBER; *w++ = 1; *w++ = -SYMBOL; *w++ = AR.PolyFunVar;
05427         }
05428         else {
05429             *w++ = -SNUMBER; *w++ = 1; *w++ = -SNUMBER; *w++ = 1;
05430         }
05431         m[1] = w - m;

```

```

05432         *w++ = 1; *w++ = 1; *w++ = 3;
05433         *term = w - term;
05434         if ( w > AT.WorkSpace && w < AT.WorkTop ) AT.WorkPointer = w;
05435         return(0);
05436     }
05437 ReStart:
05438     if ( AR.PolyFunType == 2 && ( ( AR.PolyFunExp != 2 )
05439         || ( AR.PolyFunExp == 2 && AN.PolyNormFlag > 1 ) ) ) {
05440         WORD count1 = 0, count2 = 0, count3;
05441         WORD oldtype = AR.SortType;
05442         t = term + 1; t1 = term + *term; t1 -= ABS(t1[-1]);
05443         while ( t < t1 ) {
05444             if ( *t == AR.PolyFun ) {
05445                 if ( t[2] && dirty == 0 ) { /* Any dirty flag on? */
05446                     dirty = 1;
05447                     /* ReadPolyRatFun(BHEAD term); */
05448                     /* ToPolyFunGeneral(BHEAD term); */
05449                     poly_ratfun_normalize(BHEAD term);
05450                     if ( term[0] == 0 ) return(0);
05451                     count1 = 0;
05452                     action++;
05453                     goto ReStart;
05454                 }
05455                 t2 = t + t[1]; tt2 = t+FUNHEAD; count3 = 0;
05456                 while ( tt2 < t2 ) { count3++; NEXTARG(tt2); }
05457                 if ( count3 == 2 ) {
05458                     count1++;
05459                     if ( ( t[2] & MUSTCLEANPRF ) != 0 ) { /* Better civilize this guy */
05460                         action++;
05461                         w = AT.WorkPointer;
05462                         AR.SortType = SORTHIGHFIRST;
05463                         t2 = t + t[1]; tt2 = t+FUNHEAD;
05464                         while ( tt2 < t2 ) {
05465                             if ( *tt2 > 0 ) {
05466                                 tt4 = tt2; tt1 = tt2 + ARGHEAD; tt2 += *tt2;
05467                                 NewSort(BHEAD0);
05468                                 while ( tt1 < tt2 ) {
05469                                     i = *tt1; ww = w; NCOPY(ww,tt1,i);
05470                                     AT.WorkPointer = ww;
05471                                     Normalize(BHEAD w);
05472                                     StoreTerm(BHEAD w);
05473                                 }
05474                                 EndSort(BHEAD w,1);
05475                                 ww = w; while ( *ww ) ww += *ww;
05476                                 if ( ww-w != *tt4-ARGHEAD ) { /* Little problem */
05477                                     /*
05478                                     Solution: brute force copy
05479                                     Maybe it will never come here????
05480                                     */
05481                                     WORD *r1 = TermMalloc("PolyFunMul");
05482                                     WORD ii = (ww-w)-(tt4-ARGHEAD); /* increment */
05483                                     WORD *r2 = tt4+ARGHEAD, *r3, *r4 = r1;
05484                                     i = r2 - term; r3 = term; NCOPY(r4,r3,i);
05485                                     i = ww-w; ww = w; NCOPY(r4,ww,i);
05486                                     r3 = tt2; i = term+*term-tt2; NCOPY(r4,r3,i);
05487                                     *r1 = i = r4-r1; r4 = term; r3 = r1;
05488                                     NCOPY(r4,r3,i);
05489                                     t[1] += ii; t1 += ii; *tt4 += ii;
05490                                     tt2 = tt4 + *tt4;
05491                                     TermFree(r1,"PolyFunMul");
05492                                 }
05493                                 else {
05494                                     i = ww-w; ww = w; tt1 = tt4+ARGHEAD;
05495                                     NCOPY(tt1,ww,i);
05496                                     AT.WorkPointer = w;
05497                                 }
05498                             }
05499                             else if ( *tt2 <= -FUNCTION ) tt2++;
05500                             else tt2 += 2;
05501                         }
05502                         AR.SortType = oldtype;
05503                     }
05504                 }
05505             }
05506             t += t[1];
05507         }
05508         if ( count1 <= 1 ) { goto checkaction; }
05509         if ( AR.PolyFunExp == 1 ) {
05510             t = term + *term; t -= ABS(t[-1]);
05511             *t++ = 1; *t++ = 1; *t++ = 3; *term = t - term;
05512         }
05513     }
05514     AR.SortType = SORTHIGHFIRST;
05515     /* retval = ReadPolyRatFun(BHEAD term); */
05516     /* ToPolyFunGeneral(BHEAD term); */
05517     retval = poly_ratfun_normalize(BHEAD term);
05518     if ( *term == 0 ) return(retval);

```

```

05519         AR.SortType = oldtype;
05520     }
05521
05522     t = term + 1; t1 = term + *term; t1 -= ABS(t1[-1]);
05523     while ( t < t1 ) {
05524         if ( *t == AR.PolyFun ) {
05525             t2 = t + t[1]; tt2 = t+FUNHEAD; count3 = 0;
05526             while ( tt2 < t2 ) { count3++; NEXTARG(tt2); }
05527             if ( count3 == 2 ) {
05528                 count2++;
05529             }
05530         }
05531         t += t[1];
05532     }
05533     if ( count1 >= count2 ) {
05534         t = term + 1;
05535         while ( t < t1 ) {
05536             if ( *t == AR.PolyFun ) {
05537                 t2 = t;
05538                 t = t + t[1];
05539                 t2[2] |= (DIRTYFLAG|MUSTCLEANPRF);
05540                 t2 += FUNHEAD;
05541                 while ( t2 < t ) {
05542                     if ( *t2 > 0 ) t2[1] = DIRTYFLAG;
05543                     NEXTARG(t2);
05544                 }
05545             }
05546             else t += t[1];
05547         }
05548     }
05549
05550     w = term + *term;
05551     if ( w > AT.WorkSpace && w < AT.WorkTop ) AT.WorkPointer = w;
05552 checkaction:
05553     if ( action ) retval = action;
05554     return(retval);
05555 }
05556 retry:
05557     if ( term >= AT.WorkSpace && term+*term < AT.WorkTop )
05558         AT.WorkPointer = term + *term;
05559     GETSTOP(term,tstop);
05560     t = term+1;
05561     while ( *t != AR.PolyFun && t < tstop ) t += t[1];
05562     while ( t < tstop && *t == AR.PolyFun ) {
05563         if ( t[1] > FUNHEAD ) {
05564             if ( t[FUNHEAD] < 0 ) {
05565                 if ( t[FUNHEAD] <= -FUNCTION && t[1] == FUNHEAD+1 ) break;
05566                 if ( t[FUNHEAD] > -FUNCTION && t[1] == FUNHEAD+2 ) {
05567                     if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] == 0 ) {
05568                         *term = 0;
05569                         return(0);
05570                     }
05571                     break;
05572                 }
05573             }
05574             else if ( t[FUNHEAD] == t[1] - FUNHEAD ) break;
05575         }
05576         noac = 1;
05577         t += t[1];
05578     }
05579     if ( *t != AR.PolyFun || t >= tstop ) goto done;
05580     fun1 = t;
05581     t += t[1];
05582     while ( t < tstop && *t == AR.PolyFun ) {
05583         if ( t[1] > FUNHEAD ) {
05584             if ( t[FUNHEAD] < 0 ) {
05585                 if ( t[FUNHEAD] <= -FUNCTION && t[1] == FUNHEAD+1 ) break;
05586                 if ( t[FUNHEAD] > -FUNCTION && t[1] == FUNHEAD+2 ) {
05587                     if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] == 0 ) {
05588                         *term = 0;
05589                         return(0);
05590                     }
05591                     break;
05592                 }
05593             }
05594             else if ( t[FUNHEAD] == t[1] - FUNHEAD ) break;
05595         }
05596         noac = 1;
05597         t += t[1];
05598     }
05599     if ( *t != AR.PolyFun || t >= tstop ) goto done;
05600     fun2 = t;
05601 /*
05602     We have two functions of the proper type.
05603     Count terms (needed for the specials)
05604 */
05605     t = fun1 + FUNHEAD;

```

```

05606     if ( *t < 0 ) {
05607         n1 = 1; arg1 = AT.WorkPointer;
05608         ToGeneral(t,arg1,1);
05609         AT.WorkPointer = arg1 + *arg1;
05610     }
05611     else {
05612         t += ARGHEAD;
05613         n1 = 0; t1 = fun1 + fun1[1]; arg1 = t;
05614         while ( t < t1 ) { n1++; t += *t; }
05615     }
05616     t = fun2 + FUNHEAD;
05617     if ( *t < 0 ) {
05618         n2 = 1; arg2 = AT.WorkPointer;
05619         ToGeneral(t,arg2,1);
05620         AT.WorkPointer = arg2 + *arg2;
05621     }
05622     else {
05623         t += ARGHEAD;
05624         n2 = 0; t2 = fun2 + fun2[1]; arg2 = t;
05625         while ( t < t2 ) { n2++; t += *t; }
05626     }
05627     /*
05628     Now we can start the multiplications. We first multiply the terms
05629     without coefficients, then normalize, and finally put the coefficients
05630     in place. This is because one has often truncated series and the
05631     high powers may get killed, while their coefficients are the most
05632     expensive ones.
05633     Note: We may run into fun(-SNUMBER,value)
05634     */
05635     w = AT.WorkPointer;
05636     NewSort(BHEAD0);
05637     if ( AR.PolyFunType == 2 && AR.PolyFunExp == 2 ) {
05638         AT.TrimPower = 1;
05639     /*
05640         We have to find the lowest power in both polynomials.
05641         This will be needed to temporarily correct the AR.PolyFunPow
05642     */
05643         minp1 = MAXPOWER;
05644         for ( t1 = arg1, i1 = 0; i1 < n1; i1++, t1 += *t1 ) {
05645             if ( *t1 == 4 ) {
05646                 if ( minp1 > 0 ) minp1 = 0;
05647             }
05648             else if ( ABS(t1[*t1-1]) == (*t1-1) ) {
05649                 if ( minp1 > 0 ) minp1 = 0;
05650             }
05651             else {
05652                 if ( t1[1] == SYMBOL && t1[2] == 4 && t1[3] == AR.PolyFunVar ) {
05653                     if ( t1[4] < minp1 ) minp1 = t1[4];
05654                 }
05655                 else {
05656                     MesPrint("Illegal term in expanded polyratfun.");
05657                     goto PolyCall;
05658                 }
05659             }
05660         }
05661         minp2 = MAXPOWER;
05662         for ( t2 = arg2, i2 = 0; i2 < n2; i2++, t2 += *t2 ) {
05663             if ( *t2 == 4 ) {
05664                 if ( minp2 > 0 ) minp2 = 0;
05665             }
05666             else if ( ABS(t2[*t2-1]) == (*t2-1) ) {
05667                 if ( minp2 > 0 ) minp2 = 0;
05668             }
05669             else {
05670                 if ( t2[1] == SYMBOL && t2[2] == 4 && t2[3] == AR.PolyFunVar ) {
05671                     if ( t2[4] < minp2 ) minp2 = t2[4];
05672                 }
05673                 else {
05674                     MesPrint("Illegal term in expanded polyratfun.");
05675                     goto PolyCall;
05676                 }
05677             }
05678         }
05679         AR.PolyFunPow += minp1+minp2;
05680     }
05681     for ( t1 = arg1, i1 = 0; i1 < n1; i1++, t1 += *t1 ) {
05682     for ( t2 = arg2, i2 = 0; i2 < n2; i2++, t2 += *t2 ) {
05683         m = w;
05684         m++;
05685         GETSTOP(t1,tt1);
05686         t = t1 + 1;
05687         while ( t < tt1 ) *m++ = *t++;
05688         GETSTOP(t2,tt2);
05689         t = t2+1;
05690         while ( t < tt2 ) *m++ = *t++;
05691         *m++ = 1; *m++ = 1; *m++ = 3; *w = WORDDIF(m,w);
05692         AT.WorkPointer = m;

```

```

05693         if ( Normalize(BHEAD w) ) { LowerSortLevel(); goto PolyCall; }
05694         if ( *w ) {
05695             m = w + *w;
05696             if ( m[-1] != 3 || m[-2] != 1 || m[-3] != 1 ) {
05697                 l3 = REDLENG(m[-1]);
05698                 m -= ABS(m[-1]);
05699                 t = t1 + *t1 - 1;
05700                 l1 = REDLENG(*t);
05701                 if ( MulRat(BHEAD (UWORD *)m,l3,(UWORD *)t1,l1,(UWORD *)m,&l4) ) {
05702                     LowerSortLevel(); goto PolyCall; }
05703                 if ( AN.ncmod != 0 && TakeModulus((UWORD *)m,&l4,AC.cmod,AN.ncmod,UNPACK|AC.modmode) )
05704             {
05705                 LowerSortLevel(); goto PolyCall; }
05706                 if ( l4 == 0 ) continue;
05707                 t = t2 + *t2 - 1;
05708                 l2 = REDLENG(*t);
05709                 if ( MulRat(BHEAD (UWORD *)m,l4,(UWORD *)t2,l2,(UWORD *)m,&l3) ) {
05710                     LowerSortLevel(); goto PolyCall; }
05711                 if ( AN.ncmod != 0 && TakeModulus((UWORD *)m,&l3,AC.cmod,AN.ncmod,UNPACK|AC.modmode) )
05712             {
05713                 LowerSortLevel(); goto PolyCall; }
05714             }
05715             else {
05716                 m -= 3;
05717                 t = t1 + *t1 - 1;
05718                 l1 = REDLENG(*t);
05719                 t = t2 + *t2 - 1;
05720                 l2 = REDLENG(*t);
05721                 if ( MulRat(BHEAD (UWORD *)t1,l1,(UWORD *)t2,l2,(UWORD *)m,&l3) ) {
05722                     LowerSortLevel(); goto PolyCall; }
05723                 if ( AN.ncmod != 0 && TakeModulus((UWORD *)m,&l3,AC.cmod,AN.ncmod,UNPACK|AC.modmode) )
05724             {
05725                 LowerSortLevel(); goto PolyCall; }
05726             }
05727             if ( l3 == 0 ) continue;
05728             l3 = INCLENG(l3);
05729             m += ABS(l3);
05730             m[-1] = l3;
05731             *w = WORDDIF(m,w);
05732             AT.WorkPointer = m;
05733             if ( StoreTerm(BHEAD w) ) { LowerSortLevel(); goto PolyCall; }
05734         }
05735     }
05736     if ( EndSort(BHEAD w,0) < 0 ) goto PolyCall;
05737     AR.PolyFunPow = OldPolyFunPow;
05738     AT.TrimPower = 0;
05739     if ( *w == 0 ) {
05740         *term = 0;
05741         return(0);
05742     }
05743     t = w;
05744     while ( *t ) t += *t;
05745     AT.WorkPointer = t;
05746     n1 = WORDDIF(t,w);
05747     t1 = term;
05748     while ( t1 < fun1 ) *t++ = *t1++;
05749     t2 = t;
05750     *t++ = AR.PolyFun;
05751     *t++ = FUNHEAD+ARGHEAD+n1;
05752     *t++ = 0;
05753     FILLFUN3(t)
05754     *t++ = ARGHEAD+n1;
05755     *t++ = 0;
05756     FILLARG(t)
05757     NCOPY(t,w,n1);
05758     if ( ToFast(t2+FUNHEAD,t2+FUNHEAD) ) {
05759         if ( t2[FUNHEAD] > -FUNCTION ) t2[1] = FUNHEAD+2;
05760         else t2[FUNHEAD] = FUNHEAD+1;
05761         t = t2 + t2[1];
05762     }
05763     t1 = fun1 + fun1[1];
05764     while ( t1 < fun2 ) *t++ = *t1++;
05765     t1 = fun2 + fun2[1];
05766     t2 = term + *term;
05767     while ( t1 < t2 ) *t++ = *t1++;
05768     *AT.WorkPointer = n1 = WORDDIF(t,AT.WorkPointer);
05769     if ( n1*((LONG)sizeof(WORD)) > AM.MaxTer ) {
05770         MLOCK(ErrorMessageLock);
05771         MesPrint("Term too complex (%d words). MaxTermSize (%l words) is too small.", n1,
05772             AM.MaxTer/(LONG)sizeof(WORD) );
05773         goto PolyCall2;
05774     }
05775     m = term; t = AT.WorkPointer;
05776     NCOPY(m,t,n1);
05777     action++;
05778     goto retry;

```

```

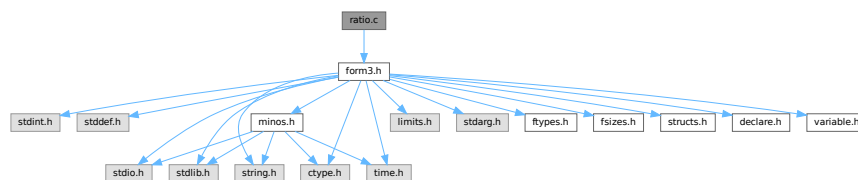
05776 done:
05777     AT.WorkPointer = term + *term;
05778     if ( action && noac ) {
05779         if ( Normalize(BHEAD term) ) goto PolyCall;
05780         AT.WorkPointer = term + *term;
05781     }
05782     return(0);
05783 PolyCall:
05784     MLOCK(ErrorMessageLock);
05785 PolyCall2:
05786     AR.PolyFunPow = OldPolyFunPow;
05787     MesCall("PolyFunMul");
05788     MUNLOCK(ErrorMessageLock);
05789     SETERROR(-1)
05790 }
05791
05792 /*
05793     #] PolyFunMul :
05794     #] Processor :
05795 */

```

4.117 ratio.c File Reference

```
#include "form3.h"
```

Include dependency graph for ratio.c:



Data Structures

- struct [ARGBUFFER](#)

Functions

- int [RatioFind](#) (PHEAD WORD *term, WORD *params)
- int [RatioGen](#) (PHEAD WORD *term, WORD *params, WORD num, WORD level)
- int [BinomGen](#) (PHEAD WORD *term, WORD level, WORD **tstops, WORD x1, WORD x2, WORD pow1, WORD pow2, WORD sign, UWORD *coef, WORD ncoef)
- int [DoSumF1](#) (PHEAD WORD *term, WORD *params, WORD replac, WORD level)
- int [Glue](#) (PHEAD WORD *term1, WORD *term2, WORD *sub, WORD insert)
- int [DoSumF2](#) (PHEAD WORD *term, WORD *params, WORD replac, WORD level)
- int [GCDfunction](#) (PHEAD WORD *term, WORD level)
- WORD * [GCDfunction3](#) (PHEAD WORD *in1, WORD *in2)
- WORD * [PutExtraSymbols](#) (PHEAD WORD *in, WORD startbuf, int *actionflag)
- WORD * [TakeExtraSymbols](#) (PHEAD WORD *in, WORD startbuf)
- WORD * [MultiplyWithTerm](#) (PHEAD WORD *in, WORD *term, WORD par)
- WORD * [TakeContent](#) (PHEAD WORD *in, WORD *term)
- int [MergeSymbolLists](#) (PHEAD WORD *old, WORD *extra, int par)
- int [MergeDotproductLists](#) (PHEAD WORD *old, WORD *extra, int par)
- WORD * [CreateExpression](#) (PHEAD WORD nexp)
- int [GCDterms](#) (PHEAD WORD *term1, WORD *term2, WORD *termout)

- int [ReadPolyRatFun](#) (PHEAD WORD *term)
- int [FromPolyRatFun](#) (PHEAD WORD *fun, WORD **numout, WORD **denout)
- WORD * [TakeSymbolContent](#) (PHEAD WORD *in, WORD *term)
- void [GCDclean](#) (PHEAD WORD *num, WORD *den)
- WORD * [PolyDiv](#) (PHEAD WORD *a, WORD *b, char *text)
- int [DIVfunction](#) (PHEAD WORD *term, WORD level, int par)
- WORD * [MULfunc](#) (PHEAD WORD *p1, WORD *p2)
- WORD * [ConvertArgument](#) (PHEAD WORD *arg, int *type)
- int [ExpandRat](#) (PHEAD WORD *fun)
- int [InvPoly](#) (PHEAD WORD *inpoly, WORD maxpow, WORD sym)

Variables

- WORD [divrem](#) [4] = { DIVFUNCTION, REMFUNCTION, INVERSEFUNCTION, MULFUNCTION }
- char * [TheErrorMessage](#) []

4.117.1 Detailed Description

A variety of routines: The ratio command for partial fractioning (rather old. Schoonschip inheritance) The sum routines.

Definition in file [ratio.c](#).

4.117.2 Function Documentation

4.117.2.1 RatioFind()

```
int RatioFind (
    PHEAD WORD * term,
    WORD * params )
```

Definition at line 62 of file [ratio.c](#).

4.117.2.2 RatioGen()

```
int RatioGen (
    PHEAD WORD * term,
    WORD * params,
    WORD num,
    WORD level )
```

Definition at line 170 of file [ratio.c](#).

4.117.2.3 BinomGen()

```
int BinomGen (
    PHEAD WORD * term,
    WORD level,
    WORD ** tstops,
    WORD x1,
    WORD x2,
    WORD pow1,
    WORD pow2,
    WORD sign,
    UWORD * coef,
    WORD ncoef )
```

Definition at line 320 of file [ratio.c](#).

4.117.2.4 DoSumF1()

```
int DoSumF1 (
    PHEAD WORD * term,
    WORD * params,
    WORD replac,
    WORD level )
```

Definition at line 395 of file [ratio.c](#).

4.117.2.5 Glue()

```
int Glue (
    PHEAD WORD * term1,
    WORD * term2,
    WORD * sub,
    WORD insert )
```

Definition at line 461 of file [ratio.c](#).

4.117.2.6 DoSumF2()

```
int DoSumF2 (
    PHEAD WORD * term,
    WORD * params,
    WORD replac,
    WORD level )
```

Definition at line 520 of file [ratio.c](#).

4.117.2.7 GCDfunction()

```
int GCDfunction (
    PHEAD WORD * term,
    WORD level )
```

Definition at line 609 of file [ratio.c](#).

4.117.2.8 GCDfunction3()

```
WORD * GCDfunction3 (
    PHEAD WORD * in1,
    WORD * in2 )
```

Definition at line 1141 of file [ratio.c](#).

4.117.2.9 PutExtraSymbols()

```
WORD * PutExtraSymbols (
    PHEAD WORD * in,
    WORD startebuf,
    int * actionflag )
```

Definition at line 1244 of file [ratio.c](#).

4.117.2.10 TakeExtraSymbols()

```
WORD * TakeExtraSymbols (
    PHEAD WORD * in,
    WORD startebuf )
```

Definition at line 1273 of file [ratio.c](#).

4.117.2.11 MultiplyWithTerm()

```
WORD * MultiplyWithTerm (
    PHEAD WORD * in,
    WORD * term,
    WORD par )
```

Definition at line 1312 of file [ratio.c](#).

4.117.2.12 TakeContent()

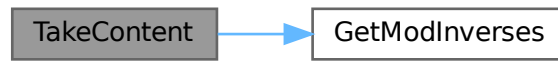
```
WORD * TakeContent (
    PHEAD WORD * in,
    WORD * term )
```

Implements part of the old ExecArg in which we take common factors from arguments with more than one term. Here the input is a sequence of terms in 'in' and the answer is a content-free sequence of terms. This sequence has been allocated by the Malloc1 routine in a call to EndSort, unless the expression was already content-free. In that case the input pointer is returned. The content is returned in term. This is supposed to be a separate allocation, made by TermMalloc in the calling routine.

Definition at line 1376 of file [ratio.c](#).

References [GetModInverses\(\)](#).

Here is the call graph for this function:



4.117.2.13 MergeSymbolLists()

```
int MergeSymbolLists (
    PHEAD WORD * old,
    WORD * extra,
    int par )
```

Definition at line 1808 of file [ratio.c](#).

4.117.2.14 MergeDotproductLists()

```
int MergeDotproductLists (
    PHEAD WORD * old,
    WORD * extra,
    int par )
```

Definition at line 1927 of file [ratio.c](#).

4.117.2.15 CreateExpression()

```
WORD * CreateExpression (
    PHEAD WORD nexp )
```

Definition at line 2020 of file [ratio.c](#).

4.117.2.16 GCDterms()

```
int GCDterms (
    PHEAD WORD * term1,
    WORD * term2,
    WORD * termout )
```

Definition at line 2076 of file [ratio.c](#).

4.117.2.17 ReadPolyRatFun()

```
int ReadPolyRatFun (
    PHEAD WORD * term )
```

Definition at line 2264 of file [ratio.c](#).

4.117.2.18 FromPolyRatFun()

```
int FromPolyRatFun (
    PHEAD WORD * fun,
    WORD ** numout,
    WORD ** denout )
```

Definition at line 2383 of file [ratio.c](#).

4.117.2.19 TakeSymbolContent()

```
WORD * TakeSymbolContent (
    PHEAD WORD * in,
    WORD * term )
```

Implements part of the old ExecArg in which we take common factors from arguments with more than one term. We allow only symbols as this code is used for the polyratfun only. We have a special routine, because the generic TakeContent does too much work and speed is at a premium here. Input: *in* is the input expression as a sequence of terms. **Output:** *term*: the content return value: the contentfree expression. it is in new allocation, made by TermMalloc. (should be in a TermMalloc space?)

Definition at line 2434 of file [ratio.c](#).

References [GetModInverses\(\)](#).

Here is the call graph for this function:



4.117.2.20 GCDclean()

```
void GCDclean (
    PHEAD WORD * num,
    WORD * den )
```

Definition at line 2644 of file [ratio.c](#).

4.117.2.21 PolyDiv()

```
WORD * PolyDiv (
    PHEAD WORD * a,
    WORD * b,
    char * text )
```

Definition at line [2741](#) of file [ratio.c](#).

4.117.2.22 DIVfunction()

```
int DIVfunction (
    PHEAD WORD * term,
    WORD level,
    int par )
```

Definition at line [2788](#) of file [ratio.c](#).

4.117.2.23 MULfunc()

```
WORD * MULfunc (
    PHEAD WORD * p1,
    WORD * p2 )
```

Definition at line [2957](#) of file [ratio.c](#).

4.117.2.24 ConvertArgument()

```
WORD * ConvertArgument (
    PHEAD WORD * arg,
    int * type )
```

Definition at line [3018](#) of file [ratio.c](#).

4.117.2.25 ExpandRat()

```
int ExpandRat (
    PHEAD WORD * fun )
```

Definition at line [3113](#) of file [ratio.c](#).

4.117.2.26 InvPoly()

```
int InvPoly (
    PHEAD WORD * inpoly,
    WORD maxpow,
    WORD sym )
```

Definition at line [3592](#) of file [ratio.c](#).

4.117.3 Variable Documentation

4.117.3.1 divrem

WORD divrem[4] = { DIVFUNCTION, REMFUNCTION, INVERSEFUNCTION, MULFUNCTION }

Definition at line 2786 of file [ratio.c](#).

4.117.3.2 TheErrorMessage

char* TheErrorMessage[]

Initial value:

```
= {
    "PolyRatFun not of a type that FORM will expand: incorrect variable inside."
    , "Division by zero in PolyRatFun encountered in ExpandRat."
    , "Irregular code in PolyRatFun encountered in ExpandRat."
    , "Called from ExpandRat."
    , "WorkSpace overflow. Change parameter WorkSpace in setup file?"
}
```

Definition at line 3105 of file [ratio.c](#).

4.118 ratio.c

[Go to the documentation of this file.](#)

```
00001
00008 /* # [ License : */
00009 /*
00010 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00011 * When using this file you are requested to refer to the publication
00012 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013 * This is considered a matter of courtesy as the development was paid
00014 * for by FOM the Dutch physics granting agency and we would like to
00015 * be able to track its scientific use to convince FOM of its value
00016 * for the community.
00017 *
00018 * This file is part of FORM.
00019 *
00020 * FORM is free software: you can redistribute it and/or modify it under the
00021 * terms of the GNU General Public License as published by the Free Software
00022 * Foundation, either version 3 of the License, or (at your option) any later
00023 * version.
00024 *
00025 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00028 * details.
00029 *
00030 * You should have received a copy of the GNU General Public License along
00031 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00032 */
00033 /* # [ License : */
00034 /*
00035 * # [ Includes : ratio.c
00036 */
00037
00038 #include "form3.h"
00039
00040 /*
00041 * # [ Includes :
00042 * # [ Ratio :
00043
00044 * These are the special operations regarding simple polynomials.
00045 * The first and most needed is the partial fractioning expansion.
00046 * Ratio,x1,x2,x3
00047
00048 * The files belonging to the ratio command serve also as a good example
00049 * of how to implement a new operation.
00050
```

```

00051      # [ RatioFind :
00052
00053      The routine that should locate the need for a ratio command.
00054      If located the corresponding symbols are removed and the
00055      operational parameters are loaded. A subexpression pointer
00056      is inserted and the code for success is returned.
00057
00058      params points at the compiler output block defined in RatioComp.
00059
00060  */
00061
00062  int RatioFind(PHEAD WORD *term, WORD *params)
00063  {
00064      GETBIDENTITY
00065      WORD *t, *m, *r;
00066      WORD x1, x2, i;
00067      WORD *y1, *y2, n1 = 0, n2 = 0;
00068      x1 = params[3];
00069      x2 = params[4];
00070      m = t = term;
00071      m += *m;
00072      m -= ABS(m[-1]);
00073      t++;
00074      if ( t < m ) do {
00075          if ( *t == SYMBOL ) {
00076              y1 = 0;
00077              y2 = 0;
00078              r = t + t[1];
00079              m = t + 2;
00080              do {
00081                  if ( *m == x1 ) { y1 = m; n1 = m[1]; }
00082                  else if ( *m == x2 ) { y2 = m; n2 = m[1]; }
00083                  m += 2;
00084              } while ( m < r );
00085              if ( !y1 || !y2 || ( n1 > 0 && n2 > 0 ) ) return(0);
00086              m -= 2;
00087              if ( y1 > y2 ) { r = y1; y1 = y2; y2 = r; }
00088              *y2 = *m; y2[1] = m[1];
00089              m -= 2;
00090              *y1 = *m; y1[1] = m[1];
00091              i = WORDDIFF(m,t);
00092              #if SUBEXPSIZE > 6
00093              We have to revise the code for the second case.
00094              #endif
00095              if ( i > 2 ) {          /* Subexpression fits exactly */
00096                  t[1] = i;
00097                  y1 = term+*term;
00098                  y2 = y1+SUBEXPSIZE-4;
00099                  r = m+4;
00100                  while ( y1 > r ) --y2 = --y1;
00101                  *m++ = SUBEXPRESSION;
00102                  *m++ = SUBEXPSIZE;
00103                  *m++ = -1;
00104                  *m++ = 1;
00105                  *m++ = DUMMYBUFFER;
00106                  FILLSUB(m)
00107                  *term += SUBEXPSIZE-4;
00108              }
00109              else {                  /* All symbols are gone. Rest has to be moved */
00110                  m -= 2;
00111                  *m++ = SUBEXPRESSION;
00112                  *m++ = SUBEXPSIZE;
00113                  *m++ = -1;
00114                  *m++ = 1;
00115                  *m++ = DUMMYBUFFER;
00116                  FILLSUB(m)
00117                  t = term;
00118                  t += *t;
00119                  *term += SUBEXPSIZE-6;
00120                  r = m + 6-SUBEXPSIZE;
00121                  do { *m++ = *r++; } while ( r < t );
00122              }
00123              t = AT.TMout;          /* Load up the TM out array for the generator */
00124              *t++ = 7;
00125              *t++ = RATIO;
00126              *t++ = x1;
00127              *t++ = x2;
00128              *t++ = params[5];
00129              *t++ = n1;
00130              *t++ = n2;
00131              return(1);
00132          }
00133          t += t[1];
00134      } while ( t < m );
00135      return(0);
00136  }
00137

```



```

00138 /*
00139      #] RatioFind :
00140      #[ RatioGen :
00141
00142      The algorithm:
00143      x1^-n1*x2^n2 ==> x2 --> x1 + x3
00144      x1^n1*x2^-n2 ==> x1 --> x2 - x3
00145      x1^-n1*x2^-n2 ==>
00146
00147          +sum(i=0,n1-1){(-1)^i*binom(n2-1+i,n2-1)
00148              *x3^-(n2+i)*x1^-(n1-i)}
00149          +sum(i=0,n2-1){(-1)^(n1)*binom(n1-1+i,n1-1)
00150              *x3^-(n1+i)*x2^-(n2-i)}
00151
00152      Actually there is an amount of arbitrariness in the first two
00153      formulae and the replacement x2 -> x1 + x3 could be made 'by hand'.
00154      It is better to use the nontrivial 'minimal change' formula:
00155
00156      x1^-n1*x2^n2:   if ( n1 >= n2 ) {
00157                      +sum(i=0,n2){x3^i*x1^-(n1-n2+i)*binom(n2,i)}
00158                      }
00159                      else {
00160                          sum(i=0,n2-n1){x2^(n2-n1-i)*x3^i*binom(n1-1+i,n1-1)}
00161                          +sum(i=0,n1-1){x3^(n2-i)*x1^-(n1-i)*binom(n2,i)}
00162                      }
00163      x1^n1*x2^-n2:   Same but x3 -> -x3.
00164
00165      The contents of the AT.TMout/params array are:
00166      length,type,x1,x2,x3,n1,n2
00167
00168 */
00169
00170 int RatioGen(PHEAD WORD *term, WORD *params, WORD num, WORD level)
00171 {
00172     GETBIDENTITY
00173     WORD *t, *m;
00174     WORD *tstops[3];
00175     WORD n1, n2, i, j;
00176     WORD x1,x2,x3;
00177     UWORD *coef;
00178     WORD ncoef, sign = 0;
00179     coef = (UWORD *)AT.WorkPointer;
00180     t = term;
00181     tstops[2] = m = t + *t;
00182     m -= ABS(m[-1]);
00183     t++;
00184     do {
00185         if ( *t == SUBEXPRESSION && t[2] == num ) break;
00186         t += t[1];
00187     } while ( t < m );
00188     tstops[0] = t;
00189     tstops[1] = t + t[1];
00190
00191     /* Copying to termout will be from term to tstop1, then the induced part
00192     and finally from tstop2 to tstop3
00193
00194     Now separate the various cases:
00195
00196 */
00197     t = params + 2;
00198     x1 = *t++;
00199     x2 = *t++;
00200     x3 = *t++;
00201     n1 = *t++;
00202     n2 = *t++;
00203     if ( n1 > 0 ) { /* Flip the variables and indicate -x3 */
00204         n2 = -n2;
00205         sign = 1;
00206         i = n1; n1 = n2; n2 = i;
00207         i = x1; x1 = x2; x2 = i;
00208         goto PosNeg;
00209     }
00210     else if ( n2 > 0 ) {
00211         n1 = -n1;
00212 PosNeg:
00213         if ( n2 <= n1 ) { /* x1 -> x2 + x3 */
00214             *coef = 1;
00215             ncoef = 1;
00216             AT.WorkPointer = (WORD *) (coef + 1);
00217             j = n2;
00218             for ( i = 0; i <= n2; i++ ) {
00219                 if ( BinomGen(BHEAD term,level,tstops,x1,x3,n2-n1-i,i,sign&i
00220                     ,coef,ncoef) ) goto RatioCall;
00221                 if ( i < n2 ) {
00222                     if ( Product(coef,&ncoef,j) ) goto RatioCall;
00223                     if ( Quotient(coef,&ncoef,i+1) ) goto RatioCall;
00224                     j--;

```

```

00225         AT.WorkPointer = (WORD *) (coef + ABS(ncoef));
00226     }
00227 }
00228 AT.WorkPointer = (WORD *) (coef);
00229 return(0);
00230 }
00231 else {
00232 /*
00233     sum(i=0,n2-n1){x2^(n2-n1-i)*x3^i*binom(n1-1+i,n1-1)}
00234 +sum(i=0,n1-1){x3^(n2-i)*x1^(n1-i)*binom(n2,i)}
00235 */
00236     *coef = 1;
00237     ncoef = 1;
00238     AT.WorkPointer = (WORD *) (coef + 1);
00239     j = n2 - n1;
00240     for ( i = 0; i <= j; i++ ) {
00241         if ( BinomGen(BHEAD term,level,tstops,x2,x3,n2-n1-i,i,sign&i
00242             ,coef,ncoef) ) goto RatioCall;
00243         if ( i < j ) {
00244             if ( Product(coef,&ncoef,n1+i) ) goto RatioCall;
00245             if ( Quotient(coef,&ncoef,i+1) ) goto RatioCall;
00246             AT.WorkPointer = (WORD *) (coef + ABS(ncoef));
00247         }
00248     }
00249     *coef = 1;
00250     ncoef = 1;
00251     AT.WorkPointer = (WORD *) (coef + 1);
00252     j = n1-1;
00253     for ( i = 0; i <= j; i++ ) {
00254         if ( BinomGen(BHEAD term,level,tstops,x1,x3,i-n1,n2-i,sign&(n2-i)
00255             ,coef,ncoef) ) goto RatioCall;
00256         if ( i < j ) {
00257             if ( Product(coef,&ncoef,n2-i) ) goto RatioCall;
00258             if ( Quotient(coef,&ncoef,i+1) ) goto RatioCall;
00259             AT.WorkPointer = (WORD *) (coef + ABS(ncoef));
00260         }
00261     }
00262     AT.WorkPointer = (WORD *) (coef);
00263     return(0);
00264 }
00265 }
00266 else {
00267     n2 = -n2;
00268     n1 = -n1;
00269 /*
00270     +sum(i=0,n1-1){(-1)^i*binom(n2-1+i,n2-1)
00271     *x3^(n2+i)*x1^(n1-i)}
00272 +sum(i=0,n2-1){(-1)^(n1)*binom(n1-1+i,n1-1)
00273     *x3^(n1+i)*x2^(n2-i)}
00274 */
00275     *coef = 1;
00276     ncoef = 1;
00277     AT.WorkPointer = (WORD *) (coef + 1);
00278     j = n1-1;
00279     for ( i = 0; i <= j; i++ ) {
00280         if ( BinomGen(BHEAD term,level,tstops,x1,x3,i-n1,-n2-i,i&1
00281             ,coef,ncoef) ) goto RatioCall;
00282         if ( i < j ) {
00283             if ( Product(coef,&ncoef,n2+i) ) goto RatioCall;
00284             if ( Quotient(coef,&ncoef,i+1) ) goto RatioCall;
00285             AT.WorkPointer = (WORD *) (coef + ABS(ncoef));
00286         }
00287     }
00288     *coef = 1;
00289     ncoef = 1;
00290     AT.WorkPointer = (WORD *) (coef + 1);
00291     j = n2-1;
00292     for ( i = 0; i <= j; i++ ) {
00293         if ( BinomGen(BHEAD term,level,tstops,x2,x3,i-n2,-n1-i,n1&1
00294             ,coef,ncoef) ) goto RatioCall;
00295         if ( i < j ) {
00296             if ( Product(coef,&ncoef,n1+i) ) goto RatioCall;
00297             if ( Quotient(coef,&ncoef,i+1) ) goto RatioCall;
00298             AT.WorkPointer = (WORD *) (coef + ABS(ncoef));
00299         }
00300     }
00301     AT.WorkPointer = (WORD *) (coef);
00302     return(0);
00303 }
00304 }
00305 RatioCall:
00306     MLOCK(ErrorMessageLock);
00307     MesCall("RatioGen");
00308     MUNLOCK(ErrorMessageLock);
00309     SETERROR(-1)
00310 }
00311

```

```

00312 /*
00313     #] RatioGen :
00314     #[ BinomGen :
00315
00316     Routine for the generation of terms in a binomialtype expansion.
00317
00318 */
00319
00320 int BinomGen(PHEAD WORD *term, WORD level, WORD **tstops, WORD x1, WORD x2,
00321             WORD pow1, WORD pow2, WORD sign, UWORD *coef, WORD ncoef)
00322 {
00323     GETBIDENTITY
00324     WORD *t, *r;
00325     WORD *termout;
00326     WORD k;
00327     termout = AT.WorkPointer;
00328     t = termout;
00329     r = term;
00330     do { *t++ = *r++; } while ( r < tstops[0] );
00331     *t++ = SYMBOL;
00332     if ( pow2 == 0 ) {
00333         if ( pow1 == 0 ) t--;
00334         else { *t++ = 4; *t++ = x1; *t++ = pow1; }
00335     }
00336     else if ( pow1 == 0 ) {
00337         *t++ = 4; *t++ = x2; *t++ = pow2;
00338     }
00339     else {
00340         *t++ = 6; *t++ = x1; *t++ = pow1; *t++ = x2; *t++ = pow2;
00341     }
00342     *t++ = LNUMBER;
00343     *t++ = ABS(ncoef) + 3;
00344     *t = ncoef;
00345     if ( sign ) *t = -*t;
00346     t++;
00347     ncoef = ABS(ncoef);
00348     for ( k = 0; k < ncoef; k++ ) *t++ = coef[k];
00349     r = tstops[1];
00350     do { *t++ = *r++; } while ( r < tstops[2] );
00351     *termout = WORDDIF(t,termout);
00352     AT.WorkPointer = t;
00353     if ( AT.WorkPointer > AT.WorkTop ) {
00354         MLOCK(ErrorMessageLock);
00355         MesWork();
00356         MUNLOCK(ErrorMessageLock);
00357         return(-1);
00358     }
00359     *AN.RepPoint = 1;
00360     AR.expchanged = 1;
00361     if ( Generator(BHEAD termout,level) ) {
00362         MLOCK(ErrorMessageLock);
00363         MesCall("BinomGen");
00364         MUNLOCK(ErrorMessageLock);
00365         SETERROR(-1)
00366     }
00367     AT.WorkPointer = termout;
00368     return(0);
00369 }
00370
00371 /*
00372     #] BinomGen :
00373     #] Ratio :
00374     #[ Sum :
00375     #[ DoSumF1 :
00376
00377     Routine expands a sum_ function.
00378     Its arguments are:
00379     The term in which the function occurs.
00380     The parameter list:
00381         length of parameter field
00382         function number (SUMNUM1)
00383         number of the symbol that is loop parameter
00384         min value
00385         max value
00386         increment
00387     the number of the subexpression to be removed
00388     the level in the generation tree.
00389
00390     Note that the insertion of the loop parameter in the argument
00391     is done via the regular wildcard substitution mechanism.
00392
00393 */
00394
00395 int DoSumF1(PHEAD WORD *term, WORD *params, WORD replac, WORD level)
00396 {
00397     GETBIDENTITY
00398     WORD *termout, *t, extractbuff = AT.TMbuff;

```

```

00399     WORD isum, ival, iinc;
00400     LONG from;
00401     CBUF *C;
00402     ival = params[3];
00403     iinc = params[5];
00404     if ( ( iinc > 0 && params[4] >= ival )
00405         || ( iinc < 0 && params[4] <= ival ) ) {
00406         isum = (params[4] - ival)/iinc + 1;
00407     }
00408     else return(0);
00409     termout = AT.WorkPointer;
00410     AT.WorkPointer = (WORD *)((UBYTE *) (AT.WorkPointer)) + AM.MaxTer;
00411     if ( AT.WorkPointer > AT.WorkTop ) {
00412         MLOCK(ErrorMessageLock);
00413         MesWork();
00414         MUNLOCK(ErrorMessageLock);
00415         return(-1);
00416     }
00417     t = term + 1;
00418     while ( *t != SUBEXPRESSION || t[2] != replac || t[4] != extractbuff )
00419         t += t[1];
00420     C = cbuf+t[4];
00421     t += SUBEXPSIZE;
00422     if ( params[2] < 0 ) {
00423         while ( *t != INDTOIND || t[2] != -params[2] ) t += t[1];
00424         *t = INDTOIND;
00425     }
00426     else {
00427         while ( *t > SYMTOSUB || t[2] != params[2] ) t += t[1];
00428         *t = SYMTONUM;
00429     }
00430     do {
00431         t[3] = ival;
00432         from = C->rhs[replac] - C->Buffer;
00433         while ( C->Buffer[from] ) {
00434             if ( InsertTerm(BHEAD term,replac,extractbuff,C->Buffer+from,termout,0) < 0 ) goto
SumFlCall;
00435             AT.WorkPointer = termout + *termout;
00436             if ( Generator(BHEAD termout,level) < 0 ) goto SumFlCall;
00437             from += C->Buffer[from];
00438         }
00439         ival += iinc;
00440     } while ( --isum > 0 );
00441     AT.WorkPointer = termout;
00442     return(0);
00443 SumFlCall:
00444     MLOCK(ErrorMessageLock);
00445     MesCall("DoSumFl");
00446     MUNLOCK(ErrorMessageLock);
00447     SETERROR(-1)
00448 }
00449
00450 /*
00451     #] DoSumFl :
00452     #[ Glue :
00453
00454     Routine multiplies two terms. The second term is subject
00455     to the wildcard substitutions in sub.
00456     Output in the first term. This routine is a variation on
00457     the routine InsertTerm.
00458
00459 */
00460
00461 int Glue(PHEAD WORD *term1, WORD *term2, WORD *sub, WORD insert)
00462 {
00463     GETBIDENTITY
00464     UWORD *coef;
00465     WORD ncoef, *t, *t1, *t2, i, nc2, nc3, old, newer;
00466     coef = (UWORD *) (TermMalloc("Glue"));
00467     t = term1;
00468     t += *t;
00469     i = t[-1];
00470     t -= ABS(i);
00471     old = WORDDIF(t,term1);
00472     ncoef = REDLENG(i);
00473     if ( i < 0 ) i = -i;
00474     i--;
00475     t1 = t;
00476     t2 = (WORD *)coef;
00477     while ( --i >= 0 ) *t2++ = *t1++;
00478     i = *--t;
00479     nc2 = WildFill(BHEAD t,term2,sub);
00480     *t = i;
00481     t += nc2;
00482     nc2 = t[-1];
00483     t -= ABS(nc2);
00484     newer = WORDDIF(t,term1);

```

```

00485     if ( MulRat (BHEAD (UWORD *)t, REDLENG(nc2), coef, ncoef, (UWORD *)t, &nc3) ) {
00486         MLOCK(ErrorMessageLock);
00487         MesCall("Glue");
00488         MUNLOCK(ErrorMessageLock);
00489         TermFree(coef, "Glue");
00490         SETERROR(-1)
00491     }
00492     i = (ABS(nc3))*2;
00493     t += i++;
00494     *t++ = (nc3 >= 0)?i:-i;
00495     *term1 = WORDDIF(t, term1);
00496 /*
00497     Switch the new piece with the old tail, so that noncommuting
00498     variables get into their proper spot.
00499 */
00500     i = old - insert;
00501     t1 = t;
00502     t2 = term1+insert;
00503     NCOPY(t1, t2, i);
00504     i = newer - old;
00505     t1 = term1+insert;
00506     t2 = term1+old;
00507     NCOPY(t1, t2, i);
00508     t2 = t;
00509     i = old - insert;
00510     NCOPY(t1, t2, i);
00511     TermFree(coef, "Glue");
00512     return(0);
00513 }
00514
00515 /*
00516     #] Glue :
00517     #[ DoSumF2 :
00518 */
00519
00520 int DoSumF2(PHEAD WORD *term, WORD *params, WORD replac, WORD level)
00521 {
00522     GETBIDENTITY
00523     WORD *termout, *t, *from, *sub, *to, extractbuff = AT.TMbuff;
00524     WORD isum, ival, iinc, insert, i;
00525     CBUF *C;
00526     ival = params[3];
00527     iinc = params[5];
00528     if ( ( iinc > 0 && params[4] >= ival )
00529         || ( iinc < 0 && params[4] <= ival ) ) {
00530         isum = (params[4] - ival)/iinc + 1;
00531     }
00532     else return(0);
00533     termout = AT.WorkPointer;
00534     AT.WorkPointer = (WORD *)(((UBYTE *) (AT.WorkPointer)) + AM.MaxTer);
00535     if ( AT.WorkPointer > AT.WorkTop ) {
00536         MLOCK(ErrorMessageLock);
00537         MesWork();
00538         MUNLOCK(ErrorMessageLock);
00539         return(-1);
00540     }
00541     t = term + 1;
00542     while ( *t != SUBEXPRESSION || t[2] != replac || t[4] != extractbuff ) t += t[1];
00543     insert = WORDDIF(t, term);
00544
00545     from = term;
00546     to = termout;
00547     while ( from < t ) *to++ = *from++;
00548     from += t[1];
00549     sub = term + *term;
00550     while ( from < sub ) *to++ = *from++;
00551     *termout -= t[1];
00552
00553     sub = t;
00554     C = cbuf+t[4];
00555     t += SUBEXPSIZE;
00556     if ( params[2] < 0 ) {
00557         while ( *t != INDTOIND || t[2] != -params[2] ) t += t[1];
00558         *t = INDTOIND;
00559     }
00560     else {
00561         while ( *t > SYMTOSUB || t[2] != params[2] ) t += t[1];
00562         *t = SYMTONUM;
00563     }
00564     t[3] = ival;
00565     for(;;) {
00566         AT.WorkPointer = termout + *termout;
00567         to = AT.WorkPointer;
00568         if ( ( to + *termout ) > AT.WorkTop ) {
00569             MLOCK(ErrorMessageLock);
00570             MesWork();
00571             MUNLOCK(ErrorMessageLock);

```

```

00572         return(-1);
00573     }
00574     from = termout;
00575     i = *termout;
00576     NCOPY(to, from, i);
00577     from = AT.WorkPointer;
00578     AT.WorkPointer = to;
00579     if ( Generator(BHEAD from, level) < 0 ) goto SumF2Call;
00580     if ( --isum <= 0 ) break;
00581     ival += iinc;
00582     t[3] = ival;
00583     if ( Glue(BHEAD termout, C->rhs[replac], sub, insert) < 0 ) goto SumF2Call;
00584 }
00585 AT.WorkPointer = termout;
00586 return(0);
00587 SumF2Call:
00588 MLOCK(ErrorMessageLock);
00589 MesCall("DoSumF2");
00590 MUNLOCK(ErrorMessageLock);
00591 SETERROR(-1)
00592 }
00593
00594 /*
00595     #] DoSumF2 :
00596     #] Sum :
00597     #[ GCDfunction :
00598     #[ GCDfunction :
00599 */
00600
00601 typedef struct {
00602     WORD *buffer;
00603     DOLLARS dollar;
00604     LONG size;
00605     int type;
00606     int dummy;
00607 } ARGBUFFER;
00608
00609 int GCDfunction(PHEAD WORD *term, WORD level)
00610 {
00611     GETBIDENTITY
00612     WORD *t, *tstop, *tf, *termout, *tin, *tout, *m, *mnext, *mstop, *mm;
00613     int todo, i, ii, j, istart, sign = 1, action = 0;
00614     WORD firstshort = 0, firstvalue = 0, gcdisone = 0, mlength, tlength, newlength;
00615     WORD totargs = 0, numargs, argsdone = 0, *mh, oldvall, *g, *gcdout = 0;
00616     WORD *argl, *arg2;
00617     UWORD x1, x2, x3;
00618     LONG args;
00619     #if ( FUNHEAD > 4 )
00620     WORD sh[FUNHEAD+5];
00621     #else
00622     WORD sh[9];
00623     #endif
00624     DOLLARS d;
00625     ARGBUFFER *abuf = 0, ab;
00626 /*
00627     #[ Find Function. Count arguments :
00628
00629     First find the proper function
00630 */
00631     t = term + *term; tlength = t[-1];
00632     tstop = t - ABS(tlength);
00633     t = term + 1;
00634     while ( t < tstop ) {
00635         if ( *t != GCDFUNCTION ) { t += t[1]; continue; }
00636         todo = 1; totargs = 0;
00637         tf = t + FUNHEAD;
00638         while ( tf < t + t[1] ) {
00639             totargs++;
00640             if ( *tf > 0 && tf[1] != 0 ) todo = 0;
00641             NEXTARG(tf);
00642         }
00643         if ( todo ) break;
00644         t += t[1];
00645     }
00646     if ( t >= tstop ) {
00647         MLOCK(ErrorMessageLock);
00648         MesPrint("Internal error. Indicated gcd_ function not encountered.");
00649         MUNLOCK(ErrorMessageLock);
00650         Terminate(-1);
00651     }
00652     WantAddPointers(totargs);
00653     args = AT.pWorkPointer; AT.pWorkPointer += totargs;
00654 /*
00655     #] Find Function. Count arguments :
00656     #[ Do short arguments :
00657
00658     The function we need, in agreement with TestSub, is now in t

```

```

00659     Make first a compilation of the short arguments (except $-s and expressions)
00660     to see whether we need to do much work.
00661     This means that after this scan we can ignore all short arguments with
00662     the exception of unevaluated $-s and expressions.
00663  */
00664     numargs = 0;
00665     firstshort = 0;
00666     tf = t + FUNHEAD;
00667     while ( tf < t + t[1] ) {
00668         if ( *tf == -SNUMBER && tf[1] == 0 ) { NEXTARG(tf); continue; }
00669         if ( *tf > 0 || *tf == -DOLLAREXPRESSION || *tf == -EXPRESSION ) {
00670             AT.pWorkspace[args+numargs++] = tf;
00671             NEXTARG(tf); continue;
00672         }
00673         if ( firstshort == 0 ) {
00674             firstshort = *tf;
00675             if ( *tf <= -FUNCTION ) { firstvalue = -( *tf ); }
00676             else { firstvalue = tf[1]; }
00677             NEXTARG(tf);
00678             argsdone++;
00679             continue;
00680         }
00681         else if ( *tf != firstshort ) {
00682             if ( *tf != -INDEX && *tf != -VECTOR && *tf != -MINVECTOR ) {
00683                 argsdone++; gcdisone = 1; break;
00684             }
00685             if ( firstshort != -INDEX && firstshort != -VECTOR && firstshort != -MINVECTOR ) {
00686                 argsdone++; gcdisone = 1; break;
00687             }
00688             if ( tf[1] != firstvalue ) {
00689                 argsdone++; gcdisone = 1; break;
00690             }
00691             if ( *t == -MINVECTOR ) { firstshort = -VECTOR; }
00692             if ( firstshort == -MINVECTOR ) { firstshort = -VECTOR; }
00693         }
00694         else if ( *tf > -FUNCTION && *tf != -SNUMBER && tf[1] != firstvalue ) {
00695             argsdone++; gcdisone = 1; break;
00696         }
00697         if ( *tf == -SNUMBER && firstvalue != tf[1] ) {
00698             /*
00699                 make a new firstvalue which is gcd_(firstvalue,tf[1])
00700             */
00701             if ( firstvalue == 1 || tf[1] == 1 ) { gcdisone = 1; break; }
00702             if ( firstvalue < 0 && tf[1] < 0 ) {
00703                 x1 = -firstvalue; x2 = -tf[1]; sign = -1;
00704             }
00705             else {
00706                 x1 = ABS(firstvalue); x2 = ABS(tf[1]); sign = 1;
00707             }
00708             while ( ( x3 = x1%x2 ) != 0 ) { x1 = x2; x2 = x3; }
00709             firstvalue = ((WORD)x2)*sign;
00710             argsdone++;
00711             if ( firstvalue == 1 ) { gcdisone = 1; break; }
00712         }
00713         NEXTARG(tf);
00714     }
00715     termout = AT.WorkPointer;
00716     AT.WorkPointer = (WORD *)(((UBYTE *) (AT.WorkPointer)) + AM.MaxTer);
00717     if ( AT.WorkPointer > AT.WorkTop ) {
00718         MLOCK(ErrorMessageLock);
00719         MesWork();
00720         MUNLOCK(ErrorMessageLock);
00721         return(-1);
00722     }
00723  /*
00724     #] Do short arguments :
00725     #[ Do trivial GCD :
00726
00727     Copy head
00728  */
00729     i = t - term; tin = term; tout = termout;
00730     NCOPY(tout,tin,i);
00731     if ( gcdisone || ( firstshort == -SNUMBER && firstvalue == 1 ) ) {
00732         sign = 1;
00733     gcdone:
00734         tin += t[1]; tstop = term + *term;
00735         while ( tin < tstop ) *tout++ = *tin++;
00736         *termout = tout - termout;
00737         if ( sign < 0 ) tout[-1] = -tout[-1];
00738         AT.WorkPointer = tout;
00739         if ( argsdone && Generator(BHEAD termout,level) < 0 ) goto CalledFrom;
00740         AT.WorkPointer = termout;
00741         AT.pWorkPointer = args;
00742         return(0);
00743     }
00744  /*
00745     #] Do trivial GCD :

```

```

00746     #[ Do short argument GCD :
00747     */
00748     if ( numargs == 0 ) {      /* basically we are done */
00749 doshort:
00750         sign = 1;
00751         if ( firstshort == 0 ) goto gcdone;
00752         if ( firstshort == -SNUMBER ) {
00753             *tout++ = SNUMBER; *tout++ = 4; *tout++ = firstvalue; *tout++ = 1;
00754             goto gcdone;
00755         }
00756         else if ( firstshort == -SYMBOL ) {
00757             *tout++ = SYMBOL; *tout++ = 4; *tout++ = firstvalue; *tout++ = 1;
00758             goto gcdone;
00759         }
00760         else if ( firstshort == -VECTOR || firstshort == -INDEX ) {
00761             *tout++ = INDEX; *tout++ = 3; *tout++ = firstvalue; goto gcdone;
00762         }
00763         else if ( firstshort == -MINVECTOR ) {
00764             sign = -1;
00765             *tout++ = INDEX; *tout++ = 3; *tout++ = firstvalue; goto gcdone;
00766         }
00767         else if ( firstshort <= -FUNCTION ) {
00768             *tout++ = firstvalue; *tout++ = FUNHEAD; FILLFUN(tout);
00769             goto gcdone;
00770         }
00771         else {
00772             MLOCK(ErrorMessageLock);
00773             MesPrint("Internal error. Illegal short argument in GCDfunction.");
00774             MUNLOCK(ErrorMessageLock);
00775             Terminate(-1);
00776         }
00777     }
00778     /*
00779     #[ Do short argument GCD :
00780     #[ Convert short argument :
00781
00782     Now we allocate space for the arguments in general notation.
00783     First the special one if there were short arguments
00784     */
00785     if ( firstshort ) {
00786         switch ( firstshort ) {
00787             case -SNUMBER:
00788                 sh[0] = 4; sh[1] = ABS(firstvalue); sh[2] = 1;
00789                 if ( firstvalue < 0 ) sh[3] = -3;
00790                 else sh[3] = 3;
00791                 sh[4] = 0;
00792                 break;
00793             case -MINVECTOR:
00794             case -VECTOR:
00795             case -INDEX:
00796                 sh[0] = 8; sh[1] = INDEX; sh[2] = 3; sh[3] = firstvalue;
00797                 sh[4] = 1; sh[5] = 1;
00798                 if ( firstshort == -MINVECTOR ) sh[6] = -3;
00799                 else sh[6] = 3;
00800                 sh[7] = 0;
00801                 break;
00802             case -SYMBOL:
00803                 sh[0] = 8; sh[1] = SYMBOL; sh[2] = 4; sh[3] = firstvalue; sh[4] = 1;
00804                 sh[5] = 1; sh[6] = 1; sh[7] = 3; sh[8] = 0;
00805                 break;
00806             default:
00807                 sh[0] = FUNHEAD+4; sh[1] = firstshort; sh[2] = FUNHEAD;
00808                 for ( i = 2; i < FUNHEAD; i++ ) sh[i+1] = 0;
00809                 sh[FUNHEAD+1] = 1; sh[FUNHEAD+2] = 1; sh[FUNHEAD+3] = 3; sh[FUNHEAD+4] = 0;
00810                 break;
00811         }
00812     }
00813     /*
00814     #[ Convert short argument :
00815     #[ Sort arguments :
00816
00817     Now we should sort the arguments in a way that the dollars and the
00818     expressions come last. That way we may never need them.
00819     */
00820     for ( i = 1; i < numargs; i++ ) {
00821         for ( ii = i; ii > 0; ii-- ) {
00822             arg1 = AT.pWorkSpace[args+ii];
00823             arg2 = AT.pWorkSpace[args+ii-1];
00824             if ( *arg1 < 0 ) {
00825                 if ( *arg2 < 0 ) {
00826                     if ( *arg1 == -EXPRESSION ) break;
00827                     if ( *arg2 == -DOLLAREXPRESSION ) break;
00828                     AT.pWorkSpace[args+ii] = arg2;
00829                     AT.pWorkSpace[args+ii-1] = arg1;
00830                 }
00831                 else break;
00832             }

```



```

00833         else if ( *arg2 < 0 ) {
00834             AT.pWorkspace[args+ii] = arg2;
00835             AT.pWorkspace[args+ii-1] = arg1;
00836         }
00837         else {
00838             if ( *arg1 > *arg2 ) {
00839                 AT.pWorkspace[args+ii] = arg2;
00840                 AT.pWorkspace[args+ii-1] = arg1;
00841             }
00842             else break;
00843         }
00844     }
00845 }
00846 /*
00847 #] Sort arguments :
00848 #[ There is a single term argument :
00849 */
00850 if ( firstshort ) {
00851     mh = sh; istart = 0;
00852 oneterm:;
00853     for ( i = istart; i < numargs; i++ ) {
00854         arg1 = AT.pWorkspace[args+i];
00855         if ( *arg1 > 0 ) {
00856             oldvall = arg1[*arg1]; arg1[*arg1] = 0;
00857             m = arg1+ARGHEAD;
00858             while ( *m ) {
00859                 GCDterms(BHEAD mh,m,mh); m += *m;
00860                 if ( mh[0] == 4 && mh[1] == 1 && mh[2] == 1 && mh[3] == 3 ) {
00861                     gcdisone = 1; sign = 1; arg1[*arg1] = oldvall; goto gcdone;
00862                 }
00863             }
00864             arg1[*arg1] = oldvall;
00865         }
00866         else if ( *arg1 == -DOLLEXPRESSION ) {
00867             if ( ( d = DolToTerms(BHEAD arg1[1]) ) != 0 ) {
00868                 m = d->where;
00869                 while ( *m ) {
00870                     GCDterms(BHEAD mh,m,mh); m += *m;
00871                     argsdone++;
00872                     if ( mh[0] == 4 && mh[1] == 1 && mh[2] == 1 && mh[3] == 3 ) {
00873                         gcdisone = 1; sign = 1;
00874                         if ( d->factors ) M_free(d->factors,"Dollar factors");
00875                         M_free(d,"Copy of dollar variable"); goto gcdone;
00876                     }
00877                 }
00878                 if ( d->factors ) M_free(d->factors,"Dollar factors");
00879                 M_free(d,"Copy of dollar variable");
00880             }
00881         }
00882         else {
00883             mm = CreateExpression(BHEAD arg1[1]);
00884             m = mm;
00885             while ( *m ) {
00886                 GCDterms(BHEAD mh,m,mh); m += *m;
00887                 argsdone++;
00888                 if ( mh[0] == 4 && mh[1] == 1 && mh[2] == 1 && mh[3] == 3 ) {
00889                     gcdisone = 1; sign = 1; M_free(mm,"CreateExpression"); goto gcdone;
00890                 }
00891             }
00892             M_free(mm,"CreateExpression");
00893         }
00894     }
00895     if ( firstshort ) {
00896         if ( mh[0] == 4 ) {
00897             firstshort = -SNUMBER; firstvalue = mh[1] * (mh[3]/3);
00898         }
00899         else if ( mh[1] == SYMBOL ) {
00900             firstshort = -SYMBOL; firstvalue = mh[3];
00901         }
00902         else if ( mh[1] == INDEX ) {
00903             firstshort = -INDEX; firstvalue = mh[3];
00904             if ( mh[6] == -3 ) firstshort = -MINVECTOR;
00905         }
00906         else if ( mh[1] >= FUNCTION ) {
00907             firstshort = -mh[1]; firstvalue = mh[1];
00908         }
00909         goto doshort;
00910     }
00911     else {
00912 /*
00913         We have a GCD that is only a single term.
00914         Paste it in and combine the coefficients.
00915 */
00916         mh[mh[0]] = 0;
00917         mm = mh;
00918         ii = 0;
00919         goto multiterms;

```

```

00920     }
00921 }
00922 /*
00923 Now we have only regular arguments.
00924 But some have not yet been expanded.
00925 Check whether there are proper long arguments and if so if there is
00926 one with just a single term
00927 */
00928 for ( i = 0; i < numargs; i++ ) {
00929     arg1 = AT.pWorkSpace[args+i];
00930     if ( *arg1 > 0 && arg1[ARGHEAD]+ARGHEAD == *arg1 ) {
00931 /*
00932         We have an argument with a single term
00933 */
00934         if ( i != 0 ) {
00935             arg2 = AT.pWorkSpace[args];
00936             AT.pWorkSpace[args] = arg1;
00937             AT.pWorkSpace[args+1] = arg2;
00938         }
00939         m = mh = AT.WorkPointer;
00940         mm = arg1+ARGHEAD; i = *mm;
00941         NCOPY(m,mm,i);
00942         AT.WorkPointer = m;
00943         istart = 1;
00944         argsdone++;
00945         goto oneterm;
00946     }
00947 }
00948 /*
00949 #] There is a single term argument :
00950 #[ Expand $ and expr :
00951
00952 We have: 1: regular multiterm arguments
00953          2: dollars
00954          3: expressions.
00955 The sum of them is numargs. Their addresses are in args. The problem is
00956 that expansion will lead to allocations that we have to return and all
00957 these allocations are different in nature.
00958 */
00959 action = 1;
00960 abuf = (ARGBUFFER *)Malloc1(numargs*sizeof(ARGBUFFER),"argbuffer");
00961 for ( i = 0; i < numargs; i++ ) {
00962     arg1 = AT.pWorkSpace[args+i];
00963     if ( *arg1 > 0 ) {
00964         m = (WORD *)Malloc1(*arg1*sizeof(WORD),"argbuffer type 0");
00965         abuf[i].buffer = m;
00966         abuf[i].type = 0;
00967         mm = arg1+ARGHEAD;
00968         j = *arg1-ARGHEAD;
00969         abuf[i].size = j;
00970         if ( j ) argsdone++;
00971         NCOPY(m,mm,j);
00972         *m = 0;
00973     }
00974     else if ( *arg1 == -DOLLAREXPRESSION ) {
00975         d = DolToTerms(BHEAD arg1[1]);
00976         abuf[i].buffer = d->where;
00977         abuf[i].type = 1;
00978         abuf[i].dollar = d;
00979         m = abuf[i].buffer;
00980         if ( *m ) argsdone++;
00981         while ( *m ) m+= *m;
00982         abuf[i].size = m-abuf[i].buffer;
00983     }
00984     else if ( *arg1 == -EXPRESSION ) {
00985         abuf[i].buffer = CreateExpression(BHEAD arg1[1]);
00986         abuf[i].type = 2;
00987         m = abuf[i].buffer;
00988         if ( *m ) argsdone++;
00989         while ( *m ) m+= *m;
00990         abuf[i].size = m-abuf[i].buffer;
00991     }
00992     else {
00993         MLOCK(ErrorMessageLock);
00994         MesPrint("What argument is this?");
00995         MUNLOCK(ErrorMessageLock);
00996         goto CalledFrom;
00997     }
00998 }
00999 for ( i = 0; i < numargs; i++ ) {
01000     arg1 = abuf[i].buffer;
01001     if ( *arg1 == 0 ) {}
01002     else if ( arg1[*arg1] == 0 ) {
01003 /*
01004         After expansion there is an argument with a single term
01005 */
01006         ab = abuf[i]; abuf[i] = abuf[0]; abuf[0] = ab;

```

```

01007         mh = abuf[0].buffer;
01008         for ( j = 1; j < numargs; j++ ) {
01009             m = abuf[j].buffer;
01010             while ( *m ) {
01011                 GCDterms(BHEAD mh,m,mh); m += *m;
01012                 argsdone++;
01013                 if ( mh[0] == 4 && mh[1] == 1 && mh[2] == 1 && mh[3] == 3 ) {
01014                     gcdisone = 1; sign = 1; break;
01015                 }
01016             }
01017             if ( *m ) break;
01018         }
01019         mm = mh + *mh; if ( mm[-1] < 0 ) { sign = -1; mm[-1] = -mm[-1]; }
01020         mstop = mm - mm[-1]; m = mh+1; mlength = mm[-1];
01021         while ( tin < t ) *tout++ = *tin++;
01022         while ( m < mstop ) *tout++ = *m++;
01023         tin += tin[1];
01024         while ( tin < tstop ) *tout++ = *tin++;
01025         tlength = REDLENG(tlength);
01026         mlength = REDLENG(mlength);
01027         if ( MulRat(BHEAD (UWORD *)tstop,tlength,(UWORD *)mstop,mlength,
01028             (UWORD *)tout,&newlength) < 0 ) goto CalledFrom;
01029         mlength = INCLENG(newlength);
01030         tout += ABS(mlength);
01031         tout[-1] = mlength*sign;
01032         *termout = tout - termout;
01033         AT.WorkPointer = tout;
01034         if ( argsdone && Generator(BHEAD termout,level) < 0 ) goto CalledFrom;
01035         goto cleanup;
01036     }
01037 }
01038 /*
01039     There are only arguments with more than one term.
01040     We order them by size to make the computations as easy as possible.
01041 */
01042 for ( i = 1; i < numargs; i++ ) {
01043     for ( ii = i; ii > 0; ii-- ) {
01044         if ( abuf[ii-1].size <= abuf[ii].size ) break;
01045         ab = abuf[ii-1]; abuf[ii-1] = abuf[ii]; abuf[ii] = ab;
01046     }
01047 }
01048 /*
01049     #] Expand $ and expr :
01050     #[ Multiterm subexpressions :
01051 */
01052 ii = 0;
01053 gcdout = abuf[ii].buffer;
01054 for ( i = 0; i < numargs; i++ ) {
01055     if ( abuf[i].buffer[0] ) { gcdout = abuf[i].buffer; ii = i; i++; argsdone++; break; }
01056 }
01057 for ( ; i < numargs; i++ ) {
01058     if ( abuf[i].buffer[0] ) {
01059         g = GCDfunction3(BHEAD gcdout,abuf[i].buffer);
01060         argsdone++;
01061         if ( gcdout != abuf[ii].buffer ) M_free(gcdout,"gcdout");
01062         gcdout = g;
01063         if ( gcdout[*gcdout] == 0 && gcdout[0] == 4 && gcdout[1] == 1
01064             && gcdout[2] == 1 && gcdout[3] == 3 ) break;
01065     }
01066 }
01067 mm = gcdout;
01068 multiterms:;
01069 tlength = REDLENG(tlength);
01070 while ( *mm ) {
01071     tin = term; tout = termout; while ( tin < t ) *tout++ = *tin++;
01072     tin += t[1];
01073     mnext = mm + *mm; mlength = mnext[-1]; mstop = mnext - ABS(mlength);
01074     mm++;
01075     while ( mm < mstop ) *tout++ = *mm++;
01076     while ( tin < tstop ) *tout++ = *tin++;
01077     mlength = REDLENG(mlength);
01078     if ( MulRat(BHEAD (UWORD *)tstop,tlength,(UWORD *)mm,mlength,
01079         (UWORD *)tout,&newlength) < 0 ) goto CalledFrom;
01080     mlength = INCLENG(newlength);
01081     tout += ABS(mlength);
01082     tout[-1] = mlength;
01083     *termout = tout - termout;
01084     AT.WorkPointer = tout;
01085     if ( argsdone && Generator(BHEAD termout,level) < 0 ) goto CalledFrom;
01086     mm = mnext; /* next term */
01087 }
01088 if ( action && ( gcdout != abuf[ii].buffer ) ) M_free(gcdout,"gcdout");
01089 /*
01090     #] Multiterm subexpressions :
01091     #[ Cleanup :
01092 */
01093 cleanup:;

```

```

01094     if ( action ) {
01095         for ( i = 0; i < numargs; i++ ) {
01096             if ( abuf[i].type == 0 ) { M_free(abuf[i].buffer,"argbuffer type 0"); }
01097             else if ( abuf[i].type == 1 ) {
01098                 d = abuf[i].dollar;
01099                 if ( d->factors ) M_free(d->factors,"Dollar factors");
01100                 M_free(d,"Copy of dollar variable");
01101             }
01102             else if ( abuf[i].type == 2 ) { M_free(abuf[i].buffer,"CreateExpression"); }
01103         }
01104         M_free(abuf,"argbuffer");
01105     }
01106 /*
01107     #] Cleanup :
01108 */
01109     AT.pWorkPointer = args;
01110     AT.WorkPointer = termout;
01111     return(0);
01112
01113 CalledFrom:
01114     MLOCK(ErrorMessageLock);
01115     MesCall("GCDfunction");
01116     MUNLOCK(ErrorMessageLock);
01117     SETERROR(-1)
01118     return(-1);
01119 }
01120
01121 /*
01122     #] GCDfunction :
01123     #[ GCDfunction3 :
01124
01125     Finds the GCD of the two arguments which are buffers with terms.
01126     In principle the first buffer can have only one term.
01127
01128     If both buffers have more than one term, we need to replace all
01129     non-symbolic objects by generated symbols and substitute that back
01130     afterwards. The rest we leave to the powerful routines.
01131     Philosophical problem: What do we do with GCD_(x/z+y,x+y*z) ?
01132
01133     Method:
01134     If we have only negative powers of z and no positive powers we let
01135     the EXTRASYMBOLS do their job. When mixed, multiply the arguments with
01136     the negative powers with enough powers of z to eliminate the negative powers.
01137     The DENOMINATOR function is always eliminated with the mechanism as we
01138     cannot tell whether there are positive powers of its contents.
01139 */
01140 WORD *GCDfunction3(PHEAD WORD *in1, WORD *in2)
01141 {
01142     GETBIDENTITY
01143     WORD oldsorttype = AR.SortType, *ow = AT.WorkPointer;;
01144     WORD *t, *tt, *gcdout, *term1, *term2, *confree1, *confree2, *gcdout1, *proper1, *proper2;
01145     int i, actionflag1, actionflag2;
01146     WORD startebuf = cbuf[AT.ebufnum].numrhs;
01147     WORD tryterm1, tryterm2;
01148     if ( in2[*in2] == 0 ) { t = in1; in1 = in2; in2 = t; }
01149     if ( in1[*in1] == 0 ) { /* First input with only one term */
01150         gcdout = (WORD *)Malloc1((*in1+1)*sizeof(WORD),"gcdout");
01151         i = *in1; t = gcdout; tt = in1; NCOPY(t,tt,i); *t = 0;
01152         t = in2;
01153         while ( *t ) {
01154             GCDterms(BHEAD gcdout,t,gcdout);
01155             if ( gcdout[0] == 4 && gcdout[1] == 1
01156                 && gcdout[2] == 1 && gcdout[3] == 3 ) break;
01157             t += *t;
01158         }
01159         AT.WorkPointer = ow;
01160         return(gcdout);
01161     }
01162 }
01163 /*
01164     We need to take out the content from the two expressions
01165     and determine their GCD. This plays with the negative powers!
01166 */
01167     AR.SortType = SORTHIGHFIRST;
01168     term1 = TermMalloc("GCDfunction3-a");
01169     term2 = TermMalloc("GCDfunction3-b");
01170
01171     confree1 = TakeContent(BHEAD in1,term1);
01172     tryterm1 = AN.tryterm; AN.tryterm = 0;
01173     confree2 = TakeContent(BHEAD in2,term2);
01174     tryterm2 = AN.tryterm; AN.tryterm = 0;
01175 /*
01176     confree1 = TakeSymbolContent(BHEAD in1,term1);
01177     confree2 = TakeSymbolContent(BHEAD in2,term2);
01178 */
01179     GCDterms(BHEAD term1,term2,term1);
01180     TermFree(term2,"GCDfunction3-b");

```

```

01181 /*
01182     Now we have to replace all non-symbols and symbols to a negative power
01183     by extra symbols.
01184 */
01185 if ( ( proper1 = PutExtraSymbols(BHEAD confree1,startebuf,&actionflag1) ) == 0 ) goto CalledFrom;
01186 if ( confree1 != in1 ) {
01187     if ( tryterm1 ) { TermFree(confree1,"TakeContent"); }
01188     else { M_free(confree1,"TakeContent"); }
01189 }
01190 /*
01191     TermFree(confree1,"TakeSymbolContent");
01192 */
01193 if ( ( proper2 = PutExtraSymbols(BHEAD confree2,startebuf,&actionflag2) ) == 0 ) goto CalledFrom;
01194 if ( confree2 != in2 ) {
01195     if ( tryterm2 ) { TermFree(confree2,"TakeContent"); }
01196     else { M_free(confree2,"TakeContent"); }
01197 }
01198 /*
01199     TermFree(confree2,"TakeSymbolContent");
01200 */
01201 /*
01202     And now the real work:
01203 */
01204 gcdout1 = poly_gcd(BHEAD proper1,proper2,0);
01205 M_free(proper1,"PutExtraSymbols");
01206 M_free(proper2,"PutExtraSymbols");
01207
01208 AR.SortType = oldsorttype;
01209 if ( actionflag1 || actionflag2 ) {
01210     if ( ( gcdout = TakeExtraSymbols(BHEAD gcdout1,startebuf) ) == 0 ) goto CalledFrom;
01211     M_free(gcdout1,"gcdout");
01212 }
01213 else {
01214     gcdout = gcdout1;
01215 }
01216
01217 cbuf[AT.ebufnum].numrhs = startebuf;
01218 /*
01219     Now multiply gcdout by term1
01220 */
01221 if ( term1[0] != 4 || term1[3] != 3 || term1[1] != 1 || term1[2] != 1 ) {
01222     AN.tryterm = -1;
01223     if ( ( gcdout1 = MultiplyWithTerm(BHEAD gcdout,term1,2) ) == 0 ) goto CalledFrom;
01224     AN.tryterm = 0;
01225     M_free(gcdout,"gcdout");
01226     gcdout = gcdout1;
01227 }
01228 TermFree(term1,"GCDfunction3-a");
01229 AT.WorkPointer = ow;
01230 return(gcdout);
01231 CalledFrom:
01232 AN.tryterm = 0;
01233 MLOCK(ErrorMessageLock);
01234 MesCall("GCDfunction3");
01235 MUNLOCK(ErrorMessageLock);
01236 return(0);
01237 }
01238
01239 /*
01240     #] GCDfunction3 :
01241     #[ PutExtraSymbols :
01242 */
01243
01244 WORD *PutExtraSymbols(PHEAD WORD *in,WORD startebuf,int *actionflag)
01245 {
01246     WORD *termout = AT.WorkPointer;
01247     int action;
01248     *actionflag = 0;
01249     NewSort(BHEAD0);
01250     while ( *in ) {
01251         if ( ( action = LocalConvertToPoly(BHEAD in,termout,startebuf,0) ) < 0 ) {
01252             LowerSortLevel();
01253             goto CalledFrom;
01254         }
01255         if ( action > 0 ) *actionflag = 1;
01256         StoreTerm(BHEAD termout);
01257         in += *in;
01258     }
01259     if ( EndSort(BHEAD (WORD *)((void *)(&termout)),2) < 0 ) goto CalledFrom;
01260     return(termout);
01261 CalledFrom:
01262     MLOCK(ErrorMessageLock);
01263     MesCall("PutExtraSymbols");
01264     MUNLOCK(ErrorMessageLock);
01265     return(0);
01266 }
01267

```

```

01268 /*
01269      #] PutExtraSymbols :
01270      #[ TakeExtraSymbols :
01271 */
01272
01273 WORD *TakeExtraSymbols(PHEAD WORD *in,WORD startebuf)
01274 {
01275     CBUF *C = cbuf+AC.cbunum;
01276     CBUF *CC = cbuf+AT.ebunum;
01277     WORD *oldworkpointer = AT.WorkPointer, *termout;
01278
01279     termout = AT.WorkPointer;
01280     NewSort(BHEAD0);
01281     while ( *in ) {
01282         if ( ConvertFromPoly(BHEAD
in,termout,numxsymbol,CC->numrhs-startebuf+numxsymbol,startebuf-numxsymbol,1) <= 0 ) {
01283             LowerSortLevel();
01284             goto CalledFrom;
01285         }
01286         in += *in;
01287         AT.WorkPointer = termout + *termout;
01288     /*
01289         ConvertFromPoly leaves terms with subexpressions. Hence:
01290     */
01291         if ( Generator(BHEAD termout,C->numlhs) ) {
01292             LowerSortLevel();
01293             goto CalledFrom;
01294         }
01295     }
01296     AT.WorkPointer = oldworkpointer;
01297     if ( EndSort(BHEAD (WORD *)((void *)(&termout)),2) < 0 ) goto CalledFrom;
01298     return(termout);
01299
01300 CalledFrom:
01301     MLOCK(ErrorMessageLock);
01302     MesCall("TakeExtraSymbols");
01303     MUNLOCK(ErrorMessageLock);
01304     return(0);
01305 }
01306
01307 /*
01308      #] TakeExtraSymbols :
01309      #[ MultiplyWithTerm :
01310 */
01311
01312 WORD *MultiplyWithTerm(PHEAD WORD *in, WORD *term, WORD par)
01313 {
01314     WORD *termout, *t, *tt, *tstop, *ttstop;
01315     WORD length, length1, length2;
01316     WORD oldsorttype = AR.SortType;
01317     COMPARE oldcompareroutine = (COMPARE) (AR.CompareRoutine);
01318     AR.CompareRoutine = (COMPAREDUMMY) (&CompareSymbols);
01319
01320     if ( par == 0 || par == 2 ) AR.SortType = SORTHIGHFIRST;
01321     else AR.SortType = SORTLOWFIRST;
01322     termout = AT.WorkPointer;
01323     NewSort(BHEAD0);
01324     while ( *in ) {
01325         tt = termout + 1;
01326         tstop = in + *in; tstop -= ABS(tstop[-1]); t = in + 1;
01327         while ( t < tstop ) *t++ = *t++;
01328         ttstop = term + *term; ttstop -= ABS(ttstop[-1]); t = term + 1;
01329         while ( t < ttstop ) *t++ = *t++;
01330         length1 = REDLENG(in[*in-1]); length2 = REDLENG(term[*term-1]);
01331         if ( MulRat(BHEAD (UWORD *)tstop,length1,
01332             (UWORD *)ttstop,length2,(UWORD *)tt,&length) ) goto CalledFrom;
01333         length = INCLENG(length);
01334         tt += ABS(length); tt[-1] = length;
01335         *termout = tt - termout;
01336         SymbolNormalize(termout);
01337         StoreTerm(BHEAD termout);
01338         in += *in;
01339     }
01340     if ( par == 2 ) {
01341     /*
01342         if ( AN.tryterm == 0 ) AN.tryterm = 1; */
01343         AN.tryterm = 0; /* For now */
01344         if ( EndSort(BHEAD (WORD *)((void *)(&termout)),2) < 0 ) goto CalledFrom;
01345     }
01346     else {
01347         if ( EndSort(BHEAD termout,1) < 0 ) goto CalledFrom;
01348     }
01349     AR.CompareRoutine = (COMPAREDUMMY)oldcompareroutine;
01350
01351     AR.SortType = oldsorttype;
01352     return(termout);
01353 }

```

```

01354 CalledFrom:
01355     MLOCK(ErrorMessageLock);
01356     MesCall("MultiplyWithTerm");
01357     MUNLOCK(ErrorMessageLock);
01358     return(0);
01359 }
01360
01361 /*
01362     #] MultiplyWithTerm :
01363     #[ TakeContent :
01364 */
01376 WORD *TakeContent(PHEAD WORD *in, WORD *term)
01377 {
01378     GETBIDENTITY
01379     WORD *t, *tstop, *tcom, *tout, *tstore, *r, *rstop, *m, *mm, *w, *ww, *wterm;
01380     WORD *tnext, *tt, *tterm, code[2];
01381     WORD *inp, a, *den;
01382     int i, j, k, action = 0, sign;
01383     UWORD *GCDbuffer, *GCDbuffer2, *LCMbuffer, *LCMbuffer2, *ap;
01384     WORD GCDlen, GCDlen2, LCMlen, LCMlen2, length, redlength, len1, len2;
01385     tout = tstore = term+1;
01386 /*
01387     #[ INDEX :
01388 */
01389     t = in;
01390     tnext = t + *t;
01391     tstop = tnext-ABS(tnext[-1]);
01392     t++;
01393     while ( t < tstop ) {
01394         if ( *t == INDEX ) {
01395             i = t[1]; NCOPY(tout,t,i); break;
01396         }
01397         else t += t[1];
01398     }
01399     if ( tout > tstore ) { /* There are indices in the first term */
01400         t = tnext;
01401         while ( *t ) {
01402             tnext = t + *t;
01403             rstop = tnext - ABS(tnext[-1]);
01404             r = t+1;
01405             if ( r == rstop ) goto noindices;
01406             while ( r < rstop ) {
01407                 if ( *r != INDEX ) { r += r[1]; continue; }
01408                 m = tstore+2;
01409                 while ( m < tout ) {
01410                     for ( i = 2; i < r[1]; i++ ) {
01411                         if ( *m == r[i] ) break;
01412                         if ( *m > r[i] ) continue;
01413                         mm = m+1;
01414                         while ( mm < tout ) { mm[-1] = mm[0]; mm++; }
01415                         tout--; tstore[1]--; m--;
01416                         break;
01417                     }
01418                     m++;
01419                 }
01420             }
01421             if ( r >= rstop || tout <= tstore+2 ) {
01422                 tout = tstore; break;
01423             }
01424         }
01425         if ( tout > tstore+2 ) { /* Now we have to take out what is in tstore */
01426             t = in; w = in;
01427             while ( *t ) {
01428                 wterm = w;
01429                 tnext = t + *t; t++; w++;
01430                 while ( *t != INDEX ) { i = t[1]; NCOPY(w,t,i); }
01431                 tt = t + t[1]; t += 2; r = tstore+2; ww = w; *w++ = INDEX; w++;
01432                 while ( r < tout && t < tt ) {
01433                     if ( *r > *t ) { *w++ = *t++; }
01434                     else if ( *r == *t ) { r++; t++; }
01435                     else goto CalledFrom;
01436                 }
01437                 if ( r < tout ) goto CalledFrom;
01438                 while ( t < tt ) *w++ = *t++;
01439                 ww[1] = w - ww;
01440                 if ( ww[1] == 2 ) w = ww;
01441                 while ( t < tnext ) *w++ = *t++;
01442                 *wterm = w - wterm;
01443             }
01444             *w = 0;
01445         }
01446     noindices:
01447         tstore = tout;
01448     }
01449 /*
01450     #] INDEX :
01451     #[ VECTOR/DELTA :

```

```

01452 */
01453 code[0] = VECTOR; code[1] = DELTA;
01454 for ( k = 0; k < 2; k++ ) {
01455     t = in;
01456     tnext = t + *t;
01457     tstop = tnext-ABS(tnext[-1]);
01458     t++;
01459     while ( t < tstop ) {
01460         if ( *t == code[k] ) {
01461             i = t[1]; NCOPY(tout,t,i); break;
01462         }
01463         else t += t[1];
01464     }
01465     if ( tout > tstore ) { /* There are vectors in the first term */
01466         t = tnext;
01467         while ( *t ) {
01468             tnext = t + *t;
01469             rstop = tnext - ABS(tnext[-1]);
01470             r = t+1;
01471             if ( r == rstop ) { tstore = tout; goto novectors; }
01472             while ( r < rstop ) {
01473                 if ( *r != code[k] ) { r += r[1]; continue; }
01474                 m = tstore+2;
01475                 while ( m < tout ) {
01476                     for ( i = 2; i < r[1]; i += 2 ) {
01477                         if ( *m == r[i] && m[1] == r[i+1] ) break;
01478                         if ( *m > r[i] || ( *m == r[i] && m[1] > r[i+1] ) ) continue;
01479                         mm = m+2;
01480                         while ( mm < tout ) { mm[-2] = mm[0]; mm[-1] = mm[1]; mm += 2; }
01481                         tout -= 2; tstore[1] -= 2; m -= 2;
01482                         break;
01483                     }
01484                     m += 2;
01485                 }
01486             }
01487             if ( r >= rstop || tout <= tstore+2 ) {
01488                 tout = tstore; break;
01489             }
01490         }
01491         if ( tout > tstore+2 ) { /* Now we have to take out what is in tstore */
01492             t = in; w = in;
01493             while ( *t ) {
01494                 wterm = w;
01495                 tnext = t + *t; t++; w++;
01496                 while ( *t != code[k] ) { i = t[1]; NCOPY(w,t,i); }
01497                 tt = t + t[1]; t += 2; r = tstore+2; ww = w; *w++ = code[k]; w++;
01498                 while ( r < tout && t < tt ) {
01499                     if ( ( *r > *t ) || ( *r == *t && r[1] > t[1] ) )
01500                         { *w++ = *t++; *w++ = *t++; }
01501                     else if ( *r == *t && r[1] == t[1] ) { r += 2; t += 2; }
01502                     else goto CalledFrom;
01503                 }
01504                 if ( r < tout ) goto CalledFrom;
01505                 while ( t < tt ) *w++ = *t++;
01506                 ww[1] = w - ww;
01507                 if ( ww[1] == 2 ) w = ww;
01508                 while ( t < tnext ) *w++ = *t++;
01509                 *wterm = w - wterm;
01510             }
01511             *w = 0;
01512         }
01513         tstore = tout;
01514     }
01515 }
01516 novectors;;
01517 /*
01518 #] VECTOR/DELTA :
01519 #[ FUNCTIONS :
01520 */
01521 t = in;
01522 tnext = t + *t;
01523 tstop = tnext-ABS(tnext[-1]);
01524 t++;
01525 tcom = 0;
01526 while ( t < tstop ) {
01527     if ( *t >= FUNCTION ) {
01528         if ( functions[*t-FUNCTION].commute ) {
01529             if ( tcom == 0 ) { tcom = tstore; }
01530             else {
01531                 for ( i = 0; i < t[1]; i++ ) {
01532                     if ( t[i] != tcom[i] ) {
01533                         MLOCK(ErrorMessageLock);
01534                         MesPrint("GCD or factorization of more than one noncommuting object not
allowed");
01535                         MUNLOCK(ErrorMessageLock);
01536                         goto CalledFrom;
01537                     }

```



```

01538         }
01539     }
01540 }
01541     i = t[1]; NCOPY(tstore,t,i);
01542 }
01543     else t += t[1];
01544 }
01545 if ( tout > tstore ) {
01546     t = tnext;
01547     while ( *t ) {
01548         tnext = t + *t; tstop = tnext - ABS(tnext[-1]); t++;
01549         if ( t == tstop ) goto nofunctions;
01550         r = tstore;
01551         while ( r < tout ) {
01552             tt = t;
01553             while ( tt < tstop ) {
01554                 for ( i = 0; i < r[1]; i++ ) {
01555                     if ( r[i] != tt[i] ) break;
01556                 }
01557                 if ( i == r[1] ) { r += r[1]; goto nextrl; }
01558             }
01559             /*
01560              Not encountered in this term. Scratch from list
01561             */
01562             m = r; mm = r + r[1];
01563             while ( mm < tout ) *m++ = *mm++;
01564             tout = m;
01565 nextrl:;
01566         }
01567         if ( tout <= tstore ) break;
01568         t += *t;
01569     }
01570 }
01571 if ( tout > tstore ) {
01572     /*
01573      Now we have one or more functions left that are common in all terms.
01574      Take them out. We do this one by one.
01575     */
01576     r = tstore;
01577     while ( r < tout ) {
01578         t = in; ww = in; w = ww+1;
01579         while ( *t ) {
01580             tnext = t + *t;
01581             t++;
01582             for(;;) {
01583                 for ( i = 0; i < r[1]; i++ ) {
01584                     if ( t[i] != r[i] ) {
01585                         j = t[1]; NCOPY(w,t,j);
01586                         break;
01587                     }
01588                 }
01589                 if ( i == r[1] ) {
01590                     t += t[1];
01591                     while ( t < tnext ) *w++ = *t++;
01592                     *ww = w - ww;
01593                     break;
01594                 }
01595             }
01596         }
01597         r += r[1];
01598         *w = 0;
01599     }
01600 nofunctions:
01601     tstore = tout;
01602 }
01603 /*
01604  #] FUNCTIONS :
01605  #[ SYMBOL :
01606
01607  We make a list of symbols and their minimal powers.
01608  This includes negative powers. In the end we have to multiply by the
01609  inverse of this list. That takes out all negative powers and leaves
01610  things ready for further processing.
01611  */
01612 tterm = AT.WorkPointer; tt = tterm+1;
01613 tout[0] = SYMBOL; tout[1] = 2;
01614 t = in;
01615 tnext = t + *t; tstop = tnext - ABS(tnext[-1]); t++;
01616 while ( t < tstop ) {
01617     if ( *t == SYMBOL ) {
01618         for ( i = 0; i < t[1]; i++ ) tout[i] = t[i];
01619         break;
01620     }
01621     t += t[1];
01622 }
01623 t = tnext;
01624 while ( *t ) {

```

```

01625         tnext = t + *t; tstop = tnext - ABS(tnext[-1]); t++;
01626         if ( t == tstop ) {
01627             tout[1] = 2;
01628             break;
01629         }
01630         else {
01631             while ( t < tstop ) {
01632                 if ( *t == SYMBOL ) {
01633                     MergeSymbolLists(BHEAD tout,t,-1);
01634                     break;
01635                 }
01636                 t += t[1];
01637             }
01638             t = tnext;
01639         }
01640     }
01641     if ( tout[1] > 2 ) {
01642         t = tout;
01643         tt[0] = t[0]; tt[1] = t[1];
01644         for ( i = 2; i < t[1]; i += 2 ) {
01645             tt[i] = t[i]; tt[i+1] = -t[i+1];
01646         }
01647         tt += tt[1];
01648         tout += tout[1];
01649         action++;
01650     }
01651     /*
01652     #] SYMBOL :
01653     #[ DOTPRODUCT :
01654
01655     We make a list of dotproducts and their minimal powers.
01656     This includes negative powers. In the end we have to multiply by the
01657     inverse of this list. That takes out all negative powers and leaves
01658     things ready for further processing.
01659     */
01660     tout[0] = DOTPRODUCT; tout[1] = 2;
01661     t = in;
01662     while ( *t ) {
01663         tnext = t + *t; tstop = tnext - ABS(tnext[-1]); t++;
01664         if ( t == tstop ) {
01665             tout[1] = 2;
01666             break;
01667         }
01668         while ( t < tstop ) {
01669             if ( *t == DOTPRODUCT ) {
01670                 MergeDotproductLists(BHEAD tout,t,-1);
01671                 break;
01672             }
01673             t += t[1];
01674         }
01675         t = tnext;
01676     }
01677     if ( tout[1] > 2 ) {
01678         t = tout;
01679         tt[0] = t[0]; tt[1] = t[1];
01680         for ( i = 2; i < t[1]; i += 3 ) {
01681             tt[i] = t[i]; tt[i+1] = t[i+1]; tt[i+2] = -t[i+2];
01682         }
01683         tt += tt[1];
01684         tout += tout[1];
01685         action++;
01686     }
01687     /*
01688     #] DOTPRODUCT :
01689     #[ Coefficient :
01690
01691     Now we have to collect the GCD of the numerators
01692     and the LCM of the denominators.
01693     */
01694     AT.WorkPointer = tt;
01695     if ( AN.cmod != 0 ) {
01696         WORD x, ix, ip;
01697         t = in; tnext = t + *t; tstop = tnext - ABS(tnext[-1]);
01698         x = tstop[0];
01699         if ( tnext[-1] < 0 ) x += AC.cmod[0];
01700         if ( GetModInverses(x, (WORD) (AN.cmod[0]), &ix, &ip) ) goto CalledFrom;
01701         *tout++ = x; *tout++ = 1; *tout++ = 3;
01702         *tt++ = ix; *tt++ = 1; *tt++ = 3;
01703     }
01704     else {
01705         GCDBuffer = NumberMalloc("MakeInteger");
01706         GCDBuffer2 = NumberMalloc("MakeInteger");
01707         LCMbuffer = NumberMalloc("MakeInteger");
01708         LCMbuffer2 = NumberMalloc("MakeInteger");
01709         t = in;
01710         tnext = t + *t; length = tnext[-1];
01711         if ( length < 0 ) { sign = -1; length = -length; }

```

```

01712     else { sign = 1; }
01713     tstop = tnext - length;
01714     redlength = (length-1)/2;
01715     for ( i = 0; i < redlength; i++ ) {
01716         GCDbuffer[i] = (UWORD)(tstop[i]);
01717         LCMbuffer[i] = (UWORD)(tstop[redlength+i]);
01718     }
01719     GCDlen = LCMlen = redlength;
01720     while ( GCDbuffer[GCDlen-1] == 0 ) GCDlen--;
01721     while ( LCMbuffer[LCMlen-1] == 0 ) LCMlen--;
01722     t = tnext;
01723     while ( *t ) {
01724         tnext = t + *t; length = ABS(tnext[-1]);
01725         tstop = tnext - length; redlength = (length-1)/2;
01726         len1 = len2 = redlength;
01727         den = tstop + redlength;
01728         while ( tstop[len1-1] == 0 ) len1--;
01729         while ( den[len2-1] == 0 ) len2--;
01730         if ( GCDlen == 1 && GCDbuffer[0] == 1 ) {}
01731         else {
01732             GcdLong(BHEAD (UWORD *)tstop, len1, GCDbuffer, GCDlen, GCDbuffer2, &GCDlen2);
01733             ap = GCDbuffer; GCDbuffer = GCDbuffer2; GCDbuffer2 = ap;
01734             a = GCDlen; GCDlen = GCDlen2; GCDlen2 = a;
01735         }
01736         if ( len2 == 1 && den[0] == 1 ) {}
01737         else {
01738             GcdLong(BHEAD LCMbuffer, LCMlen, (UWORD *)den, len2, LCMbuffer2, &LCMlen2);
01739             DivLong((UWORD *)den, len2, LCMbuffer2, LCMlen2,
01740                 GCDbuffer2, &GCDlen2, (UWORD *)AT.WorkPointer, &a);
01741             Mullong(LCMbuffer, LCMlen, GCDbuffer2, GCDlen2, LCMbuffer2, &LCMlen2);
01742             ap = LCMbuffer; LCMbuffer = LCMbuffer2; LCMbuffer2 = ap;
01743             a = LCMlen; LCMlen = LCMlen2; LCMlen2 = a;
01744         }
01745         t = tnext;
01746     }
01747     if ( GCDlen != 1 || GCDbuffer[0] != 1 || LCMlen != 1 || LCMbuffer[0] != 1 ) {
01748         redlength = GCDlen; if ( LCMlen > GCDlen ) redlength = LCMlen;
01749         for ( i = 0; i < GCDlen; i++ ) *tout++ = (WORD)(GCDbuffer[i]);
01750         for ( ; i < redlength; i++ ) *tout++ = 0;
01751         for ( i = 0; i < LCMlen; i++ ) *tout++ = (WORD)(LCMbuffer[i]);
01752         for ( ; i < redlength; i++ ) *tout++ = 0;
01753         *tout++ = (2*redlength+1)*sign;
01754         for ( i = 0; i < LCMlen; i++ ) *ttt++ = (WORD)(LCMbuffer[i]);
01755         for ( ; i < redlength; i++ ) *ttt++ = 0;
01756         for ( i = 0; i < GCDlen; i++ ) *ttt++ = (WORD)(GCDbuffer[i]);
01757         for ( ; i < redlength; i++ ) *ttt++ = 0;
01758         *ttt++ = (2*redlength+1)*sign;
01759         action++;
01760     }
01761     else {
01762         *tout++ = 1; *tout++ = 1; *tout++ = 3*sign;
01763         *ttt++ = 1; *ttt++ = 1; *ttt++ = 3*sign;
01764         if ( sign != 1 ) action++;
01765     }
01766     *tout = 0;
01767     NumberFree(LCMbuffer2, "MakeInteger");
01768     NumberFree(LCMbuffer, "MakeInteger");
01769     NumberFree(GCDbuffer2, "MakeInteger");
01770     NumberFree(GCDbuffer, "MakeInteger");
01771 }
01772 /*
01773 #] Coefficient :
01774 #[ Multiply by the inverse content :
01775 */
01776 if ( action ) {
01777     *tterm = tt - tterm;
01778     AT.WorkPointer = tt;
01779     inp = MultiplyWithTerm(BHEAD in, tterm, 2);
01780     AT.WorkPointer = tterm;
01781     in = inp;
01782 }
01783 /*
01784 #] Multiply by the inverse content :
01785 */
01786 *term = tout - term;
01787 AT.WorkPointer = tterm;
01788 return(in);
01789 CalledFrom:
01790 MLOCK(ErrorMessageLock);
01791 MesCall("TakeContent");
01792 MUNLOCK(ErrorMessageLock);
01793 return(0);
01794 }
01795
01796 /*
01797 #] TakeContent :
01798 #[ MergeSymbolLists :

```

```

01799
01800     Merges the extra list into the old.
01801     If par == -1 we take minimum powers
01802     If par == 1 we take maximum powers
01803     If par == 0 we take minimum of the absolute value of the powers
01804             if one is positive and the other negative we get zero.
01805     We assume that the symbols are in order in both lists
01806 */
01807
01808 int MergeSymbolLists(PHEAD WORD *old, WORD *extra, int par)
01809 {
01810     GETBIDENTITY
01811     WORD *new = TermMalloc("MergeSymbolLists");
01812     WORD *t1, *t2, *fill;
01813     int i1, i2;
01814     fill = new + 2; *new = SYMBOL;
01815     i1 = old[1] - 2; i2 = extra[1] - 2;
01816     t1 = old + 2; t2 = extra + 2;
01817     switch ( par ) {
01818     case -1:
01819         while ( i1 > 0 && i2 > 0 ) {
01820             if ( *t1 > *t2 ) {
01821                 if ( t2[1] < 0 ) { *fill++ = *t2++; *fill++ = *t2++; }
01822                 else t2 += 2;
01823                 i2 -= 2;
01824             }
01825             else if ( *t1 < *t2 ) {
01826                 if ( t1[1] < 0 ) { *fill++ = *t1++; *fill++ = *t1++; }
01827                 else t1 += 2;
01828                 i1 -= 2;
01829             }
01830             else if ( t1[1] < t2[1] ) {
01831                 *fill++ = *t1++; *fill++ = *t1++; t2 += 2;
01832                 i1 -= 2; i2 -= 2;
01833             }
01834             else {
01835                 *fill++ = *t2++; *fill++ = *t2++; t1 += 2;
01836                 i1 -= 2; i2 -= 2;
01837             }
01838         }
01839         for ( ; i1 > 0; i1 -= 2 ) {
01840             if ( t1[1] < 0 ) { *fill++ = *t1++; *fill++ = *t1++; }
01841             else t1 += 2;
01842         }
01843         for ( ; i2 > 0; i2 -= 2 ) {
01844             if ( t2[1] < 0 ) { *fill++ = *t2++; *fill++ = *t2++; }
01845             else t2 += 2;
01846         }
01847         break;
01848     case 1:
01849         while ( i1 > 0 && i2 > 0 ) {
01850             if ( *t1 > *t2 ) {
01851                 if ( t2[1] > 0 ) { *fill++ = *t2++; *fill++ = *t2++; }
01852                 else t2 += 2;
01853                 i2 -= 2;
01854             }
01855             else if ( *t1 < *t2 ) {
01856                 if ( t1[1] > 0 ) { *fill++ = *t1++; *fill++ = *t1++; }
01857                 else t1 += 2;
01858                 i1 -= 2;
01859             }
01860             else if ( t1[1] > t2[1] ) {
01861                 *fill++ = *t1++; *fill++ = *t1++; t2 += 2;
01862                 i1 -= 2; i2 -= 2;
01863             }
01864             else {
01865                 *fill++ = *t2++; *fill++ = *t2++; t1 += 2;
01866                 i1 -= 2; i2 -= 2;
01867             }
01868         }
01869         for ( ; i1 > 0; i1 -= 2 ) {
01870             if ( t1[1] > 0 ) { *fill++ = *t1++; *fill++ = *t1++; }
01871             else t1 += 2;
01872         }
01873         for ( ; i2 > 0; i2 -= 2 ) {
01874             if ( t2[1] > 0 ) { *fill++ = *t2++; *fill++ = *t2++; }
01875             else t2 += 2;
01876         }
01877         break;
01878     case 0:
01879         while ( i1 > 0 && i2 > 0 ) {
01880             if ( *t1 > *t2 ) {
01881                 t2 += 2; i2 -= 2;
01882             }
01883             else if ( *t1 < *t2 ) {
01884                 t1 += 2; i1 -= 2;
01885             }

```

```

01886         else if ( ( t1[1] > 0 ) && ( t2[1] < 0 ) ) { t1 += 2; t2 += 2; i1 -= 2; i2 -= 2; }
01887         else if ( ( t1[1] < 0 ) && ( t2[1] > 0 ) ) { t1 += 2; t2 += 2; i1 -= 2; i2 -= 2; }
01888         else if ( t1[1] > 0 ) {
01889             if ( t1[1] < t2[1] ) {
01890                 *fill++ = *t1++; *fill++ = *t1++; t2 += 2; i2 -= 2;
01891             }
01892             else {
01893                 *fill++ = *t2++; *fill++ = *t2++; t1 += 2; i1 -= 2;
01894             }
01895         }
01896         else {
01897             if ( t2[1] < t1[1] ) {
01898                 *fill++ = *t2++; *fill++ = *t2++; t1 += 2; i1 -= 2; i2 -= 2;
01899             }
01900             else {
01901                 *fill++ = *t1++; *fill++ = *t1++; t2 += 2; i1 -= 2; i2 -= 2;
01902             }
01903         }
01904     }
01905     for ( ; i1 > 0; i1-- ) *fill++ = *t1++;
01906     for ( ; i2 > 0; i2-- ) *fill++ = *t2++;
01907     break;
01908 }
01909 i1 = new[1] = fill - new;
01910 t2 = new; t1 = old; NCOPY(t1,t2,i1);
01911 TermFree(new,"MergeSymbolLists");
01912 return(0);
01913 }
01914
01915 /*
01916     #] MergeSymbolLists :
01917     #[ MergeDotproductLists :
01918
01919     Merges the extra list into the old.
01920     If par == -1 we take minimum powers
01921     If par == 1 we take maximum powers
01922     If par == 0 we take minimum of the absolute value of the powers
01923         if one is positive and the other negative we get zero.
01924     We assume that the dotproducts are in order in both lists
01925 */
01926
01927 int MergeDotproductLists(PHEAD WORD *old, WORD *extra, int par)
01928 {
01929     GETBIDENTITY
01930     WORD *new = TermMalloc("MergeDotproductLists");
01931     WORD *t1, *t2, *fill;
01932     int i1,i2;
01933     fill = new + 2;
01934     i1 = old[1] - 2; i2 = extra[1] - 2;
01935     t1 = old + 2; t2 = extra + 2;
01936     switch ( par ) {
01937         case -1:
01938             while ( i1 > 0 && i2 > 0 ) {
01939                 if ( ( *t1 > *t2 ) || ( *t1 == *t2 && t1[1] > t2[1] ) ) {
01940                     if ( t2[2] < 0 ) { *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; }
01941                     else t2 += 3;
01942                 }
01943                 else if ( ( *t1 < *t2 ) || ( *t1 == *t2 && t1[1] < t2[1] ) ) {
01944                     if ( t1[2] < 0 ) { *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; }
01945                     else t1 += 3;
01946                 }
01947                 else if ( t1[2] < t2[2] ) {
01948                     *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; t2 += 3;
01949                 }
01950                 else {
01951                     *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; t1 += 3;
01952                 }
01953             }
01954             break;
01955         case 1:
01956             while ( i1 > 0 && i2 > 0 ) {
01957                 if ( ( *t1 > *t2 ) || ( *t1 == *t2 && t1[1] > t2[1] ) ) {
01958                     if ( t2[2] > 0 ) { *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; }
01959                     else t2 += 3;
01960                 }
01961                 else if ( ( *t1 < *t2 ) || ( *t1 == *t2 && t1[1] < t2[1] ) ) {
01962                     if ( t1[2] > 0 ) { *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; }
01963                     else t1 += 3;
01964                 }
01965                 else if ( t1[2] > t2[2] ) {
01966                     *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; t2 += 3;
01967                 }
01968                 else {
01969                     *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; t1 += 3;
01970                 }
01971             }
01972             break;

```

```

01973         case 0:
01974             while ( i1 > 0 && i2 > 0 ) {
01975                 if ( ( *t1 > *t2 ) || ( *t1 == *t2 && t1[1] > t2[1] ) ) {
01976                     t2 += 3;
01977                 }
01978                 else if ( ( *t1 < *t2 ) || ( *t1 == *t2 && t1[1] < t2[1] ) ) {
01979                     t1 += 3;
01980                 }
01981                 else if ( ( t1[2] > 0 ) && ( t2[2] < 0 ) ) { t1 += 3; t2 += 3; }
01982                 else if ( ( t1[2] < 0 ) && ( t2[2] > 0 ) ) { t1 += 3; t2 += 3; }
01983                 else if ( t1[2] > 0 ) {
01984                     if ( t1[2] < t2[2] ) {
01985                         *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; t2 += 3;
01986                     }
01987                     else {
01988                         *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; t1 += 3;
01989                     }
01990                 }
01991                 else {
01992                     if ( t2[2] < t1[2] ) {
01993                         *fill++ = *t2++; *fill++ = *t2++; *fill++ = *t2++; t1 += 3;
01994                     }
01995                     else {
01996                         *fill++ = *t1++; *fill++ = *t1++; *fill++ = *t1++; t2 += 3;
01997                     }
01998                 }
01999             }
02000             break;
02001         }
02002         i1 = new[1] = fill - new;
02003         t2 = new; t1 = old; NCOPY(t1,t2,i1);
02004         TermFree(new, "MergeDotproductLists");
02005         return(0);
02006     }
02007
02008     /*
02009         #] MergeDotproductLists :
02010         #[ CreateExpression :
02011
02012         Looks for the expression in the argument, reads it and puts it
02013         in a buffer. Returns the address of the buffer.
02014         We send the expression through the Generator system, because there
02015         may be unsubstituted (sub)expressions as in
02016         Local F = (a+b);
02017         Local G = gcd_(F,...);
02018     */
02019
02020     WORD *CreateExpression(PHEAD WORD nexp)
02021     {
02022         GETBIDENTITY
02023         CBUF *C = cbuf+AC.cbunum;
02024         POSITION startposition, oldposition;
02025         FILEHANDLE *fi;
02026         WORD *term, *oldipointer = AR.CompressPointer;
02027     ;
02028         switch ( Expressions[nexp].status ) {
02029             case HIDDENLEXPRESSION:
02030             case HIDDENGEXPRESSION:
02031             case DROPHLEXPRESSION:
02032             case DROPHGEXPRESSION:
02033             case UNHIDELEXPRESSION:
02034             case UNHIDEGEXPRESSION:
02035                 AR.GetOneFile = 2; fi = AR.hidefile;
02036                 break;
02037             default:
02038                 AR.GetOneFile = 0; fi = AR.infile;
02039                 break;
02040         }
02041         SeekScratch(fi,&oldposition);
02042         startposition = AS.OldOnFile[nexp];
02043         term = AT.WorkPointer;
02044         if ( GetOneTerm(BHEAD term,fi,&startposition,0) <= 0 ) goto CalledFrom;
02045         NewSort (BHEAD0);
02046         AR.CompressPointer = oldipointer;
02047         while ( GetOneTerm(BHEAD term,fi,&startposition,0) > 0 ) {
02048             AT.WorkPointer = term + *term;
02049             if ( Generator(BHEAD term,C->numlhs) ) {
02050                 LowerSortLevel();
02051                 goto CalledFrom;
02052             }
02053             AR.CompressPointer = oldipointer;
02054         }
02055         AT.WorkPointer = term;
02056         if ( EndSort (BHEAD (WORD *) ((void *) (&term)),2) < 0 ) goto CalledFrom;
02057         SetScratch(fi,&oldposition);
02058         return (term);
02059     CalledFrom:

```

```

02060     MLOCK(ErrorMessageLock);
02061     MesCall("CreateExpression");
02062     MUNLOCK(ErrorMessageLock);
02063     Terminate(-1);
02064     return(0);
02065 }
02066
02067 /*
02068     #] CreateExpression :
02069     #[ GCDterms :      GCD of two terms
02070
02071     Computes the GCD of two terms.
02072     Output in termout.
02073     termout may overlap with term1.
02074 */
02075
02076 int GCDterms(PHEAD WORD *term1, WORD *term2, WORD *termout)
02077 {
02078     GETBIDENTITY
02079     WORD *t1, *t1stop, *t1next, *t2, *t2stop, *t2next, *tout, *tt1, *tt2;
02080     int count1, count2, i, ii, x1, sign;
02081     WORD length1, length2;
02082     t1 = term1 + *term1; t1stop = t1 - ABS(t1[-1]); t1 = term1+1;
02083     t2 = term2 + *term2; t2stop = t2 - ABS(t2[-1]); t2 = term2+1;
02084     tout = termout+1;
02085     while ( t1 < t1stop ) {
02086         t1next = t1 + t1[1];
02087         t2 = term2+1;
02088         if ( *t1 == SYMBOL ) {
02089             while ( t2 < t2stop && *t2 != SYMBOL ) t2 += t2[1];
02090             if ( t2 < t2stop && *t2 == SYMBOL ) {
02091                 t2next = t2+t2[1];
02092                 tt1 = t1+2; tt2 = t2+2; count1 = 0;
02093                 while ( tt1 < t1next && tt2 < t2next ) {
02094                     if ( *tt1 < *tt2 ) tt1 += 2;
02095                     else if ( *tt1 > *tt2 ) tt2 += 2;
02096                     else if ( ( tt1[1] > 0 && tt2[1] < 0 ) ||
02097                             ( tt2[1] > 0 && tt1[1] < 0 ) ) {
02098                         tt1 += 2; tt2 += 2;
02099                     }
02100                     else {
02101                         x1 = tt1[1];
02102                         if ( tt1[1] < 0 ) { if ( tt2[1] > x1 ) x1 = tt2[1]; }
02103                         else { if ( tt2[1] < x1 ) x1 = tt2[1]; }
02104                         tout[count1+2] = *tt1;
02105                         tout[count1+3] = x1;
02106                         tt1 += 2; tt2 += 2;
02107                         count1 += 2;
02108                     }
02109                 }
02110                 if ( count1 > 0 ) {
02111                     *tout = SYMBOL; tout[1] = count1+2; tout += tout[1];
02112                 }
02113             }
02114         }
02115         else if ( *t1 == DOTPRODUCT ) {
02116             while ( t2 < t2stop && *t2 != DOTPRODUCT ) t2 += t2[1];
02117             if ( t2 < t2stop && *t2 == DOTPRODUCT ) {
02118                 t2next = t2+t2[1];
02119                 tt1 = t1+2; tt2 = t2+2; count1 = 0;
02120                 while ( tt1 < t1next && tt2 < t2next ) {
02121                     if ( *tt1 < *tt2 || ( *tt1 == *tt2 && tt1[1] < tt2[1] ) ) tt1 += 3;
02122                     else if ( *tt1 > *tt2 || ( *tt1 == *tt2 && tt1[1] > tt2[1] ) ) tt2 += 3;
02123                     else if ( ( tt1[2] > 0 && tt2[2] < 0 ) ||
02124                             ( tt2[2] > 0 && tt1[2] < 0 ) ) {
02125                         tt1 += 3; tt2 += 3;
02126                     }
02127                     else {
02128                         x1 = tt1[2];
02129                         if ( tt1[2] < 0 ) { if ( tt2[2] > x1 ) x1 = tt2[2]; }
02130                         else { if ( tt2[2] < x1 ) x1 = tt2[2]; }
02131                         tout[count1+2] = *tt1;
02132                         tout[count1+3] = tt1[1];
02133                         tout[count1+4] = x1;
02134                         tt1 += 3; tt2 += 3;
02135                         count1 += 3;
02136                     }
02137                 }
02138                 if ( count1 > 0 ) {
02139                     *tout = DOTPRODUCT; tout[1] = count1+2; tout += tout[1];
02140                 }
02141             }
02142         }
02143         else if ( *t1 == VECTOR ) {
02144             while ( t2 < t2stop && *t2 != VECTOR ) t2 += t2[1];
02145             if ( t2 < t2stop && *t2 == VECTOR ) {
02146                 t2next = t2+t2[1];

```

```

02147         tt1 = t1+2; tt2 = t2+2; count1 = 0;
02148         while ( tt1 < tlnext && tt2 < t2next ) {
02149             if ( *tt1 < *tt2 || ( *tt1 == *tt2 && tt1[1] < tt2[1] ) ) tt1 += 2;
02150             else if ( *tt1 > *tt2 || ( *tt1 == *tt2 && tt1[1] > tt2[1] ) ) tt2 += 2;
02151             else {
02152                 tout[count1+2] = *tt1;
02153                 tout[count1+3] = tt1[1];
02154                 tt1 += 2; tt2 += 2;
02155                 count1 += 2;
02156             }
02157         }
02158         if ( count1 > 0 ) {
02159             *tout = VECTOR; tout[1] = count1+2; tout += tout[1];
02160         }
02161     }
02162 }
02163 else if ( *t1 == INDEX ) {
02164     while ( t2 < t2stop && *t2 != INDEX ) t2 += t2[1];
02165     if ( t2 < t2stop && *t2 == INDEX ) {
02166         t2next = t2+t2[1];
02167         tt1 = t1+2; tt2 = t2+2; count1 = 0;
02168         while ( tt1 < tlnext && tt2 < t2next ) {
02169             if ( *tt1 < *tt2 ) tt1 += 1;
02170             else if ( *tt1 > *tt2 ) tt2 += 1;
02171             else {
02172                 tout[count1+2] = *tt1;
02173                 tt1 += 1; tt2 += 1;
02174                 count1 += 1;
02175             }
02176         }
02177         if ( count1 > 0 ) {
02178             *tout = INDEX; tout[1] = count1+2; tout += tout[1];
02179         }
02180     }
02181 }
02182 else if ( *t1 == DELTA ) {
02183     while ( t2 < t2stop && *t2 != DELTA ) t2 += t2[1];
02184     if ( t2 < t2stop && *t2 == DELTA ) {
02185         t2next = t2+t2[1];
02186         tt1 = t1+2; tt2 = t2+2; count1 = 0;
02187         while ( tt1 < tlnext && tt2 < t2next ) {
02188             if ( *tt1 < *tt2 || ( *tt1 == *tt2 && tt1[1] < tt2[1] ) ) tt1 += 2;
02189             else if ( *tt1 > *tt2 || ( *tt1 == *tt2 && tt1[1] > tt2[1] ) ) tt2 += 2;
02190             else {
02191                 tout[count1+2] = *tt1;
02192                 tout[count1+3] = tt1[1];
02193                 tt1 += 2; tt2 += 2;
02194                 count1 += 2;
02195             }
02196         }
02197         if ( count1 > 0 ) {
02198             *tout = DELTA; tout[1] = count1+2; tout += tout[1];
02199         }
02200     }
02201 }
02202 else if ( *t1 >= FUNCTION ) { /* noncommuting functions? Forbidden! */
02203     /*
02204         Count how many times this function occurs.
02205         Then count how many times it is in term2.
02206     */
02207     count1 = 1;
02208     while ( tlnext < t1stop && *t1 == *tlnext && t1[1] == tlnext[1] ) {
02209         for ( i = 2; i < t1[1]; i++ ) {
02210             if ( t1[i] != tlnext[i] ) break;
02211         }
02212         if ( i < t1[1] ) break;
02213         count1++;
02214         tlnext += tlnext[1];
02215     }
02216     count2 = 0;
02217     while ( t2 < t2stop ) {
02218         if ( *t2 == *t1 && t2[1] == t1[1] ) {
02219             for ( i = 2; i < t1[1]; i++ ) {
02220                 if ( t2[i] != t1[i] ) break;
02221             }
02222             if ( i >= t1[1] ) count2++;
02223         }
02224         t2 += t2[1];
02225     }
02226     if ( count1 < count2 ) count2 = count1; /* number of common occurrences */
02227     if ( count2 > 0 ) {
02228         if ( tout == t1 ) {
02229             while ( count2 > 0 ) { tout += tout[1]; count2--; }
02230         }
02231         else {
02232             i = t1[1]*count2;
02233             NCOPY(tout,t1,i);

```



```

02234     }
02235     }
02236 }
02237 t1 = tlnext;
02238 }
02239 /*
02240     Now the coefficients. They are in t1stop and t2stop. Should go to tout.
02241 */
02242 sign = 1;
02243 length1 = term1[*term1-1]; ii = i = ABS(length1); t1 = t1stop;
02244 if ( t1 != tout ) { NCOPY(tout,t1,i); tout -= ii; }
02245 length2 = term2[*term2-1];
02246 if ( length1 < 0 && length2 < 0 ) sign = -1;
02247 if ( AccumGCD(BHEAD (UWORD *)tout,&length1,(UWORD *)t2stop,length2) ) {
02248     MLOCK(ErrorMessageLock);
02249     MesCall("GCDterms");
02250     MUNLOCK(ErrorMessageLock);
02251     SETERROR(-1)
02252 }
02253 if ( sign < 0 && length1 > 0 ) length1 = -length1;
02254 tout += ABS(length1); tout[-1] = length1;
02255 *termout = tout - termout; *tout = 0;
02256 return(0);
02257 }
02258 /*
02259     #] GCDterms :
02260     #[ ReadPolyRatFun :
02261 */
02262 /*
02263 int ReadPolyRatFun(PHEAD WORD *term)
02264 {
02265     WORD *oldworkpointer = AT.WorkPointer;
02266     int flag, i;
02267     WORD *t, *fun, *nextt, *num, *den, *t1, *t2, size, numsize, densize;
02268     WORD *term1, *term2, *confree1, *confree2, *gcd, *num1, *den1, move, *newnum, *newden;
02269     WORD *tstop, *m1, *m2;
02270     WORD oldsorttype = AR.SortType;
02271     COMPARE oldcompareroutine = (COMPARE) (AR.CompareRoutine);
02272     AR.SortType = SORTHIGFIRST;
02273     AR.CompareRoutine = (COMPAREDUMMY) (&CompareSymbols);
02274     AR.CompareRoutine = (COMPAREDUMMY) (&CompareSymbols);
02275     tstop = term + *term; tstop -= ABS(tstop[-1]);
02276     if ( term + *term == AT.WorkPointer ) flag = 1;
02277     else flag = 0;
02278     t = term+1;
02279     while ( t < tstop ) {
02280         if ( *t != AR.PolyFun ) { t += t[1]; continue; }
02281         if ( ( t[2] & MUSTCLEANPRF ) == 0 ) { t += t[1]; continue; }
02282         fun = t;
02283         nextt = t + t[1];
02284         if ( fun[1] > FUNHEAD && fun[FUNHEAD] == -SNUMBER && fun[FUNHEAD+1] == 0 )
02285             { *term = 0; break; }
02286         if ( FromPolyRatFun(BHEAD fun, &num, &den) > 0 ) { t = nextt; continue; }
02287         if ( *num == ARGHEAD ) { *term = 0; break; }
02288     }
02289     /*
02290         Now we have num and den. Both are in general argument notation,
02291         but can also be used as expressions as in num+ARGHEAD, den+ARGHEAD.
02292         We need the gcd. For this we have to take out the contents
02293         because PreGCD does not like contents.
02294     */
02295     term1 = TermMalloc("ReadPolyRatFun");
02296     term2 = TermMalloc("ReadPolyRatFun");
02297     confree1 = TakeSymbolContent(BHEAD num+ARGHEAD,term1);
02298     confree2 = TakeSymbolContent(BHEAD den+ARGHEAD,term2);
02299     GCDclean(BHEAD term1,term2);
02300     /* gcd = PreGCD(BHEAD confree1,confree2,1); */
02301     gcd = poly_gcd(BHEAD confree1,confree2,1);
02302     newnum = PolyDiv(BHEAD confree1,gcd,"ReadPolyRatFun");
02303     newden = PolyDiv(BHEAD confree2,gcd,"ReadPolyRatFun");
02304     TermFree(confree2,"ReadPolyRatFun");
02305     TermFree(confree1,"ReadPolyRatFun");
02306     num1 = MULfunc(BHEAD term1,newnum);
02307     den1 = MULfunc(BHEAD term2,newden);
02308     TermFree(newnum,"ReadPolyRatFun");
02309     TermFree(newden,"ReadPolyRatFun");
02310     /* M_free(gcd,"poly_gcd"); */
02311     TermFree(gcd,"poly_gcd");
02312     TermFree(term1,"ReadPolyRatFun");
02313     TermFree(term2,"ReadPolyRatFun");
02314     /*
02315         Now we can put the function back together.
02316         Notice that we cannot use ToFast, because there is no reservation
02317         for the header of the argument. Fortunately there are only two
02318         types of fast arguments.
02319     */
02320     if ( num1[0] == 4 && num1[4] == 0 && num1[2] == 1 && num1[1] > 0 ) {

```

```

02321         numsize = 2; numl[0] = -SNUMBER;
02322         if ( numl[3] < 0 ) numl[1] = -numl[1];
02323     }
02324     else if ( numl[0] == 8 && numl[8] == 0 && numl[7] == 3 && numl[6] == 1
02325         && numl[5] == 1 && numl[1] == SYMBOL && numl[4] == 1 ) {
02326         numsize = 2; numl[0] = -SYMBOL; numl[1] = numl[3];
02327     }
02328     else { m1 = numl; while ( *m1 ) m1 += *m1; numsize = (m1-numl)+ARGHEAD; }
02329     if ( denl[0] == 4 && denl[4] == 0 && denl[2] == 1 && denl[1] > 0 ) {
02330         densize = 2; denl[0] = -SNUMBER;
02331         if ( denl[3] < 0 ) denl[1] = -denl[1];
02332     }
02333     else if ( denl[0] == 8 && denl[8] == 0 && denl[7] == 3 && denl[6] == 1
02334         && denl[5] == 1 && denl[1] == SYMBOL && denl[4] == 1 ) {
02335         densize = 2; denl[0] = -SYMBOL; denl[1] = denl[3];
02336     }
02337     else { m2 = denl; while ( *m2 ) m2 += *m2; densize = (m2-denl)+ARGHEAD; }
02338     size = FUNHEAD+numsize+densize;
02339
02340     if ( size > fun[1] ) {
02341         move = size - fun[1];
02342         t1 = term+*term; t2 = t1+move;
02343         while ( t1 > nextt ) *--t2 = *--t1;
02344         tstop += move; nextt += move;
02345         *term += move;
02346     }
02347     else if ( size < fun[1] ) {
02348         move = fun[1]-size;
02349         t2 = fun+size; t1 = nextt;
02350         tstop -= move; nextt -= move;
02351         t = term+*term;
02352         while ( t1 < t ) *t2++ = *t1++;
02353         *term -= move;
02354     }
02355     else { /* no need to move anything */ }
02356     fun[1] = size; fun[2] = 0;
02357     t2 = fun+FUNHEAD; t1 = numl;
02358     if ( *numl < 0 ) { *t2++ = numl[0]; *t2++ = numl[1]; }
02359     else { *t2++ = numsize; *t2++ = 0; FILLARG(t2);
02360         i = numsize-ARGHEAD; NCOPY(t2,t1,i) }
02361     t1 = denl;
02362     if ( *denl < 0 ) { *t2++ = denl[0]; *t2++ = denl[1]; }
02363     else { *t2++ = densize; *t2++ = 0; FILLARG(t2);
02364         i = densize-ARGHEAD; NCOPY(t2,t1,i) }
02365
02366     TermFree(numl,"MULfunc");
02367     TermFree(denl,"MULfunc");
02368     t = nextt;
02369 }
02370
02371 if ( flag ) AT.WorkPointer = term + *term;
02372 else AT.WorkPointer = oldworkpointer;
02373 AR.CompareRoutine = (COMPAREDDUMMY)oldcompareroutine;
02374 AR.SortType = oldsorttype;
02375 return(0);
02376 }
02377
02378 /*
02379     #] ReadPolyRatFun :
02380     #[ FromPolyRatFun :
02381 */
02382
02383 int FromPolyRatFun(PHEAD WORD *fun, WORD **numout, WORD **denout)
02384 {
02385     WORD *nextfun, *tt, *num, *den;
02386     int i;
02387     nextfun = fun + fun[1];
02388     fun += FUNHEAD;
02389     num = AT.WorkPointer;
02390     if ( *fun < 0 ) {
02391         if ( *fun != -SNUMBER && *fun != -SYMBOL ) goto Improper;
02392         ToGeneral(fun,num,0);
02393         tt = num + *num; *tt++ = 0;
02394         fun += 2;
02395     }
02396     else { i = *fun; tt = num; NCOPY(tt,fun,i); *tt++ = 0; }
02397     den = tt;
02398     if ( *fun < 0 ) {
02399         if ( *fun != -SNUMBER && *fun != -SYMBOL ) goto Improper;
02400         ToGeneral(fun,den,0);
02401         tt = den + *den; *tt++ = 0;
02402         fun += 2;
02403     }
02404     else { i = *fun; tt = den; NCOPY(tt,fun,i); *tt++ = 0; }
02405     *numout = num; *denout = den;
02406     if ( fun != nextfun ) { return(1); }
02407     AT.WorkPointer = tt;

```

```

02408     return(0);
02409 Improper:
02410     MLOCK(ErrorMessageLock);
02411     MesPrint("Improper use of PolyRatFun");
02412     MesCall("FromPolyRatFun");
02413     MUNLOCK(ErrorMessageLock);
02414     SETERROR(-1);
02415 }
02416
02417 /*
02418     #] FromPolyRatFun :
02419     #[ TakeSymbolContent :
02420 */
02434 WORD *TakeSymbolContent (PHEAD WORD *in, WORD *term)
02435 {
02436     GETBIDENTITY
02437     WORD *t, *tstop, *tout, *tstore;
02438     WORD *tnext, *tt, *tterm;
02439     WORD *inp, a, *den, *oldworkpointer = AT.WorkPointer;
02440     int i, action = 0, sign, first;
02441     UWORD *GCDbuffer, *GCDbuffer2, *LCMbuffer, *LCMbuffer2, *ap;
02442     WORD GCDlen, GCDlen2, LCMlen, LCMlen2, length, redlength, len1, len2;
02443     LONG j;
02444     tout = tstore = term+1;
02445 /*
02446     #[ SYMBOL :
02447
02448     We make a list of symbols and their minimal powers.
02449     This includes negative powers. In the end we have to multiply by the
02450     inverse of this list. That takes out all negative powers and leaves
02451     things ready for further processing.
02452 */
02453     tterm = AT.WorkPointer; tt = tterm+1;
02454     tout[0] = SYMBOL; tout[1] = 2;
02455     t = in; first = 1;
02456     while ( *t ) {
02457         tnext = t + *t; tstop = tnext - ABS(tnext[-1]); t++;
02458         while ( t < tstop ) {
02459             if ( first ) {
02460                 if ( *t == SYMBOL ) {
02461                     for ( i = 0; i < t[1]; i++ ) tout[i] = t[i];
02462                     goto didwork;
02463                 }
02464                 else {
02465                     MLOCK(ErrorMessageLock);
02466                     MesPrint ((char*)"ERROR: polynomials and polyratfuns must contain symbols only");
02467                     MUNLOCK(ErrorMessageLock);
02468                     Terminate(1);
02469                 }
02470             }
02471             else if ( *t == SYMBOL ) {
02472                 MergeSymbolLists(BHEAD tout,t,-1);
02473                 goto didwork;
02474             }
02475             else {
02476                 t += t[1];
02477             }
02478         }
02479 /*
02480         Here we come when there were no symbols. Only keep the negative ones.
02481 */
02482         if ( first == 0 ) {
02483             int j = 2;
02484             for ( i = 2; i < tout[1]; i += 2 ) {
02485                 if ( tout[i+1] < 0 ) {
02486                     if ( i == j ) { j += 2; }
02487                     else { tout[j] = tout[i]; tout[j+1] = tout[i+1]; j += 2; }
02488                 }
02489             }
02490             tout[1] = j;
02491         }
02492     didwork:;
02493     first = 0;
02494     t = tnext;
02495 }
02496 if ( tout[1] > 2 ) {
02497     t = tout;
02498     tt[0] = t[0]; tt[1] = t[1];
02499     for ( i = 2; i < t[1]; i += 2 ) {
02500         tt[i] = t[i]; tt[i+1] = -t[i+1];
02501     }
02502     tt += tt[1];
02503     tout += tout[1];
02504     action++;
02505 }
02506 /*
02507     #] SYMBOL :

```

```

02508     #[ Coefficient :
02509
02510     Now we have to collect the GCD of the numerators
02511     and the LCM of the denominators.
02512 */
02513     AT.WorkPointer = tt;
02514     if ( AN.cmod != 0 ) {
02515         WORD x, ix, ip;
02516         t = in; tnext = t + *t; tstop = tnext - ABS(tnext[-1]);
02517         x = tstop[0];
02518         if ( tnext[-1] < 0 ) x += AC.cmod[0];
02519         if ( GetModInverses(x, (WORD)(AN.cmod[0]), &ix, &ip) ) goto CalledFrom;
02520         *tout++ = x; *tout++ = 1; *tout++ = 3;
02521         *tt++ = ix; *tt++ = 1; *tt++ = 3;
02522     }
02523     else {
02524         GCDbuffer = NumberMalloc("MakeInteger");
02525         GCDbuffer2 = NumberMalloc("MakeInteger");
02526         LCMbuffer = NumberMalloc("MakeInteger");
02527         LCMbuffer2 = NumberMalloc("MakeInteger");
02528         t = in;
02529         tnext = t + *t; length = tnext[-1];
02530         if ( length < 0 ) { sign = -1; length = -length; }
02531         else { sign = 1; }
02532         tstop = tnext - length;
02533         redlength = (length-1)/2;
02534         for ( i = 0; i < redlength; i++ ) {
02535             GCDbuffer[i] = (UWORD)(tstop[i]);
02536             LCMbuffer[i] = (UWORD)(tstop[redlength+i]);
02537         }
02538         GCDlen = LCMlen = redlength;
02539         while ( GCDbuffer[GCDlen-1] == 0 ) GCDlen--;
02540         while ( LCMbuffer[LCMlen-1] == 0 ) LCMlen--;
02541         t = tnext;
02542         while ( *t ) {
02543             tnext = t + *t; length = ABS(tnext[-1]);
02544             tstop = tnext - length; redlength = (length-1)/2;
02545             len1 = len2 = redlength;
02546             den = tstop + redlength;
02547             while ( tstop[len1-1] == 0 ) len1--;
02548             while ( den[len2-1] == 0 ) len2--;
02549             if ( GCDlen == 1 && GCDbuffer[0] == 1 ) {}
02550             else {
02551                 GcdLong(BHEAD (UWORD *)tstop, len1, GCDbuffer, GCDlen, GCDbuffer2, &GCDlen2);
02552                 ap = GCDbuffer; GCDbuffer = GCDbuffer2; GCDbuffer2 = ap;
02553                 a = GCDlen; GCDlen = GCDlen2; GCDlen2 = a;
02554             }
02555             if ( len2 == 1 && den[0] == 1 ) {}
02556             else {
02557                 GcdLong(BHEAD LCMbuffer, LCMlen, (UWORD *)den, len2, LCMbuffer2, &LCMlen2);
02558                 DivLong((UWORD *)den, len2, LCMbuffer2, LCMlen2,
02559                     GCDbuffer2, &GCDlen2, (UWORD *)AT.WorkPointer, &a);
02560                 MullLong(LCMbuffer, LCMlen, GCDbuffer2, GCDlen2, LCMbuffer2, &LCMlen2);
02561                 ap = LCMbuffer; LCMbuffer = LCMbuffer2; LCMbuffer2 = ap;
02562                 a = LCMlen; LCMlen = LCMlen2; LCMlen2 = a;
02563             }
02564             t = tnext;
02565         }
02566         if ( GCDlen != 1 || GCDbuffer[0] != 1 || LCMlen != 1 || LCMbuffer[0] != 1 ) {
02567             redlength = GCDlen; if ( LCMlen > GCDlen ) redlength = LCMlen;
02568             for ( i = 0; i < GCDlen; i++ ) *tout++ = (WORD)(GCDbuffer[i]);
02569             for ( ; i < redlength; i++ ) *tout++ = 0;
02570             for ( i = 0; i < LCMlen; i++ ) *tout++ = (WORD)(LCMbuffer[i]);
02571             for ( ; i < redlength; i++ ) *tout++ = 0;
02572             *tout++ = (2*redlength+1)*sign;
02573             for ( i = 0; i < LCMlen; i++ ) *tt++ = (WORD)(LCMbuffer[i]);
02574             for ( ; i < redlength; i++ ) *tt++ = 0;
02575             for ( i = 0; i < GCDlen; i++ ) *tt++ = (WORD)(GCDbuffer[i]);
02576             for ( ; i < redlength; i++ ) *tt++ = 0;
02577             *tt++ = (2*redlength+1)*sign;
02578             action++;
02579         }
02580         else {
02581             *tout++ = 1; *tout++ = 1; *tout++ = 3*sign;
02582             *tt++ = 1; *tt++ = 1; *tt++ = 3*sign;
02583             if ( sign != 1 ) action++;
02584         }
02585         NumberFree(LCMbuffer2, "MakeInteger");
02586         NumberFree(LCMbuffer, "MakeInteger");
02587         NumberFree(GCDbuffer2, "MakeInteger");
02588         NumberFree(GCDbuffer, "MakeInteger");
02589     }
02590 /*
02591     #[ Coefficient :
02592     #[ Multiply by the inverse content :
02593 */
02594     if ( action ) {

```

```

02595     *term = tout - term; *tout = 0;
02596     *tterm = tt - tterm; *tt = 0;
02597     AT.WorkPointer = tt;
02598     inp = MultiplyWithTerm(BHEAD in,tterm,2);
02599     AT.WorkPointer = tterm;
02600     t = inp; while ( *t ) t += *t;
02601     j = (t-inp); t = inp;
02602     if ( j*sizeof(WORD) > (size_t)(AM.MaxTer) ) goto OverWork;
02603     in = tout = TermMalloc("TakeSymbolContent");
02604     NCOPY(tout,t,j); *tout = 0;
02605     if ( AN.tryterm > 0 ) { TermFree(inp,"MultiplyWithTerm"); AN.tryterm = 0; }
02606     else { M_free(inp,"MultiplyWithTerm"); }
02607 }
02608 else {
02609     t = in; while ( *t ) t += *t;
02610     j = (t-in); t = in;
02611     if ( j*sizeof(WORD) > (size_t)(AM.MaxTer) ) goto OverWork;
02612     in = tout = TermMalloc("TakeSymbolContent");
02613     NCOPY(tout,t,j); *tout = 0;
02614     term[0] = 4; term[1] = 1; term[2] = 1; term[3] = 3; term[4] = 0;
02615 }
02616 /*
02617 #] Multiply by the inverse content :
02618 AT.WorkPointer = tterm + *tterm;
02619 */
02620 AT.WorkPointer = oldworkpointer;
02621
02622 return(in);
02623 OverWork:
02624 MLOCK(ErrorMessageLock);
02625 MesPrint("Term too complex. Maybe increasing MaxTermSize can help");
02626 MUNLOCK(ErrorMessageLock);
02627 CalledFrom:
02628 MLOCK(ErrorMessageLock);
02629 MesCall("TakeSymbolContent");
02630 MUNLOCK(ErrorMessageLock);
02631 Terminate(-1);
02632 return(0);
02633 }
02634
02635 /*
02636 #] TakeSymbolContent :
02637 #[ GCDclean :
02638
02639     Takes a numerator and a denominator that each consist of a
02640     single term with only a coefficient and symbols and makes them
02641     into a proper fraction. Output overwrites input.
02642 */
02643
02644 void GCDclean(PHEAD WORD *num, WORD *den)
02645 {
02646     WORD *out1 = TermMalloc("GCDclean");
02647     WORD *out2 = TermMalloc("GCDclean");
02648     WORD *t1, *t2, *r1, *r2, *t1stop, *t2stop, csizel, csize2, csize3, pow, sign;
02649     int i;
02650     t1stop = num+*num; sign = ( t1stop[-1] < 0 ) ? -1 : 1;
02651     csizel = ABS(t1stop[-1]); t1stop -= csizel;
02652     t2stop = den+*den; if ( t2stop[-1] < 0 ) sign = -sign;
02653     csize2 = ABS(t2stop[-1]); t2stop -= csize2;
02654     t1 = num+1; t2 = den+1;
02655     r1 = out1+3; r2 = out2+3;
02656     if ( t1 == t1stop ) {
02657         if ( t2 < t2stop ) {
02658             for ( i = 2; i < t2[1]; i += 2 ) {
02659                 if ( t2[i+1] < 0 ) { *r1++ = t2[i]; *r1++ = -t2[i+1]; }
02660                 else { *r2++ = t2[i]; *r2++ = t2[i+1]; }
02661             }
02662         }
02663     }
02664     else if ( t2 == t2stop ) {
02665         for ( i = 2; i < t1[1]; i += 2 ) {
02666             if ( t1[i+1] < 0 ) { *r2++ = t1[i]; *r2++ = -t1[i+1]; }
02667             else { *r1++ = t1[i]; *r1++ = t1[i+1]; }
02668         }
02669     }
02670     else {
02671         t1 += 2; t2 += 2;
02672         while ( t1 < t1stop && t2 < t2stop ) {
02673             if ( *t1 < *t2 ) {
02674                 if ( t1[1] > 0 ) { *r1++ = *t1; *r1++ = t1[1]; t1 += 2; }
02675                 else if ( t1[1] < 0 ) { *r2++ = *t1; *r2++ = -t1[1]; t1 += 2; }
02676             }
02677             else if ( *t1 > *t2 ) {
02678                 if ( t2[1] > 0 ) { *r2++ = *t2; *r2++ = t2[1]; t2 += 2; }
02679                 else if ( t2[1] < 0 ) { *r1++ = *t2; *r1++ = -t2[1]; t2 += 2; }
02680             }
02681             else {

```

```

02682         pow = t1[1]-t2[1];
02683         if ( pow > 0 ) { *r1++ = *t1; *r1++ = pow; }
02684         else if ( pow < 0 ) { *r2++ = *t1; *r2++ = -pow; }
02685         t1 += 2; t2 += 2;
02686     }
02687 }
02688 while ( t1 < t1stop ) {
02689     if ( t1[1] < 0 ) { *r2++ = *t1; *r2++ = -t1[1]; }
02690     else { *r1++ = *t1; *r1++ = t1[1]; }
02691     t1 += 2;
02692 }
02693 while ( t2 < t2stop ) {
02694     if ( t2[1] < 0 ) { *r1++ = *t2; *r1++ = -t2[1]; }
02695     else { *r2++ = *t2; *r2++ = t2[1]; }
02696     t2 += 2;
02697 }
02698 }
02699 if ( r1 > out1+3 ) { out1[1] = SYMBOL; out1[2] = r1 - out1 - 1; }
02700 else r1 = out1+1;
02701 if ( r2 > out2+3 ) { out2[1] = SYMBOL; out2[2] = r2 - out2 - 1; }
02702 else r2 = out2+1;
02703 /*
02704     Now the coefficients.
02705 */
02706 csize1 = REDLENG(csize1);
02707 csize2 = REDLENG(csize2);
02708 if ( DivRat(BHEAD (UWORD *)t1stop,csize1,(UWORD *)t2stop,csize2,(UWORD *)r1,&csize3) ) {
02709     MLOCK(ErrorMessageLock);
02710     MesCall("GCDclean");
02711     MUNLOCK(ErrorMessageLock);
02712     Terminate(-1);
02713 }
02714 UnPack((UWORD *)r1,csize3,&csize2,&csize1);
02715 t2 = r1+ABS(csize3);
02716 for ( i = 0; i < csize2; i++ ) r2[i] = t2[i];
02717 r2 += csize2; *r2++ = 1;
02718 for ( i = 1; i < csize2; i++ ) *r2++ = 0;
02719 csize2 = INCLENG(csize2); *r2++ = csize2; *out2 = r2-out2;
02720 r1 += ABS(csize1); *r1++ = 1;
02721 for ( i = 1; i < ABS(csize1); i++ ) *r1++ = 0;
02722 csize1 = INCLENG(csize1); *r1++ = csize1; *out1 = r1-out1;
02723
02724 t1 = num; t2 = out1; i = *out1; NCOPY(t1,t2,i); *t1 = 0;
02725 if ( sign < 0 ) t1[-1] = -t1[-1];
02726 t1 = den; t2 = out2; i = *out2; NCOPY(t1,t2,i); *t1 = 0;
02727
02728 TermFree(out2,"GCDclean");
02729 TermFree(out1,"GCDclean");
02730 }
02731
02732 /*
02733     #] GCDclean :
02734     #[ PolyDiv :
02735
02736     Special stub function for polynomials that should fit inside a term.
02737     We make sure that the space is allocated by TermMalloc.
02738     This makes things much easier on the calling routines.
02739 */
02740
02741 WORD *PolyDiv(PHEAD WORD *a,WORD *b,char *text)
02742 {
02743     /*
02744         Probably the following would work now
02745     */
02746     DUMMYUSE(text);
02747     return(poly_div(BHEAD a,b,1));
02748     /*
02749         WORD *quo, *qq;
02750         WORD *x, *xx;
02751         LONG i;
02752         quo = poly_div(BHEAD a,b,1);
02753         x = TermMalloc(text);
02754         qq = quo; while ( *qq ) qq += *qq;
02755         i = (qq-quo+1);
02756         if ( i*sizeof(WORD) > (size_t)(AM.MaxTer) ) {
02757             DUMMYUSE(text);
02758             MLOCK(ErrorMessageLock);
02759             MesPrint("PolyDiv: Term too complex. Maybe increasing MaxTermSize can help");
02760             MUNLOCK(ErrorMessageLock);
02761             Terminate(-1);
02762         }
02763         xx = x; qq = quo;
02764         NCOPY(xx,qq,i)
02765         TermFree(quo,"poly_div");
02766         return(x);
02767     */
02768 }

```

```

02769
02770 /*
02771     #] PolyDiv :
02772     #] GCDfunction :
02773     #] DIVfunction :
02774
02775     Input: a div_ function that has two arguments inside a term.
02776     Action: Calculates [arg1/arg2] using polynomial techniques if needed.
02777     Output: The output result is combined with the remainder of the term
02778     and sent to Generator for further processing.
02779     Note that the output can be just a number or many terms.
02780     In case par == 0 the output is [arg1/arg2]
02781     In case par == 1 the output is [arg1%arg2]
02782     In case par == 2 the output is [inverse of arg1 modulus arg2]
02783     In case par == 3 the output is [arg1*arg2]
02784 */
02785
02786 WORD divrem[4] = { DIVFUNCTION, REMFUNCTION, INVERSEFUNCTION, MULFUNCTION };
02787
02788 int DIVfunction(PHEAD WORD *term,WORD level,int par)
02789 {
02790     GETBIDENTITY
02791     WORD *t, *tt, *r, *arg1 = 0, *arg2 = 0, *arg3 = 0, *termout;
02792     WORD *tstop, *tend, *r3, *rr, *rstop, tlength, rlength, newlength;
02793     WORD *proper1, *proper2, *proper3 = 0, numdol = -1;
02794     int numargs = 0, type1, type2, actionflag1, actionflag2;
02795     WORD startebuf = cbuf[AT.ebufnum].numrhs;
02796     int division = ( par <= 2 ); /* false for mul_ */
02797     if ( par < 0 || par > 3 ) {
02798         MLOCK(ErrorMessageLock);
02799         MesPrint("Internal error. Illegal parameter %d in DIVfunction.",par);
02800         MUNLOCK(ErrorMessageLock);
02801         Terminate(-1);
02802     }
02803     /*
02804     Find the function
02805     */
02806     tend = term + *term; tstop = tend - ABS(tend[-1]);
02807     t = term+1;
02808     while ( t < tstop ) {
02809         if ( *t != divrem[par] ) { t += t[1]; continue; }
02810         r = t + FUNHEAD;
02811         tt = t + t[1]; numargs = 0;
02812         while ( r < tt ) {
02813             if ( numargs == 0 ) { arg1 = r; }
02814             if ( numargs == 1 ) { arg2 = r; }
02815             if ( numargs == 2 && *r == -DOLLAREXPRESSION ) { numdol = r[1]; }
02816             numargs++;
02817             NEXTARG(r);
02818         }
02819         if ( numargs == 2 ) break;
02820         if ( division && numargs == 3 ) break;
02821         t = tt;
02822     }
02823     if ( t >= tstop ) {
02824         MLOCK(ErrorMessageLock);
02825         MesPrint("Internal error. Indicated div_ or rem_ function not encountered.");
02826         MUNLOCK(ErrorMessageLock);
02827         Terminate(-1);
02828     }
02829     /*
02830     We have two arguments in arg1 and arg2.
02831     */
02832     if ( division && *arg1 == -SNUMBER && arg1[1] == 0 ) {
02833         if ( *arg2 == -SNUMBER && arg2[1] == 0 ) {
02834             zerozero:;
02835             MLOCK(ErrorMessageLock);
02836             MesPrint("0/0 in either div_ or rem_ function.");
02837             MUNLOCK(ErrorMessageLock);
02838             Terminate(-1);
02839         }
02840         if ( numdol >= 0 ) PutTermInDollar(0,numdol);
02841         return(0);
02842     }
02843     if ( division && *arg2 == -SNUMBER && arg2[1] == 0 ) {
02844         divzero:;
02845         MLOCK(ErrorMessageLock);
02846         MesPrint("Division by zero in either div_ or rem_ function.");
02847         MUNLOCK(ErrorMessageLock);
02848         Terminate(-1);
02849     }
02850     if ( !division ) {
02851         if ( (*arg1 == -SNUMBER && arg1[1] == 0) ||
02852             (*arg2 == -SNUMBER && arg2[1] == 0) ) {
02853             return(0);
02854         }
02855     }

```

```

02856     if ( ( arg1 = ConvertArgument(BHEAD arg1, &type1) ) == 0 ) goto CalledFrom;
02857     if ( ( arg2 = ConvertArgument(BHEAD arg2, &type2) ) == 0 ) goto CalledFrom;
02858     if ( division && *arg1 == 0 ) {
02859         if ( *arg2 == 0 ) {
02860             M_free(arg2,"DIVfunction");
02861             M_free(arg1,"DIVfunction");
02862             goto zerozero;
02863         }
02864         M_free(arg2,"DIVfunction");
02865         M_free(arg1,"DIVfunction");
02866         if ( numdol >= 0 ) PutTermInDollar(0,numdol);
02867         return(0);
02868     }
02869     if ( division && *arg2 == 0 ) {
02870         M_free(arg2,"DIVfunction");
02871         M_free(arg1,"DIVfunction");
02872         goto divzero;
02873     }
02874     if ( !division && (*arg1 == 0 || *arg2 == 0) ) {
02875         M_free(arg2,"DIVfunction");
02876         M_free(arg1,"DIVfunction");
02877         return(0);
02878     }
02879     if ( ( proper1 = PutExtraSymbols(BHEAD arg1,startebuf,&actionflag1) ) == 0 ) goto CalledFrom;
02880     if ( ( proper2 = PutExtraSymbols(BHEAD arg2,startebuf,&actionflag2) ) == 0 ) goto CalledFrom;
02881 /*
02882     if ( type2 == 0 ) M_free(arg2,"DIVfunction");
02883     else {
02884         DOLLARS d = ((DOLLARS)arg2)-1;
02885         if ( d->factors ) M_free(d->factors,"Dollar factors");
02886         M_free(d,"Copy of dollar variable");
02887     }
02888 */
02889     M_free(arg2,"DIVfunction");
02890 /*
02891     if ( type1 == 0 ) M_free(arg1,"DIVfunction");
02892     else {
02893         DOLLARS d = ((DOLLARS)arg1)-1;
02894         if ( d->factors ) M_free(d->factors,"Dollar factors");
02895         M_free(d,"Copy of dollar variable");
02896     }
02897 */
02898     M_free(arg1,"DIVfunction");
02899     if ( par == 0 ) proper3 = poly_div(BHEAD proper1, proper2,0);
02900     else if ( par == 1 ) proper3 = poly_rem(BHEAD proper1, proper2,0);
02901     else if ( par == 2 ) proper3 = poly_inverse(BHEAD proper1, proper2);
02902     else if ( par == 3 ) proper3 = poly_mul(BHEAD proper1, proper2);
02903     if ( proper3 == 0 ) goto CalledFrom;
02904     if ( actionflag1 || actionflag2 ) {
02905         if ( ( arg3 = TakeExtraSymbols(BHEAD proper3,startebuf) ) == 0 ) goto CalledFrom;
02906         M_free(proper3,"DIVfunction");
02907     }
02908     else {
02909         arg3 = proper3;
02910     }
02911     M_free(proper2,"DIVfunction");
02912     M_free(proper1,"DIVfunction");
02913     cbuf[AT.ebufnum].numrhs = startebuf;
02914     if ( *arg3 ) {
02915         termout = AT.WorkPointer;
02916         tlength = tend[-1];
02917         tlength = REDLENG(tlength);
02918         r3 = arg3;
02919         while ( *r3 ) {
02920             tt = term + 1; rr = termout + 1;
02921             while ( tt < t ) *rr++ = *tt++;
02922             r = r3 + 1;
02923             r3 = r3 + *r3;
02924             rstop = r3 - ABS(r3[-1]);
02925             while ( r < rstop ) *rr++ = *r++;
02926             tt += t[1];
02927             while ( tt < tstop ) *rr++ = *ttt++;
02928             rlength = r3[-1];
02929             rlength = REDLENG(rlength);
02930             if ( MulRat(BHEAD (UWORD *)tstop,tlength,(UWORD *)rstop,rlength,
02931                 (UWORD *)rr,&newlength) < 0 ) goto CalledFrom;
02932             rlength = INCLENG(newlength);
02933             rr += ABS(rlength);
02934             rr[-1] = rlength;
02935             *termout = rr - termout;
02936             AT.WorkPointer = rr;
02937             if ( Generator(BHEAD termout,level) ) goto CalledFrom;
02938         }
02939         AT.WorkPointer = termout;
02940     }
02941     M_free(arg3,"DIVfunction");
02942     return(0);

```



```

02943 CalledFrom:
02944     MLOCK(ErrorMessageLock);
02945     MesCall("DIVfunction");
02946     MUNLOCK(ErrorMessageLock);
02947     SETERROR(-1)
02948 }
02949
02950 /*
02951 #] DIVfunction :
02952 #[ MULfunc :
02953
02954     Multiplies two polynomials and puts the results in TermMalloc space.
02955 */
02956
02957 WORD *MULfunc(PHEAD WORD *p1, WORD *p2)
02958 {
02959     WORD *prod, size1, size2, size3, *t, *tfill, *ps1, *ps2, sign1, sign2, error, *p3;
02960     UWORD *num1, *num2, *num3;
02961     int i;
02962     WORD oldsorttype = AR.SortType;
02963     COMPARE oldcompareroutine = (COMPARE) (AR.CompareRoutine);
02964     AR.SortType = SORTHIGHFIRST;
02965     AR.CompareRoutine = (COMPAREDUMMY) (&CompareSymbols);
02966     num3 = NumberMalloc("MULfunc");
02967     prod = TermMalloc("MULfunc");
02968     NewSort(BHEAD0);
02969     while ( *p1 ) {
02970         ps1 = p1+*p1; num1 = (UWORD *) (ps1 - ABS(ps1[-1])); size1 = ps1[-1];
02971         if ( size1 < 0 ) { sign1 = -1; size1 = -size1; }
02972         else sign1 = 1;
02973         size1 = (size1-1)/2;
02974         p3 = p2;
02975         while ( *p3 ) {
02976             ps2 = p3+*p3; num2 = (UWORD *) (ps2 - ABS(ps2[-1])); size2 = ps2[-1];
02977             if ( size2 < 0 ) { sign2 = -1; size2 = -size2; }
02978             else sign2 = 1;
02979             size2 = (size2-1)/2;
02980             if ( MulLong(num1, size1, num2, size2, num3, &size3) ) {
02981                 error = 1;
02982             }
02983             CalledFrom:
02984                 MLOCK(ErrorMessageLock);
02985                 MesPrint(" Error %d", error);
02986                 MesCall("MulFunc");
02987                 MUNLOCK(ErrorMessageLock);
02988                 Terminate(-1);
02989             }
02990             tfill = prod+1;
02991             t = p1+1; while ( t < (WORD *) num1 ) *tfill++ = *t++;
02992             t = p3+1; while ( t < (WORD *) num2 ) *tfill++ = *t++;
02993             t = (WORD *) num3;
02994             for ( i = 0; i < size3; i++ ) *tfill++ = *t++;
02995             *tfill++ = 1;
02996             for ( i = 1; i < size3; i++ ) *tfill++ = 0;
02997             *tfill++ = (2*size3+1)*sign1*sign2;
02998             prod[0] = tfill - prod;
02999             if ( SymbolNormalize(prod) ) { error = 2; goto CalledFrom; }
03000             if ( StoreTerm(BHEAD prod) ) { error = 3; goto CalledFrom; }
03001             p3 += *p3;
03002         }
03003         p1 += *p1;
03004     }
03005     NumberFree(num3, "MULfunc");
03006     EndSort(BHEAD prod, 1);
03007     AR.CompareRoutine = (COMPAREDUMMY) oldcompareroutine;
03008     AR.SortType = oldsorttype;
03009     return(prod);
03010 }
03011
03012 /*
03013 #] MULfunc :
03014 #[ ConvertArgument :
03015
03016     Converts an argument to a general notation in allocated space.
03017 */
03018 WORD *ConvertArgument(PHEAD WORD *arg, int *type)
03019 {
03020     WORD *output, *t, *r;
03021     int i, size;
03022     if ( *arg > 0 ) {
03023         output = (WORD *) Malloc1((*arg)*sizeof(WORD), "ConvertArgument");
03024         i = *arg - ARGHEAD; t = arg + ARGHEAD; r = output;
03025         NCOPY(r, t, i);
03026         *r = 0; *type = 0;
03027         return(output);
03028     }
03029     if ( *arg == -EXPRESSION ) {

```

```

03030     *type = 0;
03031     return(CreateExpression(BHEAD arg[1]));
03032 }
03033 if ( *arg == -DOLLAREXPRESSION ) {
03034     DOLLARS d;
03035     *type = 1;
03036     d = DolToTerms(BHEAD arg[1]);
03037 /*
03038     The problem is that DolToTerms creates a copy of the dollar variable.
03039     If we just return d->where we create a memory leak. Hence we have to
03040     copy the contents of d->where to a new buffer
03041 */
03042     output = (WORD *)Malloc1((d->size+1)*sizeof(WORD),"Copy of dollar content");
03043     WCOPY(output,d->where,d->size+1);
03044     if ( d->factors ) { M_free(d->factors,"Dollar factors"); d->factors = 0; }
03045     M_free(d,"Copy of dollar variable");
03046     return(output);
03047 }
03048 #if ( FUNHEAD > 4 )
03049     size = FUNHEAD+5;
03050 #else
03051     size = 9;
03052 #endif
03053     output = (WORD *)Malloc1(size*sizeof(WORD),"ConvertArgument");
03054     switch(*arg) {
03055         case -SYMBOL:
03056             output[0] = 8; output[1] = SYMBOL; output[2] = 4; output[3] = arg[1];
03057             output[4] = 1; output[5] = 1; output[6] = 1; output[7] = 3;
03058             output[8] = 0;
03059             break;
03060         case -INDEX:
03061         case -VECTOR:
03062         case -MINVECTOR:
03063             output[0] = 7; output[1] = INDEX; output[2] = 3; output[3] = arg[1];
03064             output[4] = 1; output[5] = 1;
03065             if ( *arg == -MINVECTOR ) output[6] = -3;
03066             else output[6] = 3;
03067             output[7] = 0;
03068             break;
03069         case -SNUMBER:
03070             output[0] = 4;
03071             if ( arg[1] < 0 ) {
03072                 output[1] = -arg[1]; output[2] = 1; output[3] = -3;
03073             }
03074             else {
03075                 output[1] = arg[1]; output[2] = 1; output[3] = 3;
03076             }
03077             output[4] = 0;
03078             break;
03079         default:
03080             output[0] = FUNHEAD+4;
03081             output[1] = -*arg;
03082             output[2] = FUNHEAD;
03083             for ( i = 3; i <= FUNHEAD; i++ ) output[i] = 0;
03084             output[FUNHEAD+1] = 1;
03085             output[FUNHEAD+2] = 1;
03086             output[FUNHEAD+3] = 3;
03087             output[FUNHEAD+4] = 0;
03088             break;
03089     }
03090     *type = 0;
03091     return(output);
03092 }
03093
03094 /*
03095 #] ConvertArgument :
03096 #[ ExpandRat :
03097
03098 Expands the denominator of a PolyRatFun in the variable PolyFunVar.
03099 The output is a polyratfun with a single argument.
03100 In the case that there is a polyratfun with more than one argument
03101 or the dirtyflag is on, the argument(s) is/are normalized.
03102 The output overwrites the input.
03103 */
03104
03105 char *TheErrorMessage[] = {
03106     "PolyRatFun not of a type that FORM will expand: incorrect variable inside."
03107     ,"Division by zero in PolyRatFun encountered in ExpandRat."
03108     ,"Irregular code in PolyRatFun encountered in ExpandRat."
03109     ,"Called from ExpandRat."
03110     ,"Workspace overflow. Change parameter Workspace in setup file?"
03111 };
03112
03113 int ExpandRat(PHEAD WORD *fun)
03114 {
03115     WORD *r, *rr, *rrr, *tt, *tnext, *arg1, *arg2, *rmin = 0, *rmininv;
03116     WORD *rcoef, rsize, rcopy, *ow = AT.WorkPointer;

```

```

03117     WORD *numerator, *denominator, *rnext;
03118     WORD *thecopy, *rc, ncoef, newcoef, *m, *mm, nco, *outarg = 0;
03119     UWORD co[2], col[2], co2[2];
03120     WORD OldPolyFunPow = AR.PolyFunPow;
03121     int i, j, minpow = 0, eppow, first, error = 0, ipoly;
03122     if ( fun[1] == FUNHEAD ) { return(0); }
03123     tnext = fun + fun[1];
03124     if ( fun[1] == fun[FUNHEAD]+FUNHEAD ) { /* Single argument */
03125         if ( fun[2] == 0 ) { goto done; }
03126     /*
03127         We have to normalize the argument. This could make it shorter.
03128     */
03129     NormArg;;
03130         if ( outarg == 0 ) outarg = TermMalloc("ExpandRat")+ARGHEAD;
03131         AT.TrimPower = 1;
03132         NewSort(BHEAD0);
03133         r = fun+FUNHEAD+ARGHEAD;
03134         if ( AR.PolyFunExp == 2 ) { /* Find minimum power */
03135             WORD minpow2 = MAXPOWER, *rrm;
03136             rrm = r;
03137             while ( rrm < tnext ) {
03138                 if ( *rrm == 4 ) {
03139                     if ( minpow2 > 0 ) minpow2 = 0;
03140                 }
03141                 else if ( ABS(rrm[*rrm-1]) == (*rrm-1) ) {
03142                     if ( minpow2 > 0 ) minpow2 = 0;
03143                 }
03144                 else {
03145                     if ( rrm[1] == SYMBOL && rrm[2] == 4 && rrm[3] == AR.PolyFunVar ) {
03146                         if ( rrm[4] < minpow2 ) minpow2 = rrm[4];
03147                     }
03148                     else {
03149                         MesPrint("Illegal term in expanded polyratfun.");
03150                         goto onerror;
03151                     }
03152                 }
03153                 rrm += *rrm;
03154             }
03155             AR.PolyFunPow += minpow2;
03156         }
03157         while ( r < tnext ) {
03158             rr = r + *r;
03159             i = *r; rrr = outarg; NCOPY(rrr,r,i);
03160             Normalize(BHEAD outarg);
03161             if ( *outarg > 0 ) StoreTerm(BHEAD outarg);
03162         }
03163         r = fun+FUNHEAD+ARGHEAD;
03164         EndSort(BHEAD r,1);
03165         AT.TrimPower = 0;
03166         if ( *r == 0 ) {
03167             fun[FUNHEAD] = -SNUMBER; fun[FUNHEAD+1] = 0;
03168             fun[1] = FUNHEAD+2;
03169         }
03170         else {
03171             rr = fun+FUNHEAD;
03172             if ( ToFast(rr,rr) ) {
03173                 NEXTARG(rr); fun[1] = rr - fun;
03174             }
03175             else {
03176                 while ( *r ) r += *r;
03177                 *rr = r-rr; rr[1] = CLEANFLAG;
03178                 fun[1] = r - fun;
03179             }
03180         }
03181         fun[2] = CLEANFLAG;
03182         goto done;
03183     }
03184     /*
03185     First test whether we have only AR.PolyFunVar in the denominator
03186     */
03187     tt = fun + FUNHEAD;
03188     arg1 = arg2 = 0;
03189     if ( tt < tnext ) {
03190         arg1 = tt; NEXTARG(tt);
03191         if ( tt < tnext ) {
03192             arg2 = tt; NEXTARG(tt);
03193             if ( tt != tnext ) { arg1 = arg2 = 0; } /* more than two arguments */
03194         }
03195     }
03196     if ( arg2 == 0 ) {
03197         if ( *arg1 < 0 ) { fun[2] = CLEANFLAG; goto done; }
03198         if ( fun[2] == CLEANFLAG ) goto done;
03199         goto NormArg; /* Note: should not come here */
03200     }
03201     /*
03202     Produce the output argument in outarg
03203     */

```

```

03204     if ( outarg == 0 ) outarg = TermMalloc("ExpandRat")+ARGHEAD;
03205
03206     if ( *arg2 <= 0 ) {
03207     /*
03208         These cases are trivial.
03209         We try as much as possible to write the output directly into the
03210         function. We just have to be extremely careful not to overwrite
03211         relevant information before we are finished with it.
03212     */
03213     if ( *arg2 == -SYMBOL && arg2[1] == AR.PolyFunVar ) {
03214         rr = r = fun+FUNHEAD+ARGHEAD;
03215         if ( *arg1 < 0 ) {
03216             if ( *arg1 == -SYMBOL ) {
03217                 if ( arg1[1] == AR.PolyFunVar ) {
03218                     *r++ = 4; *r++ = 1; *r++ = 1; *r++ = 3; *r = 0;
03219                 }
03220             } else {
03221                 *r++ = 10; *r++ = SYMBOL; *r++ = 6;
03222                 *r++ = arg1[1]; *r++ = 1;
03223                 *r++ = AR.PolyFunVar; *r++ = -1;
03224                 *r++ = 1; *r++ = 1; *r++ = 3; *r = 0;
03225                 Normalize(BHEAD rr);
03226             }
03227         }
03228         else if ( *arg1 == -SNUMBER ) {
03229             nco = arg1[1];
03230             if ( nco == 0 ) { *r++ = 0; }
03231             else {
03232                 *r++ = 8; *r++ = SYMBOL; *r++ = 4;
03233                 *r++ = AR.PolyFunVar; *r++ = -1;
03234                 *r++ = ABS(nco); *r++ = 1;
03235                 if ( nco < 0 ) *r++ = -3;
03236                 else *r++ = 3;
03237                 *r = 0;
03238             }
03239         }
03240         else { error = 2; goto onerror; } /* should not happen! */
03241     }
03242     else { /* Multi-term numerator. */
03243         m = arg1+ARGHEAD;
03244         NewSort(BHEAD0); /* Technically maybe not needed */
03245         if ( AR.PolyFunExp == 2 ) { /* Find minimum power */
03246             WORD minpow2 = MAXPOWER, *rrm;
03247             rrm = m;
03248             while ( rrm < arg2 ) {
03249                 if ( *rrm == 4 ) {
03250                     if ( minpow2 > 0 ) minpow2 = 0;
03251                 }
03252                 else if ( ABS(rrm[*rrm-1]) == (*rrm-1) ) {
03253                     if ( minpow2 > 0 ) minpow2 = 0;
03254                 }
03255                 else {
03256                     if ( rrm[1] == SYMBOL && rrm[2] == 4 && rrm[3] == AR.PolyFunVar ) {
03257                         if ( rrm[4] < minpow2 ) minpow2 = rrm[4];
03258                     }
03259                     else {
03260                         MesPrint("Illegal term in expanded polyratfun.");
03261                         goto onerror;
03262                     }
03263                 }
03264                 rrm += *rrm;
03265             }
03266             AR.PolyFunPow += minpow2-1;
03267         }
03268         while ( m < arg2 ) {
03269             r = outarg;
03270             rrr = r++; mm = m + *m;
03271             *r++ = SYMBOL; *r++ = 4; *r++ = AR.PolyFunVar; *r++ = -1;
03272             m++; while ( m < mm ) *r++ = *m++;
03273             *rrr = r-rrr;
03274             Normalize(BHEAD rrr);
03275             StoreTerm(BHEAD rrr);
03276         }
03277         EndSort(BHEAD rr,1);
03278         r = rr; while ( *r ) r += *r;
03279     }
03280     if ( *rr == 0 ) {
03281         fun[FUNHEAD] = -SNUMBER; fun[FUNHEAD+1] = CLEANFLAG;
03282         fun[1] = FUNHEAD+2;
03283     }
03284     else {
03285         rr = fun+FUNHEAD;
03286         *rr = r-rr;
03287         rr[1] = CLEANFLAG;
03288         if ( ToFast(rr,rr) ) {
03289             NEXTARG(rr);
03290             fun[1] = rr - fun;

```

```

03291         }
03292         else { fun[1] = r - fun; }
03293     }
03294     fun[2] = CLEANFLAG;
03295     goto done;
03296 }
03297 else if ( *arg2 == -SNUMBER ) {
03298     rr = r = outarg;
03299     if ( arg2[1] == 0 ) { error = 1; goto onerror; }
03300     if ( *arg1 == -SNUMBER ) { /* Things may not be normalized */
03301         if ( arg1[1] == 0 ) { *r++ = 0; }
03302         else {
03303             col[0] = ABS(arg1[1]); col[1] = 1;
03304             co2[0] = 1; co2[1] = ABS(arg2[1]);
03305             MulRat(BHEAD col,1,co2,1,co,&nco);
03306             *r++ = 4; *r++ = (WORD)(co[0]); *r++ = (WORD)(co[1]);
03307             if ( ( arg1[1] < 0 && arg2[1] > 0 ) ||
03308                 ( arg1[1] > 0 && arg2[1] < 0 ) ) *r++ = -3;
03309             else *r++ = 3;
03310             *r = 0;
03311         }
03312     }
03313     else if ( *arg1 == -SYMBOL ) {
03314         *r++ = 8; *r++ = SYMBOL; *r++ = 4;
03315         *r++ = arg1[1]; *r++ = 1;
03316         *r++ = 1; *r++ = ABS(arg2[1]);
03317         if ( arg2[1] < 0 ) *r++ = -3;
03318         else *r++ = 3;
03319         *r = 0;
03320     }
03321     else if ( *arg1 < 0 ) { error = 2; goto onerror; }
03322     else { /* Multi-term numerator. */
03323         m = arg1+ARGHEAD;
03324         NewSort(BHEAD0); /* Technically maybe not needed */
03325         if ( AR.PolyFunExp == 2 ) { /* Find minimum power */
03326             WORD minpow2 = MAXPOWER, *rrm;
03327             rrm = m;
03328             while ( rrm < arg2 ) {
03329                 if ( *rrm == 4 ) {
03330                     if ( minpow2 > 0 ) minpow2 = 0;
03331                 }
03332                 else if ( ABS(rrm[*rrm-1]) == (*rrm-1) ) {
03333                     if ( minpow2 > 0 ) minpow2 = 0;
03334                 }
03335                 else {
03336                     if ( rrm[1] == SYMBOL && rrm[2] == 4 && rrm[3] == AR.PolyFunVar ) {
03337                         if ( rrm[4] < minpow2 ) minpow2 = rrm[4];
03338                     }
03339                     else {
03340                         MesPrint("Illegal term in expanded polyratfun.");
03341                         goto onerror;
03342                     }
03343                 }
03344                 rrm += *rrm;
03345             }
03346             AR.PolyFunPow += minpow2;
03347         }
03348         while ( m < arg2 ) {
03349             r = rr;
03350             rrr = r++; mm = m + *m;
03351             *r++ = DENOMINATOR; *r++ = FUNHEAD + 2; *r++ = DIRTYFLAG;
03352             FILLFUN3(r);
03353             *r++ = arg2[0]; *r++ = arg2[1];
03354             m++; while ( m < mm ) *r++ = *m++;
03355             *rrr = r-rrr;
03356             if ( r < AT.WorkTop && r >= AT.WorkSpace )
03357                 AT.WorkPointer = r;
03358             Normalize(BHEAD rrr);
03359             if ( ABS(rrr[*rrr-1]) == *rrr-1 ) {
03360                 if ( AR.PolyFunPow >= 0 ) {
03361                     StoreTerm(BHEAD rrr);
03362                 }
03363             }
03364             else if ( rrr[1] == SYMBOL && rrr[2] == 4 &&
03365                 rrr[3] == AR.PolyFunVar && rrr[4] <= AR.PolyFunPow ) {
03366                 StoreTerm(BHEAD rrr);
03367             }
03368         }
03369         EndSort(BHEAD rr,1);
03370     }
03371     r = rr; while ( *r ) r += *r;
03372     i = r-rr;
03373     r = fun + FUNHEAD + ARGHEAD;
03374     NCOPY(r,rr,i);
03375     rr = fun + FUNHEAD;
03376     *rr = r - rr; rr[1] = CLEANFLAG;
03377     if ( ToFast(rr,rr) ) {

```

```

03378         NEXTARG(rr);
03379         fun[1] = rr - fun;
03380     }
03381     else { fun[1] = r - fun; }
03382     fun[2] = CLEANFLAG;
03383     goto done;
03384 }
03385 else { error = 0; goto onerror; }
03386 }
03387 else {
03388     r = arg2+ARGHEAD; /* The argument ends at tnext */
03389     first = 1;
03390     while ( r < tnext ) {
03391         rr = r + *r; rr -= ABS(rr[-1]);
03392         if ( r+1 == rr ) {
03393             if ( first ) { minpow = 0; first = 0; rmin = r; }
03394             else if ( minpow > 0 ) { minpow = 0; rmin = r; }
03395         }
03396         else if ( r[1] != SYMBOL || r[2] != 4 || r[3] != AR.PolyFunVar
03397             || r[4] > MAXPOWER ) { error = 0; goto onerror; }
03398         else if ( first ) { minpow = r[4]; first = 0; rmin = r; }
03399         else if ( r[4] < minpow ) { minpow = r[4]; rmin = r; }
03400         r += *r;
03401     }
03402     /*
03403     We have now:
03404     1: a numerator in arg1 which can contain several variables.
03405     2: a denominator in arg2 with at most only AR.PolyFunVar (ep).
03406     3: the minimum power in the denominator is minpow and the
03407        term with that minimum power is in rmin.
03408     Divide numerator and denominator by this minimum power.
03409     Determine the power range in the numerator.
03410     Call InvPoly.
03411     Multiply by the inverse in such a way that we never take more
03412     powers of ep than necessary.
03413     */
03414     /*
03415     One: put 1/rmin in AT.WorkPointer -> rmininv
03416     */
03417     AT.WorkPointer += AM.MaxTer/sizeof(WORD);
03418     if ( AT.WorkPointer + (AM.MaxTer/sizeof(WORD)) >= AT.WorkTop ) {
03419         error = 4; goto onerror;
03420     }
03421     rmininv = r = AT.WorkPointer;
03422     rr = rmin; i = *rmin; NCOPY(r,rr,i)
03423     if ( minpow != 0 ) { rmininv[4] = -rmininv[4]; }
03424     rsize = ABS(r[-1]);
03425     rcoef = r - rsize;
03426     rsize = (rsize-1)/2; rr = rcoef + rsize;
03427     for ( i = 0; i < rsize; i++ ) {
03428         rcopy = rcoef[i]; rcoef[i] = rr[i]; rr[i] = rcopy;
03429     }
03430     AT.WorkPointer = r;
03431     if ( *arg1 < 0 ) {
03432         ToGeneral(arg1,r,0);
03433         arg1 = r; r += *r; *r++ = 0; rcopy = 0;
03434         AT.WorkPointer = r;
03435     }
03436     else {
03437         r = arg1 + *arg1;
03438         rcopy = *r; *r++ = 0;
03439     }
03440     /*
03441     We can use MultiplyWithTerm.
03442     */
03443     AT.LeaveNegative = 1;
03444     numerator = MultiplyWithTerm(BHEAD arg1+ARGHEAD,rmininv,0);
03445     AT.LeaveNegative = 0;
03446     r[-1] = rcopy;
03447     r = numerator; while ( *r ) r += *r;
03448     AT.WorkPointer = r+1;
03449     rcopy = arg2[*arg2]; arg2[*arg2] = 0;
03450     denominator = MultiplyWithTerm(BHEAD arg2+ARGHEAD,rmininv,1);
03451     arg2[*arg2] = rcopy;
03452     r = denominator; while ( *r ) r += *r;
03453     AT.WorkPointer = r+1;
03454     /*
03455     Now find the minimum power of ep in the numerator.
03456     */
03457     r = numerator;
03458     first = 1;
03459     while ( *r ) {
03460         rr = r + *r; rr -= ABS(rr[-1]);
03461         if ( r+1 == rr ) {
03462             if ( first ) { minpow = 0; first = 0; }
03463             else if ( minpow > 0 ) { minpow = 0; }
03464         }

```

```

03465         else if ( r[1] != SYMBOL ) { error = 0; goto onerror; }
03466     else {
03467         for ( i = 3; i < r[2]; i += 2 ) {
03468             if ( r[i] == AR.PolyFunVar ) {
03469                 if ( first ) { minpow = r[i+1]; first = 0; }
03470                 else if ( r[i+1] < minpow ) minpow = r[i+1];
03471                 break;
03472             }
03473         }
03474         if ( i >= r[2] ) {
03475             if ( first ) { minpow = 0; first = 0; }
03476             else if ( minpow > 0 ) minpow = 0;
03477         }
03478     }
03479     r += *r;
03480 }
03481 /*
03482 We can invert the denominator.
03483 Note that the return value is an offset in AT.pWorkSpace.
03484 Hence there is no need to free memory afterwards.
03485 */
03486 if ( AR.PolyFunExp == 3 ) {
03487     ipoly = InvPoly(BHEAD denominator, AR.PolyFunPow-minpow, AR.PolyFunVar);
03488 }
03489 else {
03490     ipoly = InvPoly(BHEAD denominator, AR.PolyFunPow, AR.PolyFunVar);
03491 }
03492 /*
03493 Now we start the multiplying
03494 */
03495 NewSort(BHEAD0);
03496 r = numerator;
03497 while ( *r ) {
03498     /*
03499         1: Find power of ep.
03500     */
03501     rnext = r + *r;
03502     rrr = rnext - ABS(rnext[-1]);
03503     rr = r+1;
03504     eppow = 0;
03505     if ( rr < rrr ) {
03506         j = rr[1] - 2; rr += 2;
03507         while ( j > 0 ) {
03508             if ( *rr == AR.PolyFunVar ) { eppow = rr[1]; break; }
03509             j -= 2; rr += 2;
03510         }
03511     }
03512     /*
03513         2: Multiply by the proper terms in ipoly
03514     */
03515     for ( i = 0; i <= AR.PolyFunPow-eppow+minpow; i++ ) {
03516         if ( AT.pWorkSpace[ipoly+i] == 0 ) continue;
03517         /*
03518             Copy the term, add i to the power of ep and multiply coef.
03519         */
03520         rc = r;
03521         rr = thecopy = AT.WorkPointer;
03522         while ( rc < rrr ) *rr++ = *rc++;
03523         if ( i != 0 ) {
03524             *rr++ = SYMBOL; *rr++ = 4; *rr++ = AR.PolyFunVar; *rr++ = i;
03525         }
03526         ncoef = REDLENG(rnext[-1]);
03527         MulRat(BHEAD (UWORD *)rrr, ncoef,
03528             (UWORD *) (AT.pWorkSpace[ipoly+i]+1), AT.pWorkSpace[ipoly+i][0],
03529             (UWORD *)rr, &newcoef);
03530         ncoef = ABS(newcoef); rr += 2*ncoef;
03531         newcoef = INCLENG(newcoef);
03532         *rr++ = newcoef;
03533         *thecopy = rr - thecopy;
03534         AT.WorkPointer = rr;
03535         Normalize(BHEAD thecopy);
03536         if ( *thecopy > 0 ) StoreTerm(BHEAD thecopy);
03537         AT.WorkPointer = thecopy;
03538     }
03539     r = rnext;
03540 }
03541 /*
03542 Now we have all.
03543 */
03544 rr = fun + FUNHEAD; r = rr + ARGHEAD;
03545 EndSort(BHEAD r, 1);
03546 if ( *r == 0 ) {
03547     fun[1] = FUNHEAD+2; fun[2] = CLEANFLAG;
03548     fun[FUNHEAD] = -SNUMBER; fun[FUNHEAD+1] = 0;
03549 }
03550 else {
03551     while ( *r ) r += *r;

```

```

03552         rr[0] = r-rr; rr[1] = CLEANFLAG;
03553         if ( ToFast(rr,rr) ) { NEXTARG(rr); fun[1] = rr-fun; }
03554         else { fun[1] = r-fun; }
03555         fun[2] = CLEANFLAG;
03556     }
03557 }
03558 done:
03559     if ( outarg ) TermFree(outarg-ARGHEAD,"ExpandRat");
03560     AR.PolyFunPow = OldPolyFunPow;
03561     AT.WorkPointer = ow;
03562     AN.PolyNormFlag = 1;
03563     return(0);
03564 onerror:
03565     if ( outarg ) TermFree(outarg-ARGHEAD,"ExpandRat");
03566     AR.PolyFunPow = OldPolyFunPow;
03567     AT.WorkPointer = ow;
03568     MLOCK(ErrorMessageLock);
03569     MesPrint(TheErrorMessage[error]);
03570     MUNLOCK(ErrorMessageLock);
03571     Terminate(-1);
03572     return(-1);
03573 }
03574
03575 /*
03576 #] ExpandRat :
03577 #[ InvPoly :
03578
03579 The input polynomial is represented as a sequence of terms in ascending
03580 power. The first coefficient is 1. If we call this 1-a and
03581  $a = \sum_{j=1,n} x^j a(j)$ , and  $b = 1/(1-a)$  we can find the coefficients
03582 of b with the recursion
03583  $b(0) = 1, b(n) = \sum_{j=1,n} a(j) * b(n-j)$ 
03584 The variable is the symbol sym and we need maxpow powers in the answer.
03585 The answer is an array of pointers to the coefficients of the various
03586 powers as rational numbers in the notation signedsize,numerator,denominator
03587 We put these powers in the workspace and the answer is in AT.pWorkspace.
03588 Hence the return value is an offset in the pWorkspace.
03589 A zero pointer indicates that this coefficient is zero.
03590 */
03591
03592 int InvPoly(PHEAD WORD *inpoly, WORD maxpow, WORD sym)
03593 {
03594     int needed, inpointers, outpointers, maxinput = 0, i, j;
03595     WORD *t, *tt, *ttt, *w, *c, *cc, *ccc, lenc, lenc1, lenc2, rc, *c1, *c2;
03596     /*
03597     Step 0: allocate the space
03598     */
03599     needed = (maxpow+1)*2;
03600     WantAddPointers(needed);
03601     inpointers = AT.pWorkPointer;
03602     outpointers = AT.pWorkPointer+maxpow+1;
03603     for ( i = 0; i < needed; i++ ) AT.pWorkspace[inpointers+i] = 0;
03604     /*
03605     Step 1: determine the coefficients in inpoly
03606     often there is a maximum power that is much smaller than maxpow.
03607     keeping track of this can speed up things.
03608     */
03609     t = inpoly;
03610     w = AT.WorkPointer;
03611     while ( *t ) {
03612         if ( *t == 4 ) {
03613             if ( t[1] != 1 || t[2] != 1 || t[3] != 3 ) goto onerror;
03614             AT.pWorkspace[inpointers] = 0;
03615         }
03616         else if ( t[1] != SYMBOL || t[2] != 4 || t[3] != sym || t[4] < 0 ) goto onerror;
03617         else if ( t[4] > maxpow ) {} /* power outside useful range */
03618         else {
03619             if ( t[4] > maxinput ) maxinput = t[4];
03620             AT.pWorkspace[inpointers+t[4]] = w;
03621             tt = t + *t; rc = *--tt; /* we need - the coefficient! */
03622             rc = REDLENG(rc); *w++ = rc;
03623             ttt = t+5;
03624             while ( ttt < tt ) *w++ = *ttt++;
03625         }
03626         t += *t;
03627     }
03628     /*
03629     Step 2: compute the output. b(0) = 1.
03630     then the recursion starts.
03631     */
03632     AT.pWorkspace[outpointers] = w;
03633     *w++ = 1; *w++ = 1; *w++ = 1;
03634     c = TermMalloc("InvPoly");
03635     c1 = TermMalloc("InvPoly");
03636     c2 = TermMalloc("InvPoly");
03637     for ( j = 1; j <= maxpow; j++ ) {
03638     /*

```



```

03639         Start at c = a(j)*b(0) = a(j)
03640     */
03641     if ( ( cc = AT.pWorkspace[inpointers+j] ) != 0 ) {
03642         lenc = *cc++; /* reduced length */
03643         i = 2*ABS(lenc); ccc = c;
03644         NCOPY(ccc,cc,i);
03645     }
03646     else { lenc = 0; }
03647     for ( i = Min(j-1,maxinput); i > 0; i-- ) {
03648     /*
03649         c -> c + a(i)*b(j-i)
03650     */
03651         if ( AT.pWorkspace[inpointers+i] == 0
03652             || AT.pWorkspace[outpointers+j-i] == 0 ) {
03653         }
03654         else {
03655             if ( MulRat(BHEAD (UWORD
*) (AT.pWorkspace[inpointers+i]+1),AT.pWorkspace[inpointers+i][0],
03656                 (UWORD *) (AT.pWorkspace[outpointers+j-i]+1),AT.pWorkspace[outpointers+j-i][0],
03657                 (UWORD *)c1,&lenc1) ) goto calcerror;
03658             if ( lenc == 0 ) {
03659                 cc = c; c = c1; c1 = cc;
03660                 lenc = lenc1;
03661             }
03662             else {
03663                 if ( AddRat(BHEAD (UWORD *)c,lenc, (UWORD *)c1,lenc1, (UWORD *)c2,&lenc2) )
03664                     goto calcerror;
03665                 cc = c; c = c2; c2 = cc;
03666                 lenc = lenc2;
03667             }
03668         }
03669     }
03670     /*
03671     Copy c to the proper location
03672     */
03673     if ( lenc == 0 ) AT.pWorkspace[outpointers+j] = 0;
03674     else {
03675         AT.pWorkspace[outpointers+j] = w;
03676         *w++ = lenc;
03677         i = 2*ABS(lenc); ccc = c;
03678         NCOPY(w,ccc,i);
03679     }
03680 }
03681 AT.WorkPointer = w;
03682 TermFree(c2,"InvPoly");
03683 TermFree(c1,"InvPoly");
03684 TermFree(c,"InvPoly");
03685
03686 return(outpointers);
03687 onerror:
03688 MLOCK(ErrorMessageLock);
03689 MesPrint("Incorrect symbol field in InvPoly.");
03690 MUNLOCK(ErrorMessageLock);
03691 Terminate(-1);
03692 return(-1);
03693 calcerror:
03694 MLOCK(ErrorMessageLock);
03695 MesPrint("Called from InvPoly.");
03696 MUNLOCK(ErrorMessageLock);
03697 Terminate(-1);
03698 return(-1);
03699 }
03700
03701 /*
03702 #] InvPoly :
03703 */

```

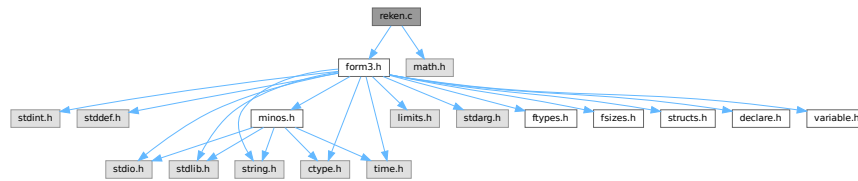
4.119 reken.c File Reference

```

#include "form3.h"
#include <math.h>

```

Include dependency graph for reken.c:



Macros

- `#define GCDMAX 3`
- `#define NEWTRICK 1`
- `#define COPYLONG(x1, nx1, x2, nx2) { int i; for(i=0;i<ABS(nx2);i++)x1[i]=x2[i];nx1=nx2; }`
- `#define WARMUP 6`

Functions

- void `Pack` (UWORD *a, WORD *na, UWORD *b, WORD nb)
- void `UnPack` (UWORD *a, WORD na, WORD *denom, WORD *numer)
- int `Mully` (PHEAD UWORD *a, WORD *na, UWORD *b, WORD nb)
- int `Divvy` (PHEAD UWORD *a, WORD *na, UWORD *b, WORD nb)
- int `AddRat` (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- int `MulRat` (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- int `DivRat` (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- int `Simplify` (PHEAD UWORD *a, WORD *na, UWORD *b, WORD *nb)
- int `AccumGCD` (PHEAD UWORD *a, WORD *na, UWORD *b, WORD nb)
- int `TakeRatRoot` (UWORD *a, WORD *n, WORD power)
- int `AddLong` (UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- int `AddPLon` (UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- void `SubPLon` (UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- int `MulLong` (UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- int `BigLong` (UWORD *a, WORD na, UWORD *b, WORD nb)
- int `DivLong` (UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc, UWORD *d, WORD *nd)
- int `RaisPow` (PHEAD UWORD *a, WORD *na, UWORD b)
- void `RaisPowCached` (PHEAD WORD x, WORD n, UWORD **c, WORD *nc)
- WORD `RaisPowMod` (WORD x, WORD n, WORD m)
- int `NormalModulus` (UWORD *a, WORD *na)
- int `MakeInverses` (void)
- int `GetModInverses` (WORD m1, WORD m2, WORD *im1, WORD *im2)
- int `GetLongModInverses` (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *ia, WORD *nia, UWORD *ib, WORD *nib)
- int `Product` (UWORD *a, WORD *na, WORD b)
- UWORD `Quotient` (UWORD *a, WORD *na, WORD b)
- WORD `Remain10` (UWORD *a, WORD *na)
- WORD `Remain4` (UWORD *a, WORD *na)
- void `PrtLong` (UWORD *a, WORD na, UBYTE *s)
- int `GetLong` (UBYTE *s, UWORD *a, WORD *na)
- int `GcdLong` (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- int `GetBinom` (UWORD *a, WORD *na, WORD i1, WORD i2)

- int [LcmLong](#) (PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
- int [TakeLongRoot](#) (UWORD *a, WORD *n, WORD power)
- int [MakeRational](#) (WORD a, WORD m, WORD *b, WORD *c)
- int [MakeLongRational](#) (PHEAD UWORD *a, WORD na, UWORD *m, WORD nm, UWORD *b, WORD *nb)
- WORD [CompCoef](#) (WORD *term1, WORD *term2)
- int [Modulus](#) (WORD *term)
- int [TakeModulus](#) (UWORD *a, WORD *na, UWORD *cmodvec, WORD ncm, WORD par)
- int [TakeNormalModulus](#) (UWORD *a, WORD *na, UWORD *c, WORD nc, WORD par)
- int [MakeModTable](#) (void)
- int [Factorial](#) (PHEAD WORD n, UWORD *a, WORD *na)
- int [Bernoulli](#) (WORD n, UWORD *a, WORD *na)
- WORD [NextPrime](#) (PHEAD WORD num)
- WORD [Moebius](#) (PHEAD WORD nn)
- void [iniwranf](#) (PHEAD0)
- UWORD [wranf](#) (PHEAD0)
- UWORD [iranf](#) (PHEAD UWORD imax)
- UBYTE * [PreRandom](#) (UBYTE *s)

4.119.1 Detailed Description

This file contains the numerical routines. The arithmetic in FORM is normally over the rational numbers. Hence there are routines for dealing with integers and with rational of 'arbitrary precision' (within limits) There are also routines for that calculus modulus an integer. In addition there are the routines for factorials and Bernoulli numbers. The random number function is currently only for internal purposes.

Definition in file [reken.c](#).

4.119.2 Macro Definition Documentation

4.119.2.1 GCDMAX

```
#define GCDMAX 3
```

Definition at line 49 of file [reken.c](#).

4.119.2.2 NEWTRICK

```
#define NEWTRICK 1
```

Definition at line 51 of file [reken.c](#).

4.119.2.3 COPYLONG

```
#define COPYLONG(
    x1,
    nx1,
    x2,
    nx2 ) { int i; for(i=0;i<ABS(nx2);i++) x1[i]=x2[i];nx1=nx2; }
```

Definition at line 2902 of file [reken.c](#).

4.119.2.4 WARMUP

```
#define WARMUP 6
```

Definition at line [3802](#) of file [reken.c](#).

4.119.3 Function Documentation

4.119.3.1 Pack()

```
void Pack (
    UWORD * a,
    WORD * na,
    UWORD * b,
    WORD nb )
```

Definition at line [62](#) of file [reken.c](#).

4.119.3.2 UnPack()

```
void UnPack (
    UWORD * a,
    WORD na,
    WORD * denom,
    WORD * numer )
```

Definition at line [100](#) of file [reken.c](#).

4.119.3.3 Mully()

```
int Mully (
    PHEAD UWORD * a,
    WORD * na,
    UWORD * b,
    WORD nb )
```

Definition at line [126](#) of file [reken.c](#).

4.119.3.4 Divvy()

```
int Divvy (
    PHEAD UWORD * a,
    WORD * na,
    UWORD * b,
    WORD nb )
```

Definition at line [172](#) of file [reken.c](#).

4.119.3.5 AddRat()

```
int AddRat (
    PHEAD UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc )
```

Definition at line 210 of file [reken.c](#).

4.119.3.6 MulRat()

```
int MulRat (
    PHEAD UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc )
```

Definition at line 378 of file [reken.c](#).

4.119.3.7 DivRat()

```
int DivRat (
    PHEAD UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc )
```

Definition at line 500 of file [reken.c](#).

4.119.3.8 Simplify()

```
int Simplify (
    PHEAD UWORD * a,
    WORD * na,
    UWORD * b,
    WORD * nb )
```

Definition at line 531 of file [reken.c](#).

4.119.3.9 AccumGCD()

```
int AccumGCD (
    PHEAD UWORD * a,
    WORD * na,
    UWORD * b,
    WORD nb )
```

Definition at line 648 of file [reken.c](#).

4.119.3.10 TakeRatRoot()

```
int TakeRatRoot (
    UWORD * a,
    WORD * n,
    WORD power )
```

Definition at line [681](#) of file [reken.c](#).

4.119.3.11 AddLong()

```
int AddLong (
    UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc )
```

Definition at line [706](#) of file [reken.c](#).

4.119.3.12 AddPLon()

```
int AddPLon (
    UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc )
```

Definition at line [751](#) of file [reken.c](#).

4.119.3.13 SubPLon()

```
void SubPLon (
    UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc )
```

Definition at line [804](#) of file [reken.c](#).

4.119.3.14 MulLong()

```
int MulLong (
    UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc )
```

Definition at line [842](#) of file [reken.c](#).

4.119.3.15 BigLong()

```
int BigLong (
    UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb )
```

Definition at line 945 of file [reken.c](#).

4.119.3.16 DivLong()

```
int DivLong (
    UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc,
    UWORD * d,
    WORD * nd )
```

Definition at line 973 of file [reken.c](#).

4.119.3.17 RaisPow()

```
int RaisPow (
    PHEAD UWORD * a,
    WORD * na,
    UWORD b )
```

Definition at line 1229 of file [reken.c](#).

4.119.3.18 RaisPowCached()

```
void RaisPowCached (
    PHEAD WORD x,
    WORD n,
    UWORD ** c,
    WORD * nc )
```

Computes power x^n and caches the value

4.119.4 Description

Calculates the power x^n and stores the results for caching purposes. The pointer c (i.e., the pointer, and not what it points to) is overwritten. What it points to should not be overwritten in the calling function.

4.119.5 Notes

- Caching is done in `AT.small_power[]`. This array is extended if necessary.

Definition at line 1297 of file [reken.c](#).

Referenced by [poly::divmod_heap\(\)](#), [poly::divmod_univar\(\)](#), and [poly::mul_heap\(\)](#).

4.119.5.1 RaisPowMod()

```
WORD RaisPowMod (
    WORD x,
    WORD n,
    WORD m )
```

Definition at line 1384 of file [reken.c](#).

4.119.5.2 NormalModulus()

```
int NormalModulus (
    UWORD * a,
    WORD * na )
```

Brings a modular representation in the range $-p/2$ to $+p/2$ The return value tells whether anything was done. Routine made in the general modulus revamp of July 2008 (JV).

Definition at line 1404 of file [reken.c](#).

Referenced by [AddCoef\(\)](#), and [MergePatches\(\)](#).

4.119.5.3 MakeInverses()

```
int MakeInverses (
    void )
```

Makes a table of inverses in modular calculus The modulus is in `AC.cmod` and `AC.ncmod` One should notice that the table of inverses can only be made if the modulus fits inside a single FORM word. Otherwise the table lookup becomes too difficult and the table too long.

Definition at line 1441 of file [reken.c](#).

References [GetModInverses\(\)](#).

Here is the call graph for this function:



4.119.5.4 GetModInverses()

```
int GetModInverses (
    WORD m1,
    WORD m2,
    WORD * im1,
    WORD * im2 )
```

Input m1 and m2, which are relative prime. determines $a*m1+b*m2 = 1$ (and 1 is the gcd of m1 and m2) then $a*m1 = 1 \bmod m2$ and hence $im1 = a$. and $b*m2 = 1 \bmod m1$ and hence $im2 = b$. Set $m1 = 0*m1+1*m2 = a1*m1+b1*m2$ $m2 = 1*m1+0*m2 = a2*m1+b2*m2$ If everything is OK, the return value is zero

Definition at line [1477](#) of file [reken.c](#).

Referenced by [poly::divmod_heap\(\)](#), [poly::divmod_univar\(\)](#), [MakeDollarMod\(\)](#), [MakeInverses\(\)](#), [MakeMod\(\)](#), [TakeContent\(\)](#), and [TakeSymbolContent\(\)](#).

4.119.5.5 GetLongModInverses()

```
int GetLongModInverses (
    PHEAD UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * ia,
    WORD * nia,
    UWORD * ib,
    WORD * nib )
```

Definition at line [1509](#) of file [reken.c](#).

4.119.5.6 Product()

```
int Product (
    UWORD * a,
    WORD * na,
    WORD b )
```

Definition at line [1586](#) of file [reken.c](#).

4.119.5.7 Quotient()

```
UWORD Quotient (
    UWORD * a,
    WORD * na,
    WORD b )
```

Definition at line [1621](#) of file [reken.c](#).

4.119.5.8 Remain10()

```
WORD Remain10 (
    UWORD * a,
    WORD * na )
```

Definition at line 1660 of file [reken.c](#).

4.119.5.9 Remain4()

```
WORD Remain4 (
    UWORD * a,
    WORD * na )
```

Definition at line 1687 of file [reken.c](#).

4.119.5.10 PrtLong()

```
void PrtLong (
    UWORD * a,
    WORD na,
    UBYTE * s )
```

Definition at line 1713 of file [reken.c](#).

4.119.5.11 GetLong()

```
int GetLong (
    UBYTE * s,
    UWORD * a,
    WORD * na )
```

Definition at line 1775 of file [reken.c](#).

4.119.5.12 GcdLong()

```
int GcdLong (
    PHEAD UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD nc )
```

Definition at line 2277 of file [reken.c](#).

4.119.5.13 GetBinom()

```
int GetBinom (
    UWORD * a,
    WORD * na,
    WORD i1,
    WORD i2 )
```

Definition at line 2668 of file [reken.c](#).

4.119.5.14 LcmLong()

```
int LcmLong (
    PHEAD UWORD * a,
    WORD na,
    UWORD * b,
    WORD nb,
    UWORD * c,
    WORD * nc )
```

Definition at line 2702 of file [reken.c](#).

4.119.5.15 TakeLongRoot()

```
int TakeLongRoot (
    UWORD * a,
    WORD * n,
    WORD power )
```

Definition at line 2736 of file [reken.c](#).

4.119.5.16 MakeRational()

```
int MakeRational (
    WORD a,
    WORD m,
    WORD * b,
    WORD * c )
```

Definition at line 2856 of file [reken.c](#).

4.119.5.17 MakeLongRational()

```
int MakeLongRational (
    PHEAD UWORD * a,
    WORD na,
    UWORD * m,
    WORD nm,
    UWORD * b,
    WORD * nb )
```

Definition at line 2904 of file [reken.c](#).

4.119.5.18 CompCoef()

```
WORD CompCoef (
    WORD * term1,
    WORD * term2 )
```

Routine takes $a_1 \bmod m_1$ and $a_2 \bmod m_2$ and returns $a \bmod m_1*m_2$ with
 $a \bmod m_1 = a_1$ and $a \bmod m_2 = a_2$

Chinese remainder: $a \bmod (m_1*m_2) = q_1*m_1 + a_1$ $a \bmod (m_1*m_2) = q_2*m_2 + a_2$ Compute n_1 such that $(n_1*m_1) \bmod m_2$ is one
 Compute n_2 such that $(n_2*m_2) \bmod m_1$ is one Then $(a_1*n_2*m_2 + a_2*n_1*m_1) \bmod (m_1*m_2)$ is $a \bmod (m_1*m_2)$

Definition at line 3048 of file [reken.c](#).

Referenced by [Compare1\(\)](#).

4.119.5.19 Modulus()

```
int Modulus (
    WORD * term )
```

Definition at line 3103 of file [reken.c](#).

4.119.5.20 TakeModulus()

```
int TakeModulus (
    UWORD * a,
    WORD * na,
    UWORD * cmodvec,
    WORD ncmod,
    WORD par )
```

Definition at line 3150 of file [reken.c](#).

4.119.5.21 TakeNormalModulus()

```
int TakeNormalModulus (
    UWORD * a,
    WORD * na,
    UWORD * c,
    WORD nc,
    WORD par )
```

Definition at line 3322 of file [reken.c](#).

4.119.5.22 MakeModTable()

```
int MakeModTable (
    void )
```

Definition at line 3364 of file [reken.c](#).

4.119.5.23 Factorial()

```
int Factorial (
    PHEAD WORD n,
    UWORD * a,
    WORD * na )
```

Definition at line 3450 of file [reken.c](#).

4.119.5.24 Bernoulli()

```
int Bernoulli (
    WORD n,
    UWORD * a,
    WORD * na )
```

Definition at line 3530 of file [reken.c](#).

4.119.5.25 NextPrime()

```
WORD NextPrime (
    PHEAD WORD num )
```

Gives the next prime number in the list of prime numbers.

If the list isn't long enough we expand it. For ease in ParForm and because these lists shouldn't be very big we let each worker keep its own list.

The list is cut off at MAXPOWER, because we don't want to get into trouble that the power of a variable gets larger than the prime number.

Definition at line 3665 of file [reken.c](#).

4.119.5.26 Moebius()

```
WORD Moebius (
    PHEAD WORD nn )
```

Definition at line 3729 of file [reken.c](#).

4.119.5.27 iniwranf()

```
void iniwranf (
    PHEAD0 )
```

Definition at line 3820 of file [reken.c](#).

4.119.5.28 wranf()

```
UWORD wranf (
    PHEAD0 )
```

Definition at line [3869](#) of file [reken.c](#).

4.119.5.29 iranf()

```
UWORD iranf (
    PHEAD UWORD imax )
```

Definition at line [3888](#) of file [reken.c](#).

4.119.5.30 PreRandom()

```
UBYTE * PreRandom (
    UBYTE * s )
```

Definition at line [3913](#) of file [reken.c](#).

4.120 reken.c

[Go to the documentation of this file.](#)

```
00001
00011 /* #[] License : */
00012 /*
00013 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00014 *   When using this file you are requested to refer to the publication
00015 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00016 *   This is considered a matter of courtesy as the development was paid
00017 *   for by FOM the Dutch physics granting agency and we would like to
00018 *   be able to track its scientific use to convince FOM of its value
00019 *   for the community.
00020 *
00021 *   This file is part of FORM.
00022 *
00023 *   FORM is free software: you can redistribute it and/or modify it under the
00024 *   terms of the GNU General Public License as published by the Free Software
00025 *   Foundation, either version 3 of the License, or (at your option) any later
00026 *   version.
00027 *
00028 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00029 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00030 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00031 *   details.
00032 *
00033 *   You should have received a copy of the GNU General Public License along
00034 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00035 */
00036 /* #[] License : */
00037 /*
00038 *   #[] Includes : reken.c
00039 */
00040
00041 #include "form3.h"
00042 #include <math.h>
00043
00044 #ifdef WITHGMP
00045 #include <gmp.h>
00046 #define GMPSPREAD (GMP_LIMB_BITS/BITSINWORD)
00047 #endif
00048
00049 #define GCDMAX 3
00050
00051 #define NEWTRICK 1
```

```

00052 /*
00053  #] Includes :
00054  #[ RekenRational :
00055      #[ Pack :          void Pack(a,na,b,nb)
00056
00057      Packs the contents of the numerator a and the denominator b into
00058      one normalized fraction a.
00059  */
00060
00061 void Pack(UWORD *a, WORD *na, UWORD *b, WORD nb)
00062 {
00063     WORD c, sgn = 1, i;
00064     UWORD *to,*from;
00065     if ( (c = *na) == 0 ) {
00066         MLOCK(ErrorMessageLock);
00067         MesPrint("Caught a zero in Pack");
00068         MUNLOCK(ErrorMessageLock);
00069         return;
00070     }
00071     if ( nb == 0 ) {
00072         MLOCK(ErrorMessageLock);
00073         MesPrint("Division by zero in Pack");
00074         MUNLOCK(ErrorMessageLock);
00075         return;
00076     }
00077     if ( *na < 0 ) { sgn = -sgn; c = -c; }
00078     if ( nb < 0 ) { sgn = -sgn; nb = -nb; }
00079     *na = MaX(c,nb);
00080     to = a + c;
00081     i = *na - c;
00082     while ( --i >= 0 ) *to++ = 0;
00083     i = *na - nb;
00084     from = b;
00085     NCOPY(to,from,nb);
00086     while ( --i >= 0 ) *to++ = 0;
00087     if ( sgn < 0 ) *na = -*na;
00088 }
00089
00090
00091 /*
00092     #] Pack :
00093     #[ UnPack :          void UnPack(a,na,denom,number)
00094
00095     Determines the sizes of the numerator and the denominator in the
00096     normalized fraction a with length na.
00097  */
00098
00099 void UnPack(UWORD *a, WORD na, WORD *denom, WORD *number)
00100 {
00101     UWORD *pos;
00102     WORD i, sgn = na;
00103     if ( na < 0 ) { na = -na; }
00104     i = na;
00105     if ( i > 1 ) {          /* Find the respective leading words */
00106         a += i;
00107         a--;
00108         pos = a + i;
00109         while ( !(*a) ) { i--; a--; }
00110         while ( !(*pos) ) { na--; pos--; }
00111     }
00112     *denom = na;
00113     if ( sgn < 0 ) i = -i;
00114     *number = i;
00115 }
00116
00117
00118 /*
00119     #] UnPack :
00120     #[ Mully :          WORD Mully(a,na,b,nb)
00121
00122     Multiplies the rational a by the Long b.
00123  */
00124
00125 int Mully(PHEAD UWORD *a, WORD *na, UWORD *b, WORD nb)
00126 {
00127     GETBIDENTITY
00128     UWORD *d,*e;
00129     WORD i, sgn = 1;
00130     WORD nd, ne, adenom, anumer;
00131     if ( !nb ) { *na = 0; return(0); }
00132     else if ( *b == 1 ) {
00133         if ( nb == 1 ) return(0);
00134         else if ( nb == -1 ) { *na = -*na; return(0); }
00135     }
00136     if ( *na < 0 ) { sgn = -sgn; *na = -*na; }
00137     if ( nb < 0 ) { sgn = -sgn; nb = -nb; }

```

```

00139     UnPack(a,*na,&adenom,&anumer);
00140     d = NumberMalloc("Mully"); e = NumberMalloc("Mully");
00141     for ( i = 0; i < nb; i++ ) { e[i] = *b++; }
00142     ne = nb;
00143     if ( Simplify(BHEAD a+*na,&adenom,e,&ne) ) goto MullyEr;
00144     if ( Mullong(a,anumer,e,ne,d,&nd) ) goto MullyEr;
00145     b = a+*na;
00146     for ( i = 0; i < *na; i++ ) { e[i] = *b++; }
00147     ne = adenom;
00148     *na = nd;
00149     b = d;
00150     *na = nd;
00151     for ( i = 0; i < *na; i++ ) { a[i] = *b++; }
00152     Pack(a,na,e,ne);
00153     if ( sgn < 0 ) *na = -*na;
00154     NumberFree(d,"Mully"); NumberFree(e,"Mully");
00155     return(0);
00156 MullyEr:
00157     MLOCK(ErrorMessageLock);
00158     MesCall("Mully");
00159     MUNLOCK(ErrorMessageLock);
00160     NumberFree(d,"Mully"); NumberFree(e,"Mully");
00161     SETERROR(-1)
00162 }
00163
00164 /*
00165     #] Mully :
00166     #[ Divvy :          WORD Divvy(a,na,b,nb)
00167
00168     Divides the rational a by the Long b.
00169 */
00170
00171 int Divvy(PHEAD UWORD *a, WORD *na, UWORD *b, WORD nb)
00172 {
00173     GETBIDENTITY
00174     UWORD *d,*e;
00175     WORD i, sgn = 1;
00176     WORD nd, ne, adenom, anumer;
00177     if ( !nb ) {
00178         MLOCK(ErrorMessageLock);
00179         MesPrint("Division by zero in Divvy");
00180         MUNLOCK(ErrorMessageLock);
00181         return(-1);
00182     }
00183     d = NumberMalloc("Divvy"); e = NumberMalloc("Divvy");
00184     if ( nb < 0 ) { sgn = -sgn; nb = -nb; }
00185     if ( *na < 0 ) { sgn = -sgn; *na = -*na; }
00186     UnPack(a,*na,&adenom,&anumer);
00187     for ( i = 0; i < nb; i++ ) { e[i] = *b++; }
00188     ne = nb;
00189     if ( Simplify(BHEAD a,&anumer,e,&ne) ) goto DivvyEr;
00190     if ( Mullong(a+*na,adenom,e,ne,d,&nd) ) goto DivvyEr;
00191     *na = anumer;
00192     Pack(a,na,d,nd);
00193     if ( sgn < 0 ) *na = -*na;
00194     NumberFree(d,"Divvy"); NumberFree(e,"Divvy");
00195     return(0);
00196 DivvyEr:
00197     MLOCK(ErrorMessageLock);
00198     MesCall("Divvy");
00199     MUNLOCK(ErrorMessageLock);
00200     NumberFree(d,"Divvy"); NumberFree(e,"Divvy");
00201     SETERROR(-1)
00202 }
00203
00204 /*
00205     #] Divvy :
00206     #[ AddRat :          WORD AddRat(a,na,b,nb,c,nc)
00207
00208 */
00209
00210 int AddRat(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
00211 {
00212     GETBIDENTITY
00213     UWORD *d, *e, *f, *g;
00214     WORD nd, ne, nf, ng, adenom, anumer, bdenom, bnumer;
00215     if ( !na ) {
00216         WORD i;
00217         *nc = nb;
00218         if ( nb < 0 ) nb = -nb;
00219         nb *= 2;
00220         for ( i = 0; i < nb; i++ ) *c++ = *b++;
00221         return(0);
00222     }
00223     else if ( !nb ) {
00224         WORD i;
00225         *nc = na;

```



```

00226     if ( na < 0 ) na = -na;
00227     na *= 2;
00228     for ( i = 0; i < na; i++ ) *c++ = *a++;
00229     return(0);
00230 }
00231 else if ( b[1] == 1 && a[1] == 1 ) {
00232     if ( na == 1 ) {
00233         if ( nb == 1 ) {
00234             *c = *a + *b;
00235             c[1] = 1;
00236             if ( *c < *a ) { c[2] = 1; c[3] = 0; *nc = 2; }
00237             else { *nc = 1; }
00238             return(0);
00239         }
00240         else if ( nb == -1 ) {
00241             if ( *b > *a ) {
00242                 *c = *b - *a; *nc = -1;
00243             }
00244             else if ( *b < *a ) {
00245                 *c = *a - *b; *nc = 1;
00246             }
00247             else *nc = 0;
00248             c[1] = 1;
00249             return(0);
00250         }
00251     }
00252     else if ( na == -1 ) {
00253         if ( nb == -1 ) {
00254             c[1] = 1;
00255             *c = *a + *b;
00256             if ( *c < *a ) { c[2] = 1; c[3] = 0; *nc = -2; }
00257             else { *nc = -1; }
00258             return(0);
00259         }
00260         else if ( nb == 1 ) {
00261             if ( *b > *a ) {
00262                 *c = *b - *a; *nc = 1;
00263             }
00264             else if ( *b < *a ) {
00265                 *c = *a - *b; *nc = -1;
00266             }
00267             else *nc = 0;
00268             c[1] = 1;
00269             return(0);
00270         }
00271     }
00272 }
00273 UnPack(a,na,&adenom,&anumer);
00274 UnPack(b,nb,&bdenom,&bnumber);
00275 if ( na < 0 ) na = -na;
00276 if ( nb < 0 ) nb = -nb;
00277 if ( na == 1 && nb == 1 ) {
00278     RLONG t1, t2, t3;
00279     t3 = ((RLONG)a[1])*((RLONG)b[1]);
00280     t1 = ((RLONG)a[0])*((RLONG)b[1]);
00281     t2 = ((RLONG)a[1])*((RLONG)b[0]);
00282     if ( ( anumer > 0 && bnumber > 0 ) || ( anumer < 0 && bnumber < 0 ) ) {
00283         if ( ( t1 = t1 + t2 ) < t2 ) {
00284             c[2] = 1;
00285             c[0] = (UWORD)t1;
00286             c[1] = (UWORD)(t1 » BITSINWORD);
00287             *nc = 3;
00288         }
00289         else {
00290             c[0] = (UWORD)t1;
00291             if ( ( c[1] = (UWORD)(t1 » BITSINWORD) ) != 0 ) *nc = 2;
00292             else *nc = 1;
00293         }
00294     }
00295     else {
00296         if ( t1 == t2 ) { *nc = 0; return(0); }
00297         if ( t1 > t2 ) {
00298             t1 -= t2;
00299         }
00300         else {
00301             t1 = t2 - t1;
00302             anumer = -anumer;
00303         }
00304         c[0] = (UWORD)t1;
00305         if ( ( c[1] = (UWORD)(t1 » BITSINWORD) ) != 0 ) *nc = 2;
00306         else *nc = 1;
00307     }
00308     if ( anumer < 0 ) *nc = -*nc;
00309     d = NumberMalloc("AddRat");
00310     d[0] = (UWORD)t3;
00311     if ( ( d[1] = (UWORD)(t3 » BITSINWORD) ) != 0 ) nd = 2;
00312     else nd = 1;

```

```

00313         if ( Simplify(BHEAD c,nc,d,&nd) ) goto AddRer1;
00314     }
00315 /*
00316     else if ( a[na] == 1 && b[nb] == 1 && adenom == 1 && bdenom == 1 ) {
00317         if ( AddLong(a,na,b,nb,c,&nc) ) goto AddRer2;
00318         i = ABS(nc); d = c + i; *d++ = 1;
00319         while ( --i > 0 ) *d++ = 0 ;
00320         return(0);
00321     }
00322 */
00323     else {
00324         d = NumberMalloc("AddRat"); e = NumberMalloc("AddRat");
00325         f = NumberMalloc("AddRat"); g = NumberMalloc("AddRat");
00326         if ( GcdLong(BHEAD a+na,adenom,b+nb,bdenom,d,&nd) ) goto AddRer;
00327         if ( *d == 1 && nd == 1 ) nd = 0;
00328         if ( nd ) {
00329             if ( DivLong(a+na,adenom,d,nd,e,&ne,c,nc) ) goto AddRer;
00330             if ( DivLong(b+nb,bdenom,d,nd,f,&nf,c,nc) ) goto AddRer;
00331             if ( MulLong(a,anumer,f,nf,c,nc) ) goto AddRer;
00332             if ( MulLong(b,bnumer,e,ne,g,&ng) ) goto AddRer;
00333         }
00334         else {
00335             if ( MulLong(a+na,adenom,b,bnumer,c,nc) ) goto AddRer;
00336             if ( MulLong(b+nb,bdenom,a,anumer,g,&ng) ) goto AddRer;
00337         }
00338         if ( AddLong(c,*nc,g,ng,c,nc) ) goto AddRer;
00339         if ( !*nc ) {
00340             NumberFree(g,"AddRat"); NumberFree(f,"AddRat");
00341             NumberFree(e,"AddRat"); NumberFree(d,"AddRat");
00342             return(0);
00343         }
00344         if ( nd ) {
00345             if ( Simplify(BHEAD c,nc,d,&nd) ) goto AddRer;
00346             if ( MulLong(e,ne,d,nd,g,&ng) ) goto AddRer;
00347             if ( MulLong(g,ng,f,nf,d,&nd) ) goto AddRer;
00348         }
00349         else {
00350             if ( MulLong(a+na,adenom,b+nb,bdenom,d,&nd) ) goto AddRer;
00351         }
00352         NumberFree(g,"AddRat"); NumberFree(f,"AddRat"); NumberFree(e,"AddRat");
00353     }
00354     Pack(c,nc,d,nd);
00355     NumberFree(d,"AddRat");
00356     return(0);
00357 AddRer:
00358     NumberFree(g,"AddRat"); NumberFree(f,"AddRat"); NumberFree(e,"AddRat");
00359 AddRer1:
00360     NumberFree(d,"AddRat");
00361 /* AddRer2: */
00362     MLOCK(ErrorMessageLock);
00363     MesCall("AddRat");
00364     MUNLOCK(ErrorMessageLock);
00365     SETERROR(-1)
00366 }
00367
00368 /*
00369     #] AddRat :
00370     #[ MulRat :          WORD MulRat(a,na,b,nb,c,nc)
00371
00372     Multiplies the rationals a and b. The Gcd of the individual
00373     pieces is divided out first to minimize the chances of spurious
00374     overflows.
00375 */
00376 */
00377
00378 int MulRat(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
00379 {
00380     WORD i;
00381     WORD sgn = 1;
00382     if ( *b == 1 && b[1] == 1 ) {
00383         if ( nb == 1 ) {
00384             *nc = na;
00385             i = ABS(na); i *= 2;
00386             while ( --i >= 0 ) *c++ = *a++;
00387             return(0);
00388         }
00389         else if ( nb == -1 ) {
00390             *nc = - na;
00391             i = ABS(na); i *= 2;
00392             while ( --i >= 0 ) *c++ = *a++;
00393             return(0);
00394         }
00395     }
00396     if ( *a == 1 && a[1] == 1 ) {
00397         if ( na == 1 ) {
00398             *nc = nb;
00399             i = ABS(nb); i *= 2;

```

```

00400         while ( --i >= 0 ) *c++ = *b++;
00401         return(0);
00402     }
00403     else if ( na == -1 ) {
00404         *nc = - nb;
00405         i = ABS(nb); i *= 2;
00406         while ( --i >= 0 ) *c++ = *b++;
00407         return(0);
00408     }
00409 }
00410 if ( na < 0 ) { na = -na; sgn = -sgn; }
00411 if ( nb < 0 ) { nb = -nb; sgn = -sgn; }
00412 if ( !na || !nb ) { *nc = 0; return(0); }
00413 if ( na != 1 || nb != 1 ) {
00414     GETBIDENTITY
00415     UWORD *xd,*xe, *xf,*xg;
00416     WORD dden, dnumr, eden, enumr;
00417     UnPack(a,na,&dden,&dnumr);
00418     UnPack(b,nb,&eden,&enumr);
00419     xd = NumberMalloc("MulRat"); xf = NumberMalloc("MulRat");
00420     for ( i = 0; i < dnumr; i++ ) xd[i] = a[i];
00421     a += na;
00422     for ( i = 0; i < dden; i++ ) xf[i] = a[i];
00423     xe = NumberMalloc("MulRat"); xg = NumberMalloc("MulRat");
00424     for ( i = 0; i < enumr; i++ ) xe[i] = b[i];
00425     b += nb;
00426     for ( i = 0; i < eden; i++ ) xg[i] = b[i];
00427     if ( Simplify(BHEAD xd,&dnumr,xg,&eden) ||
00428         Simplify(BHEAD xe,&enumr,xf,&dden) ||
00429         MulLong(xd,dnumr,xe,enumr,c,nc) ||
00430         MulLong(xf,dden,xg,eden,xd,&dnumr) ) {
00431         MLOCK(ErrorMessageLock);
00432         MesCall("MulRat");
00433         MUNLOCK(ErrorMessageLock);
00434         NumberFree(xd,"MulRat"); NumberFree(xe,"MulRat"); NumberFree(xf,"MulRat");
00435         NumberFree(xg,"MulRat");
00436         SETERROR(-1)
00437     }
00438     Pack(c,nc,xd,dnumr);
00439     NumberFree(xd,"MulRat"); NumberFree(xe,"MulRat"); NumberFree(xf,"MulRat");
00440     NumberFree(xg,"MulRat");
00441 }
00442 else {
00443     UWORD y;
00444     UWORD a0,a1,b0,b1;
00445     RLONG xx;
00446     y = a[0]; b1=b[1];
00447     do { a0 = y % b1; y = b1; } while ( ( b1 = a0 ) != 0 );
00448     if ( y != 1 ) {
00449         a0 = a[0] / y;
00450         b1 = b[1] / y;
00451     }
00452     else {
00453         a0 = a[0];
00454         b1 = b[1];
00455     }
00456     y=b[0]; a1=a[1];
00457     do { b0 = y % a1; y = a1; } while ( ( a1 = b0 ) != 0 );
00458     if ( y != 1 ) {
00459         a1 = a[1] / y;
00460         b0 = b[0] / y;
00461     }
00462     else {
00463         a1 = a[1];
00464         b0 = b[0];
00465     }
00466     xx = ((RLONG)a0)*b0;
00467     if ( xx & AWORDMASK ) {
00468         *nc = 2;
00469         c[0] = (UWORD)xx;
00470         c[1] = (UWORD)(xx >> BITSINWORD);
00471         xx = ((RLONG)a1)*b1;
00472         c[2] = (UWORD)xx;
00473         c[3] = (UWORD)(xx >> BITSINWORD);
00474     }
00475     else {
00476         c[0] = (UWORD)xx;
00477         xx = ((RLONG)a1)*b1;
00478         if ( xx & AWORDMASK ) {
00479             c[1] = 0;
00480             c[2] = (UWORD)xx;
00481             c[3] = (UWORD)(xx >> BITSINWORD);
00482             *nc = 2;
00483         }
00484         else {
00485             c[1] = (UWORD)xx;
00486             *nc = 1;

```

```

00485     }
00486     }
00487 }
00488 if ( sgn < 0 ) *nc = -*nc;
00489 return(0);
00490 }
00491
00492 /*
00493     #] MulRat :
00494     #[ DivRat :      WORD DivRat(a,na,b,nb,c,nc)
00495
00496     Divides the rational a by the rational b.
00497
00498 */
00499
00500 int DivRat(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
00501 {
00502     GETBIDENTITY
00503     WORD i, j;
00504     int ret;
00505     UWORD *xd,*xe,xx;
00506     if ( !nb ) {
00507         MLOCK(ErrorMessageLock);
00508         MesPrint("Rational division by zero");
00509         MUNLOCK(ErrorMessageLock);
00510         return(-1);
00511     }
00512     j = i = (nb >= 0)? nb: -nb;
00513     xd = b; xe = b + i;
00514     do { xx = *xd; *xd++ = *xe; *xe++ = xx; } while ( --j > 0 );
00515     ret = MulRat(BHEAD a,na,b,nb,c,nc);
00516     xd = b; xe = b + i;
00517     do { xx = *xd; *xd++ = *xe; *xe++ = xx; } while ( --i > 0 );
00518     return(ret);
00519 }
00520
00521 /*
00522     #] DivRat :
00523     #[ Simplify :      WORD Simplify(a,na,b,nb)
00524
00525     Determines the greatest common denominator of a and b and
00526     divides both by it. A possible sign is put in a. This is
00527     the simplification of the fraction a/b.
00528
00529 */
00530
00531 int Simplify(PHEAD UWORD *a, WORD *na, UWORD *b, WORD *nb)
00532 {
00533     GETBIDENTITY
00534     UWORD *x1,*x2,*x3;
00535     UWORD *x4;
00536     WORD n1,n2,n3,n4,sgn = 1;
00537     WORD i;
00538     UWORD *Siscrat5, *Siscrat6, *Siscrat7, *Siscrat8;
00539     if ( *na < 0 ) { *na = -*na; sgn = -sgn; }
00540     if ( *nb < 0 ) { *nb = -*nb; sgn = -sgn; }
00541     Siscrat5 = NumberMalloc("Simplify"); Siscrat6 = NumberMalloc("Simplify");
00542     Siscrat7 = NumberMalloc("Simplify"); Siscrat8 = NumberMalloc("Simplify");
00543     x1 = Siscrat8; x2 = Siscrat7;
00544     if ( *nb == 1 ) {
00545         x3 = Siscrat6;
00546         if ( DivLong(a,*na,b,*nb,x1,&n1,x2,&n2) ) goto SimpErr;
00547         if ( !n2 ) {
00548             for ( i = 0; i < n1; i++ ) *a++ = *x1++;
00549             *na = n1;
00550             *b = 1;
00551         }
00552     } else {
00553         UWORD y1, y2, y3;
00554         y2 = *b;
00555         y3 = *x2;
00556         do { y1 = y2 % y3; y2 = y3; } while ( ( y3 = y1 ) != 0 );
00557         if ( ( *x2 = y2 ) != 1 ) {
00558             *b /= y2;
00559             if ( DivLong(a,*na,x2,(WORD)1,x1,&n1,x3,&n3) ) goto SimpErr;
00560             for ( i = 0; i < n1; i++ ) *a++ = *x1++;
00561             *na = n1;
00562         }
00563     }
00564 }
00565 #ifndef NEWTRICK
00566 else if ( *na >= GCDMAX && *nb >= GCDMAX ) {
00567     n1 = i = *na; x3 = a;
00568     NCOPY(x1,x3,i);
00569     x3 = b; n2 = i = *nb;
00570     NCOPY(x2,x3,i);
00571     x4 = Siscrat5;

```

```

00572     x2 = Siscrat6;
00573     x3 = Siscrat7;
00574     if ( GcdLong(BHEAD Siscrat8,n1,Siscrat7,n2,x2,&n3) ) goto SimpErr;
00575     n2 = n3;
00576     if ( *x2 != 1 || n2 != 1 ) {
00577         DivLong(a,*na,x2,n2,x1,&n1,x4,&n4);
00578         *na = i = n1;
00579         NCOPY(a,x1,i);
00580         DivLong(b,*nb,x2,n2,x3,&n3,x4,&n4);
00581         *nb = i = n3;
00582         NCOPY(b,x3,i);
00583     }
00584 }
00585 #endif
00586 else {
00587     x4 = Siscrat5;
00588     n1 = i = *na; x3 = a;
00589     NCOPY(x1,x3,i);
00590     x3 = b; n2 = i = *nb;
00591     NCOPY(x2,x3,i);
00592     x1 = Siscrat8; x2 = Siscrat7; x3 = Siscrat6;
00593     for(;;) {
00594         if ( DivLong(x1,n1,x2,n2,x4,&n4,x3,&n3) ) goto SimpErr;
00595         if ( !n3 ) break;
00596         if ( n2 == 1 ) {
00597             while ( ( *x1 = (*x2) % (*x3) ) != 0 ) { *x2 = *x3; *x3 = *x1; }
00598             *x2 = *x3;
00599             break;
00600         }
00601         if ( DivLong(x2,n2,x3,n3,x4,&n4,x1,&n1) ) goto SimpErr;
00602         if ( !n1 ) { x2 = x3; n2 = n3; x3 = Siscrat7; break; }
00603         if ( n3 == 1 ) {
00604             while ( ( *x2 = (*x3) % (*x1) ) != 0 ) { *x3 = *x1; *x1 = *x2; }
00605             *x2 = *x1;
00606             n2 = 1;
00607             break;
00608         }
00609         if ( DivLong(x3,n3,x1,n1,x4,&n4,x2,&n2) ) goto SimpErr;
00610         if ( !n2 ) { x2 = x1; n2 = n1; x1 = Siscrat7; break; }
00611         if ( n1 == 1 ) {
00612             while ( ( *x3 = (*x1) % (*x2) ) != 0 ) { *x1 = *x2; *x2 = *x3; }
00613             break;
00614         }
00615     }
00616     if ( *x2 != 1 || n2 != 1 ) {
00617         DivLong(a,*na,x2,n2,x1,&n1,x4,&n4);
00618         *na = i = n1;
00619         NCOPY(a,x1,i);
00620         DivLong(b,*nb,x2,n2,x3,&n3,x4,&n4);
00621         *nb = i = n3;
00622         NCOPY(b,x3,i);
00623     }
00624 }
00625 if ( sgn < 0 ) *na = -*na;
00626 NumberFree(Siscrat5,"Simplify"); NumberFree(Siscrat6,"Simplify");
00627 NumberFree(Siscrat7,"Simplify"); NumberFree(Siscrat8,"Simplify");
00628 return(0);
00629 SimpErr:
00630 MLOCK(ErrorMessageLock);
00631 MesCall("Simplify");
00632 MUNLOCK(ErrorMessageLock);
00633 NumberFree(Siscrat5,"Simplify"); NumberFree(Siscrat6,"Simplify");
00634 NumberFree(Siscrat7,"Simplify"); NumberFree(Siscrat8,"Simplify");
00635 SETERROR(-1)
00636 }
00637
00638 /*
00639 #] Simplify :
00640 #[ AccumGCD :      WORD AccumGCD(PHEAD a,na,b,nb)
00641
00642 Routine takes the rational GCD of the fractions in a and b and
00643 replaces a by the GCD of the two.
00644 The rational GCD is defined as the rational that consists of
00645 the GCD of the numerators divided by the GCD of the denominators
00646 */
00647
00648 int AccumGCD(PHEAD UWORD *a, WORD *na, UWORD *b, WORD nb)
00649 {
00650     GETBIDENTITY
00651     WORD nna,nnb,numa,numb,dena,denb,numc,denc;
00652     UWORD *GCDbuffer = NumberMalloc("AccumGCD");
00653     int i;
00654     nna = *na; if ( nna < 0 ) nna = -nna; nna = (nna-1)/2;
00655     nnb = nb; if ( nnb < 0 ) nnb = -nnb; nnb = (nnb-1)/2;
00656     UnPack(a,nna,&dena,&numa);
00657     UnPack(b,nnb,&denb,&numb);
00658     if ( GcdLong(BHEAD a,numa,b,numb,GCDbuffer,&numc) ) goto AccErr;

```

```

00659     numa = numc;
00660     for ( i = 0; i < numa; i++ ) a[i] = GCDbuffer[i];
00661     if ( GcdLong(BHEAD a+nna,dena,b+nnb,denb,GCDbuffer,&denc) ) goto AccErr;
00662     dena = denc;
00663     for ( i = 0; i < dena; i++ ) a[i+nna] = GCDbuffer[i];
00664     Pack(a,&numa,a+nna,dena);
00665     *na = INCLENG(numa);
00666     NumberFree(GCDbuffer,"AccumGCD");
00667     return(0);
00668 AccErr:
00669     MLOCK(ErrorMessageLock);
00670     MesCall("AccumGCD");
00671     MUNLOCK(ErrorMessageLock);
00672     NumberFree(GCDbuffer,"AccumGCD");
00673     SETERROR(-1)
00674 }
00675
00676 /*
00677     #] AccumGCD :
00678     #[ TakeRatRoot:
00679 */
00680
00681 int TakeRatRoot(WORD *a, WORD *n, WORD power)
00682 {
00683     WORD number,denom, nn;
00684     if ( ( power & 1 ) == 0 && *n < 0 ) return(1);
00685     if ( ABS(*n) == 1 && a[0] == 1 && a[1] == 1 ) return(0);
00686     nn = ABS(*n);
00687     UnPack(a,nn,&denom,&number);
00688     if ( TakeLongRoot(a+nn,&denom,power) ) return(1);
00689     if ( TakeLongRoot(a,&number,power) ) return(1);
00690     Pack(a,&number,a+nn,denom);
00691     if ( *n < 0 ) *n = -number;
00692     else          *n = number;
00693     return(0);
00694 }
00695
00696 /*
00697     #] TakeRatRoot:
00698     #] RekenRational :
00699     #[ RekenLong :
00700     #[ AddLong :          WORD AddLong(a,na,b,nb,c,nc)
00701
00702     Long addition. Uses addition and subtraction of positive numbers.
00703
00704 */
00705
00706 int AddLong(WORD *a, WORD na, WORD *b, WORD nb, WORD *c, WORD *nc)
00707 {
00708     WORD sgn, res;
00709     if ( na < 0 ) {
00710         if ( nb < 0 ) {
00711             if ( AddPLon(a,-na,b,-nb,c,nc) ) return(-1);
00712             *nc = -*nc;
00713             return(0);
00714         }
00715         else {
00716             na = -na;
00717             sgn = -1;
00718         }
00719     }
00720     else {
00721         if ( nb < 0 ) {
00722             nb = -nb;
00723             sgn = 1;
00724         }
00725         else { return( AddPLon(a,na,b,nb,c,nc) ); }
00726     }
00727     if ( ( res = BigLong(a,na,b,nb) ) > 0 ) {
00728         SubPLon(a,na,b,nb,c,nc);
00729         if ( sgn < 0 ) *nc = -*nc;
00730     }
00731     else if ( res < 0 ) {
00732         SubPLon(b,nb,a,na,c,nc);
00733         if ( sgn > 0 ) *nc = -*nc;
00734     }
00735     else {
00736         *nc = 0;
00737         *c = 0;
00738     }
00739     return(0);
00740 }
00741
00742 /*
00743     #] AddLong :
00744     #[ AddPLon :          WORD AddPLon(a,na,b,nb,c,nc)
00745

```

```

00746     Adds two long integers a and b and puts the result in c.
00747     The length of a and b are na and nb. The length of c is returned in nc.
00748     c can be a or b.
00749  */
00750
00751 int AddPLon(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
00752 {
00753     UWORD carry = 0, e, nd = 0;
00754     while ( na && nb ) {
00755         e = *a;
00756         *c = e + *b + carry;
00757         if ( carry ) {
00758             if ( e < *c ) carry = 0;
00759         }
00760         else {
00761             if ( e > *c ) carry = 1;
00762         }
00763         a++; b++; c++; nd++; na--; nb--;
00764     }
00765     while ( na ) {
00766         if ( carry ) {
00767             *c = *a++ + carry;
00768             if ( *c++ ) carry = 0;
00769         }
00770         else *c++ = *a++;
00771         nd++; na--;
00772     }
00773     while ( nb ) {
00774         if ( carry ) {
00775             *c = *b++ + carry;
00776             if ( *c++ ) carry = 0;
00777         }
00778         else *c++ = *b++;
00779         nd++; nb--;
00780     }
00781     if ( carry ) {
00782         nd++;
00783         if ( nd > (UWORD)AM.MaxTal ) {
00784             MLOCK(ErrorMessageLock);
00785             MesPrint("Overflow in addition");
00786             MUNLOCK(ErrorMessageLock);
00787             return(-1);
00788         }
00789         *c++ = carry;
00790     }
00791     *nc = nd;
00792     return(0);
00793 }
00794
00795 /*
00796     #] AddPLon :
00797     #[ SubPLon :      void SubPLon(a,na,b,nb,c,nc)
00798
00799     Subtracts b from a. Assumes that a > b. Result in c.
00800     c can be a or b.
00801
00802  */
00803
00804 void SubPLon(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
00805 {
00806     UWORD borrow = 0, e, nd = 0;
00807     while ( nb ) {
00808         e = *a;
00809         if ( borrow ) {
00810             *c = e - *b - borrow;
00811             if ( *c < e ) borrow = 0;
00812         }
00813         else {
00814             *c = e - *b;
00815             if ( *c > e ) borrow = 1;
00816         }
00817         a++; b++; c++; na--; nb--; nd++;
00818     }
00819     while ( na ) {
00820         if ( borrow ) {
00821             if ( *a ) { *c++ = *a++ - 1; borrow = 0; }
00822             else { *c++ = (UWORD)(-1); a++; }
00823         }
00824         else *c++ = *a++;
00825         na--; nd++;
00826     }
00827     while ( nd && !*--c ) { nd--; }
00828     *nc = (WORD)nd;
00829 }
00830
00831 /*
00832     #] SubPLon :

```

```

00833      # [ MulLong :          WORD MulLong(a,na,b,nb,c,nc)
00834
00835      Does a Long multiplication. Assumes that WORD is half the size
00836      of a LONG to work out the scheme! The number of operations is
00837      the canonical na*nb multiplications.
00838      c should not overlap with a or b.
00839
00840      */
00841
00842      int MulLong(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
00843      {
00844          WORD sgn = 1;
00845          UWORD i, *ic, *ia;
00846          RLONG t, bb;
00847          if ( !na || !nb ) { *nc = 0; return(0); }
00848          if ( na < 0 ) { na = -na; sgn = -sgn; }
00849          if ( nb < 0 ) { nb = -nb; sgn = -sgn; }
00850          *nc = i = na + nb;
00851          if ( i > (UWORD)(AM.MaxTal+1) ) goto MulLov;
00852          ic = c;
00853      /*
00854      # [ GMP stuff :
00855      */
00856      #ifdef WITHGMP
00857          if (na > 3 && nb > 3) {
00858              /* mp_limb_t res; */
00859              UWORD *to, *from;
00860              int j;
00861              GETIDENTITY
00862              UWORD *DLscrat9 = NumberMalloc("MulLong"), *DLscratA = NumberMalloc("MulLong"), *DLscratB =
00863              NumberMalloc("MulLong");
00864              if ( na & 1 ) {
00865                  from = a; a = to = DLscrat9; j = na; NCOPY(to, from, j);
00866                  a[na++] = 0;
00867                  ++*nc;
00868              } else
00869              #endif
00870              if ( (LONG)a & (sizeof(mp_limb_t)-1) ) {
00871                  from = a; a = to = DLscrat9; j = na; NCOPY(to, from, j);
00872              }
00873
00874              if ( GMPSPREAD != 1 )
00875              if ( nb & 1 ) {
00876                  from = b; b = to = DLscratA; j = nb; NCOPY(to, from, j);
00877                  b[nb++] = 0;
00878                  ++*nc;
00879              } else
00880              #endif
00881              if ( (LONG)b & (sizeof(mp_limb_t)-1) ) {
00882                  from = b; b = to = DLscratA; j = nb; NCOPY(to, from, j);
00883              }
00884
00885              if ( ( *nc > (WORD)i ) || ( (LONG)c & (LONG)(sizeof(mp_limb_t)-1) ) ) {
00886                  ic = DLscratB;
00887              }
00888              if ( na < nb ) {
00889                  /* res = */
00890                  mpn_mul((mp_ptr)ic, (mp_srcptr)b, nb/GMPSPREAD, (mp_srcptr)a, na/GMPSPREAD);
00891              } else {
00892                  /* res = */
00893                  mpn_mul((mp_ptr)ic, (mp_srcptr)a, na/GMPSPREAD, (mp_srcptr)b, nb/GMPSPREAD);
00894              }
00895              while ( ic[i-1] == 0 ) i--;
00896              *nc = i;
00897          /*
00898              if ( res == 0 ) *nc -= GMPSPREAD;
00899              else if ( res <= WORDMASK ) --*nc;
00900          */
00901              if ( ic != c ) {
00902                  j = *nc; NCOPY(c, ic, j);
00903              }
00904              if ( sgn < 0 ) *nc = -( *nc );
00905              NumberFree(DLscrat9, "MulLong"); NumberFree(DLscratA, "MulLong");
00906              NumberFree(DLscratB, "MulLong");
00907              return(0);
00908          #endif
00909      /*
00910      # ] GMP stuff :
00911      */
00912      do { *ic++ = 0; } while ( --i > 0 );
00913      do {
00914          ia = a;
00915          ic = c++;
00916          t = 0;
00917          i = na;

```



```

00918         bb = (RLONG) (*b++);
00919         do {
00920             t = (*ia++) * bb + t + *ic;
00921             *ic++ = (WORD)t;
00922             t >= BITSINWORD;          /* should actually be a swap */
00923         } while ( --i > 0 );
00924         if ( t ) *ic = (UWORD)t;
00925     } while ( --nb > 0 );
00926     if ( !*ic ) (*nc)--;
00927     if ( *nc > AM.MaxTal ) goto MulLov;
00928     if ( sgn < 0 ) *nc = -(*nc);
00929     return(0);
00930 MulLov:
00931     MLOCK(ErrorMessageLock);
00932     MesPrint("Overflow in Multiplication");
00933     MUNLOCK(ErrorMessageLock);
00934     return(-1);
00935 }
00936
00937 /*
00938     #] MulLong :
00939     #[ BigLong :          WORD BigLong(a,na,b,nb)
00940
00941     Returns > 0 if a > b, < 0 if b > a and 0 if a == b
00942
00943 */
00944
00945 int BigLong(UWORD *a, WORD na, UWORD *b, WORD nb)
00946 {
00947     a += na;
00948     b += nb;
00949     while ( na && !--a ) na--;
00950     while ( nb && !--b ) nb--;
00951     if ( nb < na ) return(1);
00952     if ( nb > na ) return(-1);
00953     while ( --na >= 0 ) {
00954         if ( *a > *b ) return(1);
00955         else if ( *b > *a ) return(-1);
00956         a--; b--;
00957     }
00958     return(0);
00959 }
00960
00961 /*
00962     #] BigLong :
00963     #[ DivLong :          WORD DivLong(a,na,b,nb,c,nc,d,nd)
00964
00965     This is the long division which knows a couple of exceptions.
00966     It uses therefore a recursive call for the renormalization.
00967     The quotient comes in c and the remainder in d.
00968     d may be overlapping with b. It may also be identical to a.
00969     c should not overlap with a, but it can overlap with b.
00970
00971 */
00972
00973 int DivLong(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c,
00974             WORD *nc, UWORD *d, WORD *nd)
00975 {
00976     WORD sgnA = 1, sgnb = 1, ne, nf, ng, nh;
00977     WORD i, ni;
00978     UWORD *w1, *w2;
00979     RLONG t, v;
00980     UWORD *e, *f, *ff, *g, norm, estim;
00981 #ifdef WITHGMP
00982     UWORD *DLscrat9, *DLscratA, *DLscratB, *DLscratC;
00983 #endif
00984     RLONG esthelp;
00985     if ( !nb ) {
00986         MLOCK(ErrorMessageLock);
00987         MesPrint("Division by zero");
00988         MUNLOCK(ErrorMessageLock);
00989         return(-1);
00990     }
00991     if ( !na ) { *nc = *nd = 0; return(0); }
00992     if ( na < 0 ) { sgnA = -sgnA; na = -na; }
00993     if ( nb < 0 ) { sgnb = -sgnb; nb = -nb; }
00994     if ( na < nb ) {
00995         for ( i = 0; i < na; i++ ) *d++ = *a++;
00996         *nd = na;
00997         *nc = 0;
00998     }
00999     else if ( nb == na && ( i = BigLong(b,nb,a,na) ) >= 0 ) {
01000         if ( i > 0 ) {
01001             for ( i = 0; i < na; i++ ) *d++ = *a++;
01002             *nd = na;
01003             *nc = 0;
01004         }

```

```

01005         else {
01006             *c = 1;
01007             *nc = 1;
01008             *nd = 0;
01009         }
01010     }
01011     else if ( nb == 1 ) {
01012         if ( *b == 1 ) {
01013             for ( i = 0; i < na; i++ ) *c++ = *a++;
01014             *nc = na;
01015             *nd = 0;
01016         }
01017         else {
01018             w1 = a+na;
01019             *nc = ni = na;
01020             *nd = 1;
01021             w2 = c+ni;
01022             v = (RLONG) (*b);
01023             t = (RLONG) (*--w1);
01024             while ( --ni >= 0 ) {
01025                 *--w2 = t / v;
01026                 t -= v * (*w2);
01027                 if ( ni ) {
01028                     t <= BITSINWORD;
01029                     t += *--w1;
01030                 }
01031             }
01032             if ( ( *d = (UWORD)t ) == 0 ) *nd = 0;
01033             if ( !*(c+na-1) ) (*nc)--;
01034         }
01035     }
01036     else {
01037         GETIDENTITY
01038     /*
01039         #! GMP stuff :
01040
01041         We start with copying a and b.
01042         Then we make space for c and d.
01043         Next we call mpn_tdiv_qr
01044         We adjust sizes and copy to c and d if needed.
01045         Finally the signs are settled.
01046     */
01047     #ifdef WITHGMP
01048         if ( na > 4 && nb > 3 ) {
01049             UWORD *ic, *id, *to, *from;
01050             int j = na - nb;
01051             DLscrat9 = NumberMalloc("DivLong"); DLscratA = NumberMalloc("DivLong");
01052             DLscratB = NumberMalloc("DivLong"); DLscratC = NumberMalloc("DivLong");
01053
01054             #if ( GMPSPREAD != 1 )
01055                 if ( na & 1 ) {
01056                     from = a; a = to = DLscrat9; i = na; NCOPY(to, from, i);
01057                     a[na++] = 0;
01058                 } else
01059             #endif
01060                 if ( (LONG)a & (sizeof(mp_limb_t)-1) ) {
01061                     from = a; a = to = DLscrat9; i = na; NCOPY(to, from, i);
01062                 }
01063
01064             #if ( GMPSPREAD != 1 )
01065                 if ( nb & 1 ) {
01066                     from = b; b = to = DLscratA; i = nb; NCOPY(to, from, i);
01067                     b[nb++] = 0;
01068                 } else
01069             #endif
01070                 if ( ( (LONG)b & (sizeof(mp_limb_t)-1) ) != 0 ) {
01071                     from = b; b = to = DLscratA; i = nb; NCOPY(to, from, i);
01072                 }
01073             #if ( GMPSPREAD != 1 )
01074             /*
01075                 Not recognizing this case was a nasty and extremely rare bug.
01076                 In the case that c == a and na is odd and nc == na and b
01077                 still needs to be used, the least significant UWORD of b got
01078                 overwritten by zero. (22-mar-2023)
01079             */
01080             ic = DLscratB; id = DLscratC;
01081             #else
01082                 if ( ( (LONG)c & (sizeof(mp_limb_t)-1) ) != 0 ) ic = DLscratB;
01083                 else ic = c;
01084
01085                 if ( ( (LONG)d & (sizeof(mp_limb_t)-1) ) != 0 ) id = DLscratC;
01086                 else id = d;
01087             #endif
01088             mpn_tdiv_qr((mp_limb_t *)ic, (mp_limb_t *)id, (mp_size_t)0,
01089                 (const mp_limb_t *)a, (mp_size_t)(na/GMPSPREAD),
01090                 (const mp_limb_t *)b, (mp_size_t)(nb/GMPSPREAD));
01091             while ( j >= 0 && ic[j] == 0 ) j--;

```

```

01092         j++; *nc = j;
01093         if ( c != ic ) { NCOPY(c,ic,j); }
01094         j = nb-1;
01095         while ( j >= 0 && id[j] == 0 ) j--;
01096         j++; *nd = j;
01097         if ( d != id ) { NCOPY(d,id,j); }
01098         if ( sgna < 0 ) { *nc = -(*nc); *nd = -(*nd); }
01099         if ( sgnb < 0 ) { *nc = -(*nc); }
01100         NumberFree(DLscrat9,"DivLong"); NumberFree(DLscratA,"DivLong");
01101         NumberFree(DLscratB,"DivLong"); NumberFree(DLscratC,"DivLong");
01102         return(0);
01103     }
01104 #endif
01105 /*
01106 #] GMP stuff :
01107 */
01108 /* Start with normalization operation */
01109
01110 e = NumberMalloc("DivLong"); f = NumberMalloc("DivLong"); g = NumberMalloc("DivLong");
01111 if ( b[nb-1] == (FULLMAX-1) ) norm = 1;
01112 else {
01113     norm = (UWORD) (((ULONG)FULLMAX) / (ULONG) ((b[nb-1]+1L)));
01114 }
01115 f[na] = 0;
01116 if ( MulLong(b,nb,&norm,1,e,&ne) ||
01117     MulLong(a,na,&norm,1,f,&nf) ) {
01118     NumberFree(e,"DivLong"); NumberFree(f,"DivLong"); NumberFree(g,"DivLong");
01119     return(-1);
01120 }
01121 if ( BigLong(f+nf-ne,ne,e,ne) >= 0 ) {
01122     SubPLon(f+nf-ne,ne,e,ne,f+nf-ne,&nh);
01123     w1 = c + (nf-ne);
01124     *nc = nf-ne+1;
01125 }
01126 else {
01127     nh = ne;
01128     *nc = nf-ne;
01129     w1 = 0;
01130 }
01131 w2 = c; i = *nc; do { *w2++ = 0; } while ( --i > 0 );
01132 nf = na;
01133 ni = nf-ne;
01134 esthelp = (RLONG) (e[ne-1]) + 1L;
01135 while ( nf >= ne ) {
01136     if ( (WORD)esthelp == 0 ) {
01137         estim = (WORD) ((( (RLONG) (f[nf])) «BITSINWORD) + f[nf-1] »BITSINWORD);
01138     }
01139     else {
01140         estim = (WORD) ((( (RLONG) (f[nf])) «BITSINWORD) + f[nf-1] ) / esthelp);
01141     }
01142     /* This estimate may be up to two too small */
01143     if ( estim ) {
01144         MulLong(e,ne,&estim,1,g,&ng);
01145         nh = ne + 1; if ( !f[ni+ne] ) nh--;
01146         SubPLon(f+ni,nh,g,ng,f+ni,&nh);
01147     }
01148     else {
01149         w2 = f+ni+ne; nh = ne+1;
01150         while ( ( nh > 0 ) && !*w2 ) { nh--; w2--; }
01151     }
01152     if ( BigLong(f+ni,nh,e,ne) >= 0 ) {
01153         estim++;
01154         SubPLon(f+ni,nh,e,ne,f+ni,&nh);
01155         if ( BigLong(f+ni,nh,e,ne) >= 0 ) {
01156             estim++;
01157             SubPLon(f+ni,nh,e,ne,f+ni,&nh);
01158             if ( BigLong(f+ni,nh,e,ne) >= 0 ) {
01159                 MLOCK(ErrorMessageLock);
01160                 MesPrint("Problems in DivLong");
01161                 AO.OutSkip = 3;
01162                 FiniLine();
01163                 i = na;
01164                 while ( --i >= 0 ) { TalToLine((UWORD) (*a++)); TokenToLine((UBYTE *) " "); }
01165                 FiniLine();
01166                 i = nb;
01167                 while ( --i >= 0 ) { TalToLine((UWORD) (*b++)); TokenToLine((UBYTE *) " "); }
01168                 AO.OutSkip = 0;
01169                 FiniLine();
01170                 MUNLOCK(ErrorMessageLock);
01171                 NumberFree(e,"DivLong"); NumberFree(f,"DivLong"); NumberFree(g,"DivLong");
01172                 return(-1);
01173             }
01174         }
01175     }
01176     c[ni] = estim;
01177     nf--;
01178     ni--;

```

```

01179     }
01180     if ( w1 ) *w1 = 1;
01181
01182     /* Finish with the renormalization operation */
01183
01184     if ( nh > 0 ) {
01185         if ( norm == 1 ) {
01186             *nd = i = nh; ff = f;
01187             NCOPY(d,ff,i);
01188         }
01189         else {
01190             w1 = f+nh;
01191             *nd = ni = nh;
01192             w2 = d+ni;
01193             v = norm;
01194             t = (RLONG) (*--w1);
01195             while ( --ni >= 0 ) {
01196                 *--w2 = t / v;
01197                 t -= v * (*w2);
01198                 if ( ni ) {
01199                     t <<= BITSINWORD;
01200                     t += *--w1;
01201                 }
01202             }
01203             if ( t ) {
01204                 MLOCK(ErrorMessageLock);
01205                 MesPrint("Error in DivLong");
01206                 MUNLOCK(ErrorMessageLock);
01207                 NumberFree(e, "DivLong"); NumberFree(f, "DivLong"); NumberFree(g, "DivLong");
01208                 return(-1);
01209             }
01210             if ( !(d+nh-1) ) (*nd)--;
01211         }
01212     }
01213     else { *nd = 0; }
01214     NumberFree(e, "DivLong"); NumberFree(f, "DivLong"); NumberFree(g, "DivLong");
01215 }
01216 if ( sgna < 0 ) { *nc = -(*nc); *nd = -(*nd); }
01217 if ( sgnb < 0 ) { *nc = -(*nc); }
01218 return(0);
01219 }
01220
01221 /*
01222     #] DivLong :
01223     #[ RaisPow :      WORD RaisPow(a,na,b)
01224
01225     Raises a to the power b. a is a Long integer and b >= 0.
01226     The method that is used works with a bitdecomposition of b.
01227 */
01228
01229 int RaisPow(PHEAD UWORD *a, WORD *na, UWORD b)
01230 {
01231     GETBIDENTITY
01232     WORD i, nu;
01233     UWORD *it, *iu, c;
01234     UWORD *is, *iss;
01235     WORD ns, nt, nmod;
01236     nmod = ABS(AN.ncmod);
01237     if ( !*na || ( ( *na == 1 ) && ( *a == 1 ) ) ) return(0);
01238     if ( !b ) { *na=1; *a=1; return(0); }
01239     is = NumberMalloc("RaisPow");
01240     it = NumberMalloc("RaisPow");
01241     for ( i = 0; i < ABS(*na); i++ ) is[i] = a[i];
01242     ns = *na;
01243     c = b;
01244     for ( i = 0; i < BITSINWORD; i++ ) {
01245         if ( !c ) break;
01246         c /= 2;
01247     }
01248     i--;
01249     c = 1 << i;
01250     while ( --i >= 0 ) {
01251         c /= 2;
01252         if (MulLong(is,ns,is,ns,it,&nt)) goto RaisOvl;
01253         if ( b & c ) {
01254             if ( MulLong(it,nt,a,*na,is,&ns) ) goto RaisOvl;
01255         }
01256         else {
01257             iu = is; is = it; it = iu;
01258             nu = ns; ns = nt; nt = nu;
01259         }
01260         if ( nmod != 0 ) {
01261             if ( DivLong(is,ns,(UWORD *)AC.cmod,nmod,it,&nt,is,&ns) ) goto RaisOvl;
01262         }
01263     }
01264     if ( ( nmod != 0 ) && ( ( AC.modmode & POSNEG ) != 0 ) ) {
01265         NormalModulus(is,&ns);

```

```

01266     }
01267     if ( ( *na = i = ns ) != 0 ) { iss = is; i=ABS(i); NCOPY(a,iss,i); }
01268     NumberFree(is,"RaisPow"); NumberFree(it,"RaisPow");
01269     return(0);
01270 RaisOvl:
01271     MLOCK(ErrorMessageLock);
01272     MesCall("RaisPow");
01273     MUNLOCK(ErrorMessageLock);
01274     NumberFree(is,"RaisPow"); NumberFree(it,"RaisPow");
01275     SETERROR(-1)
01276 }
01277
01278 /*
01279     #] RaisPow :
01280     #[ RaisPowCached :
01281 */
01282
01297 void RaisPowCached (PHEAD WORD x, WORD n, UWORD **c, WORD *nc) {
01298
01299     int i,j;
01300     WORD new_small_power_maxx, new_small_power_maxn, ID;
01301     WORD *new_small_power_n;
01302     UWORD **new_small_power;
01303
01304     /* check whether to extend the array */
01305     if (x>=AT.small_power_maxx || n>=AT.small_power_maxn) {
01306
01307         new_small_power_maxx = AT.small_power_maxx;
01308         if (x>=AT.small_power_maxx)
01309             new_small_power_maxx = MaX(2*AT.small_power_maxx, x+1);
01310
01311         new_small_power_maxn = AT.small_power_maxn;
01312         if (n>=AT.small_power_maxn)
01313             new_small_power_maxn = MaX(2*AT.small_power_maxn, n+1);
01314
01315         new_small_power_n = (WORD*)
01316         Malloc1(new_small_power_maxx*new_small_power_maxn*sizeof(WORD),"RaisPowCached");
01317         new_small_power = (UWORD **) Malloc1(new_small_power_maxx*new_small_power_maxn*sizeof(UWORD
01318         *),"RaisPowCached");
01319
01320         for (i=0; i<new_small_power_maxx * new_small_power_maxn; i++) {
01321             new_small_power_n[i] = 0;
01322             new_small_power [i] = NULL;
01323         }
01324
01325         for (i=0; i<AT.small_power_maxx; i++)
01326             for (j=0; j<AT.small_power_maxn; j++) {
01327                 new_small_power_n[i*new_small_power_maxn+j] =
01328                 AT.small_power_n[i*AT.small_power_maxn+j];
01329                 new_small_power [i*new_small_power_maxn+j] = AT.small_power
01330                 [i*AT.small_power_maxn+j];
01331             }
01332
01333         if (AT.small_power_n != NULL) {
01334             M_free(AT.small_power_n,"RaisPowCached");
01335             M_free(AT.small_power,"RaisPowCached");
01336         }
01337
01338         AT.small_power_maxx = new_small_power_maxx;
01339         AT.small_power_maxn = new_small_power_maxn;
01340         AT.small_power_n    = new_small_power_n;
01341         AT.small_power      = new_small_power;
01342     }
01343
01344     /* check whether the results is already calculated */
01345     ID = x * AT.small_power_maxn + n;
01346
01347     if (AT.small_power[ID] == NULL) {
01348         #ifdef OLDRAISPOWCACHED
01349             AT.small_power[ID] = NumberMalloc("RaisPowCached");
01350             AT.small_power_n[ID] = 1;
01351             AT.small_power[ID][0] = x;
01352             RaisPow(BHEAD AT.small_power[ID], &AT.small_power_n[ID], n);
01353         #else
01354             UWORD *c = NumberMalloc("RaisPowCached");
01355             WORD i, k = 1;
01356             c[0] = x;
01357             RaisPow(BHEAD c, &k, n);
01358         /*
01359             And now get the proper amount.
01360         */
01361         if ( AT.InNumMem < k ) { /* We should start a new buffer */
01362             AT.InNumMem = 5*AM.MaxTal;
01363             AT.NumMem = (UWORD *) Malloc1(AT.InNumMem*sizeof(UWORD), "RaisPowCached");
01364         /*
01365             MesPrint(" Got an extra %l UWORDS in RaisPowCached",AT.InNumMem);
01366         */

```

```

01363     }
01364     for ( i = 0; i < k; i++ ) AT.NumMem[i] = c[i];
01365     AT.small_power[ID] = AT.NumMem;
01366     AT.small_power_n[ID] = k;
01367     AT.NumMem += k;
01368     AT.InNumMem -= k;
01369     NumberFree(c, "RaisPowCached");
01370 #endif
01371 }
01372
01373 /* return the result */
01374 *c = AT.small_power[ID];
01375 *nc = AT.small_power_n[ID];
01376 }
01377
01378 /*
01379     #] RaisPowCached :
01380     #[ RaisPowMod :
01381
01382     Computes the power x^n mod m
01383 */
01384 WORD RaisPowMod (WORD x, WORD n, WORD m) {
01385     LONG y=1, z=x;
01386     while (n) {
01387         if (n&1) { y*=z; y%=m; }
01388         z*=z; z%=m;
01389         n /= 2;
01390     }
01391     return (WORD)y;
01392 }
01393
01394 /*
01395     #] RaisPowMod :
01396     #[ NormalModulus : int NormalModulus(UWORD *a,WORD *na)
01397 */
01404 int NormalModulus(UWORD *a,WORD *na)
01405 {
01406     WORD n;
01407     if ( AC.halfmod == 0 ) {
01408         LOCK(AC.halfmodlock);
01409         if ( AC.halfmod == 0 ) {
01410             UWORD two[1],remain[1];
01411             WORD dummy;
01412             two[0] = 2;
01413             AC.halfmod = (UWORD *)Malloc1((ABS(AC.ncmod))*sizeof(UWORD),"halfmod");
01414             DivLong((UWORD *)AC.cmod,(ABS(AC.ncmod)),two,1
01415                 ,(UWORD *)AC.halfmod,&(AC.nhalfmod),remain,&dummy);
01416         }
01417         UNLOCK(AC.halfmodlock);
01418     }
01419     n = ABS(*na);
01420     if ( BigLong(a,n,AC.halfmod,AC.nhalfmod) > 0 ) {
01421         SubPLon((UWORD *)AC.cmod,(ABS(AC.ncmod)),a,n,a,&n);
01422         if ( *na > 0 ) { *na = -n; }
01423         else { *na = n; }
01424         return(1);
01425     }
01426     return(0);
01427 }
01428
01429 /*
01430     #] NormalModulus :
01431     #[ MakeInverses :
01432 */
01441 int MakeInverses(void)
01442 {
01443     WORD n = AC.cmod[0], i, inv2;
01444     if ( AC.ncmod != 1 ) return(1);
01445     if ( AC.modinverses == 0 ) {
01446         LOCK(AC.halfmodlock);
01447         if ( AC.modinverses == 0 ) {
01448             AC.modinverses = (UWORD *)Malloc1(n*sizeof(UWORD),"modinverses");
01449             AC.modinverses[0] = 0;
01450             AC.modinverses[1] = 1;
01451             for ( i = 2; i < n; i++ ) {
01452                 if ( GetModInverses(i,n,
01453                     (WORD *)(&(AC.modinverses[i])),&inv2) ) {
01454                     SETERROR(-1)
01455                 }
01456             }
01457         }
01458         UNLOCK(AC.halfmodlock);
01459     }
01460     return(0);
01461 }
01462
01463 /*

```

```

01464         #] MakeInverses :
01465         #[ GetModInverses :
01466     */
01477 int GetModInverses(WORD m1, WORD m2, WORD *im1, WORD *im2)
01478 {
01479     WORD a1, a2, a3;
01480     WORD b1, b2, b3;
01481     WORD x = m1, y, c, d = m2;
01482     if ( x < 1 || d <= 1 ) goto somethingwrong;
01483     a1 = 0; a2 = 1;
01484     b1 = 1; b2 = 0;
01485     for(;;) {
01486         c = d/x; y = d%x; /* a good compiler makes this faster than y=d-c*x */
01487         if ( y == 0 ) break;
01488         a3 = a1-c*a2; a1 = a2; a2 = a3;
01489         b3 = b1-c*b2; b1 = b2; b2 = b3;
01490         d = x; x = y;
01491     }
01492     if ( x != 1 ) goto somethingwrong;
01493     if ( a2 < 0 ) a2 += m2;
01494     if ( b2 < 0 ) b2 += m1;
01495     if (im1!=NULL) *im1 = a2;
01496     if (im2!=NULL) *im2 = b2;
01497     return(0);
01498 somethingwrong:
01499     MLOCK(ErrorMessageLock);
01500     MesPrint("Error trying to determine inverses in GetModInverses");
01501     MUNLOCK(ErrorMessageLock);
01502     return(-1);
01503 }
01504 /*
01505     #] GetModInverses :
01506     #[ GetLongModInverses :
01507 */
01508
01509 int GetLongModInverses(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *ia, WORD *nia, UWORD *ib,
    WORD *nib) {
01510
01511     UWORD *s, *t, *sa, *sb, *ta, *tb, *x, *y, *swap1;
01512     WORD ns, nt, nsa, nsb, nta, ntb, nx, ny, swap2;
01513
01514     s = NumberMalloc("GetLongModInverses");
01515     ns = na;
01516     WCOPY(s, a, ABS(ns));
01517
01518     t = NumberMalloc("GetLongModInverses");
01519     nt = nb;
01520     WCOPY(t, b, ABS(nt));
01521
01522     sa = NumberMalloc("GetLongModInverses");
01523     nsa = 1;
01524     sa[0] = 1;
01525
01526     sb = NumberMalloc("GetLongModInverses");
01527     nsb = 0;
01528
01529     ta = NumberMalloc("GetLongModInverses");
01530     nta = 0;
01531
01532     tb = NumberMalloc("GetLongModInverses");
01533     ntb = 1;
01534     tb[0] = 1;
01535
01536     x = NumberMalloc("GetLongModInverses");
01537     y = NumberMalloc("GetLongModInverses");
01538
01539     while (nt != 0) {
01540         DivLong(s, ns, t, nt, x, &nx, y, &ny);
01541         swap1=s; s=y; y=swap1;
01542         ns=ny;
01543         MulLong(x, nx, ta, nta, y, &ny);
01544         AddLong(sa, nsa, y, -ny, sa, &nsa);
01545         MulLong(x, nx, tb, ntb, y, &ny);
01546         AddLong(sb, nsb, y, -ny, sb, &nsb);
01547
01548         swap1=s; s=t; t=swap1;
01549         swap2=ns; ns=nt; nt=swap2;
01550         swap1=sa; sa=ta; ta=swap1;
01551         swap2=nsa; nsa=nta; nta=swap2;
01552         swap1=sb; sb=tb; tb=swap1;
01553         swap2=nsb; nsb=ntb; ntb=swap2;
01554     }
01555
01556     if (ia!=NULL) {
01557         *nia = nsa*ns;
01558         WCOPY(ia, sa, ABS(*nia));
01559     }

```

```

01560
01561     if (ib!=NULL) {
01562         *nib = nsb*ns;
01563         WCOPY(ib,sb,ABS(*nib));
01564     }
01565
01566     NumberFree(s,"GetLongModInverses");
01567     NumberFree(t,"GetLongModInverses");
01568     NumberFree(sa,"GetLongModInverses");
01569     NumberFree(sb,"GetLongModInverses");
01570     NumberFree(ta,"GetLongModInverses");
01571     NumberFree(tb,"GetLongModInverses");
01572     NumberFree(x,"GetLongModInverses");
01573     NumberFree(y,"GetLongModInverses");
01574
01575     return 0;
01576 }
01577
01578 /*
01579     #] GetLongModInverses :
01580     #[ Product :          WORD Product(a,na,b)
01581
01582     Multiplies the Long number in a with the WORD b.
01583
01584 */
01585
01586 int Product(UWORD *a, WORD *na, WORD b)
01587 {
01588     WORD i, sgn = 1;
01589     RLONG t, u;
01590     if ( *na < 0 ) { *na = -(*na); sgn = -sgn; }
01591     if ( b < 0 ) { b = -b; sgn = -sgn; }
01592     t = 0;
01593     u = (RLONG)b;
01594     for ( i = 0; i < *na; i++ ) {
01595         t += *a * u;
01596         *a++ = (UWORD)t;
01597         t >>= BITSINWORD;
01598     }
01599     if ( t > 0 ) {
01600         if ( ++(*na) > AM.MaxTal ) {
01601             MLOCK(ErrorMessageLock);
01602             MesPrint("Overflow in Product");
01603             MUNLOCK(ErrorMessageLock);
01604             return(-1);
01605         }
01606         *a = (UWORD)t;
01607     }
01608     if ( sgn < 0 ) *na = -(*na);
01609     return(0);
01610 }
01611
01612 /*
01613     #] Product :
01614     #[ Quotient :          UWORD Quotient(a,na,b)
01615
01616     Routine divides the long number a by b with the assumption that
01617     there is no remainder (like while computing binomials).
01618
01619 */
01620
01621 UWORD Quotient(UWORD *a, WORD *na, WORD b)
01622 {
01623     RLONG v, t;
01624     WORD i, j, sgn = 1;
01625     if ( ( i = *na ) < 0 ) { sgn = -1; i = -i; }
01626     if ( b < 0 ) { b = -b; sgn = -sgn; }
01627     if ( i == 1 ) {
01628         if ( ( *a /= (UWORD)b ) == 0 ) *na = 0;
01629         if ( sgn < 0 ) *na = -*na;
01630         return(0);
01631     }
01632     a += i;
01633     j = i;
01634     v = (RLONG)b;
01635     t = (RLONG)(--a);
01636     while ( --i >= 0 ) {
01637         *a = t / v;
01638         t -= v * (*a);
01639         if ( i ) {
01640             t <= BITSINWORD;
01641             t += --a;
01642         }
01643     }
01644     a += j - 1;
01645     if ( !*a ) j--;
01646     if ( sgn < 0 ) j = -j;

```



```

01647     *na = j;
01648     return(0);
01649 }
01650
01651 /*
01652     #] Quotient :
01653     #[ Remain10 :      WORD Remain10(a,na)
01654
01655     Routine divides a by 10 and gives the remainder as return value.
01656     The value of a will be the quotient! a must be positive.
01657
01658 */
01659
01660 WORD Remain10(UWORD *a, WORD *na)
01661 {
01662     WORD i;
01663     RLONG t, u;
01664     UWORD *b;
01665     i = *na;
01666     t = 0;
01667     b = a + i - 1;
01668     while ( --i >= 0 ) {
01669         t += *b;
01670         *b-- = u = t / 10;
01671         t -= u * 10;
01672         if ( i > 0 ) t <= BITSINWORD;
01673     }
01674     if ( ( *na > 0 ) && !a[*na-1] ) (*na)--;
01675     return((WORD)t);
01676 }
01677
01678 /*
01679     #] Remain10 :
01680     #[ Remain4 :      WORD Remain4(a,na)
01681
01682     Routine divides a by 10000 and gives the remainder as return value.
01683     The value of a will be the quotient! a must be positive.
01684
01685 */
01686
01687 WORD Remain4(UWORD *a, WORD *na)
01688 {
01689     WORD i;
01690     RLONG t, u;
01691     UWORD *b;
01692     i = *na;
01693     t = 0;
01694     b = a + i - 1;
01695     while ( --i >= 0 ) {
01696         t += *b;
01697         *b-- = u = t / 10000;
01698         t -= u * 10000;
01699         if ( i > 0 ) t <= BITSINWORD;
01700     }
01701     if ( ( *na > 0 ) && !a[*na-1] ) (*na)--;
01702     return((WORD)t);
01703 }
01704
01705 /*
01706     #] Remain4 :
01707     #[ PrtLong :      void PrtLong(a,na,s)
01708
01709     Puts the long number a in string s.
01710
01711 */
01712
01713 void PrtLong(UWORD *a, WORD na, UBYTE *s)
01714 {
01715     GETIDENTITY
01716     WORD q, i;
01717     UBYTE *sa, *sb;
01718     UBYTE c;
01719     UWORD *bb, *b;
01720
01721     if ( na < 0 ) {
01722         *s++ = '-';
01723         na = -na;
01724     }
01725
01726     b = NumberMalloc("PrtLong");
01727     bb = b;
01728     i = na; while ( --i >= 0 ) *bb++ = *a++;
01729     a = b;
01730     if ( na > 2 ) {
01731         sa = s;
01732         do {
01733             q = Remain4(a,&na);

```

```

01734         *sa++ = (UBYTE) ('0' + (q%10));
01735         q /= 10;
01736         *sa++ = (UBYTE) ('0' + (q%10));
01737         q /= 10;
01738         *sa++ = (UBYTE) ('0' + (q%10));
01739         q /= 10;
01740         *sa++ = (UBYTE) ('0' + (q%10));
01741     } while ( na );
01742     while ( sa[-1] == '0' ) sa--;
01743     sb = s;
01744     s = sa;
01745     sa--;
01746     while ( sa > sb ) { c = *sa; *sa = *sb; *sb = c; sa--; sb++; }
01747 }
01748 else if ( na ) {
01749     sa = s;
01750     do {
01751         q = Remain10(a,&na);
01752         *sa++ = (UBYTE) ('0' + q);
01753     } while ( na );
01754     sb = s;
01755     s = sa;
01756     sa--;
01757     while ( sa > sb ) { c = *sa; *sa = *sb; *sb = c; sa--; sb++; }
01758 }
01759 else *s++ = '0';
01760 *s = '\0';
01761 NumberFree(b,"PrtLong");
01762 }
01763
01764 /*
01765     #] PrtLong :
01766     #[ GetLong :          WORD GetLong(s,a,na)
01767
01768     Reads a long number from a string.
01769     The string is zero terminated and contains only digits!
01770
01771     New algorithm: try to read 4 digits together before the result
01772     is accumulated.
01773 */
01774
01775 int GetLong(UBYTE *s, UWORD *a, WORD *na)
01776 {
01777     /*
01778         UWORD digit;
01779         *a = 0;
01780         *na = 0;
01781         while ( FG.cTable[*s] == 1 ) {
01782             digit = *s++ - '0';
01783             if ( *na && Product(a,na,(WORD)10) ) return(-1);
01784             if ( digit && AddLong(a,*na,&digit,(WORD)1,a,na) ) return(-1);
01785         }
01786         return(0);
01787     */
01788     UWORD digit, x = 0, y = 0;
01789     *a = 0;
01790     *na = 0;
01791     while ( FG.cTable[*s] == 1 ) {
01792         x = *s++ - '0';
01793         if ( FG.cTable[*s] != 1 ) { y = 10; break; }
01794         x = 10*x + *s++ - '0';
01795         if ( FG.cTable[*s] != 1 ) { y = 100; break; }
01796         x = 10*x + *s++ - '0';
01797         if ( FG.cTable[*s] != 1 ) { y = 1000; break; }
01798         x = 10*x + *s++ - '0';
01799         if ( *na && Product(a,na,(WORD)10000) ) return(-1);
01800         if ( ( digit = x ) != 0 && AddLong(a,*na,&digit,(WORD)1,a,na) )
01801             return(-1);
01802         y = 0;
01803     }
01804     if ( y ) {
01805         if ( *na && Product(a,na,(WORD)y) ) return(-1);
01806         if ( ( digit = x ) != 0 && AddLong(a,*na,&digit,(WORD)1,a,na) )
01807             return(-1);
01808     }
01809     return(0);
01810 }
01811
01812 /*
01813     #] GetLong :
01814     #[ GCD :          WORD GCD(a,na,b,nb,c,nc)
01815
01816     Algorithm to compute the GCD of two long numbers.
01817     See Knuth, sec 4.5.2 algorithm L.
01818
01819     We assume that both numbers are positive
01820

```

```

01821     NOTE!!!!. NumberMalloc gets called and it may not be freed
01822 */
01823
01824 #ifdef EXTRAGCD
01825
01826 #define Convert(ia,aa,naa) \
01827     if ( (LONG)ia < 0 ) { \
01828         ia = (ULONG)(~(LONG)ia); \
01829         aa[0] = ia; \
01830         if ( ( aa[1] = ia » BITSINWORD ) != 0 ) naa = -2; \
01831         else naa = -1; \
01832     } \
01833     else if ( ia == 0 ) { aa[0] = 0; naa = 0; } \
01834     else { \
01835         aa[0] = ia; \
01836         if ( ( aa[1] = ia » BITSINWORD ) != 0 ) naa = 2; \
01837         else naa = 1; \
01838     }
01839
01840 void GCD(UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
01841 {
01842     int ja = 0, jb = 0, j;
01843     UWORD *r,*t;
01844     UWORD *x1, *x2, *x3;
01845     WORD nd,naa,nbb;
01846     ULONG ia,ib,ic,id,u,v,w,q,T;
01847     UWORD aa[2], bb[2];
01848 /*
01849     First eliminate easy powers of 2^...
01850 */
01851     while ( a[0] == 0 ) { na--; ja++; a++; }
01852     while ( b[0] == 0 ) { nb--; jb++; b++; }
01853     if ( ja > jb ) ja = jb;
01854     if ( ja > 0 ) {
01855         j = ja;
01856         do { *c++ = 0; } while ( --j > 0 );
01857     }
01858 /*
01859     Now arrange things such that a >= b
01860 */
01861     if ( na < nb ) {
01862         jb = na; na = nb; nb = jb;
01863     }
01864     exch:
01865     r = a; a = b; b = r;
01866     }
01867     else if ( na == nb ) {
01868         r = a+na;
01869         t = b+nb;
01870         j = na;
01871         while ( --j >= 0 ) {
01872             if ( *--r > *--t ) break;
01873             if ( *r < *t ) goto exch;
01874         }
01875         if ( j < 0 ) {
01876             out:
01877             j = nb;
01878             NCOPY(c,b,j);
01879             *nc = nb+ja;
01880             return;
01881         }
01882     }
01883     {
01884         MLOCK(ErrorMessageLock);
01885         MesPrint("Ordered input, ja = %d", (WORD)ja);
01886         AO.OutSkip = 3;
01887         FiniLine();
01888         j = na; r = a;
01889         while ( --j >= 0 ) { TalToLine((UWORD)(*r++)); TokenToLine((UBYTE *) " "); }
01890         FiniLine();
01891         j = nb; r = b;
01892         while ( --j >= 0 ) { TalToLine((UWORD)(*r++)); TokenToLine((UBYTE *) " "); }
01893         AO.OutSkip = 0;
01894         FiniLine();
01895         MUNLOCK(ErrorMessageLock);
01896     }
01897 */
01898 /*
01899     We have now that A > B
01900     The loop recognizes the case that na-nb >= 1
01901     In that case we just have to divide!
01902 */
01903     r = x1 = NumberMalloc("GCD"); t = x2 = NumberMalloc("GCD"); x3 = NumberMalloc("GCD");
01904     j = na;
01905     NCOPY(r,a,j);
01906     j = nb;
01907     NCOPY(t,b,j);

```

```

01908
01909     for(;;) {
01910
01911         while ( na > nb ) {
01912 toobad:
01913             DivLong(x1,na,x2,nb,c,nc,x3,&nd);
01914             if ( nd == 0 ) { b = x2; goto out; }
01915             t = x1; x1 = x2; x2 = x3; x3 = t; na = nb; nb = nd;
01916             if ( na == 2 ) break;
01917         }
01918     /*
01919     Here we can use the shortcut.
01920 */
01921     if ( na == 2 ) {
01922         v = x1[0] + ( ((ULONG)x1[1]) << BITSINWORD );
01923         w = x2[0];
01924         if ( nb == 2 ) w += ((ULONG)x2[1]) << BITSINWORD;
01925 #ifdef EXTRAGCD2
01926         v = GCD2(v,w);
01927 #else
01928         do { u = v%w; v = w; w = u; } while ( w );
01929 #endif
01930         c[0] = (UWORD)v;
01931         if ( ( c[1] = (UWORD)(v >> BITSINWORD) ) != 0 ) *nc = 2+ja;
01932         else *nc = 1+ja;
01933         NumberFree(x1,"GCD"); NumberFree(x2,"GCD"); NumberFree(x3,"GCD");
01934         return;
01935     }
01936     if ( na == 1 ) {
01937         UWORD ui, uj;
01938         ui = x1[0]; uj = x2[0];
01939 #ifdef EXTRAGCD2
01940         ui = (UWORD)GCD2((ULONG)ui,(ULONG)uj);
01941 #else
01942         do { nd = ui%uj; ui = uj; uj = nd; } while ( nd );
01943 #endif
01944         c[0] = ui;
01945         *nc = 1 + ja;
01946         NumberFree(x1,"GCD"); NumberFree(x2,"GCD"); NumberFree(x3,"GCD");
01947         return;
01948     }
01949     ia = 1; ib = 0; ic = 0; id = 1;
01950     u = ( ((ULONG)x1[na-1]) << BITSINWORD ) + x1[na-2];
01951     v = ( ((ULONG)x2[nb-1]) << BITSINWORD ) + x2[nb-2];
01952
01953     while ( v+ic != 0 && v+id != 0 &&
01954             ( q = (u+ia)/(v+ic) ) == (u+ib)/(v+id) ) {
01955         T = ia-q*ic; ia = ic; ic = T;
01956         T = ib-q*id; ib = id; id = T;
01957         T = u - q*v; u = v; v = T;
01958     }
01959     if ( ib == 0 ) goto toobad;
01960     Convert(ia,aa,naa);
01961     Convert(ib,bb,nbb);
01962     MulLong(x1,na,aa,naa,x3,&nd);
01963     MulLong(x2,nb,bb,nbb,c,nc);
01964     AddLong(x3,nd,c,*nc,c,nc);
01965     Convert(ic,aa,naa);
01966     Convert(id,bb,nbb);
01967     MulLong(x1,na,aa,naa,x3,&nd);
01968     t = c; na = j = *nc; r = x1;
01969     NCOPY(r,t,j);
01970     MulLong(x2,nb,bb,nbb,c,nc);
01971     AddLong(x3,nd,c,*nc,x2,&nb);
01972     }
01973 }
01974
01975 #endif
01976
01977 /*
01978     #] GCD :
01979     #[ GcdLong :          WORD GcdLong(a,na,b,nb,c,nc)
01980
01981     Returns the Greatest Common Divider of a and b in c.
01982     If a and or b are zero an error message will be returned.
01983     The answer is always positive.
01984     In principle a and c can be the same.
01985 */
01986
01987 #ifndef NEWTRICK
01988 /*
01989     #[ Old Routine :
01990 */
01991
01992 int GcdLong(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
01993 {
01994     GETBIDENTITY

```

```

01995     if ( !na || !nb ) {
01996         if ( !na && !nb ) {
01997             MLOCK(ErrorMessageLock);
01998             MesPrint("Cannot take gcd");
01999             MUNLOCK(ErrorMessageLock);
02000             return(-1);
02001         }
02002     }
02003     if ( !na ) {
02004         *nc = abs(nb);
02005         NCOPY(c,b,*nc);
02006         *nc = abs(nb);
02007         return(0);
02008     }
02009
02010     *nc = abs(na);
02011     NCOPY(c,a,*nc);
02012     *nc = abs(na);
02013     return(0);
02014 }
02015 if ( na < 0 ) na = -na;
02016 if ( nb < 0 ) nb = -nb;
02017 if ( na == 1 && nb == 1 ) {
02018 #ifdef EXTRAGCD2
02019     *c = (UWORD)GCD2((ULONG)*a,(ULONG)*b);
02020 #else
02021     UWORD x,y,z;
02022     x = *a;
02023     y = *b;
02024     do { z = x % y; x = y; } while ( ( y = z ) != 0 );
02025     *c = x;
02026 #endif
02027     *nc = 1;
02028 }
02029 else if ( na <= 2 && nb <= 2 ) {
02030     RLONG lx,ly,lz;
02031     if ( na == 2 ) { lx = ((RLONG)(a[1]))«BITSINWORD + *a; }
02032     else { lx = *a; }
02033     if ( nb == 2 ) { ly = ((RLONG)(b[1]))«BITSINWORD + *b; }
02034     else { ly = *b; }
02035 #ifdef LONGWORD
02036 #ifdef EXTRAGCD2
02037     lx = GCD2(lx,ly);
02038 #else
02039     do { lz = lx % ly; lx = ly; } while ( ( ly = lz ) != 0 );
02040 #endif
02041 #else
02042     if ( lx < ly ) { lz = lx; lx = ly; ly = lz; }
02043     do {
02044         lz = lx % ly; lx = ly;
02045     } while ( ( ly = lz ) != 0 && ( lx & AWORDMASK ) != 0 );
02046     if ( ly ) {
02047         do { *c = ((UWORD)lx)%((UWORD)ly); lx = ly; } while ( ( ly = *c ) != 0 );
02048         *c = (UWORD)lx;
02049         *nc = 1;
02050     }
02051     else
02052 #endif
02053     {
02054         *c++ = (UWORD)lx;
02055         if ( ( *c = (UWORD)(lx » BITSINWORD) ) != 0 ) *nc = 2;
02056         else *nc = 1;
02057     }
02058 }
02059 else {
02060 #ifdef EXTRAGCD
02061     GCD(a,na,b,nb,c,nc);
02062 #else
02063 #ifdef NEWGCD
02064     UWORD *x3,*x1,*x2, *GLscrat7, *GLscrat8;
02065     WORD n1,n2,n3,n4;
02066     WORD i, j;
02067     x1 = c; x3 = a; n1 = i = na;
02068     NCOPY(x1,x3,i);
02069     GLscrat7 = NumberMalloc("GcdLong"); GLscrat8 = NumberMalloc("GcdLong");
02070     x2 = GLscrat8; x3 = b; n2 = i = nb;
02071     NCOPY(x2,x3,i);
02072     x1 = c; i = 0;
02073     while ( x1[0] == 0 ) { i += BITSINWORD; x1++; n1--; }
02074     while ( ( x1[0] & 1 ) == 0 ) { i++; SCHUIF(x1,n1) }
02075     x2 = GLscrat8; j = 0;
02076     while ( x2[0] == 0 ) { j += BITSINWORD; x2++; n2--; }
02077     while ( ( x2[0] & 1 ) == 0 ) { j++; SCHUIF(x2,n2) }
02078     if ( j > i ) j = i; /* powers of two in GCD */
02079     for(;;){
02080         if ( n1 > n2 ) {
02081 firstbig:

```

```

02082         SubPLon(x1,n1,x2,n2,x1,&n3);
02083         n1 = n3;
02084         if ( n1 == 0 ) {
02085             x1 = c;
02086             n1 = i = n2; NCOPY(x1,x2,i);
02087             break;
02088         }
02089         while ( ( x1[0] & 1 ) == 0 ) SCHUIF(x1,n1)
02090         if ( n1 == 1 ) {
02091             if ( DivLong(x2,n2,x1,n1,GLscrat7,&n3,x2,&n4) ) goto GcdErr;
02092             n2 = n4;
02093             if ( n2 == 0 ) {
02094                 i = n1; x2 = c; NCOPY(x2,x1,i);
02095                 break;
02096             }
02097 #ifdef EXTRAGCD2
02098             *c = (UWORD)GCD2( (ULONG)x1[0], (ULONG)x2[0] );
02099 #else
02100             {
02101                 UWORD x,y,z;
02102                 x = x1[0];
02103                 y = x2[0];
02104                 do { z = x % y; x = y; } while ( ( y = z ) != 0 );
02105                 *c = x;
02106             }
02107 #endif
02108             n1 = 1;
02109             break;
02110         }
02111     }
02112     else if ( n1 < n2 ) {
02113 lastbig:
02114         SubPLon(x2,n2,x1,n1,x2,&n3);
02115         n2 = n3;
02116         if ( n2 == 0 ) {
02117             i = n1; x2 = c; NCOPY(x2,x1,i);
02118             break;
02119         }
02120         while ( ( x2[0] & 1 ) == 0 ) SCHUIF(x2,n2)
02121         if ( n2 == 1 ) {
02122             if ( DivLong(x1,n1,x2,n2,GLscrat7,&n3,x1,&n4) ) goto GcdErr;
02123             n1 = n4;
02124             if ( n1 == 0 ) {
02125                 x1 = c;
02126                 n1 = i = n2; NCOPY(x1,x2,i);
02127                 break;
02128             }
02129 #ifdef EXTRAGCD2
02130             *c = (UWORD)GCD2( (ULONG)x2[0], (ULONG)x1[0] );
02131 #else
02132             {
02133                 UWORD x,y,z;
02134                 x = x2[0];
02135                 y = x1[0];
02136                 do { z = x % y; x = y; } while ( ( y = z ) != 0 );
02137                 *c = x;
02138             }
02139 #endif
02140             n1 = 1;
02141             break;
02142         }
02143     }
02144     else {
02145         for ( i = n1-1; i >= 0; i-- ) {
02146             if ( x1[i] > x2[i] ) goto firstbig;
02147             else if ( x1[i] < x2[i] ) goto lastbig;
02148         }
02149         i = n1; x2 = c; NCOPY(x2,x1,i);
02150         break;
02151     }
02152 }
02153 /*
02154 Now the GCD is in c but still needs j powers of 2.
02155 */
02156 x1 = c;
02157 while ( j >= BITSINWORD ) {
02158     for ( i = n1; i > 0; i-- ) x1[i] = x1[i-1];
02159     x1[0] = 0; n1++;
02160     j -= BITSINWORD;
02161 }
02162 if ( j > 0 ) {
02163     ULONG a1,a2 = 0;
02164     for ( i = 0; i < n1; i++ ) {
02165         a1 = x1[i]; a1 <= j;
02166         a2 += a1;
02167         x1[i] = a2;
02168         a2 >= BITSINWORD;

```

```

02169         }
02170         if ( a2 != 0 ) {
02171             x1[n1++] = a2;
02172         }
02173     }
02174     *nc = n1;
02175     NumberFree (GLscrat7,"GcdLong"); NumberFree (GLscrat8,"GcdLong");
02176 #else
02177     UWORD *x1,*x2,*x3,*x4,*c1,*c2;
02178     WORD n1,n2,n3,n4,i;
02179     x1 = c; x3 = a; n1 = i = na;
02180     NCOPY(x1,x3,i);
02181     x1 = c; c1 = x2 = NumberMalloc("GcdLong"); x3 = NumberMalloc("GcdLong"); x4 =
NumberMalloc("GcdLong");
02182     c2 = b; n2 = i = nb;
02183     NCOPY(c1,c2,i);
02184     for(;;){
02185         if ( DivLong(x1,n1,x2,n2,x4,&n4,x3,&n3) ) goto GcdErr;
02186         if ( !n3 ) { x1 = x2; n1 = n2; break; }
02187         if ( DivLong(x2,n2,x3,n3,x4,&n4,x1,&n1) ) goto GcdErr;
02188         if ( !n1 ) { x1 = x3; n1 = n3; break; }
02189         if ( DivLong(x3,n3,x1,n1,x4,&n4,x2,&n2) ) goto GcdErr;
02190         if ( !n2 ) {
02191             *nc = n1;
02192             NumberFree(x2,"GcdLong"); NumberFree(x3,"GcdLong"); NumberFree(x4,"GcdLong");
02193             return(0);
02194         }
02195     }
02196     *nc = i = n1;
02197     NCOPY(c,x1,i);
02198     NumberFree(x2,"GcdLong"); NumberFree(x3,"GcdLong"); NumberFree(x4,"GcdLong");
02199 #endif
02200 #endif
02201     }
02202     return(0);
02203 GcdErr:
02204     MLOCK(ErrorMessageLock);
02205     MesCall("GcdLong");
02206     MUNLOCK(ErrorMessageLock);
02207     SETERROR(-1)
02208 }
02209 /*
02210 #] Old Routine :
02211 */
02212 #else
02213 /*
02214     New routine for GcdLong that uses smart shortcuts.
02215     Algorithm by J. Vermaseren 15-nov-2006.
02216     It runs faster for very big numbers but only by a fixed factor.
02217     There is no improvement in the power behaviour.
02218     Improvement on the whole of hf9 (multiple zeta values at weight 9):
02219         Better than a factor 2 on a 32 bits architecture and 2.76 on a
02220         64 bits architecture.
02221     On hf10 (MZV's at weight 10), 64 bits architecture: factor 7.
02222
02223     If we have two long numbers (na,nb > GCDMAX) we will work in a
02224     truncated way. At the moment of writing (15-nov-2006) it isn't
02225     clear whether this algorithm is an invention or a reinvention.
02226     A short search on the web didn't show anything.
02227
02228     31-jul-2007:
02229     A better search shows that this is an adaptation of the Lehmer-Euclid
02230     algorithm, already described in Knuth. Here we can work without upper
02231     and lower limit because we are only interested in the GCD, not the
02232     extra numbers. Also it takes already some features of the double
02233     digit Lehmer-Euclid algorithm of Jeebelean it seems.
02234
02235     Maybe this can be programmed slightly better and we can get another
02236     few percent speed increase. Further improvements for the asymptotic
02237     case come from splitting the calculation as in Karatsuba and working
02238     with FFT divisions and multiplications etc. But this is when hundreds
02239     of words are involved at the least.
02240
02241     Algorithm
02242
02243     1: while ( na > nb || nb < GCDMAX ) {
02244         if ( nb == 0 ) { result in a }
02245         c = a % b;
02246         a = b;
02247         b = c;
02248     }
02249
02250     2: Make the truncated values in which a and b are the combinations
02251     of the top two words of a and b. The whole numbers are aa and bb now.
02252     3: ma1 = 1; ma2 = 0; mb1 = 0; mb2 = 1;
02253     4: A = a; B = b; m = a/b; c = a - m*b;
02254         c = ma1*a+ma2*b-m*(mb1*a+mb2*b) = (ma1-m*mb1)*a+(ma2-m*mb2)*b

```

```

02255     mc1 = ma1-m*mb1; mc2 = ma2-m*mb2;
02256 5: a = b; ma1 = mb1; ma2 = mb2;
02257   b = c; mb1 = mc1; mb2 = mc2;
02258 6: if ( b != 0 && nb >= FULLMAX ) goto 4;
02259 7: Now construct the new quantities
02260     ma1*aa+ma2*bb and mb1*aa+mb2*bb
02261 8: goto 1;
02262
02263 The essence of the above algorithm is that we do the divisions only
02264 on relatively short numbers. Also usually there are many steps 4&5
02265 for each step 7. This eliminates many operations.
02266 The termination at FULLMAX is that we make errors by not considering
02267 the tail of the number. If we run b down all the way, the errors combine
02268 in such a way that the new numbers may be of the same order as the old
02269 numbers. By stopping halfway we don't get the error beyond halfway
02270 either. Unfortunately this means that a >= FULLMAX and hence na > nb
02271 which means that next we will have a complete division. But just once.
02272 Running the steps 4-6 till a < FULLMAX runs already into problems.
02273 It may be necessary to experiment a bit to obtain the optimum value
02274 of GCDMAX.
02275 */
02276
02277 int GcdLong(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
02278 {
02279     GETBIDENTITY
02280     UWORD x,y,z;
02281     UWORD *x1,*x2,*x3,*x4,*x5,*d;
02282     UWORD *GLscrat6,*GLscrat7,*GLscrat8,*GLscrat9,*GLscrat10;
02283     WORD n1,n2,n3,n4,n5,i;
02284     RLONG lx,ly,lz;
02285     LONG ma1, ma2, mb1, mb2, mc1, mc2, m;
02286     if ( !na || !nb ) {
02287         if ( !na && !nb ) {
02288             MLOCK(ErrorMessageLock);
02289             MesPrint("Cannot take gcd");
02290             MUNLOCK(ErrorMessageLock);
02291             return(-1);
02292         }
02293
02294         if ( !na ) {
02295             *nc = abs(nb);
02296             NCOPY(c,b,*nc);
02297             *nc = abs(nb);
02298             return(0);
02299         }
02300
02301         *nc = abs(na);
02302         NCOPY(c,a,*nc);
02303         *nc = abs(na);
02304         return(0);
02305     }
02306     if ( na < 0 ) na = -na;
02307     if ( nb < 0 ) nb = -nb;
02308 /*
02309     # [ GMP stuff :
02310 */
02311 #ifdef WITHGMP
02312     if ( na > 3 && nb > 3 ) {
02313         int ii;
02314         mp_limb_t *upa, *upb, *upc, xx;
02315         UWORD *uw, *u1, *u2;
02316         unsigned int tcounta, tcountb, tcountal, tcountbl;
02317         mp_size_t ana, anb, anc;
02318
02319         u1 = uw = NumberMalloc("GcdLong");
02320         upa = (mp_limb_t *)u1;
02321         ana = na; tcountal = 0;
02322         while ( a[0] == 0 ) { a++; ana--; tcountal++; }
02323         for ( ii = 0; ii < ana; ii++ ) { *uw++ = *a++; }
02324         if ( ( ana & 1 ) != 0 ) { *uw = 0; ana++; }
02325         ana /= 2;
02326
02327         u2 = uw = NumberMalloc("GcdLong");
02328         upb = (mp_limb_t *)u2;
02329         anb = nb; tcountbl = 0;
02330         while ( b[0] == 0 ) { b++; anb--; tcountbl++; }
02331         for ( ii = 0; ii < anb; ii++ ) { *uw++ = *b++; }
02332         if ( ( anb & 1 ) != 0 ) { *uw = 0; anb++; }
02333         anb /= 2;
02334
02335         xx = upa[0]; tcounta = 0;
02336         while ( ( xx & 15 ) == 0 ) { tcounta += 4; xx >>= 4; }
02337         while ( ( xx & 1 ) == 0 ) { tcounta += 1; xx >>= 1; }
02338         xx = upb[0]; tcountb = 0;
02339         while ( ( xx & 15 ) == 0 ) { tcountb += 4; xx >>= 4; }
02340         while ( ( xx & 1 ) == 0 ) { tcountb += 1; xx >>= 1; }
02341

```



```

02342     if ( tcounta ) {
02343         mpn_rshift(upa,upa,ana,tcounta);
02344         if ( upa[ana-1] == 0 ) ana--;
02345     }
02346     if ( tcountb ) {
02347         mpn_rshift(upb,upb,anb,tcountb);
02348         if ( upb[anb-1] == 0 ) anb--;
02349     }
02350
02351     upc = (mp_limb_t *) (NumberMalloc("GcdLong"));
02352     if ( ( ana > anb ) || ( ( ana == anb ) && ( upa[ana-1] >= upb[ana-1] ) ) ) {
02353         anc = mpn_gcd(upc,upa,ana,upb,anb);
02354     }
02355     else {
02356         anc = mpn_gcd(upc,upb,anb,upa,ana);
02357     }
02358
02359     tcounta = tcounta*BITSINWORD + tcounta;
02360     tcountb = tcountb*BITSINWORD + tcountb;
02361     if ( tcountb > tcounta ) tcountb = tcounta;
02362     tcounta = tcountb/BITSINWORD;
02363     tcountb = tcountb%BITSINWORD;
02364
02365     if ( tcountb ) {
02366         xx = mpn_lshift(upc,upc,anc,tcountb);
02367         if ( xx ) { upc[anc] = xx; anc++; }
02368     }
02369
02370     uw = (UWORD *)upc; anc *= 2;
02371     while ( uw[anc-1] == 0 ) anc--;
02372     for ( ii = 0; ii < (int)tcounta; ii++ ) *c++ = 0;
02373     for ( ii = 0; ii < anc; ii++ ) *c++ = *uw++;
02374     *nc = anc + tcounta;
02375     NumberFree(u1,"GcdLong"); NumberFree(u2,"GcdLong"); NumberFree((UWORD *) (upc),"GcdLong");
02376     return(0);
02377 }
02378 #endif
02379 /*
02380 #] GMP stuff :
02381 */
02382 /*
02383 #[ Easy cases :
02384 */
02385     if ( na == 1 && nb == 1 ) {
02386         x = *a;
02387         y = *b;
02388         do { z = x % y; x = y; } while ( ( y = z ) != 0 );
02389         *c = x;
02390         *nc = 1;
02391         return(0);
02392     }
02393     else if ( na <= 2 && nb <= 2 ) {
02394         if ( na == 2 ) { lx = ((RLONG) (a[1]))«BITSINWORD + *a; }
02395         else { lx = *a; }
02396         if ( nb == 2 ) { ly = ((RLONG) (b[1]))«BITSINWORD + *b; }
02397         else { ly = *b; }
02398         if ( lx < ly ) { lz = lx; lx = ly; ly = lz; }
02399     #if ( BITSINWORD == 16 )
02400         do {
02401             lz = lx % ly; lx = ly;
02402         } while ( ( ly = lz ) != 0 );
02403     #else
02404         do {
02405             lz = lx % ly; lx = ly;
02406         } while ( ( ly = lz ) != 0 && ( lx & AWORDMASK ) != 0 );
02407         if ( ly ) {
02408             x = (UWORD)lx; y = (UWORD)ly;
02409             do { *c = x % y; x = y; } while ( ( y = *c ) != 0 );
02410             *c = x;
02411             *nc = 1;
02412         }
02413         else
02414     #endif
02415     {
02416         *c++ = (UWORD)lx;
02417         if ( ( *c = (UWORD) (lx » BITSINWORD) ) != 0 ) *nc = 2;
02418         else *nc = 1;
02419     }
02420     return(0);
02421 }
02422 /*
02423 #] Easy cases :
02424 */
02425     GLscrat6 = NumberMalloc("GcdLong"); GLscrat7 = NumberMalloc("GcdLong");
02426     GLscrat8 = NumberMalloc("GcdLong");
02427     GLscrat9 = NumberMalloc("GcdLong"); GLscrat10 = NumberMalloc("GcdLong");
02428     restart;;

```

```

02429 /*
02430      #] Easy cases :
02431 */
02432     if ( na == 1 && nb == 1 ) {
02433         x = *a;
02434         y = *b;
02435         do { z = x % y; x = y; } while ( ( y = z ) != 0 );
02436         *c = x;
02437         *nc = 1;
02438     }
02439     else if ( na <= 2 && nb <= 2 ) {
02440         if ( na == 2 ) { lx = ((RLONG) (a[1]))«BITSINWORD) + *a; }
02441         else { lx = *a; }
02442         if ( nb == 2 ) { ly = ((RLONG) (b[1]))«BITSINWORD) + *b; }
02443         else { ly = *b; }
02444         if ( lx < ly ) { lz = lx; lx = ly; ly = lz; }
02445     #if ( BITSINWORD == 16 )
02446         do {
02447             lz = lx % ly; lx = ly;
02448         } while ( ( ly = lz ) != 0 );
02449     #else
02450         do {
02451             lz = lx % ly; lx = ly;
02452         } while ( ( ly = lz ) != 0 && ( lx & AWORDMASK ) != 0 );
02453         if ( ly ) {
02454             x = (UWORD)lx; y = (UWORD)ly;
02455             do { *c = x % y; x = y; } while ( ( y = *c ) != 0 );
02456             *c = x;
02457             *nc = 1;
02458         }
02459         else
02460     #endif
02461     {
02462         *c++ = (UWORD)lx;
02463         if ( ( *c = (UWORD) (lx » BITSINWORD) ) != 0 ) *nc = 2;
02464         else *nc = 1;
02465     }
02466 }
02467 /*
02468      #] Easy cases :
02469      #] Original code :
02470 */
02471     else if ( na < GCDMAX || nb < GCDMAX || na != nb ) {
02472         if ( na < nb ) {
02473             x2 = GLscrat8; x3 = a; n2 = i = na;
02474             NCOPY(x2,x3,i);
02475             x1 = c; x3 = b; n1 = i = nb;
02476             NCOPY(x1,x3,i);
02477         }
02478         else {
02479             x1 = c; x3 = a; n1 = i = na;
02480             NCOPY(x1,x3,i);
02481             x2 = GLscrat8; x3 = b; n2 = i = nb;
02482             NCOPY(x2,x3,i);
02483         }
02484         x1 = c; x2 = GLscrat8; x3 = GLscrat7; x4 = GLscrat6;
02485         for(;;){
02486             if ( DivLong(x1,n1,x2,n2,x4,&n4,x3,&n3) ) goto GcdErr;
02487             if ( !n3 ) { x1 = x2; n1 = n2; break; }
02488             if ( n2 <= 2 ) { a = x2; b = x3; na = n2; nb = n3; goto restart; }
02489             if ( n3 >= GCDMAX && n2 == n3 ) {
02490                 a = GLscrat9; b = GLscrat10; na = n2; nb = n3;
02491                 for ( i = 0; i < na; i++ ) a[i] = x2[i];
02492                 for ( i = 0; i < nb; i++ ) b[i] = x3[i];
02493                 goto newtrick;
02494             }
02495             if ( DivLong(x2,n2,x3,n3,x4,&n4,x1,&n1) ) goto GcdErr;
02496             if ( !n1 ) { x1 = x3; n1 = n3; break; }
02497             if ( n3 <= 2 ) { a = x3; b = x1; na = n3; nb = n1; goto restart; }
02498             if ( n1 >= GCDMAX && n1 == n3 ) {
02499                 a = GLscrat9; b = GLscrat10; na = n3; nb = n1;
02500                 for ( i = 0; i < na; i++ ) a[i] = x3[i];
02501                 for ( i = 0; i < nb; i++ ) b[i] = x1[i];
02502                 goto newtrick;
02503             }
02504             if ( DivLong(x3,n3,x1,n1,x4,&n4,x2,&n2) ) goto GcdErr;
02505             if ( !n2 ) { *nc = n1; goto normalend; }
02506             if ( n1 <= 2 ) { a = x1; b = x2; na = n1; nb = n2; goto restart; }
02507             if ( n2 >= GCDMAX && n2 == n1 ) {
02508                 a = GLscrat9; b = GLscrat10; na = n1; nb = n2;
02509                 for ( i = 0; i < na; i++ ) a[i] = x1[i];
02510                 for ( i = 0; i < nb; i++ ) b[i] = x2[i];
02511                 goto newtrick;
02512             }
02513         }
02514         *nc = i = n1;
02515         NCOPY(c,x1,i);

```

```

02516     }
02517 /*
02518     #] Original code :
02519     #[ New code :
02520 */
02521     else {
02522 /*
02523         This is the new algorithm starting at step 3.
02524
02525         3: ma1 = 1; ma2 = 0; mb1 = 0; mb2 = 1;
02526         4: A = a; B = b; m = a/b; c = a - m*b;
02527           c = ma1*a+ma2*b-m*(mb1*a+mb2*b) = (ma1-m*mb1)*a+(ma2-m*mb2)*b
02528           mc1 = ma1-m*mb1; mc2 = ma2-m*mb2;
02529         5: a = b; ma1 = mb1; ma2 = mb2;
02530           b = c; mb1 = mc1; mb2 = mc2;
02531         6: if ( b != 0 ) goto 4;
02532 */
02533 newtrick;;
02534     ma1 = 1; ma2 = 0; mb1 = 0; mb2 = 1;
02535     lx = ((RLONG)(a[na-1]))«BITSINWORD + a[na-2];
02536     ly = ((RLONG)(b[nb-1]))«BITSINWORD + b[nb-2];
02537     if ( ly > lx ) { lz = lx; lx = ly; ly = lz; d = a; a = b; b = d; }
02538     do {
02539         m = lx/ly;
02540         mc1 = ma1-m*mb1; mc2 = ma2-m*mb2;
02541         ma1 = mb1; ma2 = mb2; mb1 = mc1; mb2 = mc2;
02542         lz = lx - m*ly; lx = ly; ly = lz;
02543     } while ( ly >= FULLMAX );
02544 /*
02545     Next the construction of the two new numbers
02546
02547     7: Now construct the new quantities
02548        a = ma1*aa+ma2*bb and b = mb1*aa+mb2*bb
02549 */
02550     x1 = GLscrat6;
02551     x2 = GLscrat7;
02552     x3 = GLscrat8;
02553     x5 = GLscrat10;
02554     if ( ma1 < 0 ) {
02555         ma1 = -ma1;
02556         x1[0] = (UWORD)ma1;
02557         x1[1] = (UWORD)(ma1 » BITSINWORD);
02558         if ( x1[1] ) n1 = -2;
02559         else n1 = -1;
02560     }
02561     else {
02562         x1[0] = (UWORD)ma1;
02563         x1[1] = (UWORD)(ma1 » BITSINWORD);
02564         if ( x1[1] ) n1 = 2;
02565         else n1 = 1;
02566     }
02567     if ( MulLong(a,na,x1,n1,x2,&n2) ) goto GcdErr;
02568     if ( ma2 < 0 ) {
02569         ma2 = -ma2;
02570         x1[0] = (UWORD)ma2;
02571         x1[1] = (UWORD)(ma2 » BITSINWORD);
02572         if ( x1[1] ) n1 = -2;
02573         else n1 = -1;
02574     }
02575     else {
02576         x1[0] = (UWORD)ma2;
02577         x1[1] = (UWORD)(ma2 » BITSINWORD);
02578         if ( x1[1] ) n1 = 2;
02579         else n1 = 1;
02580     }
02581     if ( MulLong(b,nb,x1,n1,x3,&n3) ) goto GcdErr;
02582     if ( AddLong(x2,n2,x3,n3,c,&n4) ) goto GcdErr;
02583     if ( mb1 < 0 ) {
02584         mb1 = -mb1;
02585         x1[0] = (UWORD)mb1;
02586         x1[1] = (UWORD)(mb1 » BITSINWORD);
02587         if ( x1[1] ) n1 = -2;
02588         else n1 = -1;
02589     }
02590     else {
02591         x1[0] = (UWORD)mb1;
02592         x1[1] = (UWORD)(mb1 » BITSINWORD);
02593         if ( x1[1] ) n1 = 2;
02594         else n1 = 1;
02595     }
02596     if ( MulLong(a,na,x1,n1,x2,&n2) ) goto GcdErr;
02597     if ( mb2 < 0 ) {
02598         mb2 = -mb2;
02599         x1[0] = (UWORD)mb2;
02600         x1[1] = (UWORD)(mb2 » BITSINWORD);
02601         if ( x1[1] ) n1 = -2;
02602         else n1 = -1;

```

```

02603     }
02604     else {
02605         x1[0] = (UWORD)mb2;
02606         x1[1] = (UWORD)(mb2 » BITSINWORD);
02607         if ( x1[1] ) n1 = 2;
02608         else      n1 = 1;
02609     }
02610     if ( MulLong(b,nb,x1,n1,x3,&n3) ) goto GcdErr;
02611     if ( AddLong(x2,n2,x3,n3,x5,&n5) ) goto GcdErr;
02612     a = c; na = n4; b = x5; nb = n5;
02613     if ( nb == 0 ) { *nc = n4; goto normalend; }
02614     x4 = GLscrat9;
02615     for ( i = 0; i < na; i++ ) x4[i] = a[i];
02616     a = x4;
02617     if ( na < 0 ) na = -na;
02618     if ( nb < 0 ) nb = -nb;
02619     /*
02620     The typical case now is that in a we have the last step to go
02621     to loose the leading word, while in b we have lost the leading word.
02622     We could go to DivLong now but we can also add an extra step that
02623     is less wasteful.
02624     In the case that the new leading word of b is extrememly short (like 1)
02625     we make a rather large error of course. In the worst case the whole
02626     will be intercepted by DivLong after all, but that is so rare that
02627     it shouldn't influence any timing in a measurable way.
02628     */
02629     if ( nb >= GCDMAX && na == nb+1 && b[nb-1] >= HALFMAX && b[nb-1] > a[na-1] ) {
02630         lx = (((RLONG)(a[na-1]))«BITSINWORD) + a[na-2];
02631         x1[0] = lx/b[nb-1]; n1 = 1;
02632         MulLong(b,nb,x1,n1,x2,&n2);
02633         n2 = -n2;
02634         AddLong(a,na,x2,n2,x4,&n4);
02635         if ( n4 == 0 ) {
02636             *nc = nb;
02637             for ( i = 0; i < nb; i++ ) c[i] = b[i];
02638             goto normalend;
02639         }
02640         if ( n4 < 0 ) n4 = -n4;
02641         a = b; na = nb; b = x4; nb = n4;
02642     }
02643     goto restart;
02644     /*
02645     #] New code :
02646     */
02647     }
02648 normalend:
02649     NumberFree(GLscrat6,"GcdLong"); NumberFree(GLscrat7,"GcdLong"); NumberFree(GLscrat8,"GcdLong");
02650     NumberFree(GLscrat9,"GcdLong"); NumberFree(GLscrat10,"GcdLong");
02651     return(0);
02652 GcdErr:
02653     MLOCK(ErrorMessageLock);
02654     MesCall("GcdLong");
02655     MUNLOCK(ErrorMessageLock);
02656     NumberFree(GLscrat6,"GcdLong"); NumberFree(GLscrat7,"GcdLong"); NumberFree(GLscrat8,"GcdLong");
02657     NumberFree(GLscrat9,"GcdLong"); NumberFree(GLscrat10,"GcdLong");
02658     SETERROR(-1)
02659 }
02660
02661 #endif
02662
02663 /*
02664     #] GcdLong :
02665     #[ GetBinom :      WORD GetBinom(a,na,i1,i2)
02666     */
02667
02668 int GetBinom(UWORD *a, WORD *na, WORD i1, WORD i2)
02669 {
02670     GETIDENTITY
02671     WORD j, k, l;
02672     UWORD *GBscrat3, *GBscrat4;
02673     if ( i1-i2 < i2 ) i2 = i1-i2;
02674     if ( i2 == 0 ) { *a = 1; *na = 1; return(0); }
02675     if ( i2 > i1 ) { *a = 0; *na = 0; return(0); }
02676     *a = i1; *na = 1;
02677     GBscrat3 = NumberMalloc("GetBinom"); GBscrat4 = NumberMalloc("GetBinom");
02678     for ( j = 2; j <= i2; j++ ) {
02679         GBscrat3[0] = i1+1-j;
02680         if ( MulLong(a,*na,GBscrat3,(WORD)1,GBscrat4,&k) ) goto CalledFrom;
02681         GBscrat3[0] = j;
02682         if ( DivLong(GBscrat4,k,GBscrat3,(WORD)1,a,na,GBscrat3,&l) ) goto CalledFrom;
02683     }
02684     NumberFree(GBscrat3,"GetBinom"); NumberFree(GBscrat4,"GetBinom");
02685     return(0);
02686 CalledFrom:
02687     MLOCK(ErrorMessageLock);
02688     MesCall("GetBinom");
02689     MUNLOCK(ErrorMessageLock);

```

```

02690     NumberFree(GBscrat3,"GetBinom"); NumberFree(GBscrat4,"GetBinom");
02691     SETERROR(-1)
02692 }
02693
02694 /*
02695     #] GetBinom :
02696     #[ LcmLong :      WORD LcmLong(a,na,b,nb)
02697
02698     Computes the LCM of the long numbers a and b and puts the result
02699     in c. c is allowed to be equal to a.
02700 */
02701
02702 int LcmLong(PHEAD UWORD *a, WORD na, UWORD *b, WORD nb, UWORD *c, WORD *nc)
02703 {
02704     int error = 0;
02705     UWORD *d = NumberMalloc("LcmLong");
02706     UWORD *e = NumberMalloc("LcmLong");
02707     UWORD *f = NumberMalloc("LcmLong");
02708     WORD nd, ne, nf;
02709     GcdLong(BHEAD a, na, b, nb, d, &nd);
02710     DivLong(a,na,d,nd,e,&ne,f,&nf);
02711     if ( MulLong(b,nb,e,ne,c,nc) ) {
02712         MLOCK(ErrorMessageLock);
02713         MesCall("LcmLong");
02714         MUNLOCK(ErrorMessageLock);
02715         error = -1;
02716     }
02717     NumberFree(f,"LcmLong");
02718     NumberFree(e,"LcmLong");
02719     NumberFree(d,"LcmLong");
02720     return(error);
02721 }
02722
02723 /*
02724     #] LcmLong :
02725     #[ TakeLongRoot:   int TakeLongRoot(a,n,power)
02726
02727     Takes the 'power'-root of the long number in a.
02728     If the root could be taken the return value is zero.
02729     If the root could not be taken, the return value is 1.
02730     The root will be in a if it could be taken, otherwise there will be garbage
02731     Algorithm: (assume b is guess of root, b' better guess)
02732         b' = (a-(power-1)*b^power)/(n*b^(power-1))
02733     Note: power should be positive!
02734 */
02735
02736 int TakeLongRoot(UWORD *a, WORD *n, WORD power)
02737 {
02738     GETIDENTITY
02739     int numbits, guessbits, i, retval = 0;
02740     UWORD x, *b, *c, *d, *e;
02741     WORD na, nb, nc, nd, ne;
02742     if ( *n < 0 && ( power & 1 ) == 0 ) return(1);
02743     if ( power == 1 ) return(0);
02744     if ( *n < 0 ) { na = -*n; }
02745     else { na = *n; }
02746     if ( na == 1 ) {
02747         /* Special cases that are the most frequent */
02748         if ( a[0] == 1 ) return(0);
02749         if ( power < BITSINWORD && na == 1 && a[0] == (UWORD)(1<<power) ) {
02750             a[0] = 2; return(0);
02751         }
02752         if ( 2*power < BITSINWORD && na == 1 && a[0] == (UWORD)(1<<(2*power)) ) {
02753             a[0] = 4; return(0);
02754         }
02755     }
02756     /*
02757     Step 1: make a guess. We count the number of bits.
02758     numbits will be the 1+2log(a)
02759     */
02760     numbits = BITSINWORD*(na-1);
02761     x = a[na-1];
02762     while ( ( x >> 8 ) != 0 ) { numbits += 8; x >>= 8; }
02763     if ( ( x >> 4 ) != 0 ) { numbits += 4; x >>= 4; }
02764     if ( ( x >> 2 ) != 0 ) { numbits += 2; x >>= 2; }
02765     if ( ( x >> 1 ) != 0 ) numbits++;
02766     guessbits = numbits / power;
02767     if ( guessbits <= 0 ) return(1); /* root < 2 and 1 we did already */
02768     nb = guessbits/BITSINWORD;
02769     /*
02770     The recursion is:
02771     (b'-b) = (a/b^(power-1)-b)/n
02772             = (a/c-b)/n
02773             = (d-b)/n    (remainder of a/c is e)
02774             = c/n    (we reuse the scratch array c)
02775     Termination can be tricky. When a/c has no remainder and = b we have a root.
02776     When d = b but the remainder of a/c != 0, there is definitely no root.

```

```

02777 */
02778 b = NumberMalloc("TakeLongRoot"); c = NumberMalloc("TakeLongRoot");
02779 d = NumberMalloc("TakeLongRoot"); e = NumberMalloc("TakeLongRoot");
02780 for ( i = 0; i < nb; i++ ) { b[i] = 0; }
02781 b[nb] = 1 « (guessbits%BITSINWORD);
02782 nb++;
02783 for(;;) {
02784     nc = nb;
02785     for ( i = 0; i < nb; i++ ) c[i] = b[i];
02786     if ( RaisPow(BHEAD c,&nc,power-1) ) goto TLcall;
02787     if ( DivLong(a,na,c,nc,d,&nd,e,&ne) ) goto TLcall;
02788     nb = -nb;
02789     if ( AddLong(d,nd,b,nb,c,&nc) ) goto TLcall;
02790     nb = -nb;
02791     if ( nc == 0 ) {
02792         if ( ne == 0 ) break;
02793         retval = 1; break;
02794     /*
02795         else {
02796             NumberFree(b,"TakeLongRoot"); NumberFree(c,"TakeLongRoot");
02797             NumberFree(d,"TakeLongRoot"); NumberFree(e,"TakeLongRoot");
02798             return(1);
02799         }
02800     */
02801     }
02802     DivLong(c,nc,(UWORD *)(&power),1,d,&nd,e,&ne);
02803     if ( nd == 0 ) {
02804         retval = 1;
02805         break;
02806     /*
02807         NumberFree(b,"TakeLongRoot"); NumberFree(c,"TakeLongRoot");
02808         NumberFree(d,"TakeLongRoot"); NumberFree(e,"TakeLongRoot");
02809         return(1);
02810     */
02811     /*
02812         This code tries b+1 as a final possibility.
02813         We believe this is not needed
02814         UWORD one = 1;
02815         if ( AddLong(b,nb,&one,1,c,&nc) ) goto TLcall;
02816         if ( RaisPow(BHEAD c,&nc,power-1) ) goto TLcall;
02817         if ( DivLong(a,na,c,nc,d,&nd,e,&ne) ) goto TLcall;
02818         if ( ne != 0 ) return(1);
02819         nb = -nb;
02820         if ( SubLong(d,nd,b,nb,c,&nc) ) goto TLcall;
02821         nb = -nb;
02822         if ( nc != 0 ) {
02823             NumberFree(b,"TakeLongRoot"); NumberFree(c,"TakeLongRoot");
02824             NumberFree(d,"TakeLongRoot"); NumberFree(e,"TakeLongRoot");
02825             return(1);
02826         }
02827         break;
02828     */
02829     }
02830     if ( AddLong(b,nb,d,nd,b,&nb) ) goto TLcall;
02831 }
02832 for ( i = 0; i < nb; i++ ) a[i] = b[i];
02833 if ( *n < 0 ) *n = -nb;
02834 else *n = nb;
02835 NumberFree(b,"TakeLongRoot"); NumberFree(c,"TakeLongRoot");
02836 NumberFree(d,"TakeLongRoot"); NumberFree(e,"TakeLongRoot");
02837 return(retval);
02838 TLcall:
02839 MLOCK(ErrorMessageLock);
02840 MesCall("TakeLongRoot");
02841 MUNLOCK(ErrorMessageLock);
02842 NumberFree(b,"TakeLongRoot"); NumberFree(c,"TakeLongRoot");
02843 NumberFree(d,"TakeLongRoot"); NumberFree(e,"TakeLongRoot");
02844 Terminate(-1);
02845 return(-1);
02846 }
02847
02848 /*
02849     #] TakeLongRoot:
02850     #[ MakeRational:
02851
02852     Makes the integer a mod m into a traction b/c with |b|,|c| < sqrt(m)
02853     For the algorithm, see MakeLongRational.
02854 */
02855
02856 int MakeRational(WORD a,WORD m, WORD *b, WORD *c)
02857 {
02858     LONG x1,x2,x3,x4,y1,y2;
02859     if ( a < 0 ) { a = a+m; }
02860     if ( a <= 1 ) {
02861         if ( a > m/2 ) a = a-m;
02862         *b = a; *c = 1; return(0);
02863     }

```

```

02864     x1 = m; x2 = a;
02865     if ( x2*x2 >= m ) {
02866         y1 = x1/x2; y2 = x1%x2; x3 = 1; x4 = -y1; x1 = x2; x2 = y2;
02867         while ( x2*x2 >= m ) {
02868             y1 = x1/x2; y2 = x1%x2; x1 = x2; x2 = y2; y2 = x3-y1*x4; x3 = x4; x4 = y2;
02869         }
02870     }
02871     else x4 = 1;
02872     if ( x2 == 0 ) { return(1); }
02873     if ( x2 > m/2 ) *b = x2-m;
02874     else *b = x2;
02875     if ( x4 > m/2 ) { *c = x4-m; *c = -*c; *b = -*b; }
02876     else if ( x4 <= -m/2 ) { x4 += m; *c = x4; }
02877     else if ( x4 < 0 ) { x4 = -x4; *c = x4; *b = -*b; }
02878     else *c = x4;
02879     return(0);
02880 }
02881
02882 /*
02883     #] MakeRational:
02884     #[ MakeLongRational:
02885
02886     Converts the long number a mod m into the fraction b
02887     One of the properties of b is that num,den < sqrt(m)
02888     The algorithm: Start with:   m 0
02889                                a 1
02890     Make now c=m/a, c1=m/a      c c2=0-c1*1
02891     Make now d=a/c, d1=a/c      d d2=1-d1*c2
02892     Make now e=c/d, e1=c/d      e e2=1-e1*d2
02893     etc till in the first column we get a number < sqrt(m)
02894     We have then f,f2 and the fraction is f/f2.
02895     If at any moment we get a zero, m contained an unlucky prime.
02896
02897     Note that this can be made a lot faster when we make the same
02898     improvements as in the GCD routine. That is something for later.
02899 #ifndef WITHMAKERATIONAL
02900 */
02901
02902 #define COPYLONG(x1,nx1,x2,nx2) { int i; for(i=0;i<ABS(nx2);i++)x1[i]=x2[i];nx1=nx2; }
02903
02904 int MakeLongRational(PHEAD UWORD *a, WORD na, UWORD *m, WORD nm, UWORD *b, WORD *nb)
02905 {
02906     UWORD *root = NumberMalloc("MakeRational");
02907     UWORD *x1 = NumberMalloc("MakeRational");
02908     UWORD *x2 = NumberMalloc("MakeRational");
02909     UWORD *x3 = NumberMalloc("MakeRational");
02910     UWORD *x4 = NumberMalloc("MakeRational");
02911     UWORD *y1 = NumberMalloc("MakeRational");
02912     UWORD *y2 = NumberMalloc("MakeRational");
02913     WORD nroot,nx1,nx2,nx3,nx4,ny1,ny2 = 0,retval = 0;
02914     WORD sign = 1;
02915     /*
02916     Step 1: Take the square root of m
02917     */
02918     COPYLONG(root,nroot,m,nm)
02919     TakeLongRoot(root,&nroot,2);
02920     /*
02921     Step 2: Set the start values
02922     */
02923     if ( na < 0 ) { na = -na; sign = -sign; }
02924     COPYLONG(x1,nx1,m,nm)
02925     COPYLONG(x2,nx2,a,na)
02926     /*
02927     x3[0] = 0, nx3 = 0;
02928     x4[0] = 1, nx4 = 1;
02929     */
02930     /*
02931     The start operation needs some special attention because of the zero.
02932     */
02933     if ( BigLong(x2,nx2,root,nroot) <= 0 ) {
02934         x4[0] = 1, nx4 = 1;
02935         goto gottheanswer;
02936     }
02937     DivLong(x1,nx1,x2,nx2,y1,&ny1,y2,&ny2);
02938     if ( ny2 == 0 ) { retval = 1; goto cleanup; }
02939     COPYLONG(x1,nx1,x2,nx2)
02940     COPYLONG(x2,nx2,y2,ny2)
02941     x3[0] = 1; nx3 = 1;
02942     COPYLONG(x4,nx4,y1,ny1)
02943     nx4 = -nx4;
02944     /*
02945     Now the loop.
02946     */
02947     while ( BigLong(x2,nx2,root,nroot) > 0 ) {
02948         DivLong(x1,nx1,x2,nx2,y1,&ny1,y2,&ny2);
02949         if ( ny2 == 0 ) { retval = 1; goto cleanup; }
02950         COPYLONG(x1,nx1,x2,nx2)

```

```

02951     COPYLONG(x2,nx2,y2,ny2)
02952     MulLong(y1,ny1,x4,nx4,y2,&ny2);
02953     ny2 = -ny2;
02954     AddLong(x3,nx3,y2,ny2,y1,&ny1);
02955     COPYLONG(x3,nx3,x4,nx4)
02956     COPYLONG(x4,nx4,y1,ny1)
02957 }
02958 /*
02959     Now we have the answer. It is x2/x4. It has to be packed into b.
02960 */
02961 gottheanswer:
02962     if ( nx4 < 0 ) { sign = -sign; nx4 = -nx4; }
02963     COPYLONG(b,*nb,x2,nx2)
02964     Pack(b,nb,x4,nx4);
02965     if ( sign < 0 ) *nb = -*nb;
02966 cleanup:
02967     NumberFree(y2,"MakeRational");
02968     NumberFree(y1,"MakeRational");
02969     NumberFree(x4,"MakeRational");
02970     NumberFree(x3,"MakeRational");
02971     NumberFree(x2,"MakeRational");
02972     NumberFree(x1,"MakeRational");
02973     NumberFree(root,"MakeRational");
02974     return(retval);
02975 }
02976
02977 /*
02978 #endif
02979     #] MakeLongRational:
02980     #[ ChineseRemainder:
02981 */
02993 #ifdef WITHCHINESEREMAINDER
02994
02995 int ChineseRemainder(PHEAD MODNUM *a1, MODNUM *a2, MODNUM *a)
02996 {
02997     UWORD *inv1 = NumberMalloc("ChineseRemainder");
02998     UWORD *inv2 = NumberMalloc("ChineseRemainder");
02999     UWORD *fac1 = NumberMalloc("ChineseRemainder");
03000     UWORD *fac2 = NumberMalloc("ChineseRemainder");
03001     UWORD two[1];
03002     WORD ninv1, ninv2, nfac1, nfac2;
03003     if ( a1->na < 0 ) {
03004         AddLong(a1->a,a1->na,a1->m,a1->nm,a1->a,&(a1->na));
03005     }
03006     if ( a2->na < 0 ) {
03007         AddLong(a2->a,a2->na,a2->m,a2->nm,a2->a,&(a2->na));
03008     }
03009     MulLong(a1->m,a1->nm,a2->m,a2->nm,a->m,&(a->nm));
03010
03011     GetLongModInverses(BHEAD a1->m,a1->nm,a2->m,a2->nm,inv1,&ninv1,inv2,&ninv2);
03012     MulLong(inv1,ninv1,a1->m,a1->nm,fac1,&nfac1);
03013     MulLong(inv2,ninv2,a2->m,a2->nm,fac2,&nfac2);
03014
03015     MulLong(fac1,nfac1,a2->a,a2->na,inv1,&ninv1);
03016     MulLong(fac2,nfac2,a1->a,a1->na,inv2,&ninv2);
03017     AddLong(inv1,ninv1,inv2,ninv2,a->a,&(a->na));
03018
03019     two[0] = 2;
03020     MulLong(a->a,a->na,two,1,fac1,&nfac1);
03021     if ( BigLong(fac1,nfac1,a->m,a->nm) > 0 ) {
03022         a->nm = -a->nm;
03023         AddLong(a->a,a->na,a->m,a->nm,a->a,&(a->na));
03024         a->nm = -a->nm;
03025     }
03026     NumberFree(fac2,"ChineseRemainder");
03027     NumberFree(fac1,"ChineseRemainder");
03028     NumberFree(inv2,"ChineseRemainder");
03029     NumberFree(inv1,"ChineseRemainder");
03030     return(0);
03031 }
03032
03033 #endif
03034
03035 /*
03036     #] ChineseRemainder:
03037     #[ RekenLong :
03038     #[ RekenTerms :
03039     #[ CompCoef :      WORD CompCoef(term1,term2)
03040
03041     Compares the coefficients of term1 and term2 by subtracting them.
03042     This does more work than needed but this routine is only called
03043     when sorting functions and function arguments.
03044     (and comparing values
03045 */
03046 /* #define 64SAVE */
03047
03048 WORD CompCoef(WORD *term1, WORD *term2)

```



```

03049 {
03050     GETIDENTITY
03051     UWORD *c;
03052     WORD n1,n2,n3,*a;
03053     GETCOEF(term1,n1);
03054     GETCOEF(term2,n2);
03055     if ( term1[1] == 0 && n1 == 1 ) {
03056         if ( term2[1] == 0 && n2 == 1 ) return(0);
03057         if ( n2 < 0 ) return(1);
03058         return(-1);
03059     }
03060     else if ( term2[1] == 0 && n2 == 1 ) {
03061         if ( n1 < 0 ) return(-1);
03062         return(1);
03063     }
03064     if ( n1 > 0 ) {
03065         if ( n2 < 0 ) return(1);
03066     }
03067     else {
03068         if ( n2 > 0 ) return(-1);
03069         a = term1; term1 = term2; term2 = a;
03070         n3 = -n1; n1 = -n2; n2 = n3;
03071     }
03072     if ( term1[1] == 1 && term2[1] == 1 && n1 == 1 && n2 == 1 ) {
03073         if ( (UWORD)*term1 > (UWORD)*term2 ) return(1);
03074         else if ( (UWORD)*term1 < (UWORD)*term2 ) return(-1);
03075         else return(0);
03076     }
03077
03078     /*
03079     The next call should get dedicated code, as AddRat does more than
03080     strictly needed. Also more attention should be given to overflow.
03081     */
03082     c = NumberMalloc("CompCoef");
03083     if ( AddRat(BHEAD (UWORD *)term1,n1,(UWORD *)term2,-n2,c,&n3) ) {
03084         MLOCK(ErrorMessageLock);
03085         MesCall("CompCoef");
03086         MUNLOCK(ErrorMessageLock);
03087         NumberFree(c,"CompCoef");
03088         SETERROR(-1)
03089     }
03090     NumberFree(c,"CompCoef");
03091     return(n3);
03092 }
03093
03094 /*
03095     #] CompCoef :
03096     #[ Modulus :      WORD Modulus(term)
03097
03098     Routine takes the coefficient of term modulus b. The answer
03099     is in term again and the length of term is adjusted.
03100
03101     */
03102
03103 int Modulus(WORD *term)
03104 {
03105     WORD *t;
03106     WORD n1;
03107     t = term;
03108     GETCOEF(t,n1);
03109     if ( TakeModulus((UWORD *)t,&n1,AC.cmod,AC.ncmod,UNPACK) ) {
03110         MLOCK(ErrorMessageLock);
03111         MesCall("Modulus");
03112         MUNLOCK(ErrorMessageLock);
03113         SETERROR(-1)
03114     }
03115     if ( !n1 ) {
03116         *term = 0;
03117         return(0);
03118     }
03119     else if ( n1 > 0 ) {
03120         n1 *= 2;
03121         t += n1;          /* Note that n1 >= 0 */
03122         n1++;
03123     }
03124     else if ( n1 < 0 ) {
03125         n1 *= 2;
03126         t += -n1;
03127         n1--;
03128     }
03129     *t++ = n1;
03130     *term = WORDDIF(t,term);
03131     return(0);
03132 }
03133
03134 /*
03135     #] Modulus :

```

```

03136      # [ TakeModulus :      WORD TakeModulus(a,na,cmodvec,nmod,par)
03137
03138      Routine gets the rational number in a with reduced length na.
03139      It is called when AC.nmod != 0 and the number in AC.cmod is the
03140      number wrt which we need the modulus.
03141      The result is returned in a and na again.
03142
03143      If par == NOUNPACK we only do a single number, not a fraction.
03144      In addition we don't do fancy. We want a positive number and
03145      the input was supposed to be positive.
03146      We don't pack the result. The calling routine is responsible for that.
03147      This may not be a good idea. To be checked.
03148  */
03149
03150  int TakeModulus(UWORD *a, WORD *na, UWORD *cmodvec, WORD nmod, WORD par)
03151  {
03152      GETIDENTITY
03153      UWORD *c, *d, *e, *f, *g, *h;
03154      UWORD *x4,*x2;
03155      UWORD *x3,*x1,*x5,*x6,*x7,*x8;
03156      WORD y3,y1,y5,y6;
03157      WORD n1, i, y2, y4;
03158      WORD nh, tdenom, tnum, nmod;
03159      LONG x;
03160      if ( nmod == 0 ) return(0);          /* No modulus operation */
03161      nmod = ABS(nmod);
03162      n1 = *na;
03163      if ( ( par & UNPACK ) != 0 ) UnPack(a,n1,&tdenom,&tnum);
03164      else { tnum = n1; }
03165  /*
03166      We fish out the special case that the coefficient is short as well.
03167      There is no need to make lots of calls etc
03168  */
03169      if ( ( ( par & UNPACK ) == 0 ) && nmod == 1 && ( n1 == 1 || n1 == -1 ) ) {
03170          goto simplecase;
03171      }
03172      else if ( nmod == 1 && ( n1 == 1 || n1 == -1 ) ) {
03173          if ( a[1] != 1 ) {
03174              a[1] = a[1] % cmodvec[0];
03175              if ( a[1] == 0 ) {
03176                  MesPrint("Division by zero in short modulus arithmetic");
03177                  return(-1);
03178              }
03179              y1 = 0;
03180              if ( ( AC.modinverses != 0 ) && ( ( par & NOINVERSES ) == 0 ) ) {
03181                  y1 = AC.modinverses[a[1]];
03182              }
03183              else {
03184                  GetModInverses(a[1],cmodvec[0],&y1,&y2);
03185              }
03186              x = a[0];
03187              a[0] = (x*y1) % cmodvec[0];
03188              a[1] = 1;
03189          }
03190          else {
03191  simplecase:
03192              a[0] = a[0] % cmodvec[0];
03193          }
03194          if ( a[0] == 0 ) { *na = 0; return(0); }
03195          if ( ( AC.modmode & POSNEG ) != 0 ) {
03196              if ( a[0] > (UWORD)(cmodvec[0]/2) ) {
03197                  a[0] = cmodvec[0] - a[0];
03198                  *na = -*na;
03199              }
03200          }
03201          else if ( *na < 0 ) {
03202              *na = 1; a[0] = cmodvec[0] - a[0];
03203          }
03204          return(0);
03205      }
03206      c = NumberMalloc("TakeModulus"); d = NumberMalloc("TakeModulus"); e = NumberMalloc("TakeModulus");
03207      f = NumberMalloc("TakeModulus"); g = NumberMalloc("TakeModulus"); h = NumberMalloc("TakeModulus");
03208      n1 = ABS(n1);
03209      if ( DivLong(a,tnum,(UWORD *)cmodvec,nmod,
03210                  c,&nh,a,&tnum) ) goto ModErr;
03211      if ( tnum == 0 ) { *na = 0; goto normalreturn; }
03212      if ( ( par & UNPACK ) == 0 ) {
03213          if ( ( AC.modmode & POSNEG ) != 0 ) {
03214              NormalModulus(a,&tnum);
03215          }
03216          else if ( tnum < 0 ) {
03217              SubPLon((UWORD *)cmodvec,nmod,a,-tnum,a,&tnum);
03218          }
03219          *na = tnum;
03220          goto normalreturn;
03221      }
03222      if ( tdenom == 1 && a[n1] == 1 ) {

```

```

03223         if ( ( AC.modmode & POSNEG ) != 0 ) {
03224             NormalModulus(a,&tnumer);
03225         }
03226     else if ( tnumer < 0 ) {
03227         SubPLon((UWORD *)cmodvec,nmod,a,-tnumer,a,&tnumer);
03228     }
03229     *na = tnumer;
03230     i = ABS(tnumer);
03231     a += i;
03232     *a++ = 1;
03233     while ( --i > 0 ) *a++ = 0;
03234     goto normalreturn;
03235 }
03236 if ( DivLong(a+n1,tdenom,(UWORD *)cmodvec,nmod,c,&nh,a+n1,&tdenom) ) goto ModErr;
03237 if ( !tdenom ) {
03238     MLOCK(ErrorMessageLock);
03239     MesPrint("Division by zero in modulus arithmetic");
03240     if ( AP.DebugFlag ) {
03241         AO.OutSkip = 3;
03242         FiniLine();
03243         i = *na;
03244         if ( i < 0 ) i = -i;
03245         while ( --i >= 0 ) { TalToLine((UWORD)(*a++)); TokenToLine((UBYTE *)" "); }
03246         i = *na;
03247         if ( i < 0 ) i = -i;
03248         while ( --i >= 0 ) { TalToLine((UWORD)(*a++)); TokenToLine((UBYTE *)" "); }
03249         TalToLine((UWORD)(*na));
03250         AO.OutSkip = 0;
03251         FiniLine();
03252     }
03253     MUNLOCK(ErrorMessageLock);
03254     NumberFree(c,"TakeModulus"); NumberFree(d,"TakeModulus"); NumberFree(e,"TakeModulus");
03255     NumberFree(f,"TakeModulus"); NumberFree(g,"TakeModulus"); NumberFree(h,"TakeModulus");
03256     return(-1);
03257 }
03258 if ( ( AC.modinverses != 0 ) && ( ( par & NOINVERSES ) == 0 )
03259 && ( tdenom == 1 || tdenom == -1 ) ) {
03260     *d = AC.modinverses[a[n1]]; y1 = 1; y2 = tdenom;
03261     if ( MulLong(a,tnumer,d,y1,c,&y3) ) goto ModErr;
03262     if ( DivLong(c,y3,(UWORD *)cmodvec,nmod,d,&y5,a,&tdenom) ) goto ModErr;
03263     if ( y2 < 0 ) tdenom = -tdenom;
03264 }
03265 else {
03266     x2 = (UWORD *)cmodvec; x1 = c; i = nmod; while ( --i >= 0 ) *x1++ = *x2++;
03267     x1 = c; x2 = a+n1; x3 = d; x4 = e; x5 = f; x6 = g;
03268     y1 = nmod; y2 = tdenom; y4 = 0; y5 = 1; *x5 = 1;
03269     for(;;) {
03270         if ( DivLong(x1,y1,x2,y2,h,&nh,x3,&y3) ) goto ModErr;
03271         if ( MulLong(x5,y5,h,nh,x6,&y6) ) goto ModErr;
03272         if ( AddLong(x4,y4,x6,-y6,x6,&y6) ) goto ModErr;
03273         if ( !y3 ) {
03274             if ( y2 != 1 || *x2 != 1 ) {
03275                 MLOCK(ErrorMessageLock);
03276                 MesPrint("Inverse in modulus arithmetic doesn't exist");
03277                 MesPrint("Denominator and modulus are not relative prime");
03278                 MUNLOCK(ErrorMessageLock);
03279                 goto ModErr;
03280             }
03281             break;
03282         }
03283         x7 = x1; x1 = x2; y1 = y2; x2 = x3; y2 = y3; x3 = x7;
03284         x8 = x4; x4 = x5; y4 = y5; x5 = x6; y5 = y6; x6 = x8;
03285     }
03286     if ( y5 < 0 && AddLong((UWORD *)cmodvec,nmod,x5,y5,x5,&y5) ) goto ModErr;
03287     if ( MulLong(a,tnumer,x5,y5,c,&y3) ) goto ModErr;
03288     if ( DivLong(c,y3,(UWORD *)cmodvec,nmod,d,&y5,a,&tdenom) ) goto ModErr;
03289 }
03290 if ( !tdenom ) { *na = 0; goto normalreturn; }
03291 if ( ( ( AC.modmode & POSNEG ) != 0 ) && ( ( par & FROMFUNCTION ) == 0 ) ) {
03292     NormalModulus(a,&tdenom);
03293 }
03294 else if ( tdenom < 0 ) {
03295     SubPLon((UWORD *)cmodvec,nmod,a,-tdenom,a,&tdenom);
03296 }
03297 *na = tdenom;
03298 i = ABS(tdenom);
03299 a += i;
03300 *a++ = 1;
03301 while ( --i > 0 ) *a++ = 0;
03302 normalreturn:
03303     NumberFree(c,"TakeModulus"); NumberFree(d,"TakeModulus"); NumberFree(e,"TakeModulus");
03304     NumberFree(f,"TakeModulus"); NumberFree(g,"TakeModulus"); NumberFree(h,"TakeModulus");
03305     return(0);
03306 ModErr:
03307     MLOCK(ErrorMessageLock);
03308     MesCall("TakeModulus");
03309     MUNLOCK(ErrorMessageLock);

```

```

03310     NumberFree(c,"TakeModulus"); NumberFree(d,"TakeModulus"); NumberFree(e,"TakeModulus");
03311     NumberFree(f,"TakeModulus"); NumberFree(g,"TakeModulus"); NumberFree(h,"TakeModulus");
03312     SETERROR(-1)
03313 }
03314
03315 /*
03316     #] TakeModulus :
03317     #[ TakeNormalModulus : WORD TakeNormalModulus(a,na,par)
03318
03319     added by Jan [01-09-2010]
03320 */
03321
03322 int TakeNormalModulus (UWORD *a, WORD *na, UWORD *c, WORD nc, WORD par)
03323 {
03324     WORD n;
03325     WORD nhalfc;
03326     UWORD *halfc;
03327
03328     GETIDENTITY;
03329
03330     /* determine c/2 by right shifting */
03331     halfc = NumberMalloc("TakeNormalModulus");
03332     nhalfc=nc;
03333     WCOPY(halfc,c,nc);
03334
03335     for (n=0; n<nhalfc; n++) {
03336         halfc[n] /= 2;
03337         if (n+1<nc) halfc[n] |= c[n+1] << (BITSINWORD-1);
03338     }
03339
03340     if (halfc[nhalfc-1]==0)
03341         nhalfc--;
03342
03343     /* takes care of the number never expanding, e.g., -1(mod 100) -> 99 -> -1 */
03344     if (BigLong(a,ABS(*na),halfc,nhalfc) > 0) {
03345
03346         TakeModulus(a,na,c,nc,par);
03347
03348         n = ABS(*na);
03349         if (BigLong(a,n,halfc,nhalfc) > 0) {
03350             SubPLon(c,nc,a,n,a,&n);
03351             *na = (*na > 0 ? -n : n);
03352         }
03353     }
03354
03355     NumberFree(halfc,"TakeNormalModulus");
03356     return(0);
03357 }
03358
03359 /*
03360     #] TakeNormalModulus :
03361     #[ MakeModTable : WORD MakeModTable()
03362 */
03363
03364 int MakeModTable(void)
03365 {
03366     LONG size, i, j, n;
03367     n = ABS(AC.ncmod);
03368     if ( AC.modpowers ) {
03369         M_free(AC.modpowers,"AC.modpowers");
03370         AC.modpowers = NULL;
03371     }
03372     if ( n > 2 ) {
03373         MLOCK(ErrorMessageLock);
03374         MesPrint("%No memory for modulus generator power table");
03375         MUNLOCK(ErrorMessageLock);
03376         Terminate(-1);
03377     }
03378     if ( n == 0 ) return(0);
03379     size = (LONG) (*AC.cmod);
03380     if ( n == 2 ) size += ((LONG)AC.cmod[1]<<BITSINWORD);
03381     AC.modpowers = (UWORD *)Malloca(size*n*sizeof(UWORD),"table for powers of modulus");
03382     if ( n == 1 ) {
03383         j = 1;
03384         for ( i = 0; i < size; i++ ) AC.modpowers[i] = 0;
03385         for ( i = 0; i < size; i++ ) {
03386             AC.modpowers[j] = (WORD)i;
03387             j *= *AC.powmod;
03388             j %= *AC.cmod;
03389         }
03390         for ( i = 2; i < size; i++ ) {
03391             if ( AC.modpowers[i] == 0 ) {
03392                 MLOCK(ErrorMessageLock);
03393                 MesPrint("%improper generator for this modulus");
03394                 MUNLOCK(ErrorMessageLock);
03395                 M_free(AC.modpowers,"AC.modpowers");
03396                 return(-1);

```

```

03397     }
03398 }
03399     AC.modpowers[1] = 0;
03400 }
03401 else {
03402     GETIDENTITY
03403     WORD nSkrat, n2;
03404     UWORD *MMskrat7 = NumberMalloc("MakeModTable"), *MMskratC = NumberMalloc("MakeModTable");
03405     *MMskratC = 1;
03406     nSkrat = 1;
03407     j = size * 2;
03408     for ( i = 0; i < j; i+=2 ) { AC.modpowers[i] = 0; AC.modpowers[i+1] = 0; }
03409     for ( i = 0; i < size; i++ ) {
03410         j = *MMskratC + (((LONG)MMskratC[1])«BITSINWORD);
03411         j *= 2;
03412         AC.modpowers[j] = (WORD)(i & WORDMASK);
03413         AC.modpowers[j+1] = (WORD)(i » BITSINWORD);
03414         Mullong((UWORD *)MMskratC,nSkrat,(UWORD *)AC.powmod,
03415             AC.npowmod,(UWORD *)MMskrat7,&n2);
03416         TakeModulus(MMskrat7,&n2,AC.cmod,AC.ncmod,NOUNPACK);
03417         *MMskratC = *MMskrat7; MMskratC[1] = MMskrat7[1]; nSkrat = n2;
03418     }
03419     NumberFree(MMskrat7,"MakeModTable"); NumberFree(MMskratC,"MakeModTable");
03420     j = size * 2;
03421     for ( i = 4; i < j; i+=2 ) {
03422         if ( AC.modpowers[i] == 0 && AC.modpowers[i+1] == 0 ) {
03423             MLOCK(ErrorMessageLock);
03424             MesPrint("&improper generator for this modulus");
03425             MUNLOCK(ErrorMessageLock);
03426             M_free(AC.modpowers,"AC.modpowers");
03427             return(-1);
03428         }
03429     }
03430     AC.modpowers[2] = AC.modpowers[3] = 0;
03431 }
03432 return(0);
03433 }
03434
03435 /*
03436     #] MakeModTable :
03437     #] RekenTerms :
03438     #[ Functions :
03439         #[ Factorial :      WORD Factorial(n,a,na)
03440
03441     Starts with only the value of fac_(0).
03442     Builds up what is needed and remembers it for the next time.
03443
03444     We have:
03445         AT.nfac: the number of the highest stored factorial
03446         AT.pfac: the array of locations in the array of stored factorials
03447         AT.factorials: the array with stored factorials
03448 */
03449
03450 int Factorial(PHEAD WORD n, UWORD *a, WORD *na)
03451 {
03452     GETBIDENTITY
03453     UWORD *b, *c;
03454     WORD nc;
03455     int i, j;
03456     LONG ii;
03457     if ( n > AT.nfac ) {
03458         if ( AT.factorials == 0 ) {
03459             AT.nfac = 0; AT.mfac = 50; AT.sfact = 400;
03460             AT.pfac = (LONG *)Mallocl1((AT.mfac+2)*sizeof(LONG),"factorials");
03461             AT.factorials = (UWORD *)Mallocl1(AT.sfact*sizeof(UWORD),"factorials");
03462             AT.factorials[0] = 1; AT.pfac[0] = 0; AT.pfac[1] = 1;
03463         }
03464         b = a;
03465         c = AT.factorials+AT.pfac[AT.nfac];
03466         nc = i = AT.pfac[AT.nfac+1] - AT.pfac[AT.nfac];
03467         while ( --i >= 0 ) *b++ = *c++;
03468         for ( j = AT.nfac+1; j <= n; j++ ) {
03469             Product(a,&nc,j);
03470             if ( nc > AM.MaxTal ) {
03471                 MLOCK(ErrorMessageLock);
03472                 MesPrint("Overflow in factorial. MaxTal = %d",AM.MaxTal);
03473                 MesPrint("Increase MaxTerm in %s",setupfilename);
03474                 MUNLOCK(ErrorMessageLock);
03475                 return(-1);
03476             }
03477             if ( j > AT.mfac ) { /* double the pfac buffer */
03478                 LONG *p;
03479                 p = (LONG *)Mallocl1((AT.mfac*2+2)*sizeof(LONG),"factorials");
03480                 i = AT.mfac;
03481                 for ( i = AT.mfac+1; i >= 0; i-- ) p[i] = AT.pfac[i];
03482                 M_free(AT.pfac,"factorial offsets"); AT.pfac = p; AT.mfac *= 2;
03483             }

```

```

03484         if ( AT.pfac[j] + nc >= AT.sfact ) { /* double the factorials buffer */
03485             UWORD *f;
03486             f = (UWORD *)Malloc1(AT.sfact*2*sizeof(UWORD),"factorials");
03487             ii = AT.sfact;
03488             c = AT.factorials; b = f;
03489             while ( --ii >= 0 ) *b++ = *c++;
03490             M_free(AT.factorials,"factorials");
03491             AT.factorials = f;
03492             AT.sfact *= 2;
03493         }
03494         b = a; c = AT.factorials + AT.pfac[j]; i = nc;
03495         while ( --i >= 0 ) *c++ = *b++;
03496         AT.pfac[j+1] = AT.pfac[j] + nc;
03497     }
03498     *na = nc;
03499     AT.nfac = n;
03500 }
03501 else if ( n == 0 ) {
03502     *a = 1; *na = 1;
03503 }
03504 else {
03505     *na = i = AT.pfac[n+1] - AT.pfac[n];
03506     b = AT.factorials + AT.pfac[n];
03507     while ( --i >= 0 ) *a++ = *b++;
03508 }
03509 return(0);
03510 }
03511
03512 /*
03513     #] Factorial :
03514     #[ Bernoulli :      WORD Bernoulli(n,a,na)
03515
03516     Starts with only the value of bernoulli_(0).
03517     Builds up what is needed and remembers it for the next time.
03518     b_0 = 1
03519     (n+1)*b_n = -b_{n-1}-sum_(i,1,n-1,b_i*b_{n-i})
03520     The n-1 plays only a role for b_2.
03521     We have hard coded b_0,b_1,b_2 and b_odd. After that:
03522     (2n+1)*b_{2n} = -sum_(i,1,n-1,b_{2i}*b_{2n-2i})
03523
03524     We have:
03525         AT.nBer: the number of the highest stored Bernoulli number
03526         AT.pBer: the array of locations in the array of stored Bernoulli numbers
03527         AT.bernoullis: the array with stored Bernoulli numbers
03528 */
03529
03530 int Bernoulli(WORD n, UWORD *a, WORD *na)
03531 {
03532     GETIDENTITY
03533     UWORD *b, *c, *scrib, *ntop, *ntopl;
03534     WORD i, i1, i2, nhalf, nqua, nscrib, nntop, nntopl, *oldworkpointer;
03535     UWORD twee = 2, twonplus1;
03536     int j;
03537     LONG ii;
03538     if ( n <= 1 ) {
03539         if ( n == 0 ) { a[0] = a[1] = 1; *na = 3; }
03540         else if ( n == 1 ) { a[0] = 1; a[1] = 2; *na = 3; }
03541         return(0);
03542     }
03543     if ( ( n & 1 ) != 0 ) { a[0] = a[1] = 0; *na = 0; return(0); }
03544     nhalf = n/2;
03545     if ( nhalf > AT.nBer ) {
03546         oldworkpointer = AT.WorkPointer;
03547         if ( AT.bernoullis == 0 ) {
03548             AT.nBer = 1; AT.mBer = 50; AT.sBer = 400;
03549             AT.pBer = (LONG *)Malloc1((AT.mBer+2)*sizeof(LONG),"bernoullis");
03550             AT.bernoullis = (UWORD *)Malloc1(AT.sBer*sizeof(UWORD),"bernoullis");
03551             AT.pBer[1] = 0; AT.pBer[2] = 3;
03552             AT.bernoullis[0] = 3; AT.bernoullis[1] = 1; AT.bernoullis[2] = 12;
03553             if ( nhalf == 1 ) {
03554                 a[0] = 1; a[1] = 12; *na = 3; return(0);
03555             }
03556         }
03557         while ( nhalf > AT.mBer ) {
03558             LONG *p;
03559             p = (LONG *)Malloc1((AT.mBer*2+1)*sizeof(LONG),"bernoullis");
03560             i = AT.mBer;
03561             for ( i = AT.mBer; i >= 0; i-- ) p[i] = AT.pBer[i];
03562             M_free(AT.pBer,"factorial pointers"); AT.pBer = p; AT.mBer *= 2;
03563         }
03564         for ( n = AT.nBer+1; n <= nhalf; n++ ) {
03565             scrib = (UWORD *) (AT.WorkPointer);
03566             nqua = n/2;
03567             if ( ( n & 1 ) == 1 ) {
03568                 nscrib = 0; ntop = scrib;
03569             }
03570             else {

```

```

03571         b = AT.bernoullis + AT.pBer[nqua];
03572         nscrib = *b++;
03573         i = (WORD)(REDLENG(nscrib));
03574         MulRat(BHEAD b,i,b,i,scrib,&nscrib);
03575         ntop = scrib + 2*nscrib;
03576         nqua--;
03577     }
03578     for ( j = 1; j <= nqua; j++ ) {
03579         b = AT.bernoullis + AT.pBer[j];
03580         c = AT.bernoullis + AT.pBer[n-j];
03581         i1 = (WORD)(*b); i2 = (WORD)(*c);
03582         i1 = REDLENG(i1);
03583         i2 = REDLENG(i2);
03584         MulRat(BHEAD b+1,i1,c+1,i2,ntop,&nntop);
03585         Mullt(BHEAD ntop,&nntop,&twee,1);
03586         if ( nscrib ) {
03587             i = (WORD)nntop; if ( i < 0 ) i = -i;
03588             ntop1 = ntop + 2*i;
03589             AddRat(BHEAD ntop,nntop,scrib,nscrib,ntop1,&nntop1);
03590         }
03591         else {
03592             ntop1 = ntop; nntop1 = nntop;
03593         }
03594         nscrib = i1 = (WORD)nntop1;
03595         if ( i1 < 0 ) i1 = - i1;
03596         i1 = 2*i1;
03597         for ( i = 0; i < i1; i++ ) scrib[i] = ntop1[i];
03598         ntop = scrib + i1;
03599     }
03600     twonplus1 = 2*n+1;
03601     Divvy(BHEAD scrib,&nscrib,&twonplus1,-1);
03602     i1 = INCLENG(nscrib);
03603     i2 = i1; if ( i2 < 0 ) i2 = -i2;
03604     i = (WORD)(AT.bernoullis[AT.pBer[n-1]]);
03605     if ( i < 0 ) i = -i;
03606     AT.pBer[n] = AT.pBer[n-1]+i;
03607     if ( AT.pBer[n] + i2 >= AT.sBer ) {
03608         UWORD *f;
03609         f = (UWORD *)Malloca(AT.sBer*2*sizeof(UWORD),"bernoullis");
03610         ii = AT.sBer;
03611         c = AT.bernoullis; b = f;
03612         while ( --ii >= 0 ) *b++ = *c++;
03613         M_free(AT.bernoullis,"bernoullis");
03614         AT.bernoullis = f;
03615         AT.sBer *= 2;
03616     }
03617     c = AT.bernoullis + AT.pBer[n]; b = scrib;
03618     *c++ = i1;
03619     for ( i = 1; i < i2; i++ ) *c++ = *b++;
03620 }
03621 AT.nBer = nhalf;
03622 AT.WorkPointer = oldworkpointer;
03623 }
03624 b = AT.bernoullis + AT.pBer[nhalf];
03625 *na = i = (WORD)(*b++);
03626 if ( i < 0 ) i = -i;
03627 i--;
03628 while ( --i >= 0 ) *a++ = *b++;
03629 return(0);
03630 }
03631
03632 /*
03633     #] Bernoulli :
03634     #[ NextPrime :
03635 */
03647 #if ( BITSINWORD == 32 )
03648
03649 void StartPrimeList(PHEAD0)
03650 {
03651     int i, j;
03652     AR.PrimeList[AR.numinprimelist++] = 3;
03653     for ( i = 5; i < 46340; i += 2 ) {
03654         for ( j = 0; j < AR.numinprimelist && AR.PrimeList[j]*AR.PrimeList[j] <= i; j++ ) {
03655             if ( i % AR.PrimeList[j] == 0 ) goto nexti;
03656         }
03657         AR.PrimeList[AR.numinprimelist++] = i;
03658     nexti:;
03659     }
03660     AR.notfirstprime = 1;
03661 }
03662 #endif
03663
03664 WORD NextPrime(PHEAD WORD num)
03665 {
03666     int i, j;
03667     WORD *newpl;

```

```

03669     LONG newsize, x;
03670 #if ( BITSINWORD == 32 )
03671     if ( AR.notfirstprime == 0 ) StartPrimeList(BHEAD0);
03672 #endif
03673     if ( num > AT.inprimelist ) {
03674         while ( AT.inprimelist < num ) {
03675             if ( num >= AT.sizeprimelist ) {
03676                 if ( AT.sizeprimelist == 0 ) newsize = 32;
03677                 else newsize = 2*AT.sizeprimelist;
03678                 while ( num >= newsize ) newsize = newsize*2;
03679                 newpl = (WORD *)Malloca1(newsize*sizeof(WORD),"NextPrime");
03680                 for ( i = 0; i < AT.sizeprimelist; i++ ) {
03681                     newpl[i] = AT.primelist[i];
03682                 }
03683                 if ( AT.sizeprimelist > 0 ) {
03684                     M_free(AT.primelist,"NextPrime");
03685                 }
03686                 AT.sizeprimelist = newsize;
03687                 AT.primelist = newpl;
03688             }
03689             if ( AT.inprimelist < 0 ) { i = MAXPOSITIVE; }
03690             else { i = AT.primelist[AT.inprimelist]; }
03691             while ( i > MAXPOWER ) {
03692                 i -= 2; x = i;
03693             }
03694             #if ( BITSINWORD == 32 )
03695                 for ( j = 0; j < AR.numinprimelist && AR.PrimeList[j]*(LONG)(AR.PrimeList[j]) <= x;
03696                     j++ ) {
03697                     if ( x % AR.PrimeList[j] == 0 ) goto nexti;
03698                 }
03699             #else
03700                 for ( j = 3; j*((LONG)j) <= x; j += 2 ) {
03701                     if ( x % j == 0 ) goto nexti;
03702                 }
03703             #endif
03704             AT.inprimelist++;
03705             AT.primelist[AT.inprimelist] = i;
03706             break;
03707         }
03708         nexti:;
03709     }
03710     if ( i < MAXPOWER ) {
03711         MLOCK(ErrorMessageLock);
03712         MesPrint("There are not enough short prime numbers for this calculation");
03713         MesPrint("Try to use a computer with a %d-bits architecture",
03714             (int)(BITSINWORD*4));
03715         MUNLOCK(ErrorMessageLock);
03716         Terminate(-1);
03717     }
03718     return(AT.primelist[num]);
03719 }
03720 /*
03721     #] NextPrime :
03722     #[ Moebius :
03723
03724     Returns the value of the Moebius function if n fits inside
03725     a WORD.
03726     The method we use is a bit like the sieve of Erathostenes.
03727 */
03728
03729 WORD Moebius(PHEAD WORD nn)
03730 {
03731     WORD i,n = nn, x;
03732     LONG newsize;
03733     SBYTE *newtable, mu;
03734     #if ( BITSINWORD == 32 )
03735         if ( AR.notfirstprime == 0 ) StartPrimeList(BHEAD0);
03736     #endif
03737     /*
03738         First we make sure that:
03739         a: the table is big enough.
03740         b: the number is not already in the table.
03741     */
03742     if ( nn >= AR.moebiustablesizesize ) {
03743         if ( AR.moebiustablesizesize <= 0 ) { newsize = (LONG)nn + 20; }
03744         else { newsize = (LONG)nn*2; }
03745         if ( newsize > MAXPOSITIVE ) newsize = MAXPOSITIVE;
03746         newtable = (SBYTE *)Malloca1(newsize*sizeof(SBYTE),"Moebius");
03747         for ( i = 0; i < AR.moebiustablesizesize; i++ ) newtable[i] = AR.moebiustable[i];
03748         for ( ; i < newsize; i++ ) newtable[i] = 2;
03749         if ( AR.moebiustablesizesize > 0 ) M_free(AR.moebiustable,"Moebius");
03750         AR.moebiustable = newtable;
03751         AR.moebiustablesizesize = newsize;
03752     }
03753     /* NOTE: nn == MAXPOSITIVE never fits in moebiustable. */
03754     if ( nn != MAXPOSITIVE && AR.moebiustable[nn] != 2 ) return((WORD)AR.moebiustable[nn]);

```



```

03755     mu = 1;
03756     if ( n == 1 ) goto putvalue;
03757     if ( n % 2 == 0 ) {
03758         n /= 2;
03759         if ( n % 2 == 0 ) { mu = 0; goto putvalue; }
03760         if ( AR.moebiustable[n] != 2 ) { mu = -AR.moebiustable[n]; goto putvalue; }
03761         mu = -mu;
03762         if ( n == 1 ) goto putvalue;
03763     }
03764     #if ( BITSINWORD == 32 )
03765     for ( i = 0; i < AR.numinprimelist; i++ ) {
03766         x = AR.PrimeList[i];
03767     #else
03768     for ( x = 3; x < MAXPOSITIVE; x += 2 ) {
03769     #endif
03770         if ( n % x == 0 ) {
03771             n /= x;
03772             if ( n % x == 0 ) { mu = 0; goto putvalue; }
03773             if ( AR.moebiustable[n] != 2 ) { mu = -AR.moebiustable[n]; goto putvalue; }
03774             mu = -mu;
03775             if ( n == 1 ) goto putvalue;
03776         }
03777         if ( n < x*x ) break; /* notice that x*x always fits inside a WORD */
03778     }
03779     mu = -mu;
03780     putvalue:
03781     if ( nn != MAXPOSITIVE ) AR.moebiustable[nn] = mu;
03782     return ( (WORD)mu );
03783 }
03784
03785 /*
03786     #] Moebius :
03787     #[ wranf :
03788
03789     A random number generator that generates random WORDs with a very
03790     long sequence. It is based on the Knuth generator.
03791
03792     We take some care that each thread can run its own, but each
03793     uses its own startup. Hence the seed includes the identity of
03794     the thread.
03795
03796     For NPAIR1, NPAIR2 we can use any pair from the table on page 28.
03797     Candidates are 24,55 (the example on the pages 171,172)
03798     or (33,97) or (38,89)
03799     These values are defined in fsizes.h and used in startup.c and threads.c
03800 */
03801
03802 #define WARMUP 6
03803
03804 static void wranfnew(PHEAD0)
03805 {
03806     int i;
03807     LONG j;
03808     for ( i = 0; i < AR.wranfnpair1; i++ ) {
03809         j = AR.wranfia[i] - AR.wranfia[i+(AR.wranfnpair2-AR.wranfnpair1)];
03810         if ( j < 0 ) j += (LONG)1 « (2*BITSINWORD-2);
03811         AR.wranfia[i] = j;
03812     }
03813     for ( i = AR.wranfnpair1; i < AR.wranfnpair2; i++ ) {
03814         j = AR.wranfia[i] - AR.wranfia[i-AR.wranfnpair1];
03815         if ( j < 0 ) j += (LONG)1 « (2*BITSINWORD-2);
03816         AR.wranfia[i] = j;
03817     }
03818 }
03819
03820 void iniwranf(PHEAD0)
03821 {
03822     int imax = AR.wranfnpair2-1;
03823     ULONG i, ii, seed = AR.wranfseed;
03824     LONG j, k;
03825     ULONG offset = 12345;
03826     #ifdef PARALLELCODE
03827     int id;
03828     #if defined(WITHPTHREADS)
03829     id = AT.identity;
03830     #elif defined(WITHMPI)
03831     id = PF.me;
03832     #endif
03833     seed += id;
03834     i = id + 1;
03835     if ( i > 1 ) {
03836         ULONG pow, accu;
03837         pow = offset; accu = 1;
03838         while ( i ) {
03839             if ( ( i & 1 ) != 0 ) accu *= pow;
03840             i /= 2; pow = pow*pow;
03841         }

```

```

03842         offset = accu;
03843     }
03844 #endif
03845     if ( seed < ((LONG)1«(BITSINWORD-1)) ) {
03846         j = ( (seed+31459L) « (BITSINWORD-2))+offset;
03847     }
03848     else if ( seed < ((LONG)1«(BITSINWORD+10-1)) ) {
03849         j = ( (seed+31459L) « (BITSINWORD-10-2))+offset;
03850     }
03851     else {
03852         j = ( (seed+31459L) « 1)+offset;
03853     }
03854     if ( ( seed & 1 ) == 1 ) seed++;
03855     j += seed;
03856     AR.wranfia[imax] = j;
03857     k = 1;
03858     for ( i = 0; i <= (ULONG)(imax); i++ ) {
03859         ii = (AR.wranfnpair1*i)%AR.wranfnpair2;
03860         AR.wranfia[ii] = k;
03861         k = ULONGToLong((ULONG)j - (ULONG)k);
03862         if ( k < 0 ) k += (LONG)1 « (2*BITSINWORD-2);
03863         j = AR.wranfia[ii];
03864     }
03865     for ( i = 0; i < WARMUP; i++ ) wranfnew(BHEAD0);
03866     AR.wranfcall = 0;
03867 }
03868
03869 UWORD wranf(PHEAD0)
03870 {
03871     UWORD wval;
03872     if ( AR.wranfia == 0 ) {
03873         AR.wranfia = (ULONG *)Malloc1(AR.wranfnpair2*sizeof(ULONG),"wranf");
03874         iniwranf(BHEAD0);
03875     }
03876     if ( AR.wranfcall >= AR.wranfnpair2 ) {
03877         wranfnew(BHEAD0);
03878         AR.wranfcall = 0;
03879     }
03880     wval = (UWORD) (AR.wranfia[AR.wranfcall++])«(BITSINWORD-1));
03881     return(wval);
03882 }
03883
03884 /*
03885  Returns a random UWORD in the range (0,...,imax-1)
03886 */
03887
03888 UWORD iranf(PHEAD UWORD imax)
03889 {
03890     UWORD i;
03891     ULONG x, xmax;
03892     if (imax < 2) return 0;
03893     x = (LONG)1 « BITSINWORD;
03894     xmax = x - x%imax;
03895     while ( ( i = wranf(BHEAD0) ) >= xmax ) {}
03896     return(i%imax);
03897 }
03898
03899 /*
03900     #] wranf :
03901     #[ PreRandom :
03902
03903     The random number generator of the preprocessor.
03904     This one is completely different from the execution time generator
03905     random_(number). In the preprocessor we generate a floating point
03906     number in a string according to a distribution.
03907     Currently allowed are:
03908         RANDOM_(log,min,max)
03909         RANDOM_(lin,min,max)
03910     The return value is a string with the floating point number.
03911 */
03912
03913 UBYTE *PreRandom(UBYTE *s)
03914 {
03915     GETIDENTITY
03916     UBYTE *mode,*mins = 0,*maxs = 0, *outval;
03917     float num;
03918     double minval, maxval, value = 0;
03919     int linlog = -1;
03920     mode = s;
03921     while ( FG.cTable[*s] <= 1 ) s++;
03922     if ( *s == ',' ) { *s = 0; s++; }
03923     mins = s;
03924     while ( *s && *s != ',' ) s++;
03925     if ( *s == ',' ) { *s = 0; s++; }
03926     maxs = s;
03927     while ( *s && *s != ',' ) s++;
03928     if ( *s || *maxs == 0 || *mins == 0 ) {

```

```

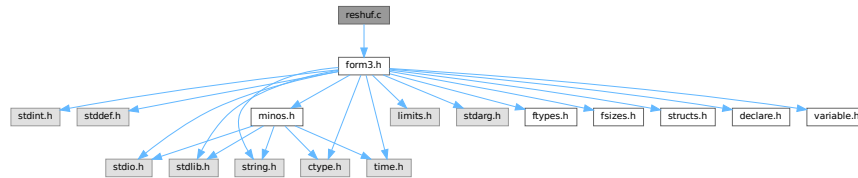
03929     MesPrint("@Illegal arguments in macro RANDOM_");
03930     Terminate(-1);
03931 }
03932 if ( StrICmp(mode, (UBYTE *)"lin") == 0 ) {
03933     linlog = 0;
03934 }
03935 else if ( StrICmp(mode, (UBYTE *)"log") == 0 ) {
03936     linlog = 1;
03937 }
03938 else {
03939     MesPrint("@Illegal mode argument in macro RANDOM_");
03940     Terminate(-1);
03941 }
03942
03943 sscanf((char *)mins, "%f", &num); minval = num;
03944 sscanf((char *)maxs, "%f", &num); maxval = num;
03945
03946 /*
03947  * Note on ParFORM: we should use the same random number on all the
03948  * processes in the compilation phase. The random number is generated
03949  * on the master and broadcast to the other processes.
03950  */
03951 {
03952     UWORD x;
03953     double xx;
03954 #ifdef WITHMPI
03955     x = 0;
03956     if ( PF.me == MASTER ) {
03957         x = wranf(BHEAD0);
03958     }
03959     x = (UWORD)PF_BroadcastNumber((LONG)x);
03960 #else
03961     x = wranf(BHEAD0);
03962 #endif
03963     xx = x/pow(2.0, (double) (BITSINWORD-1));
03964     if ( linlog == 0 ) {
03965         value = minval + (maxval-minval)*xx;
03966     }
03967     else if ( linlog == 1 ) {
03968         value = minval * pow(maxval/minval, xx);
03969     }
03970 }
03971
03972 outval = (UBYTE *)Malloca(64, "PreRandom");
03973 if ( ABS(value) < 0.00001 || ABS(value) > 1000000. ) {
03974     snprintf((char *)outval, 64, "%e", value);
03975 }
03976 else if ( ABS(value) < 0.0001 ) { snprintf((char *)outval, 64, "%10f", value); }
03977 else if ( ABS(value) < 0.001 ) { snprintf((char *)outval, 64, "%9f", value); }
03978 else if ( ABS(value) < 0.01 ) { snprintf((char *)outval, 64, "%8f", value); }
03979 else if ( ABS(value) < 0.1 ) { snprintf((char *)outval, 64, "%7f", value); }
03980 else if ( ABS(value) < 1. ) { snprintf((char *)outval, 64, "%6f", value); }
03981 else if ( ABS(value) < 10. ) { snprintf((char *)outval, 64, "%5f", value); }
03982 else if ( ABS(value) < 100. ) { snprintf((char *)outval, 64, "%4f", value); }
03983 else if ( ABS(value) < 1000. ) { snprintf((char *)outval, 64, "%3f", value); }
03984 else if ( ABS(value) < 10000. ) { snprintf((char *)outval, 64, "%2f", value); }
03985 else { snprintf((char *)outval, 64, "%1f", value); }
03986 return(outval);
03987 }
03988
03989 /*
03990     #] PreRandom :
03991     #] Functions :
03992 */

```

4.121 reshuf.c File Reference

```
#include "form3.h"
```

Include dependency graph for reshuf.c:



Functions

- WORD [ReNumber](#) (PHEAD WORD *term)
- void [FunLevel](#) (PHEAD WORD *term)
- WORD [DetCurDum](#) (PHEAD WORD *t)
- int [FullRenumber](#) (PHEAD WORD *term, WORD par)
- void [MoveDummies](#) (PHEAD WORD *term, WORD shift)
- void [AdjustRenumScratch](#) (PHEAD0)
- WORD [CountDo](#) (WORD *term, WORD *instruct)
- WORD [CountFun](#) (WORD *term, WORD *countfun)
- WORD [DimensionSubterm](#) (WORD *subterm)
- WORD [DimensionTerm](#) (WORD *term)
- WORD [DimensionExpression](#) (PHEAD WORD *expr)
- int [MultDo](#) (PHEAD WORD *term, WORD *pattern)
- int [TryDo](#) (PHEAD WORD *term, WORD *pattern, WORD level)
- int [DoDistrib](#) (PHEAD WORD *term, WORD level)
- int [EqualArg](#) (WORD *parms, WORD num1, WORD num2)
- int [DoDelta3](#) (PHEAD WORD *term, WORD level)
- int [TestPartitions](#) (WORD *tfun, [PARTI](#) *parti)
- int [DoPartitions](#) (PHEAD WORD *term, WORD level)
- int [DoPermutations](#) (PHEAD WORD *term, WORD level)
- int [DoShuffle](#) (WORD *term, WORD level, WORD fun, WORD option)
- int [Shuffle](#) (WORD *from1, WORD *from2, WORD *to)
- int [FinishShuffle](#) (WORD *fini)
- int [DoStuff](#) (WORD *term, WORD level, WORD fun, WORD option)
- int [Stuff](#) (WORD *from1, WORD *from2, WORD *to)
- int [FinishStuff](#) (WORD *fini)
- WORD * [StuffRootAdd](#) (WORD *t1, WORD *t2, WORD *to)

4.121.1 Detailed Description

Mixed routines: Routines for relabelling dummy indices. The multiply command The distrib_ function The tryreplace statement

Definition in file [reshuf.c](#).

4.121.2 Macro Definition Documentation

4.121.2.1 NEWCODE

```
#define NEWCODE
```

Definition at line 35 of file [reshuf.c](#).

4.121.3 Function Documentation

4.121.3.1 ReNumber()

```
WORD ReNumber (
    PHEAD WORD * term )
```

Definition at line 80 of file [reshuf.c](#).

4.121.3.2 FunLevel()

```
void FunLevel (
    PHEAD WORD * term )
```

Definition at line 130 of file [reshuf.c](#).

4.121.3.3 DetCurDum()

```
WORD DetCurDum (
    PHEAD WORD * t )
```

Definition at line 252 of file [reshuf.c](#).

4.121.3.4 FullRenumber()

```
int FullRenumber (
    PHEAD WORD * term,
    WORD par )
```

Definition at line 324 of file [reshuf.c](#).

4.121.3.5 MoveDummies()

```
void MoveDummies (
    PHEAD WORD * term,
    WORD shift )
```

Definition at line 436 of file [reshuf.c](#).

4.121.3.6 AdjustRenumScratch()

```
void AdjustRenumScratch (
    PHEAD0 )
```

Definition at line 506 of file [reshuf.c](#).

4.121.3.7 CountDo()

```
WORD CountDo (
    WORD * term,
    WORD * instruct )
```

Definition at line 552 of file [reshuf.c](#).

4.121.3.8 CountFun()

```
WORD CountFun (
    WORD * term,
    WORD * countfun )
```

Definition at line 706 of file [reshuf.c](#).

4.121.3.9 DimensionSubterm()

```
WORD DimensionSubterm (
    WORD * subterm )
```

Definition at line 867 of file [reshuf.c](#).

4.121.3.10 DimensionTerm()

```
WORD DimensionTerm (
    WORD * term )
```

Definition at line 968 of file [reshuf.c](#).

4.121.3.11 DimensionExpression()

```
WORD DimensionExpression (
    PHEAD WORD * expr )
```

Definition at line 1000 of file [reshuf.c](#).

4.121.3.12 MultDo()

```
int MultDo (  
    PHEAD WORD * term,  
    WORD * pattern )
```

Definition at line [1044](#) of file [reshuf.c](#).

4.121.3.13 TryDo()

```
int TryDo (  
    PHEAD WORD * term,  
    WORD * pattern,  
    WORD level )
```

Definition at line [1072](#) of file [reshuf.c](#).

4.121.3.14 DoDistrib()

```
int DoDistrib (  
    PHEAD WORD * term,  
    WORD level )
```

Definition at line [1119](#) of file [reshuf.c](#).

4.121.3.15 EqualArg()

```
int EqualArg (  
    WORD * parms,  
    WORD num1,  
    WORD num2 )
```

Definition at line [1379](#) of file [reshuf.c](#).

4.121.3.16 DoDelta3()

```
int DoDelta3 (  
    PHEAD WORD * term,  
    WORD level )
```

Definition at line [1405](#) of file [reshuf.c](#).

4.121.3.17 TestPartitions()

```
int TestPartitions (  
    WORD * tfun,  
    PARTI * parti )
```

Definition at line [1607](#) of file [reshuf.c](#).

4.121.3.18 DoPartitions()

```
int DoPartitions (
    PHEAD WORD * term,
    WORD level )
```

Definition at line 1722 of file [reshuf.c](#).

4.121.3.19 DoPermutations()

```
int DoPermutations (
    PHEAD WORD * term,
    WORD level )
```

Definition at line 2007 of file [reshuf.c](#).

4.121.3.20 DoShuffle()

```
int DoShuffle (
    WORD * term,
    WORD level,
    WORD fun,
    WORD option )
```

Definition at line 2095 of file [reshuf.c](#).

4.121.3.21 Shuffle()

```
int Shuffle (
    WORD * from1,
    WORD * from2,
    WORD * to )
```

Definition at line 2229 of file [reshuf.c](#).

4.121.3.22 FinishShuffle()

```
int FinishShuffle (
    WORD * fini )
```

Definition at line 2421 of file [reshuf.c](#).

4.121.3.23 DoStuffle()

```
int DoStuffle (
    WORD * term,
    WORD level,
    WORD fun,
    WORD option )
```

Definition at line 2487 of file [reshuf.c](#).

4.121.3.24 Stuffle()

```
int Stuffle (
    WORD * from1,
    WORD * from2,
    WORD * to )
```

Definition at line [2702](#) of file [reshuf.c](#).

4.121.3.25 FinishStuffle()

```
int FinishStuffle (
    WORD * fini )
```

Definition at line [2829](#) of file [reshuf.c](#).

4.121.3.26 StuffRootAdd()

```
WORD * StuffRootAdd (
    WORD * t1,
    WORD * t2,
    WORD * to )
```

Definition at line [2876](#) of file [reshuf.c](#).

4.122 reshuf.c

[Go to the documentation of this file.](#)

```
00001
00009 /* #[ License : */
00010 /*
00011 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00012 *   When using this file you are requested to refer to the publication
00013 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00014 *   This is considered a matter of courtesy as the development was paid
00015 *   for by FOM the Dutch physics granting agency and we would like to
00016 *   be able to track its scientific use to convince FOM of its value
00017 *   for the community.
00018 *
00019 *   This file is part of FORM.
00020 *
00021 *   FORM is free software: you can redistribute it and/or modify it under the
00022 *   terms of the GNU General Public License as published by the Free Software
00023 *   Foundation, either version 3 of the License, or (at your option) any later
00024 *   version.
00025 *
00026 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00027 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00028 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00029 *   details.
00030 *
00031 *   You should have received a copy of the GNU General Public License along
00032 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00033 */
00034 /* #[ License : */
00035 #define NEWCODE
00036 /*
00037 *   #[ Includes : reshuf.c
00038 */
00039
00040 #include "form3.h"
00041
00042 /*
00043 *   #[ Includes :
```

```

00044      #[ Reshuf :
00045
00046      Routines to rearrange dummy indices, so that
00047      a: The notation becomes reasonably unique (the perfect job
00048         may consume very much time).
00049      b: The value of AR.CurDum is reset.
00050
00051      Also some routines are needed to aid in the reading of stored
00052      expressions. Also in those expressions there can be dummy
00053      indices, and there should be no conflict with the already
00054      existing dummies.
00055
00056      #[ ReNumber :
00057
00058      Reads the term, tests for dummies, and puts them in order.
00059      Note that this is kind of a first order approximation.
00060      There is quite some room to make this routine 'smart'
00061      First order:
00062          First index will be lowest, second will be next etc.
00063      Second order:
00064          Functions with more than one index and symmetry properties
00065          have some look ahead to see which index is the first to
00066          find its partner.
00067      Third order:
00068          Take the ordering of the functions into account.
00069      Fourth order:
00070          Try all permutations and see which one gives the 'minimal' term.
00071      Currently we use only the first order.
00072
00073      We need a scratch array for the numbers we find, and one for
00074      the addresses at which these numbers are.
00075      We can use the space for the Scrut arrays. There are 13 of those
00076      and each is AM.MaxTal UWORDS long.
00077
00078  */
00079
00080 WORD ReNumber(PHEAD WORD *term)
00081 {
00082     GETBIDENTITY
00083     WORD *d, *e, **p, **f;
00084     WORD n, i, j, old;
00085     AN.DumFound = AN.RenumScratch;
00086     AN.DumPlace = AN.PoinScratch;
00087     AN.DumFunPlace = AN.FunScratch;
00088     AN.NumFound = 0;
00089     FunLevel(BHEAD term);
00090     d = AN.RenumScratch;
00091     p = AN.PoinScratch;
00092     f = AN.FunScratch;
00093  /*
00094     Now the first level sorting.
00095  */
00096     i = AN.IndDum;
00097     n = AN.NumFound;
00098     while ( --n >= 0 ) {
00099         if ( *d > 0 ) {
00100             old = **p;
00101             **p = ++i;
00102             if ( *f ) **f |= DIRTYSYMFLAG;
00103             e = d;
00104             e++;
00105             for ( j = 1; j <= n; j++ ) {
00106                 if ( *e && *(p[j]) == old ) {
00107                     *(p[j]) = i;
00108                     if ( f[j] ) *(f[j]) |= DIRTYSYMFLAG;
00109                     *e = 0;
00110                 }
00111                 e++;
00112             }
00113         }
00114         p++;
00115         d++;
00116         f++;
00117     }
00118     return(i);
00119 }
00120
00121  /*
00122     #] ReNumber :
00123     #[ FunLevel :
00124
00125     Does one term in determining where the dummies are.
00126     Made to work recursively for functions.
00127
00128  */
00129
00130 void FunLevel(PHEAD WORD *term)

```

```

00131 {
00132     GETBIDENTITY
00133     WORD *t, *tstop, *r, *fun;
00134     WORD *m;
00135     t = r = term;
00136     r += *r;
00137     tstop = r - ABS(r[-1]);
00138     t++;
00139     if ( t < tstop ) do {
00140         r = t + t[1];
00141         switch ( *t ) {
00142             case SYMBOL:
00143             case DOTPRODUCT:
00144                 break;
00145             case VECTOR:
00146                 t += 3;
00147                 do {
00148                     if ( *t > AN.IndDum ) {
00149                         if ( AN.NumFound >= AN.MaxRenumScratch ) AdjustRenumScratch(BHEAD0);
00150                         AN.NumFound++;
00151                         *AN.DumFound++ = *t;
00152                         *AN.DumPlace++ = t;
00153                         *AN.DumFunPlace++ = 0;
00154                     }
00155                     t += 2;
00156                 } while ( t < r );
00157                 break;
00158             case SUBEXPRESSION:
00159 /*
00160             Still must hunt down the wildcards(?)
00161 */
00162                 break;
00163             case GAMMA:
00164                 t += FUNHEAD-2;
00165                 /* fall through */
00166             case DELTA:
00167             case INDEX:
00168                 t += 2;
00169                 while ( t < r ) {
00170                     if ( *t > AN.IndDum ) {
00171                         if ( AN.NumFound >= AN.MaxRenumScratch ) AdjustRenumScratch(BHEAD0);
00172                         AN.NumFound++;
00173                         *AN.DumFound++ = *t;
00174                         *AN.DumPlace++ = t;
00175                         *AN.DumFunPlace++ = 0;
00176                     }
00177                     t++;
00178                 }
00179                 break;
00180             case HAAKJE:
00181             case EXPRESSION:
00182             case SNUMBER:
00183             case LNUMBER:
00184                 break;
00185             default:
00186                 if ( *t < FUNCTION ) {
00187                     MLOCK(ErrorMessageLock);
00188                     MesPrint("Unexpected code in ReNumber");
00189                     MUNLOCK(ErrorMessageLock);
00190                     Terminate(-1);
00191                 }
00192                 fun = t+2;
00193                 if ( *t >= FUNCTION && functions[*t-FUNCTION].spec
00194                 >= TENSORFUNCTION ) {
00195                     t += FUNHEAD;
00196                     while ( t < r ) {
00197                         if ( *t > AN.IndDum ) {
00198                             if ( AN.NumFound >= AN.MaxRenumScratch ) AdjustRenumScratch(BHEAD0);
00199                             AN.NumFound++;
00200                             *AN.DumFound++ = *t;
00201                             *AN.DumPlace++ = t;
00202                             *AN.DumFunPlace++ = fun;
00203                         }
00204                         t++;
00205                     }
00206                     break;
00207                 }
00208                 t += FUNHEAD;
00209                 while ( t < r ) {
00210                     if ( *t > 0 ) {
00211
00212                         /* General function. Enter 'recursion'. */
00213
00214                         m = t + *t;
00215                         t += ARGHEAD;
00216                         while ( t < m ) {

```

```

00218             FunLevel(BHEAD t);
00219             t += *t;
00220         }
00221     }
00222     else {
00223         if ( *t == -INDEX ) {
00224             t++;
00225             if ( *t > AN.IndDum ) {
00226                 if ( AN.NumFound >= AN.MaxRenumScratch ) AdjustRenumScratch(BHEAD0);
00227                 AN.NumFound++;
00228                 *AN.DumFound++ = *t;
00229                 *AN.DumPlace++ = t;
00230                 *AN.DumFunPlace++ = fun;
00231             }
00232             t++;
00233         }
00234         else if ( *t <= -FUNCTION ) t++;
00235         else t += 2;
00236     }
00237 }
00238 break;
00239 }
00240 t = r;
00241 } while ( t < tstop );
00242 }
00243
00244 /*
00245     #] FunLevel :
00246     #[ DetCurDum :
00247
00248     We look for indices in the range AM.IndDum to AM.IndDum+MAXDUMMIES.
00249     The maximum value is returned.
00250 */
00251
00252 WORD DetCurDum(PHEAD WORD *t)
00253 {
00254     GETBIDENTITY
00255     WORD maxval = AN.IndDum;
00256     WORD maxtop = AM.IndDum + WILDOFFSET;
00257     WORD *tstop, *m, *r, i;
00258     tstop = t + *t - 1;
00259     tstop -= ABS(*tstop);
00260     t++;
00261     while ( t < tstop ) {
00262         if ( *t == VECTOR ) {
00263             m = t + 3;
00264             t += t[1];
00265             while ( m < t ) {
00266                 if ( *m > maxval && *m < maxtop ) maxval = *m;
00267                 m += 2;
00268             }
00269         }
00270         else if ( *t == DELTA || *t == INDEX ) {
00271             m = t + 2;
00272         Singles:
00273             t += t[1];
00274             while ( m < t ) {
00275                 if ( *m > maxval && *m < maxtop ) maxval = *m;
00276                 m++;
00277             }
00278         }
00279         else if ( *t >= FUNCTION ) {
00280             if ( functions[*t-FUNCTION].spec >= TENSORFUNCTION ) {
00281                 m = t + FUNHEAD;
00282                 goto Singles;
00283             }
00284             r = t + FUNHEAD;
00285             t += t[1];
00286             while ( r < t ) {          /* The arguments */
00287                 if ( *r < 0 ) {
00288                     if ( *r <= -FUNCTION ) r++;
00289                     else if ( *r == -INDEX ) {
00290                         if ( r[1] > maxval && r[1] < maxtop ) maxval = r[1];
00291                         r += 2;
00292                     }
00293                     else r += 2;
00294                 }
00295                 else {
00296                     m = r + ARGHEAD;
00297                     r += *r;
00298                     while ( m < r ) { /* Terms in the argument */
00299                         i = DetCurDum(BHEAD m);
00300                         if ( i > maxval && i < maxtop ) maxval = i;
00301                         m += *m;
00302                     }
00303                 }
00304             }

```

```

00305     }
00306     else {
00307         t += t[1];
00308     }
00309 }
00310 return(maxval);
00311 }
00312
00313 /*
00314     #] DetCurDum :
00315     #[ FullRenumber :
00316
00317     Does a full renumbering. May be slow if there are many indices.
00318     par = 1 Goes with a factorial!
00319     par = 0 All single exchanges only till there is no more improvement.
00320     Notice that there is a hole in the defense with respect to
00321     arguments inside functions inside functions.
00322 */
00323
00324 int FullRenumber(PHEAD WORD *term, WORD par)
00325 {
00326     GETBIDENTITY
00327     WORD *d, **p, **f, *w, *t, *best, *stac, *perm, a, *termtry;
00328     WORD n, i, j, k, ii;
00329     WORD *oldworkpointer = AT.WorkPointer;
00330     n = ReNumber(BHEAD term) - AM.IndDum;
00331     if ( n <= 1 ) return(0);
00332     Normalize(BHEAD term);
00333     if ( *term == 0 ) return(0);
00334     n = ReNumber(BHEAD term) - AM.IndDum;
00335     d = AN.RenumScratch;
00336     p = AN.PoinScratch;
00337     f = AN.FunScratch;
00338     if ( AT.WorkPointer < term + *term ) AT.WorkPointer = term + *term;
00339     k = AN.NumFound;
00340     best = w = AT.WorkPointer; t = term;
00341     for ( i = *term; i > 0; i-- ) *w++ = *t++;
00342     AT.WorkPointer = w;
00343     Normalize(BHEAD best);
00344     AT.WorkPointer = w = best + *best;
00345     stac = w+100;
00346     perm = stac + n + 1;
00347     termtry = perm + n + 1;
00348     for ( i = 1; i <= n; i++ ) perm[i] = i + AM.IndDum;
00349     for ( i = 1; i <= n; i++ ) stac[i] = i;
00350     for ( i = 0; i < k; i++ ) d[i] = *(p[i]) - AM.IndDum;
00351     if ( par == 0 ) { /* All single exchanges */
00352         for ( i = 1; i < n; i++ ) {
00353             for ( j = i+1; j <= n; j++ ) {
00354                 a = perm[j]; perm[j] = perm[i]; perm[i] = a;
00355                 for ( ii = 0; ii < k; ii++ ) {
00356                     *(p[ii]) = perm[d[ii]];
00357                     if ( f[ii] ) *(f[ii]) |= DIRTYSYMFLAG;
00358                 }
00359                 t = term; w = termtry;
00360                 for ( ii = 0; ii < *term; ii++ ) *w++ = *t++;
00361                 AT.WorkPointer = w;
00362                 if ( Normalize(BHEAD termtry) == 0 ) {
00363                     if ( *termtry == 0 ) goto Return0;
00364                     if ( ( ii = CompareTerms(BHEAD termtry,best,0) ) > 0 ) {
00365                         t = termtry; w = best;
00366                         for ( ii = 0; ii < *termtry; ii++ ) *w++ = *t++;
00367                         i = 0; break; /* restart from beginning */
00368                     }
00369                     else if ( ii == 0 && CompCoef(termtry,best) != 0 )
00370                         goto Return0;
00371                 }
00372                 /* if no success, set back */
00373                 a = perm[j]; perm[j] = perm[i]; perm[i] = a;
00374             }
00375         }
00376     }
00377     else if ( par == 1 ) { /* all permutations */
00378         j = n-1;
00379         for (;;) {
00380             if ( stac[j] == n ) {
00381                 a = perm[j]; perm[j] = perm[n]; perm[n] = a;
00382                 stac[j] = j;
00383                 j--;
00384                 if ( j <= 0 ) break;
00385                 continue;
00386             }
00387             if ( j != stac[j] ) {
00388                 a = perm[j]; perm[j] = perm[stac[j]]; perm[stac[j]] = a;
00389             }
00390             (stac[j])++;
00391             a = perm[j]; perm[j] = perm[stac[j]]; perm[stac[j]] = a;

```

```

00392         {
00393             for ( i = 0; i < k; i++ ) {
00394                 *(p[i]) = perm[d[i]];
00395                 if ( f[i] ) *(f[i]) |= DIRTSYMFFLAG;
00396             }
00397             t = term; w = termtry;
00398             for ( i = 0; i < *term; i++ ) *w++ = *t++;
00399             AT.WorkPointer = w;
00400             if ( Normalize(BHEAD termtry) == 0 ) {
00401                 if ( *termtry == 0 ) goto Return0;
00402                 if ( ( ii = CompareTerms(BHEAD termtry,best,0) ) > 0 ) {
00403                     t = termtry; w = best;
00404                     for ( i = 0; i < *termtry; i++ ) *w++ = *t++;
00405                 }
00406                 else if ( ii == 0 && CompCoef(termtry,best) != 0 )
00407                     goto Return0;
00408             }
00409         }
00410         if ( j < n-1 ) { j = n-1; }
00411     }
00412 }
00413 t = term; w = best;
00414 n = *best;
00415 for ( i = 0; i < n; i++ ) *t++ = *w++;
00416 AT.WorkPointer = oldworkpointer;
00417 return(0);
00418 Return0:
00419 *term = 0;
00420 AT.WorkPointer = oldworkpointer;
00421 return(0);
00422 }
00423
00424 /*
00425     #] FullRenumber :
00426     #[ MoveDummies :
00427
00428     Routine shifts the dummy indices by an amount 'shift'.
00429     It is an adaptation of DetCurDum.
00430     Needed for = ...*expression^power*...
00431     in which expression contains dummy indices.
00432     Note that this code should have been in ver1 already and has
00433     always been missing. Routine made 29-jan-2007.
00434 */
00435
00436 void MoveDummies(PHEAD WORD *term, WORD shift)
00437 {
00438     GETBIDENTITY
00439     WORD maxval = AN.IndDum;
00440     WORD maxtop = AM.IndDum + WILDOFFSET;
00441     WORD *tstop, *m, *r;
00442     tstop = term + *term - 1;
00443     tstop -= ABS(*tstop);
00444     term++;
00445     while ( term < tstop ) {
00446         if ( *term == VECTOR ) {
00447             m = term + 3;
00448             term += term[1];
00449             while ( m < term ) {
00450                 if ( *m > maxval && *m < maxtop ) *m += shift;
00451                 m += 2;
00452             }
00453         }
00454         else if ( *term == DELTA || *term == INDEX ) {
00455             m = term + 2;
00456         Singles:
00457             term += term[1];
00458             while ( m < term ) {
00459                 if ( *m > maxval && *m < maxtop ) *m += shift;
00460                 m++;
00461             }
00462         }
00463         else if ( *term >= FUNCTION ) {
00464             if ( functions[*term-FUNCTION].spec >= TENSORFUNCTION ) {
00465                 m = term + FUNHEAD;
00466                 goto Singles;
00467             }
00468             r = term + FUNHEAD;
00469             term += term[1];
00470             while ( r < term ) { /* The arguments */
00471                 if ( *r < 0 ) {
00472                     if ( *r <= -FUNCTION ) r++;
00473                     else if ( *r == -INDEX ) {
00474                         if ( r[1] > maxval && r[1] < maxtop ) r[1] += shift;
00475                         r += 2;
00476                     }
00477                     else r += 2;
00478                 }

```

```

00479         else {
00480             m = r + ARGHEAD;
00481             r += *r;
00482             while ( m < r ) { /* Terms in the argument */
00483                 MoveDummies(BHEAD m, shift);
00484                 m += *m;
00485             }
00486         }
00487     }
00488 }
00489 else {
00490     term += term[1];
00491 }
00492 }
00493 }
00494
00495 /*
00496     #] MoveDummies :
00497     #[ AdjustRenumScratch :
00498
00499     Extends the buffer for number of dummies that can be found in
00500     a term. Originally we had a fixed buffer at size 300 in the AN
00501     struct. Thomas Hahn ran out of that. Hence we have now made it
00502     a dynamical buffer.
00503     Note that the pointers used in FunLevel need adjustment as well.
00504 */
00505 void AdjustRenumScratch(PHEAD0)
00506 {
00507     GETBIDENTITY
00508     WORD newsize;
00509     int i;
00510     WORD **newpoin, *newnum;
00511     if ( AN.MaxRenumScratch == 0 ) newsize = 100;
00512     else newsize = AN.MaxRenumScratch*2;
00513     if ( newsize > MAXPOSITIVE/2 ) newsize = MAXPOSITIVE/2+1;
00514
00515     newpoin = (WORD **)Malloca1(newsize*sizeof(WORD *), "PoinScratch");
00516     for ( i = 0; i < AN.NumFound; i++ ) newpoin[i] = AN.PoinScratch[i];
00517     for ( ; i < newsize; i++ ) newpoin[i] = 0;
00518     if ( AN.PoinScratch ) M_free(AN.PoinScratch, "PoinScratch");
00519     AN.PoinScratch = newpoin;
00520     AN.DumPlace = newpoin + AN.NumFound;
00521
00522     newpoin = (WORD **)Malloca1(newsize*sizeof(WORD *), "FunScratch");
00523     for ( i = 0; i < AN.NumFound; i++ ) newpoin[i] = AN.FunScratch[i];
00524     for ( ; i < newsize; i++ ) newpoin[i] = 0;
00525     if ( AN.FunScratch ) M_free(AN.FunScratch, "FunScratch");
00526     AN.FunScratch = newpoin;
00527     AN.DumFunPlace = newpoin + AN.NumFound;
00528
00529     newnum = (WORD *)Malloca1(newsize*sizeof(WORD), "RenumScratch");
00530     for ( i = 0; i < AN.NumFound; i++ ) newnum[i] = AN.RenumScratch[i];
00531     for ( ; i < newsize; i++ ) newnum[i] = 0;
00532     if ( AN.RenumScratch ) M_free(AN.RenumScratch, "RenumScratch");
00533     AN.RenumScratch = newnum;
00534     AN.DumFound = newnum + AN.NumFound;
00535
00536     AN.MaxRenumScratch = newsize;
00537 }
00538
00539 /*
00540     #] AdjustRenumScratch :
00541     #[ Reshuf :
00542     #[ Count :
00543     #[ CountDo :
00544
00545     This function executes the counting action in a count
00546     operation. The return value is the count of the term.
00547     Input is the term and a pointer to the instruction.
00548 */
00549 WORD CountDo(WORD *term, WORD *instruct)
00550 {
00551     WORD *m, *r, i, j, count = 0;
00552     WORD *stopper, *tstop, *r1 = 0, *r2 = 0;
00553     m = instruct;
00554     stopper = m + m[1];
00555     instruct += 3;
00556     tstop = term + *term; tstop -= ABS(tstop[-1]); term++;
00557     while ( term < tstop ) {
00558         switch ( *term ) {
00559             case SYMBOL:
00560                 i = term[1] - 2;
00561                 term += 2;
00562                 while ( i > 0 ) {

```

```

00566         m = instruct;
00567         while ( m < stopper ) {
00568             if ( *m == SYMBOL && m[2] == *term ) {
00569                 count += m[3] * term[1];
00570             }
00571             m += m[1];
00572         }
00573         term += 2;
00574         i -= 2;
00575     }
00576     break;
00577 case DOTPRODUCT:
00578     i = term[1] - 2;
00579     term += 2;
00580     while ( i > 0 ) {
00581         m = instruct;
00582         while ( m < stopper ) {
00583             if ( *m == DOTPRODUCT && ( ( m[2] == *term &&
00584                 m[3] == term[1] ) || ( m[2] == term[1] &&
00585                 m[3] == *term ) ) ) {
00586                 count += m[4] * term[2];
00587                 break;
00588             }
00589             m += m[1];
00590         }
00591         m = instruct;
00592         while ( m < stopper ) {
00593             if ( *m == VECTOR && m[2] == *term &&
00594                 ( m[3] & DOTPBIT ) != 0 ) {
00595                 count += m[m[1]-1] * term[2];
00596             }
00597             m += m[1];
00598         }
00599         m = instruct;
00600         while ( m < stopper ) {
00601             if ( *m == VECTOR && m[2] == term[1] &&
00602                 ( m[3] & DOTPBIT ) != 0 ) {
00603                 count += m[m[1]-1] * term[2];
00604             }
00605             m += m[1];
00606         }
00607         term += 3;
00608         i -= 3;
00609     }
00610     break;
00611 case INDEX:
00612     j = 1;
00613     goto VectInd;
00614 case VECTOR:
00615     j = 2;
00616 VectInd:
00617     i = term[1] - 2;
00618     term += 2;
00619     while ( i > 0 ) {
00620         m = instruct;
00621         while ( m < stopper ) {
00622             if ( *m == VECTOR && m[2] == *term &&
00623                 ( m[3] & VECTBIT ) != 0 ) {
00624                 count += m[m[1]-1];
00625             }
00626             m += m[1];
00627         }
00628         term += j;
00629         i -= j;
00630     }
00631     break;
00632 default:
00633     if ( *term >= FUNCTION ) {
00634         i = *term;
00635         m = instruct;
00636         while ( m < stopper ) {
00637             if ( *m == FUNCTION && m[2] == i ) count += m[3];
00638             m += m[1];
00639         }
00640         if ( functions[i-FUNCTION].spec >= TENSORFUNCTION ) {
00641             i = term[1] - FUNHEAD;
00642             term += FUNHEAD;
00643             while ( i > 0 ) {
00644                 if ( *term < 0 ) {
00645                     m = instruct;
00646                     while ( m < stopper ) {
00647                         if ( *m == VECTOR && m[2] == *term &&
00648                             ( m[3] & FUNBIT ) != 0 ) {
00649                             count += m[m[1]-1];
00650                         }
00651                         m += m[1];
00652                     }

```



```

00653             term++;
00654             i--;
00655         }
00656     }
00657     else {
00658         r = term + term[1];
00659         term += FUNHEAD;
00660         while ( term < r ) {
00661             if ( ( *term == -INDEX || *term == -VECTOR
00662                 || *term == -MINVECTOR ) && term[1] < MINSPEC ) {
00663                 m = instruct;
00664                 while ( m < stopper ) {
00665                     if ( *m == VECTOR && term[1] == m[2]
00666                         && ( m[3] & SETBIT ) != 0 ) {
00667                         r1 = SetElements + Sets[m[4]].first;
00668                         r2 = SetElements + Sets[m[4]].last;
00669                         while ( r1 < r2 ) {
00670                             if ( *r1 == i ) {
00671                                 count += m[m[1]-1];
00672                                 goto NextFF;
00673                             }
00674                             r1++;
00675                         }
00676                     }
00677                     m += m[1];
00678                 }
00679             NextFF:
00680                 term += 2;
00681             }
00682             else { NEXTARG(term) }
00683         }
00684     }
00685     break;
00686 }
00687 else {
00688     term += term[1];
00689 }
00690 break;
00691 }
00692 }
00693 return(count);
00694 }
00695
00696 /*
00697 #] CountDo :
00698 #[ CountFun :
00699
00700 This is the count function.
00701 The return value is the count of the term.
00702 Input is the term and a pointer to the count function.
00703
00704 */
00705
00706 WORD CountFun(WORD *term, WORD *countfun)
00707 {
00708     WORD *m, *r, i, j, count = 0, *instruct, *stopper, *tstop;
00709     m = countfun;
00710     stopper = m + m[1];
00711     instruct = countfun + FUNHEAD;
00712     tstop = term + *term; tstop -= ABS(tstop[-1]); term++;
00713     while ( term < tstop ) {
00714         switch ( *term ) {
00715             case SYMBOL:
00716                 i = term[1] - 2;
00717                 term += 2;
00718                 while ( i > 0 ) {
00719                     m = instruct;
00720                     while ( m < stopper ) {
00721                         if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
00722                         if ( *m == -SYMBOL && m[1] == *term
00723                             && m[2] == -SNUMBER && ( m + 2 ) < stopper ) {
00724                             count += m[3] * term[1]; m += 4;
00725                         }
00726                         else { NEXTARG(m) }
00727                     }
00728                     term += 2;
00729                     i -= 2;
00730                 }
00731                 break;
00732             case DOTPRODUCT:
00733                 i = term[1] - 2;
00734                 term += 2;
00735                 while ( i > 0 ) {
00736                     m = instruct;
00737                     while ( m < stopper ) {
00738                         if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
00739                         if ( *m == 9+ARGHEAD && m[ARGHEAD] == 9

```

```

00740         && m[ARGHEAD+1] == DOTPRODUCT
00741         && m[ARGHEAD+9] == -SNUMBER && ( m + ARGHEAD+9 ) < stopper
00742         && ( ( m[ARGHEAD+3] == *term &&
00743         m[ARGHEAD+4] == term[1] ) ||
00744         ( m[ARGHEAD+3] == term[1] &&
00745         m[ARGHEAD+4] == *term ) ) {
00746             count += m[ARGHEAD+10] * term[2];
00747             m += ARGHEAD+11;
00748         }
00749         else { NEXTARG(m) }
00750     }
00751     m = instruct;
00752     while ( m < stopper ) {
00753         if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
00754         if ( ( *m == -VECTOR || *m == -MINVECTOR )
00755         && m[1] == *term &&
00756         m[2] == -SNUMBER && ( m+2 ) < stopper ) {
00757             count += m[3] * term[2]; m += 4;
00758         }
00759         NEXTARG(m)
00760     }
00761     m = instruct;
00762     while ( m < stopper ) {
00763         if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
00764         if ( ( *m == -VECTOR || *m == -MINVECTOR )
00765         && m[1] == term[1] &&
00766         m[2] == -SNUMBER && ( m+2 ) < stopper ) {
00767             count += m[3] * term[2];
00768             m += 4;
00769         }
00770         NEXTARG(m)
00771     }
00772     term += 3;
00773     i -= 3;
00774 }
00775 break;
00776 case INDEX:
00777     j = 1;
00778     goto VectInd;
00779 case VECTOR:
00780     j = 2;
00781 VectInd:
00782     i = term[1] - 2;
00783     term += 2;
00784     while ( i > 0 ) {
00785         m = instruct;
00786         while ( m < stopper ) {
00787             if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
00788             if ( ( *m == -VECTOR || *m == -MINVECTOR )
00789             && m[1] == *term &&
00790             m[2] == -SNUMBER && ( m+2 ) < stopper ) {
00791                 count += m[3]; m += 4;
00792             }
00793             NEXTARG(m)
00794         }
00795         term += j;
00796         i -= j;
00797     }
00798     break;
00799 default:
00800     if ( *term >= FUNCTION ) {
00801         i = *term;
00802         m = instruct;
00803         while ( m < stopper ) {
00804             if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
00805             if ( *m == -i && m[1] == -SNUMBER && ( m+1 ) < stopper ) {
00806                 count += m[2]; m += 3;
00807             }
00808             NEXTARG(m)
00809         }
00810         if ( functions[i-FUNCTION].spec >= TENSORFUNCTION ) {
00811             i = term[1] - FUNHEAD;
00812             term += FUNHEAD;
00813             while ( i > 0 ) {
00814                 if ( *term < 0 ) {
00815                     m = instruct;
00816                     while ( m < stopper ) {
00817                         if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
00818                         if ( ( *m == -VECTOR || *m == -INDEX
00819                         || *m == -MINVECTOR ) && m[1] == *term &&
00820                         m[2] == -SNUMBER && ( m+2 ) < stopper ) {
00821                             count += m[3]; m += 4;
00822                         }
00823                         else { NEXTARG(m) }
00824                     }
00825                 }
00826                 term++;
00827                 i--;

```

```

00827     }
00828     }
00829     else {
00830         r = term + term[1];
00831         term += FUNHEAD;
00832         while ( term < r ) {
00833             if ( ( *term == -INDEX || *term == -VECTOR
00834                 || *term == -MINVECTOR ) && term[1] < MINSPEC ) {
00835                 m = instruct;
00836                 while ( m < stopper ) {
00837                     if ( *m == -SNUMBER ) { NEXTARG(m) continue; }
00838                     if ( *m == -VECTOR && m[1] == term[1]
00839                         && m[2] == -SNUMBER && (m+2) < stopper ) {
00840                         count += m[3];
00841                         m += 4;
00842                     }
00843                     else { NEXTARG(m) }
00844                 }
00845                 term += 2;
00846             }
00847             else { NEXTARG(term) }
00848         }
00849     }
00850     break;
00851 }
00852 else {
00853     term += term[1];
00854 }
00855 break;
00856 }
00857 }
00858 return(count);
00859 }
00860
00861 /*
00862     #] CountFun :
00863     #] Count :
00864     #[ DimensionSubterm :
00865 */
00866
00867 WORD DimensionSubterm(WORD *subterm)
00868 {
00869     WORD *r, *rstop, dim, i;
00870     LONG x = 0;
00871     rstop = subterm + subterm[1];
00872     if ( *subterm == SYMBOL ) {
00873         r = subterm + 2;
00874         while ( r < rstop ) {
00875             if ( *r <= NumSymbols && *r > -MAXPOWER ) {
00876                 dim = symbols[*r].dimension;
00877                 if ( dim == MAXPOSITIVE ) goto undefined;
00878                 x += dim * r[1];
00879                 if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
00880                 r += 2;
00881             }
00882             else if ( *r <= MAXVARIABLES ) {
00883 /*
00884                 Here we have an extra symbol. Store dimension in the compiler buffer
00885 */
00886                 i = MAXVARIABLES - *r;
00887                 dim = cbuf[AM.sbufnum].dimension[i];
00888                 if ( dim == MAXPOSITIVE ) goto undefined;
00889                 if ( dim == -MAXPOSITIVE ) goto outofrange;
00890                 x += dim * r[1];
00891                 if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
00892                 r += 2;
00893             }
00894             else { r += 2; }
00895         }
00896     }
00897     else if ( *subterm == DOTPRODUCT ) {
00898         r = subterm + 2;
00899         while ( r < rstop ) {
00900             dim = vectors[*r-AM.OffsetVector].dimension;
00901             if ( dim == MAXPOSITIVE ) goto undefined;
00902             x += dim * r[2];
00903             if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
00904             dim = vectors[r[1]-AM.OffsetVector].dimension;
00905             if ( dim == MAXPOSITIVE ) goto undefined;
00906             x += dim * r[2];
00907             if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
00908             r += 3;
00909         }
00910     }
00911     else if ( *subterm == VECTOR ) {
00912         r = subterm + 2;
00913         while ( r < rstop ) {

```

```

00914         dim = vectors[*r-AM.OffsetVector].dimension;
00915         if ( dim == MAXPOSITIVE ) goto undefined;
00916         x += dim;
00917         if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
00918         r += 2;
00919     }
00920 }
00921 else if ( *subterm == INDEX ) {
00922     r = subterm + 2;
00923     while ( r < rstop ) {
00924         if ( *r < 0 ) {
00925             dim = vectors[*r-AM.OffsetVector].dimension;
00926             if ( dim == MAXPOSITIVE ) goto undefined;
00927             x += dim;
00928             if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
00929         }
00930         r++;
00931     }
00932 }
00933 else if ( *subterm >= FUNCTION ) {
00934     dim = functions[*subterm-FUNCTION].dimension;
00935     if ( dim == MAXPOSITIVE ) goto undefined;
00936     x += dim;
00937     if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
00938     if ( functions[*subterm-FUNCTION].spec > 0 ) { /* tensor */
00939         r = subterm + FUNHEAD;
00940         while ( r < rstop ) {
00941             if ( *r < MINSPEC ) {
00942                 dim = vectors[*r-AM.OffsetVector].dimension;
00943                 if ( dim == MAXPOSITIVE ) goto undefined;
00944                 x += dim;
00945                 if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
00946             }
00947             r++;
00948         }
00949     }
00950 }
00951 return((WORD)x);
00952 undefined:
00953 return((WORD)MAXPOSITIVE);
00954 outofrange:
00955 return(-(WORD)MAXPOSITIVE);
00956 }
00957 /*
00958 #] DimensionSubterm :
00959 #[ DimensionTerm :
00960
00961 Returns the dimension of the given term.
00962 If there is any variable of which the dimension is not defined
00963 we return the code for undefined which is MAXPOSITIVE
00964 When the value is out of range we return -MAXPOSITIVE
00965 */
00966
00967 WORD DimensionTerm(WORD *term)
00968 {
00969     WORD *t, *tstop, dim;
00970     LONG x = 0;
00971     tstop = term + *term; tstop -= ABS(tstop[-1]);
00972     t = term+1;
00973     while ( t < tstop ) {
00974         dim = DimensionSubterm(t);
00975         if ( dim == MAXPOSITIVE ) goto undefined;
00976         if ( dim == -MAXPOSITIVE ) goto outofrange;
00977         x += dim;
00978         if ( x >= MAXPOSITIVE || x <= -MAXPOSITIVE ) goto outofrange;
00979         t += t[1];
00980     }
00981     return((WORD)x);
00982 undefined:
00983 return((WORD)MAXPOSITIVE);
00984 outofrange:
00985 return(-(WORD)MAXPOSITIVE);
00986 }
00987 /*
00988 #] DimensionTerm :
00989 #[ DimensionExpression :
00990
00991 Returns the dimension of the given expression.
00992 If there is any variable of which the dimension is not defined
00993 we return the code for undefined which is MAXPOSITIVE
00994 When the value is out of range we return -MAXPOSITIVE
00995 When the value is not consistent we return -MAXPOSITIVE.
00996 */
00997
00998 WORD DimensionExpression(PHEAD WORD *expr)

```

```

01001 {
01002     WORD dim, *term, *old, x = 0;
01003     int first = 1;
01004     term = expr;
01005     while ( *term ) {
01006         dim = DimensionTerm(term);
01007         if ( dim == MAXPOSITIVE ) goto undefined;
01008         if ( dim == -MAXPOSITIVE ) goto outofrange;
01009         if ( first ) { x = dim; }
01010         else if ( x != dim ) {
01011             old = AN.currentTerm;
01012             MLOCK(ErrorMessageLock);
01013             MesPrint("Dimension is not the same in the terms of the expression");
01014             term = expr;
01015             while ( *term ) {
01016                 AN.currentTerm = term;
01017                 MesPrint(" %T");
01018             }
01019             MUNLOCK(ErrorMessageLock);
01020             AN.currentTerm = old;
01021             return(-(WORD)MAXPOSITIVE);
01022         }
01023         term += *term;
01024     }
01025     return((WORD)x);
01026 undefined:
01027     return((WORD)MAXPOSITIVE);
01028 outofrange:
01029     old = AN.currentTerm;
01030     AN.currentTerm = term;
01031     MLOCK(ErrorMessageLock);
01032     MesPrint("Dimension out of range in %t in subexpression");
01033     MUNLOCK(ErrorMessageLock);
01034     AN.currentTerm = old;
01035     return(-(WORD)MAXPOSITIVE);
01036 }
01037
01038 /*
01039 #] DimensionExpression :
01040 #[ Multiply Term :
01041     #[ MultDo :
01042 */
01043
01044 int MultDo(PHEAD WORD *term, WORD *pattern)
01045 {
01046     GETBIDENTITY
01047     WORD *t, *r, i;
01048     t = term + *term;
01049     if ( pattern[2] > 0 ) { /* Left multiply */
01050         i = *term - 1;
01051     }
01052     else { /* Right multiply */
01053         i = ABS(t[-1]);
01054     }
01055     *term += SUBEXPSIZE;
01056     r = t + SUBEXPSIZE;
01057     do { *--r = *--t; } while ( --i > 0 );
01058     r = pattern + 3;
01059     i = r[1];
01060     while ( --i >= 0 ) *t++ = *r++;
01061     AT.WorkPointer = term + *term;
01062     return(0);
01063 }
01064
01065 /*
01066 #] MultDo :
01067 #[ Multiply Term :
01068 #[ Try Term(s) :
01069     #[ TryDo :
01070 */
01071
01072 int TryDo(PHEAD WORD *term, WORD *pattern, WORD level)
01073 {
01074     GETBIDENTITY
01075     WORD *t, *r, *m, i, j;
01076     ReNumber(BHEAD term);
01077     Normalize(BHEAD term);
01078     m = r = term + *term;
01079     m++;
01080     i = pattern[2];
01081     t = pattern + 3;
01082     NCOPY(m,t,i)
01083     j = *term - 1;
01084     t = term + 1;
01085     NCOPY(m,t,j)
01086     *r = WORDDIFF(m,r);
01087     AT.WorkPointer = m;

```

```

01088     if ( ( j = Normalize(BHEAD r) ) == 0 || j == 1 ) {
01089         if ( *r == 0 ) return(0);
01090         ReNumber(BHEAD r); Normalize(BHEAD r);
01091         if ( *r == 0 ) return(0);
01092         if ( ( i = CompareTerms(BHEAD term,r,0) ) < 0 ) {
01093             *AN.RepPoint = 1;
01094             AR.expchanged = 1;
01095             return(Generator(BHEAD r,level));
01096         }
01097         if ( i == 0 && CompCoef(term,r) != 0 ) { return(0); }
01098     }
01099     AT.WorkPointer = r;
01100     return(Generator(BHEAD term,level));
01101 }
01102
01103 /*
01104     #] TryDo :
01105     #] Try Term(s) :
01106     #[ Distribute :
01107     #[ DoDistrib :
01108
01109     The routine that generates the terms ordered by a distrib_ function.
01110     The presence of a replaceable distrib_ function has been sensed
01111     in the routine TestSub and has been passed on to Generator.
01112     It is then Generator that calls this function in a way that is
01113     similar to calling the trace routines, except for that for the
01114     trace routines and the Levi-Civita tensors the arguments are put
01115     in temporary storage and here we leave them inside the term,
01116     because there is no knowing how long the field will be.
01117 */
01118
01119 int DoDistrib(PHEAD WORD *term, WORD level)
01120 {
01121     GETBIDENTITY
01122     WORD *t, *m, *r = 0, *stop, *tstop, *termout, *endhead, *starttail, *parms;
01123     WORD i, j, k, n, nn, ntype, fun1 = 0, fun2 = 0, typ1 = 0, typ2 = 0;
01124     WORD *arg, *oldwork, *mf, ktype = 0, atype = 0;
01125     WORD sgn, dirtyflag;
01126     AN.TeInFun = AR.TePos = 0;
01127     t = term;
01128     tstop = t + *t;
01129     stop = tstop - ABS(tstop[-1]);
01130     t++;
01131     while ( t < stop ) {
01132         r = t + t[1];
01133         if ( *t == DISTRIBUTION && t[FUNHEAD] == -SNUMBER
01134             && t[FUNHEAD+1] >= -2 && t[FUNHEAD+1] <= 2
01135             && t[FUNHEAD+2] == -SNUMBER
01136             && t[FUNHEAD+4] <= -FUNCTION
01137             && t[FUNHEAD+5] <= -FUNCTION ) {
01138             WORD *ttt = t+FUNHEAD+6, *tttstop = t+t[1];
01139             while ( ttt < tttstop ) {
01140                 if ( *ttt == -DOLLEXPRESSION ) break;
01141                 NEXTARG(ttt);
01142             }
01143             if ( ttt >= tttstop ) {
01144                 fun1 = -t[FUNHEAD+4];
01145                 fun2 = -t[FUNHEAD+5];
01146                 typ1 = functions[fun1-FUNCTION].spec;
01147                 typ2 = functions[fun2-FUNCTION].spec;
01148                 if ( typ1 > 0 || typ2 > 0 ) {
01149                     m = t + FUNHEAD+6;
01150                     r = t + t[1];
01151                     while ( m < r ) {
01152                         if ( *m != -INDEX && *m != -VECTOR && *m != -MINVECTOR )
01153                             break;
01154                         m += 2;
01155                     }
01156                     if ( m < r ) {
01157                         MLOCK(ErrorMessageLock);
01158                         MesPrint("Incompatible function types and arguments in distrib_");
01159                         MUNLOCK(ErrorMessageLock);
01160                         SETERROR(-1)
01161                     }
01162                 }
01163                 break;
01164             }
01165         }
01166         t = r;
01167     }
01168     dirtyflag = t[2];
01169     ntype = t[FUNHEAD+1];
01170     n = t[FUNHEAD+3];
01171 /*
01172     t points at the distrib_ function to be expanded.
01173     fun1,fun2 and typ1,typ2 are the two functions and their types.
01174     ntype indicates the action:

```

```

01175      0:  Make all possible divisions:  2^nargs
01176      1:  fun1 should get n arguments:  nargs! / ( n! (nargs-n)! )
01177      2:  fun2 should get n arguments:  nargs! / ( n! (nargs-n)! )
01178  The distinction between 1 and two is for noncommuting objects.
01179      3:  fun1 should get n arguments. Super symmetric option.
01180      4:  fun2 idem
01181  The super symmetric option involves:
01182      a:  arguments get sorted
01183      b:  identical arguments are seen as such. Hence not all their
01184          distributions are taken into account. It is as if after the
01185          distrib there is a symmetrize fun1; symmetrize fun2;
01186      c:  Hence if the occurrence of each argument is a,b,c,...
01187          and their occurrence in fun1 is a1,b1,c1,... and in fun2
01188          is a2,b2,c2,... then each term is generated (a1+a2)!/a1!/a2!
01189          (b1+b2)!/b1!/b2! (c1+c2)!/c1!/c2! ... times.
01190      d:  We have to make an array of occurrences and positions.
01191      e:  Then we sort the arguments indirectly.
01192      f:  Next we generate the argument lists in the same way as we
01193          generate powers of expressions with binomials. Hence we need
01194          a third array to keep track of the 'powers'
01195  */
01196      endhead = t;
01197      starttail = r;
01198      parms = m = t + FUNHEAD+6;
01199      i = 0;
01200      while ( m < r ) { /* Count arguments */
01201          i++;
01202          NEXTARG(m);
01203      }
01204      oldwork = AT.WorkPointer;
01205      arg = AT.WorkPointer + 1;
01206      arg[-1] = 0;
01207      termout = arg + i;
01208      switch ( ntype ) {
01209          case 0: ktype = 1; atype = n < 0 ? 1: 0; n = 0; break;
01210          case 1: ktype = 1; atype = 0; break;
01211          case 2: ktype = 0; atype = 0; break;
01212          case -1: ktype = 1; atype = 1; break;
01213          case -2: ktype = 0; atype = 1; break;
01214      }
01215      do {
01216  /*
01217          All distributions with n elements. We generate the array arg with
01218          all possible 1 and 0 patterns. 1 means in fun1 and 0 means in fun2.
01219  */
01220          if ( n > i ) return(0); /* 0 elements */
01221
01222          for ( j = 0; j < n; j++ ) arg[j] = 1;
01223          for ( j = n; j < i; j++ ) arg[j] = 0;
01224          for(;;) {
01225              sgn = 0;
01226              t = term;
01227              m = termout;
01228              while ( t < endhead ) *m++ = *t++;
01229              mf = m;
01230              *m++ = fun1;
01231              *m++ = FUNHEAD;
01232              *m++ = dirtyflag;
01233              #if FUNHEAD > 3
01234                  k = FUNHEAD -3;
01235                  while ( k-- > 0 ) *m++ = 0;
01236              #endif
01237              r = parms;
01238              for ( k = 0; k < i; k++ ) {
01239                  if ( arg[k] == ktype ) {
01240                      if ( *r <= -FUNCTION ) *m++ = *r++;
01241                      else if ( *r < 0 ) {
01242                          if ( typ1 > 0 ) {
01243                              if ( *r == -MINVECTOR ) sgn ^= 1;
01244                              r++;
01245                              *m++ = *r++;
01246                          }
01247                          else { *m++ = *r++; *m++ = *r++; }
01248                      }
01249                      else {
01250                          nn = *r;
01251                          NCOPY(m,r,nn);
01252                      }
01253                  }
01254                  else { NEXTARG(r) }
01255              }
01256              mf[1] = WORDDIFF(m,mf);
01257              mf = m;
01258              *m++ = fun2;
01259              *m++ = FUNHEAD;
01260              *m++ = dirtyflag;
01261              #if FUNHEAD > 3

```

```

01262         k = FUNHEAD -3;
01263         while ( k-- > 0 ) *m++ = 0;
01264     #endif
01265     r = parms;
01266     for ( k = 0; k < i; k++ ) {
01267         if ( arg[k] != ktype ) {
01268             if ( *r <= -FUNCTION ) *m++ = *r++;
01269             else if ( *r < 0 ) {
01270                 if ( typ2 > 0 ) {
01271                     if ( *r == -MINVECTOR ) sgn ^= 1;
01272                     r++;
01273                     *m++ = *r++;
01274                 }
01275                 else { *m++ = *r++; *m++ = *r++; }
01276             }
01277             else {
01278                 nn = *r;
01279                 NCOPY(m,r,nn);
01280             }
01281         }
01282         else { NEXTARG(r) }
01283     }
01284     mf[1] = WORDDIF(m,mf);
01285     #ifndef NUOVO
01286     if ( atype == 0 ) {
01287         WORD k1,k2;
01288         for ( k = 0; k < i-1; k++ ) {
01289             if ( arg[k] == 0 ) continue;
01290             k1 = 1; k2 = k;
01291             while ( k < i-1 && EqualArg(parms,k,k+1) ) { k++; k1++; }
01292             while ( k2 <= k && arg[k2] == 1 ) k2++;
01293             k2 = k-k2+1;
01294             /*
01295             Now we need k1!/(k2! (k1-k2)!)
01296             */
01297             if ( k2 != k1 && k2 != 0 ) {
01298                 if ( GetBinom((UWORD *)m+3,m+2,k1,k2) ) {
01299                     MLOCK(ErrorMessageLock);
01300                     MesCall("DoDistrib");
01301                     MUNLOCK(ErrorMessageLock);
01302                     SETERROR(-1)
01303                 }
01304                 m[1] = ( m[2] < 0 ? -m[2]: m[2] ) + 3;
01305                 *m = LNUMBER;
01306                 m += m[1];
01307             }
01308         }
01309     }
01310     #endif
01311     r = starttail;
01312     while ( r < tstop ) *m++ = *r++;
01313
01314     if ( atype ) { /* antisymmetric field */
01315         k = n;
01316         nn = 0;
01317         for ( j = 0; j < i && k > 0; j++ ) {
01318             if ( arg[j] == 1 ) k--;
01319             else nn += k;
01320         }
01321         sgn ^= nn & 1;
01322     }
01323
01324     if ( sgn ) m[-1] = -m[-1];
01325     *termout = WORDDIF(m,termout);
01326     AT.WorkPointer = m;
01327     if ( AT.WorkPointer > AT.WorkTop ) {
01328         MLOCK(ErrorMessageLock);
01329         MesWork();
01330         MUNLOCK(ErrorMessageLock);
01331         return(-1);
01332     }
01333     *AN.RepPoint = 1;
01334     AR.expchanged = 1;
01335     if ( Generator(BHEAD termout,level) ) Terminate(-1);
01336     #ifndef NUOVO
01337     {
01338         WORD k1;
01339         j = i - 1;
01340         k = 0;
01341         redok: while ( arg[j] == 1 && j >= 0 ) { j--; k++; }
01342         while ( arg[j] == 0 && j >= 0 ) j--;
01343         if ( j < 0 ) break;
01344         k1 = j;
01345         arg[j] = 0;
01346         while ( !atype && EqualArg(parms,j,j+1) ) {
01347             j++;
01348             if ( j >= i - k - 1 ) { j = k1; k++; goto redok; }

```



```

01349         arg[j] = 0;
01350     }
01351     while ( k >= 0 ) { j++; arg[j] = 1; k--; }
01352     j++;
01353     while ( j < i ) { arg[j] = 0; j++; }
01354 }
01355 #else
01356     j = i - 1;
01357     k = 0;
01358     while ( arg[j] == 1 && j >= 0 ) { j--; k++; }
01359     while ( arg[j] == 0 && j >= 0 ) j--;
01360     if ( j < 0 ) break;
01361     arg[j] = 0;
01362     while ( k >= 0 ) { j++; arg[j] = 1; k--; }
01363     j++;
01364     while ( j < i ) { arg[j] = 0; j++; }
01365 #endif
01366 }
01367 } while ( ntype == 0 && ++n <= i );
01368 AT.WorkPointer = oldwork;
01369 return(0);
01370 }
01371
01372 /*
01373     #] DoDistrib :
01374     #[ EqualArg :
01375
01376     Returns 1 if the arguments in the field are identical.
01377 */
01378
01379 int EqualArg(WORD *parms, WORD num1, WORD num2)
01380 {
01381     WORD *t1, *t2;
01382     WORD i;
01383     t1 = parms;
01384     while ( --num1 >= 0 ) { NEXTARG(t1); }
01385     t2 = parms;
01386     while ( --num2 >= 0 ) { NEXTARG(t2); }
01387     if ( *t1 != *t2 ) return(0);
01388     if ( *t1 < 0 ) {
01389         if ( *t1 <= -FUNCTION || t1[1] == t2[1] ) return(1);
01390         return(0);
01391     }
01392     i = *t1;
01393     while ( --i >= 0 ) {
01394         if ( *t1 != *t2 ) return(0);
01395         t1++; t2++;
01396     }
01397     return(1);
01398 }
01399
01400 /*
01401     #] EqualArg :
01402     #[ DoDelta3 :
01403 */
01404
01405 int DoDelta3(PHEAD WORD *term, WORD level)
01406 {
01407     GETBIDENTITY
01408     WORD *t, *m, *m1, *m2, *stopper, *tstop, *termout, *dels, *taken;
01409     WORD *ic, *jc, *factors;
01410     WORD num, num2, i, j, k, knum, a;
01411     AN.TeInFun = AR.TePos = 0;
01412     tstop = term + *term;
01413     stopper = tstop - ABS(tstop[-1]);
01414     t = term+1;
01415     while ( ( *t != DELTA3 || ((t[1]-FUNHEAD) & 1) != 0 ) && t < stopper )
01416         t += t[1];
01417     if ( t >= stopper ) {
01418         MLOCK(ErrorMessageLock);
01419         MesPrint("Internal error with dd_ function");
01420         MUNLOCK(ErrorMessageLock);
01421         Terminate(-1);
01422     }
01423     m1 = t; m2 = t + t[1];
01424     num = t[1] - FUNHEAD;
01425     if ( num == 0 ) {
01426         termout = t = AT.WorkPointer;
01427         m = term;
01428         while ( m < m1 ) *t++ = *m++;
01429         m = m2; while ( m < tstop ) *t++ = *m++;
01430         *termout = WORDDIF(t,termout);
01431         AT.WorkPointer = t;
01432         *AN.RepPoint = 1;
01433         AR.expchanged = 1;
01434         if ( Generator(BHEAD termout,level) ) {
01435             MLOCK(ErrorMessageLock);

```

```

01436         MesCall("Do dd_");
01437         MUNLOCK(ErrorMessageLock);
01438         SETERROR(-1)
01439     }
01440     AT.WorkPointer = termout;
01441     return(0);
01442 }
01443 t += FUNHEAD;
01444 /*
01445 Step 1: sort the arguments
01446 */
01447 for ( i = 1; i < num; i++ ) {
01448     if ( t[i] < t[i-1] ) {
01449         a = t[i]; t[i] = t[i-1]; t[i-1] = a;
01450         j = i - 1;
01451         while ( j > 0 ) {
01452             if ( t[j] >= t[j-1] ) break;
01453             a = t[j]; t[j] = t[j-1]; t[j-1] = a;
01454             j--;
01455         }
01456     }
01457 }
01458 /*
01459 Step 2: Order them by occurrence
01460 In 'taken' we have the array with the number of occurrences.
01461 in 'dels' is the type of object.
01462 */
01463 m = taken = AT.WorkPointer;
01464 for ( i = 0; i < num; i++ ) *m++ = 0;
01465 dels = m; knum = 0;
01466 for ( i = 0; i < num; knum++ ) {
01467     *m++ = t[i]; i++; taken[knum] = 1;
01468     while ( i < num ) {
01469         if ( t[i] != t[i-1] ) break;
01470         i++; (taken[knum])++;
01471     }
01472 }
01473 for ( i = 0; i < knum; i++ ) *m++ = taken[i];
01474 ic = m; num2 = num/2;
01475 jc = ic + num2;
01476 factors = jc + num2;
01477 termout = factors + num2;
01478 /*
01479 The recursion has num/2 steps
01480 */
01481 k = 0;
01482 while ( k >= 0 ) {
01483     if ( k >= num2 ) {
01484         t = termout; m = term;
01485         while ( m < m1 ) *t++ = *m++;
01486         *t++ = DELTA; *t++ = num+2;
01487         for ( i = 0; i < num2; i++ ) {
01488             *t++ = dels[ic[i]]; *t++ = dels[jc[i]];
01489         }
01490         for ( i = 0; i < num2; i++ ) {
01491             if ( ic[i] == jc[i] ) {
01492                 j = 1;
01493                 while ( i < num2-1 && ic[i] == ic[i+1] && ic[i] == jc[i+1] )
01494                     { i++; j++; }
01495                 for ( a = 1; a < j; a++ ) {
01496                     *t++ = SNUMBER; *t++ = 4; *t++ = 2*a+1; *t++ = 1;
01497                 }
01498                 for ( a = 0; a+1+i < num2; a++ ) {
01499                     if ( ic[a+i] != ic[a+i+1] ) break;
01500                 }
01501                 if ( a > 0 ) {
01502                     if ( GetBinom((UWORD *) (t+3), t+2, 2*j+a, a) ) {
01503                         MLOCK(ErrorMessageLock);
01504                         MesCall("Do dd_");
01505                         MUNLOCK(ErrorMessageLock);
01506                         SETERROR(-1)
01507                     }
01508                     t[1] = ( t[2] < 0 ? -t[2]: t[2] ) + 3;
01509                     *t = LNUMBER;
01510                     t += t[1];
01511                 }
01512             }
01513             else if ( factors[i] != 1 ) {
01514                 *t++ = SNUMBER; *t++ = 4; *t++ = factors[i]; *t++ = 1;
01515             }
01516         }
01517         for ( i = 0; i < num2-1; i++ ) {
01518             if ( ic[i] == jc[i] ) continue;
01519             j = 1;
01520             while ( i < num2-1 && jc[i] == jc[i+1] && ic[i] == ic[i+1] ) {
01521                 i++; j++;
01522             }

```

```

01523         for ( a = 0; a+i < num2-1; a++ ) {
01524             if ( ic[i+a] != ic[i+a+1] ) break;
01525         }
01526         if ( a > 0 ) {
01527             if ( GetBinom((UWORD *) (t+3),t+2,j+a,a) ) {
01528                 MLOCK(ErrorMessageLock);
01529                 MesCall("Do dd_");
01530                 MUNLOCK(ErrorMessageLock);
01531                 SETERROR(-1)
01532             }
01533             t[1] = ( t[2] < 0 ? -t[2]: t[2] ) + 3;
01534             *t = LNUMBER;
01535             t += t[1];
01536         }
01537     }
01538     m = m2;
01539     while ( m < tstop ) *t++ = *m++;
01540     *termout = WORDDIFF(t,termout);
01541     AT.WorkPointer = t;
01542     *AN.RepPoint = 1;
01543     AR.expchanged = 1;
01544     if ( Generator(BHEAD termout,level) ) {
01545         MLOCK(ErrorMessageLock);
01546         MesCall("Do dd_");
01547         MUNLOCK(ErrorMessageLock);
01548         SETERROR(-1)
01549     }
01550     k--;
01551     if ( k >= 0 ) goto nextj;
01552     else break;
01553 }
01554 for ( ic[k] = 0; ic[k] < knum; ic[k]++ ) {
01555     if ( taken[ic[k]] > 0 ) break;
01556 }
01557 if ( k > 0 && ic[k-1] == ic[k] ) jc[k] = jc[k-1];
01558 else jc[k] = ic[k];
01559 for ( ; jc[k] < knum; jc[k]++ ) {
01560     if ( taken[jc[k]] <= 0 ) continue;
01561     if ( ic[k] == jc[k] ) {
01562         if ( taken[jc[k]] <= 1 ) continue;
01563     /*
01564         factors[k] = taken[ic[k]];
01565         if ( ( factors[k] & 1 ) == 0 ) (factors[k])--;
01566     */
01567         taken[ic[k]] -= 2;
01568     }
01569     else {
01570         factors[k] = taken[jc[k]];
01571         (taken[ic[k]])--; (taken[jc[k]])--;
01572     }
01573     k++;
01574     goto nextk; /* This is the simulated recursion */
01575 nextj::
01576     (taken[ic[k]])++; (taken[jc[k]])++;
01577 }
01578 k--;
01579 if ( k >= 0 ) goto nextj;
01580 nextk::
01581 }
01582 AT.WorkPointer = taken;
01583 return(0);
01584 }
01585
01586 /*
01587 #] DoDelta3 :
01588 #[ TestPartitions :
01589
01590 Checks whether the function in tfun is a partitions_ function
01591 that can be expanded. If it can a number of relevant objects is
01592 inside the struct parti.
01593 This test is not entirely trivial because there are many restrictions
01594 w.r.t. the arguments.
01595 Syntax (still to be implemented)
01596 partitions_(number_of_partition_entries,[function,number,]^nope,arguments)
01597 [function,number,]: can be
01598     f,3     for a partition of 3 arguments
01599     f,0     for the remaining arguments (should be last)
01600     num1,f,num2 with num1 effectively a number of partitions but this
01601                counts as num1 entries.
01602     0,f,num2: all partitions have num2 arguments. No number of partition
01603                entries needed. If num2 does not divide the number of
01604                arguments there will be no action.
01605 */
01606
01607 int TestPartitions(WORD *tfun, PARTI *parti)
01608 {
01609     WORD *tnext = tfun + tfun[1];

```

```

01610     WORD *t, *tt;
01611     WORD argcount = 0, sum = 0, i, ipart, argremain;
01612     WORD tensorflag = 0;
01613     parti->psize = parti->nfun = parti->args = parti->nargs = 0;
01614     parti->numargs = parti->numpart = parti->where = 0;
01615     tt = t = tfun + FUNHEAD;
01616     while ( t < tnext ) { argcount++; NEXTARG(t); }
01617     if ( argcount < 1 ) goto No;
01618     t = tt;
01619     if ( *t != -SNUMBER ) goto No;
01620     if ( t[1] == 0 ) {
01621         t += 2;
01622         if ( *t <= -FUNCTION && t[1] == -SNUMBER && t[2] > 0 ) {
01623             if ( functions[-*t-FUNCTION].spec > 0 ) tensorflag = 1;
01624             if ( argcount-3 < 0 ) goto No;
01625             if ( ( argcount-3 ) % t[2] != 0 ) goto No;
01626         }
01627         else goto No;
01628         parti->numpart = (argcount-3)/t[2];
01629         parti->numargs = argcount - 3;
01630         parti->psize = (WORD *)Malloc1((parti->numpart*2+parti->numargs*2+2)
01631             *sizeof(WORD),"partitions");
01632         parti->nfun = parti->psize + parti->numpart;
01633         parti->args = parti->nfun + parti->numpart;
01634         parti->nargs = parti->args + parti->numargs;
01635         for ( i = 0; i < parti->numpart; i++ ) {
01636             parti->psize[i] = t[2];
01637             parti->nfun[i] = -t[0];
01638         }
01639         t += 3;
01640     }
01641     else if ( t[1] > 0 ) { /* Number of partitions */
01642 /*
01643     We can have sequences of function,number for one partition
01644     or number1,function,number2 for number1 partitions of size number2.
01645     The last partition can have number=0. It must be a single partition
01646     and it will take all remaining arguments.
01647     If any of the functions is a tensor, all arguments must be either
01648     vector or index.
01649 */
01650     parti->numpart = t[1]; t += 2;
01651     ipart = sum = 0; argremain = argcount - 1;
01652 /*
01653     At this point is seems better to make an allocation already that
01654     may be too big. The alternative is having to pass this code twice.
01655 */
01656     parti->psize = (WORD *)Malloc1((argcount*4+2)*sizeof(WORD),"partitions");
01657     parti->nfun = parti->psize+argcount;
01658     parti->args = parti->nfun+argcount;
01659     parti->nargs = parti->args+argcount;
01660     while ( ipart < parti->numpart ) {
01661         if ( *t <= -FUNCTION && t[1] == -SNUMBER && t[2] >= 0 ) {
01662             if ( functions[-*t-FUNCTION].spec > 0 ) tensorflag = 1;
01663             if ( t[2] == 0 ) {
01664                 if ( ipart+1 != parti->numpart ) goto WhatAPity;
01665                 argremain -= 2;
01666                 parti->nfun[ipart] = -*t;
01667                 parti->psize[ipart++] = argremain-sum;
01668                 ipart++;
01669                 sum = argremain;
01670             }
01671             else {
01672                 parti->nfun[ipart] = -*t;
01673                 parti->psize[ipart++] = t[2];
01674                 argremain -= 2;
01675                 sum += t[2];
01676             }
01677             t += 3;
01678         }
01679         else if ( *t == -SNUMBER && t[1] > 0 && ipart+t[1] <= parti->numpart
01680             && t[2] <= -FUNCTION && t[3] == -SNUMBER && t[4] > 0 ) {
01681             if ( functions[-t[2]-FUNCTION].spec > 0 ) tensorflag = 1;
01682             argremain -= 3;
01683             for ( i = 0; i < t[1]; i++ ) {
01684                 parti->nfun[ipart] = -t[2];
01685                 parti->psize[ipart++] = t[4];
01686                 sum += t[4];
01687             }
01688             if ( sum > argremain ) goto WhatAPity;
01689             t += 5;
01690         }
01691         else goto WhatAPity;
01692     }
01693     if ( sum != argremain ) goto WhatAPity;
01694     parti->numargs = argremain;
01695 }
01696 else goto No;

```

```

01697 /*
01698     Now load the offsets of the arguments and check if needed whether OK with tensor
01699 */
01700 for ( i = 0; i < parti->numargs; i++ ) {
01701     parti->args[i] = t - tfun;
01702     if ( tensorflag && ( *t != -VECTOR && *t != -INDEX ) ) goto WhatAPity;
01703     NEXTARG(t);
01704 }
01705 return(1);
01706 WhatAPity:
01707     M_free(parti->psize,"partitions");
01708     parti->psize = parti->nfun = parti->args = parti->nargs = 0;
01709     parti->numargs = parti->numpart = parti->where = 0;
01710 No:
01711     return(0);
01712 }
01713
01714 /*
01715     #] TestPartitions :
01716     #[ DoPartitions :
01717
01718     As we have only one AT.partitions we need to copy it locally
01719     if we keep needing it.
01720 */
01721
01722 int DoPartitions(PHEAD WORD *term, WORD level)
01723 {
01724     WORD x, i, j, im, *fun, ndiff, siz, tensorflag = 0;
01725     PARTI part = AT.partitions;
01726     WORD *array, **j3, **j3fill, **j3where;
01727     WORD a, pfill, *j2, *j2fill, j3size, ncoeff, ncoeffnum, nfac, ncoeff2, ncoeff3, n;
01728     UWORD *coeff, *coeffnum, *cfac, *coeff2, *coeff3, *c;
01729     /* Make AT.partitions ready for future use (if there is another function) */
01730     AT.partitions.psize = AT.partitions.nfun = AT.partitions.args = AT.partitions.nargs = 0;
01731     AT.partitions.numargs = AT.partitions.numpart = AT.partitions.where = 0;
01732 /*
01733     Start with bubble sorting the list of arguments. And the list of partitions.
01734 */
01735     fun = term + part.where;
01736     if ( functions[*fun-FUNCTION].spec > 0 ) tensorflag = 1;
01737     for ( i = 1; i < part.numargs; i++ ) {
01738         for ( j = i-1; j >= 0; j-- ) {
01739             if ( CompArg(fun+part.args[j+1],fun+part.args[j]) >= 0 ) break;
01740             x = part.args[j+1]; part.args[j+1] = part.args[j]; part.args[j] = x;
01741         }
01742     }
01743     for ( i = 1; i < part.numpart; i++ ) {
01744         for ( j = i-1; j >= 0; j-- ) {
01745             if ( part.psize[j+1] < part.psize[j] ) break;
01746             if ( part.psize[j+1] == part.psize[j] && part.nfun[j+1] <= part.nfun[j] ) break;
01747             x = part.psize[j+1]; part.psize[j+1] = part.psize[j]; part.psize[j] = x;
01748             x = part.nfun[j+1]; part.nfun[j+1] = part.nfun[j]; part.nfun[j] = x;
01749         }
01750     }
01751 /*
01752     Now we have the partitions sorted from high to low and the arguments
01753     have been sorted the regular way arguments are sorted in a symmetrize.
01754     The important thing is that identical arguments are adjacent.
01755     Assign the numbers (identical arguments have identical numbers).
01756 */
01757     ndiff = 1; part.nargs[0] = ndiff;
01758     for ( i = 1; i < part.numargs; i++ ) {
01759         if ( CompArg(fun+part.args[i],fun+part.args[i-1]) != 0 ) ndiff++;
01760         part.nargs[i] = ndiff;
01761     }
01762     part.nargs[part.numargs] = 0;
01763     coeffnum = NumberMalloc("partitionsn");
01764     coeff = NumberMalloc("partitions");
01765     coeff2 = NumberMalloc("partitions2");
01766     coeff3 = NumberMalloc("partitions3");
01767     cfac = NumberMalloc("partitions!");
01768     ncoeffnum = 1; coeffnum[0] = 1;
01769 /*
01770     The numerator of the coefficient will be n1!*n2!*...*n(ndiff)!
01771     We compute it only once.
01772 */
01773     j = 0;
01774     for ( i = 1; i <= ndiff; i++ ) {
01775         n = 0;
01776         while ( part.nargs[j] == i ) { n++; j++; }
01777         if ( n > 1 ) { /* 1! needs no attention */
01778             if ( Factorial(BHEAD n, cfac, &nfac) ) Terminate(-1);
01779             if ( MulLong(coeffnum,ncoeffnum,cfac,nfac,coeff2,&ncoeff2) ) Terminate(-1);
01780             c = coeffnum; coeffnum = coeff2; coeff2 = c;
01781             n = ncoeffnum; ncoeffnum = ncoeff2; ncoeff2 = n;
01782         }
01783     }

```

```

01784 /*
01785     Now comes the part where we have to make sure that
01786     a: we generate all partitions.
01787     b: we generate only different partitions.
01788     c: we get the proper combinatorics factor.
01789     Method:
01790     Suppose the largest partition needs n objects and there are m partitions.
01791     We allocate m arrays of n 'digits'. Make in the smaller partitions the
01792     appropriate leading digits zero.
01793     Divide the largest numbers (of the arguments) over the partitions as
01794     leftmost digits (after possible zeroes). The arrays, seen as numbers,
01795     should be such that each is less or equal to its left neighbour. Take the
01796     next largest numbers, etc. This generates unique partitions and all of
01797     them. Because we have a formula for the multiplicity, this should do it.
01798
01799     The general case. At a later stage we might put in a more economical
01800     version for special cases.
01801 */
01802     siz = part.psize[0];
01803     j3size = 2*(part.numpart+1)+2*(part.numargs+1);
01804     array = (WORD *)Malloc1((part.numpart+1)*siz*sizeof(WORD), "parts");
01805     j3 = (WORD **)Malloc1(j3size*sizeof(WORD *), "parts3");
01806     j2 = (WORD *)Malloc1((part.numpart+part.numargs+2)*sizeof(WORD), "parts2");
01807     j3fill = j3+(part.numpart+1);
01808     j3where = j3fill+(part.numpart+1);
01809     for ( i = 0; i < j3size; i++ ) j3[i] = 0;
01810     j2fill = j2+(part.numpart+1);
01811     for ( i = 0; i < part.numargs; i++ ) j2fill[i] = 0;
01812     for ( i = 0; i < part.numpart; i++ ) {
01813         j3[i] = array+i*siz;
01814         for ( j = 0; j < siz; j++ ) j3[i][j] = 0;
01815         j3fill[i] = j3[i]+(siz-part.psize[i]);
01816         j2[i] = part.psize[i]; /* Number of places still available */
01817     }
01818     j3[part.numpart] = array+part.numpart*siz;
01819     j2[part.numpart] = 0;
01820 /*
01821     Now comes a complicated two-level recursion in a and pfill.
01822 */
01823     a = part.numargs-1;
01824     pfill = 0;
01825 /*
01826     We start putting the last number in part.nargs in the first partition in array.
01827     For backtracking we need to know where we put this number. Hence j3where.
01828 */
01829     while ( a < part.numargs ) {
01830         while ( j2[pfill] <= 0 ) {
01831             pfill++;
01832             while ( pfill >= part.numpart ) { /* we have to pop */
01833                 a++;
01834                 if ( a >= part.numargs ) goto Done;
01835                 pfill = j2fill[a];
01836                 j2[pfill]++;
01837                 j3where[a][0] = 0;
01838                 j3fill[pfill]--;
01839                 pfill++;
01840             }
01841         }
01842         j3where[a] = j3fill[pfill];
01843         *(j3fill[pfill])++ = part.nargs[a];
01844         j2[pfill]--; j2fill[a] = pfill;
01845 /*
01846         Now test whether this is allowed.
01847 */
01848         if ( pfill > 0 && part.psize[pfill] == part.psize[pfill-1]
01849             && part.nfun[pfill] == part.nfun[pfill-1] ) { /* First check whether allowed */
01850             for ( im = 0; im < siz; im++ ) {
01851                 if ( j3[pfill-1][im] < j3[pfill][im] ) break;
01852                 if ( j3[pfill-1][im] > j3[pfill][im] ) im = siz;
01853             }
01854             if ( im < siz ) { /* not ordered. undo and raise pfill */
01855                 pfill = j2fill[a];
01856                 j2[pfill]++;
01857                 j3where[a][0] = 0;
01858                 j3fill[pfill]--;
01859                 pfill++;
01860                 continue; /* Note that j2[part.numpart] = 0 */
01861             }
01862         }
01863         a--;
01864         if ( a < 0 ) { /* Solution */
01865 /*
01866             #[ Solution :
01867
01868             Now we compose the output term. The input term contains
01869             three parts: head, partitions_, tail.
01870             partitions_ starts at term+part.where.

```

```

01871      We first put the function parts and worry about the coefficient later.
01872  */
01873      WORD *t, *to, *twhere = term+part.where, *t2, *tend = term+term, *termout;
01874      WORD num, jj, *targ, *tfun;
01875      t2 = twhere+twhere[1];
01876      to = termout = AT.WorkPointer;
01877      if ( termout + *term + part.numpart*FUNHEAD + AM.MaxTal >= AT.WorkTop ) {
01878          MesWork();
01879      }
01880      for ( i = 0; i < ncoeffnum; i++ ) coeff[i] = coeffnum[i];
01881      ncoeff = ncoeffnum;
01882      t = term; while ( t < twhere ) *to++ = *t++;
01883  /*
01884      Now the partitions
01885  */
01886      for ( i = 0; i < part.numpart; i++ ) {
01887          tfun = to;
01888          *to++ = part.nfun[i]; to++; FILLFUN(to);
01889          for ( j = 1; j <= part.psize[i]; j++ ) {
01890              num = j3[i][siz-j]; /* now we need an argument with this number */
01891              for ( jj = num-1; jj < part.numargs; jj++ ) {
01892                  if ( part.nargs[jj] == num ) break;
01893              }
01894              targ = part.args[jj]+twhere;
01895              if ( *targ < 0 ) {
01896                  if ( tensorflag ) targ++;
01897                  else if ( *targ > -FUNCTION ) *to++ = *targ++;
01898                  *to++ = *targ++;
01899              }
01900              else { jj = *targ; NCOPY(to,targ,jj); }
01901          }
01902          tfun[1] = to - tfun;
01903      }
01904  /*
01905      Now the denominators of the coefficient
01906      First identical functions/partitions
01907  */
01908      j = 1; n = 1;
01909      while ( j < part.numpart ) {
01910          for ( im = 0; im < siz; im++ ) {
01911              if ( part.nfun[j-1] != part.nfun[j] ) break;
01912              if ( j3[j-1][im] < j3[j][im] ) break;
01913              if ( j3[j-1][im] > j3[j][im] ) im = 2*siz+2;
01914          }
01915          if ( im == siz ) { n++; j++; continue; }
01916          if ( n > 1 ) {
01917              div1:
01918                  if ( Factorial(BHEAD n, cfac, &nfac) ) Terminate(-1);
01919                  if ( DivLong(coeff,ncoeff,cfac,nfac,coeff2,&ncoeff2,coeff3,&ncoeff3) )
01920                      Terminate(-1);
01921                  c = coeff; coeff = coeff2; coeff2 = c;
01922                  n = ncoeff; ncoeff = ncoeff2; ncoeff2 = n;
01923              }
01924              n = 1; j++;
01925          }
01926          if ( n > 1 ) goto div1;
01927  /*
01928      Now identical elements inside the partitions
01929  */
01930      for ( i = 0; i < part.numpart; i++ ) {
01931          j = 0; while ( j3[i][j] == 0 ) j++;
01932          n = 1; j++;
01933          while ( j < siz ) {
01934              if ( j3[i][j-1] == j3[i][j] ) { n++; j++; }
01935              else {
01936                  if ( n > 1 ) {
01937                      div2:
01938                          if ( Factorial(BHEAD n, cfac, &nfac) ) Terminate(-1);
01939                          if ( DivLong(coeff,ncoeff,cfac,nfac,coeff2,&ncoeff2,coeff3,&ncoeff3) )
01940                              Terminate(-1);
01941                          c = coeff; coeff = coeff2; coeff2 = c;
01942                          n = ncoeff; ncoeff = ncoeff2; ncoeff2 = n;
01943                      }
01944                      n = 1; j++;
01945                  }
01946                  if ( n > 1 ) goto div2;
01947              }
01948          }
01949          if ( ncoeff != 1 || coeff[0] > 1 ) {
01950              if ( ncoeff == 1 && coeff[0] <= MAXPOSITIVE ) {
01951                  *to++ = SNUMBER; *to++ = 4; *to++ = (WORD)(coeff[0]); *to++ = 1;
01952              }
01953              else {
01954                  *to++ = LNUMBER; *to++ = ncoeff+3; *to++ = ncoeff;
01955                  for ( i = 0; i < ncoeff; i++ ) *to++ = ((WORD *)coeff)[i];
01956              }
01957          }
01958      }
01959  }

```

```

01956         }
01957     /*
01958         And the tail
01959     */
01960     while ( t2 < tend ) *to++ = *t2++;
01961     *termout = to-termout;
01962     AT.WorkPointer = to;
01963     if ( Generator(BHEAD termout,level) ) Terminate(-1);
01964     AT.WorkPointer = termout;
01965 /*
01966     #] Solution :
01967
01968     Now we can pop all a with the lowest value and one more.
01969 */
01970     a = 0;
01971     while ( part.nargs[a] == 1 ) {
01972         pfill = j2fill[a]; j2[pfill]++; j3where[a][0] = 0; j3fill[pfill]--; a++;
01973     }
01974     if ( a < part.numargs ) {
01975         pfill = j2fill[a]; j2[pfill]++; j3where[a][0] = 0; j3fill[pfill]--; a++;
01976     }
01977     a--;
01978     pfill++;
01979 }
01980 else if ( part.nargs[a] == part.nargs[a+1] ) {}
01981 else { pfill = 0; }
01982 }
01983 Done:
01984     M_free(j2,"parts2");
01985     M_free(j3,"parts3");
01986     M_free(array,"parts");
01987     NumberFree(cfac,"partitions!");
01988     NumberFree(coeff3,"partitions3");
01989     NumberFree(coeff2,"partitions2");
01990     NumberFree(coeff,"partitions");
01991     NumberFree(coeffnum,"partitionsn");
01992     M_free(part.psize,"partitions");
01993     part.psize = part.nfun = part.args = part.nargs = 0;
01994     part.numargs = part.numpart = part.where = 0;
01995     return(0);
01996 }
01997
01998 /*
01999     #] DoPartitions :
02000     #] Distribute :
02001     #[ DoPermutations :
02002
02003     Routine replaces the function perm_(f,args) by occurrences of f with
02004     all permutations of the args. This should always fit!
02005 */
02006
02007 int DoPermutations(PHEAD WORD *term, WORD level)
02008 {
02009     PERMP perm;
02010     WORD *oldworkpointer = AT.WorkPointer, *termout = AT.WorkPointer;
02011     WORD *t, *tstop, *tt, *ttstop, odd = 0;
02012     WORD *args[MAXMATCH], nargs, i, first, skip, *to, *from;
02013 /*
02014     Find function and count arguments. Check for odd/even
02015 */
02016     tstop = term+*term; tstop -= ABS(tstop[-1]);
02017     t = term+1;
02018     while ( t < tstop ) {
02019         if ( *t == PERMUTATIONS ) {
02020             if ( t[1] >= FUNHEAD+1 && t[FUNHEAD] <= -FUNCTION ) {
02021                 odd = 0; skip = 1;
02022             }
02023             else if ( t[1] >= FUNHEAD+3 && t[FUNHEAD] == -SNUMBER && t[FUNHEAD+2] <= -FUNCTION ) {
02024                 if ( t[FUNHEAD+1] % 2 == 1 ) odd = -1;
02025                 else odd = 0;
02026                 skip = 3;
02027             }
02028             else { t += t[1]; continue; }
02029             tt = t+FUNHEAD+skip; ttstop = t + t[1];
02030             nargs = 0;
02031             while ( tt < ttstop ) { NEXTARG(tt); nargs++; }
02032             tt = t+FUNHEAD+skip;
02033             if ( nargs > MAXMATCH ) {
02034                 MLOCK(ErrorMessageLock);
02035                 MesPrint("Too many arguments in function perm_. %d! is way too big", (WORD)MAXMATCH);
02036                 MUNLOCK(ErrorMessageLock);
02037                 SETERROR(-1)
02038             }
02039             i = 0;
02040             while ( tt < ttstop ) { args[i++] = tt; NEXTARG(tt); }
02041             perm.n = nargs;
02042             perm.sign = 0;

```



```

02043     perm.objects = args;
02044     first = 1;
02045     while ( (first = PermuteP(&perm,first) ) == 0 ) {
02046 /*
02047         Compose the output term
02048 */
02049         to = termout; from = term;
02050         while ( from < t ) *to++ = *from++;
02051         *to++ = -t[FUNHEAD+skip-1];
02052         *to++ = t[1] - skip;
02053         for ( i = 2; i < FUNHEAD; i++ ) *to++ = t[i];
02054         for ( i = 0; i < nargs; i++ ) {
02055             from = args[i];
02056             COPY1ARG(to,from);
02057         }
02058         from = t+t[1];
02059         tstop = term + *term;
02060         while ( from < tstop ) *to++ = *from++;
02061         if ( odd && ( ( perm.sign & 1 ) != 0 ) ) to[-1] = -to[-1];
02062         *termout = to - termout;
02063         AT.WorkPointer = to;
02064         if ( Generator(BHEAD termout,level) ) Terminate(-1);
02065         AT.WorkPointer = oldworkpointer;
02066     }
02067     return(0);
02068 }
02069 t += t[1];
02070 }
02071 return(0);
02072 }
02073
02074 /*
02075 #] DoPermutations :
02076 #[ DoShuffle :
02077
02078 Merges the arguments of all occurrences of function fun into a
02079 single occurrence of fun. The opposite of Distrib_
02080 Syntax:
02081     Shuffle[,once|all],fun;
02082     Shuffle[,once|all],$fun;
02083 The expansion of the dollar should give a single function.
02084 The dollar is indicated as usual with a negative value.
02085 option = 1 (once): generate identical results only once
02086 option = 0 (all): generate identical results with combinatorics (default)
02087 */
02088
02089 /*
02090 We use the Shuffle routine which has a large amount of combinatorics.
02091 It doesn't have grouped combinatorics as in (0,1,2)*(0,1,3) where the
02092 groups (0,1) also cause double terms.
02093 */
02094
02095 int DoShuffle(WORD *term, WORD level, WORD fun, WORD option)
02096 {
02097     GETIDENTITY
02098     SHvariables SHback, *SH = &(AN.SHvar);
02099     WORD *t1, *t2, *tstop, ncoef, n = fun, *to, *from;
02100     int i, error;
02101     LONG k;
02102     UWORD *newcombi;
02103
02104     if ( n < 0 ) {
02105         if ( ( n = DolToFunction(BHEAD -n) ) == 0 ) {
02106             MLOCK(ErrorMessageLock);
02107             MesPrint("$-variable in merge statement did not evaluate to a function.");
02108             MUNLOCK(ErrorMessageLock);
02109             return(1);
02110         }
02111     }
02112     if ( AT.WorkPointer + 3*(*term) + AM.MaxTal > AT.WorkTop ) {
02113         MLOCK(ErrorMessageLock);
02114         MesWork();
02115         MUNLOCK(ErrorMessageLock);
02116         return(-1);
02117     }
02118
02119     tstop = term + *term;
02120     ncoef = tstop[-1];
02121     tstop -= ABS(ncoef);
02122     t1 = term + 1;
02123     while ( t1 < tstop ) {
02124         if ( ( *t1 == n ) && ( t1+t1[1] < tstop ) && ( t1[1] > FUNHEAD ) ) {
02125             t2 = t1 + t1[1];
02126             if ( t2 >= tstop ) {
02127                 return(Generator(BHEAD term,level));
02128             }
02129             while ( t2 < tstop ) {

```

```

02130         if ( ( *t2 == n ) && ( t2[1] > FUNHEAD ) ) break;
02131         t2 += t2[1];
02132     }
02133     if ( t2 < tstop ) break;
02134 }
02135 t1 += t1[1];
02136 }
02137 if ( t1 >= tstop ) {
02138     return(Generator(BHEAD term,level));
02139 }
02140 *AN.RepPoint = 1;
02141 /*
02142 Now we have two occurrences of the function.
02143 Back up all relevant variables and load all the stuff that needs to be
02144 passed on.
02145 */
02146 SHback = AN.SHvar;
02147 SH->finishuf = &FinishShuffle;
02148 SH->do_uffle = &DoShuffle;
02149 SH->outterm = AT.WorkPointer;
02150 AT.WorkPointer += *term;
02151 SH->stop1 = t1 + t1[1];
02152 SH->stop2 = t2 + t2[1];
02153 SH->thefunction = n;
02154 SH->option = option;
02155 SH->level = level;
02156 SH->incoef = tstop;
02157 SH->nincoef = ncoef;
02158
02159 if ( AN.SHcombi == 0 || AN.SHcombisize == 0 ) {
02160     AN.SHcombisize = 200;
02161     AN.SHcombi = (UWORD *)Malloca(AN.SHcombisize*sizeof(UWORD), "AN.SHcombi");
02162     SH->combilast = 0;
02163     SHback.combilast = 0;
02164 }
02165 else {
02166     SH->combilast += AN.SHcombi[SH->combilast]+1;
02167     if ( SH->combilast >= AN.SHcombisize - 100 ) {
02168         newcombi = (UWORD *)Malloca(2*AN.SHcombisize*sizeof(UWORD), "AN.SHcombi");
02169         for ( k = 0; k < AN.SHcombisize; k++ ) newcombi[k] = AN.SHcombi[k];
02170         M_free(AN.SHcombi, "AN.SHcombi");
02171         AN.SHcombi = newcombi;
02172         AN.SHcombisize *= 2;
02173     }
02174 }
02175 AN.SHcombi[SH->combilast] = 1;
02176 AN.SHcombi[SH->combilast+1] = 1;
02177
02178 i = t1-term; to = SH->outterm; from = term;
02179 NCOPY(to,from,i)
02180 SH->outfun = to;
02181 for ( i = 0; i < FUNHEAD; i++ ) { *to++ = t1[i]; }
02182
02183 error = Shuffle(t1+FUNHEAD,t2+FUNHEAD,to);
02184
02185 AT.WorkPointer = SH->outterm;
02186 AN.SHvar = SHback;
02187 if ( error ) {
02188     MesCall("DoShuffle");
02189     return(-1);
02190 }
02191 return(0);
02192 }
02193
02194 /*
02195 #] DoShuffle :
02196 #[ Shuffle :
02197
02198 How to make shuffles:
02199
02200 We have two lists of arguments. We have to make a single
02201 shuffle of them. All combinations. Doubles should have as
02202 much as possible a combinatorics factor. Sometimes this is
02203 very difficult as in:
02204 (0,1,2)x(0,1,3) = -> (0,1) is a repeated pattern and the
02205 factor on that is difficult
02206 Simple way: (without combinatorics)
02207 repeat id f0(?c)*f(x1?,?a)*f(x2?,?b) =
02208         +f0(?c,x1)*f(?a)*f(x2,?b)
02209         +f0(?c,x2)*f(x1,?a)*f(?b);
02210
02211 Refinement:
02212 if ( x1 == x2 ) check how many more there are of the same.
02213 --> (n1,x) and (n2,x)
02214 id f0(?c)*f1((n1,x),?b)*f2((n2,x),?c) =
02215         +binom_(n1+n2,n1)*f0(?c,(n1+n2,x))*f1(?a)*f2(?b)
02216         +sum_(j,0,n1-1,binom_(n2+j,j)*f0(?c,(j+n2,x))
02217             *f1((n1-j),?a)*f2(?b))*force2

```

```

02217             +sum_(j,0,n2-1,binom_(n1+j,j)*f0(?c,(j+n1,x))
02218                 *f1(?a)*f2((n2-j),?b))*force1
02219     The force operation can be executed directly
02220
02221     The next question is how to program this: recursively or linearly
02222     which would require simulation of a recursion. Recursive is clearest
02223     but we need to pass a number of arguments from the calling routine
02224     to the final routine. This is done with AN.SHvar.
02225
02226     We need space for the accumulation of the combinatoric factors.
02227 */
02228
02229 int Shuffle(WORD *from1, WORD *from2, WORD *to)
02230 {
02231     GETIDENTITY
02232     WORD *t, *fr, *next1, *next2, na, *fn1, *fn2, *tt;
02233     int i, n, n1, n2, j;
02234     LONG combilast;
02235     SHvariables *SH = &(AN.SHvar);
02236     if ( from1 == SH->stop1 && from2 == SH->stop2 ) {
02237         return(FiniShuffle(to));
02238     }
02239     else if ( from1 == SH->stop1 ) {
02240         i = SH->stop2 - from2; t = to; tt = from2; NCOPY(t,tt,i)
02241         return(FiniShuffle(t));
02242     }
02243     else if ( from2 == SH->stop2 ) {
02244         i = SH->stop1 - from1; t = to; tt = from1; NCOPY(t,tt,i)
02245         return(FiniShuffle(t));
02246     }
02247 /*
02248     Compare lead arguments
02249 */
02250     if ( AreArgsEqual(from1,from2) ) {
02251 /*
02252         First find out how many of each
02253 */
02254         next1 = from1; n1 = 1; NEXTARG(next1)
02255         while ( ( next1 < SH->stop1 ) && AreArgsEqual(from1,next1) ) {
02256             n1++; NEXTARG(next1)
02257         }
02258         next2 = from2; n2 = 1; NEXTARG(next2)
02259         while ( ( next2 < SH->stop2 ) && AreArgsEqual(from2,next2) ) {
02260             n2++; NEXTARG(next2)
02261         }
02262         combilast = SH->combilast;
02263 /*
02264         +binom_(n1+n2,n1)*f0(?c,(n1+n2,x))*f1(?a)*f2(?b)
02265 */
02266         t = to;
02267         n = n1 + n2;
02268         while ( --n >= 0 ) { fr = from1; CopyArg(t,fr) }
02269         if ( GetBinom((UWORD *) (t),&na,n1+n2,n1) ) goto shuffcall;
02270         if ( combilast + AN.SHcombi[combilast] + na + 2 >= AN.SHcombisize ) {
02271 /*
02272             We need more memory in this stack. Fortunately this is the
02273             only place where we have to do this, because the other factors
02274             are definitely smaller.
02275             Layout:  size, LongInteger, size, LongInteger, ....
02276             We start pointing at the last one.
02277 */
02278             UWORD *combi = (UWORD *)Malloca(2*AN.SHcombisize*2,"AN.SHcombi");
02279             LONG jj;
02280             for ( jj = 0; jj < AN.SHcombisize; jj++ ) combi[jj] = AN.SHcombi[jj];
02281             AN.SHcombisize *= 2;
02282             M_free(AN.SHcombi,"AN.SHcombi");
02283             AN.SHcombi = combi;
02284         }
02285         if ( MulLong((UWORD *) (AN.SHcombi+combilast+1),AN.SHcombi[combilast],
02286             (UWORD *) (t),na,
02287             (UWORD *) (AN.SHcombi+combilast+AN.SHcombi[combilast]+2),
02288             (WORD *) (AN.SHcombi+combilast+AN.SHcombi[combilast]+1)) ) goto shuffcall;
02289         SH->combilast = combilast + AN.SHcombi[combilast] + 1;
02290         if ( next1 >= SH->stop1 ) {
02291             fr = next2; i = SH->stop2 - fr;
02292             NCOPY(t,fr,i)
02293             if ( FiniShuffle(t) ) goto shuffcall;
02294         }
02295         else if ( next2 >= SH->stop2 ) {
02296             fr = next1; i = SH->stop1 - fr;
02297             NCOPY(t,fr,i)
02298             if ( FiniShuffle(t) ) goto shuffcall;
02299         }
02300         else {
02301             if ( Shuffle(next1,next2,t) ) goto shuffcall;
02302         }
02303         SH->combilast = combilast;

```

```

02304 /*
02305         +sum_(j,0,n1-1,binom_(n2+j,j)*f0(?c,(j+n2,x))
02306         *f1((n1-j),?a)*f2(?b))*force2
02307 */
02308     if ( next2 < SH->stop2 ) {
02309         t = to;
02310         n = n2;
02311         while ( --n >= 0 ) { fr = from1; CopyArg(t,fr) }
02312         for ( j = 0; j < n1; j++ ) {
02313             if ( GetBinom((UWORD *) (t), &na, n2+j, j) ) goto shuffcall;
02314             if ( Mullong((UWORD *) (AN.SHcombi+combilast+1), AN.SHcombi[combilast],
02315                 (UWORD *) (t), na,
02316                 (UWORD *) (AN.SHcombi+combilast+AN.SHcombi[combilast]+2),
02317                 (WORD *) (AN.SHcombi+combilast+AN.SHcombi[combilast]+1)) ) goto shuffcall;
02318             SH->combilast = combilast + AN.SHcombi[combilast] + 1;
02319             if ( j > 0 ) { fr = from1; CopyArg(t,fr) }
02320             fn2 = next2; tt = t;
02321             CopyArg(tt,fn2)
02322         }
02323         if ( fn2 >= SH->stop2 ) {
02324             n = n1-j;
02325             while ( --n >= 0 ) { fr = from1; CopyArg(tt,fr) }
02326             fr = next1; i = SH->stop1 - fr;
02327             NCOPY(tt,fr,i)
02328             if ( FiniShuffle(tt) ) goto shuffcall;
02329         }
02330         else {
02331             n = j; fn1 = from1; while ( --n >= 0 ) { NEXTARG(fn1) }
02332             if ( Shuffle(fn1,fn2,tt) ) goto shuffcall;
02333         }
02334         SH->combilast = combilast;
02335     }
02336 }
02337 /*
02338         +sum_(j,0,n2-1,binom_(n1+j,j)*f0(?c,(j+n1,x))
02339         *f1(?a)*f2((n2-j),?b))*force1
02340 */
02341     if ( next1 < SH->stop1 ) {
02342         t = to;
02343         n = n1;
02344         while ( --n >= 0 ) { fr = from1; CopyArg(t,fr) }
02345         for ( j = 0; j < n2; j++ ) {
02346             if ( GetBinom((UWORD *) (t), &na, n1+j, j) ) goto shuffcall;
02347             if ( Mullong((UWORD *) (AN.SHcombi+combilast+1), AN.SHcombi[combilast],
02348                 (UWORD *) (t), na,
02349                 (UWORD *) (AN.SHcombi+combilast+AN.SHcombi[combilast]+2),
02350                 (WORD *) (AN.SHcombi+combilast+AN.SHcombi[combilast]+1)) ) goto shuffcall;
02351             SH->combilast = combilast + AN.SHcombi[combilast] + 1;
02352             if ( j > 0 ) { fr = from1; CopyArg(t,fr) }
02353             fn1 = next1; tt = t;
02354             CopyArg(tt,fn1)
02355         }
02356         if ( fn1 >= SH->stop1 ) {
02357             n = n2-j;
02358             while ( --n >= 0 ) { fr = from1; CopyArg(tt,fr) }
02359             fr = next2; i = SH->stop2 - fr;
02360             NCOPY(tt,fr,i)
02361             if ( FiniShuffle(tt) ) goto shuffcall;
02362         }
02363         else {
02364             n = j; fn2 = from2; while ( --n >= 0 ) { NEXTARG(fn2) }
02365             if ( Shuffle(fn1,fn2,tt) ) goto shuffcall;
02366         }
02367         SH->combilast = combilast;
02368     }
02369 }
02370 }
02371 else {
02372 /*
02373     Argument from first list
02374 */
02375     t = to;
02376     fr = from1;
02377     CopyArg(t,fr)
02378     if ( fr >= SH->stop1 ) {
02379         fr = from2; i = SH->stop2 - fr;
02380         NCOPY(t,fr,i)
02381         if ( FiniShuffle(t) ) goto shuffcall;
02382     }
02383     else {
02384         if ( Shuffle(fr,from2,t) ) goto shuffcall;
02385     }
02386 /*
02387     Argument from second list
02388 */
02389     t = to;
02390     fr = from2;

```

```

02391     CopyArg(t,fr)
02392     if ( fr >= SH->stop2 ) {
02393         fr = from1; i = SH->stop1 - fr;
02394         NCOPY(t,fr,i)
02395         if ( FiniShuffle(t) ) goto shuffcall;
02396     }
02397     else {
02398         if ( Shuffle(from1,fr,t) ) goto shuffcall;
02399     }
02400 }
02401 return(0);
02402 shuffcall:
02403 MesCall("Shuffle");
02404 return(-1);
02405 }
02406
02407 /*
02408 #] Shuffle :
02409 #[ FinishShuffle :
02410
02411 The complications here are:
02412 1: We want to save space. We put the output term in 'out' straight
02413    on top of what we produced thusfar. We have to copy the early
02414    piece because once the term goes back to Generator, Normalize can
02415    change it in situ
02416 2: There can be other occurrence of the function between the two
02417    that we did. For shuffles that isn't likely, but we use this
02418    routine also for the stuffles and there it can happen.
02419 */
02420
02421 int FinishShuffle(WORD *fini)
02422 {
02423     GETIDENTITY
02424     WORD *t, *t1, *oldworkpointer = AT.WorkPointer, *tcoef, ntcoef, *out;
02425     int i;
02426     SHvariables *SH = &(AN.SHvar);
02427     SH->outfun[1] = fini - SH->outfun;
02428     if ( functions[SH->outfun[0]-FUNCTION].symmetric != 0 )
02429         SH->outfun[2] |= DIRTYSYMFLAG;
02430     out = fini; i = fini - SH->outterm; t = SH->outterm;
02431     NCOPY(fini,t,i)
02432     t = SH->stop1;
02433     t1 = t + t[1];
02434     while ( t1 < SH->stop2 ) { t = t1; t1 = t + t[1]; }
02435     t1 = SH->stop1;
02436     while ( t1 < t ) *fini++ = *t1++;
02437     t = SH->stop2;
02438     while ( t < SH->incoef ) *fini++ = *t++;
02439     tcoef = fini;
02440     ntcoef = SH->nincoef;
02441     i = ABS(ntcoef);
02442     NCOPY(fini,t,i);
02443     ntcoef = REDLENG(ntcoef);
02444     Mully(BHEAD (UWORD *)tcoef,&ntcoef,
02445         (UWORD *) (AN.SHcombi+SH->combilast+1),AN.SHcombi[SH->combilast]));
02446     ntcoef = INCLENG(ntcoef);
02447     fini = tcoef + ABS(ntcoef);
02448     if ( ( ( SH->option & 2 ) != 0 ) && ( ( SH->option & 256 ) != 0 ) ) ntcoef = -ntcoef;
02449     fini[-1] = ntcoef;
02450     i = *out = fini - out;
02451 /*
02452 Now check whether we have to do more
02453 */
02454 AT.WorkPointer = out + *out;
02455 if ( ( SH->option & 1 ) == 1 ) {
02456     if ( Generator(BHEAD out,SH->level) ) goto Finicall;
02457 }
02458 else {
02459     if ( DoStuffle(out,SH->level,SH->thefunction,SH->option) ) goto Finicall;
02460 }
02461 AT.WorkPointer = oldworkpointer;
02462 return(0);
02463 Finicall:
02464 AT.WorkPointer = oldworkpointer;
02465 MesCall("FinishShuffle");
02466 return(-1);
02467 }
02468
02469 /*
02470 #] FinishShuffle :
02471 #[ DoStuffle :
02472
02473 Stuffling is a variation of shuffling.
02474 In the stuffling we insist that the arguments are (short) integers. nonzero.
02475 The stuffle sum is  $x \text{ st } y = \text{sig}_-(x) * \text{sig}_-(y) * (\text{abs}(x) + \text{abs}(y))$ 
02476 The way we do this is:
02477 1: count the arguments in each function: n1, n2

```

```

02478         2: take the minimum minval = min(n1,n2).
02479         3: for ( j = 0; j <= min; j++ ) take j elements in each of the lists.
02480         4: the j+1 groups of remaining arguments have to each be shuffled
02481         5: the j selected pairs have to be stuffle added.
02482         We can use many of the shuffle things.
02483         Considering the recursive nature of the generation we actually don't
02484         need to know n1, n2, minval.
02485     */
02486
02487 int DoStuffle(WORD *term, WORD level, WORD fun, WORD option)
02488 {
02489     GETIDENTITY
02490     SHvariables SHback, *SH = &(AN.SHvar);
02491     WORD *t1, *t2, *tstop, *t1stop, *t2stop, ncoef, n = fun, *to, *from;
02492     WORD *r1, *r2;
02493     int i, error;
02494     LONG k;
02495     UWORD *newcombi;
02496 #ifdef NEWCODE
02497     WORD *rr1, *rr2, i1, i2;
02498 #endif
02499     if ( n < 0 ) {
02500         if ( ( n = DolToFunction(BHEAD -n) ) == 0 ) {
02501             MLOCK(ErrorMessageLock);
02502             MesPrint("$-variable in merge statement did not evaluate to a function.");
02503             MUNLOCK(ErrorMessageLock);
02504             return(1);
02505         }
02506     }
02507     if ( AT.WorkPointer + 3*(term) + AM.MaxTal > AT.WorkTop ) {
02508         MLOCK(ErrorMessageLock);
02509         MesWork();
02510         MUNLOCK(ErrorMessageLock);
02511         return(-1);
02512     }
02513
02514     tstop = term + *term;
02515     ncoef = tstop[-1];
02516     tstop -= ABS(ncoef);
02517     t1 = term + 1;
02518 retry1:;
02519     while ( t1 < tstop ) {
02520         if ( ( *t1 == n ) && ( t1+t1[1] < tstop ) && ( t1[1] > FUNHEAD ) ) {
02521             t2 = t1 + t1[1];
02522             if ( t2 >= tstop ) {
02523                 return(Generator(BHEAD term,level));
02524             }
02525 retry2:;
02526             while ( t2 < tstop ) {
02527                 if ( ( *t2 == n ) && ( t2[1] > FUNHEAD ) ) break;
02528                 t2 += t2[1];
02529             }
02530             if ( t2 < tstop ) break;
02531         }
02532         t1 += t1[1];
02533     }
02534     if ( t1 >= tstop ) {
02535         return(Generator(BHEAD term,level));
02536     }
02537     /*
02538     Next we have to check that the arguments are of the correct type
02539     At the same time we can count them.
02540     */
02541 #ifndef NEWCODE
02542     t1stop = t1 + t1[1];
02543     r1 = t1 + FUNHEAD;
02544     while ( r1 < t1stop ) {
02545         if ( *r1 != -SNUMBER ) break;
02546         if ( r1[1] == 0 ) break;
02547         r1 += 2;
02548     }
02549     if ( r1 < t1stop ) { t1 = t2; goto retry1; }
02550     t2stop = t2 + t2[1];
02551     r2 = t2 + FUNHEAD;
02552     while ( r2 < t2stop ) {
02553         if ( *r2 != -SNUMBER ) break;
02554         if ( r2[1] == 0 ) break;
02555         r2 += 2;
02556     }
02557     if ( r2 < t2stop ) { t2 = t2 + t2[1]; goto retry2; }
02558 #else
02559     t1stop = t1 + t1[1];
02560     r1 = t1 + FUNHEAD;
02561     while ( r1 < t1stop ) {
02562         if ( *r1 == -SNUMBER ) {
02563             if ( r1[1] == 0 ) break;
02564             r1 += 2; continue;

```

```

02565     }
02566     else if ( *r1 == -SYMBOL ) {
02567         if ( ( symbols[r1[1]].complex & VARTYPEROOTOFUNITY ) != VARTYPEROOTOFUNITY )
02568             break;
02569         r1 += 2; continue;
02570     }
02571     if ( *r1 > 0 && *r1 == r1[ARGHEAD]+ARGHEAD ) {
02572         if ( ABS(r1[r1[0]-1]) == r1[0]-ARGHEAD-1 ) {}
02573         else if ( r1[ARGHEAD+1] == SYMBOL ) {
02574             rr1 = r1 + ARGHEAD + 3;
02575             i1 = rr1[-1]-2;
02576             while ( i1 > 0 ) {
02577                 if ( ( symbols[*rr1].complex & VARTYPEROOTOFUNITY ) != VARTYPEROOTOFUNITY )
02578                     break;
02579                 i1 -= 2; rr1 += 2;
02580             }
02581             if ( i1 > 0 ) break;
02582         }
02583         else break;
02584         rr1 = r1+*r1-1;
02585         i1 = (ABS(*rr1)-1)/2;
02586         while ( i1 > 1 ) {
02587             if ( rr1[-1] ) break;
02588             i1--; rr1--;
02589         }
02590         if ( i1 > 1 || rr1[-1] != 1 ) break;
02591         r1 += *r1;
02592     }
02593     else break;
02594 }
02595 if ( r1 < t1stop ) { t1 = t2; goto retry1; }
02596 t2stop = t2 + t2[1];
02597 r2 = t2 + FUNHEAD;
02598
02599 while ( r2 < t2stop ) {
02600     if ( *r2 == -SNUMBER ) {
02601         if ( r2[1] == 0 ) break;
02602         r2 += 2; continue;
02603     }
02604     else if ( *r2 == -SYMBOL ) {
02605         if ( ( symbols[r2[1]].complex & VARTYPEROOTOFUNITY ) != VARTYPEROOTOFUNITY )
02606             break;
02607         r2 += 2; continue;
02608     }
02609     if ( *r2 > 0 && *r2 == r2[ARGHEAD]+ARGHEAD ) {
02610         if ( ABS(r2[r2[0]-1]) == r2[0]-ARGHEAD-1 ) {}
02611         else if ( r2[ARGHEAD+1] == SYMBOL ) {
02612             rr2 = r2 + ARGHEAD + 3;
02613             i2 = rr2[-1]-2;
02614             while ( i2 > 0 ) {
02615                 if ( ( symbols[*rr2].complex & VARTYPEROOTOFUNITY ) != VARTYPEROOTOFUNITY )
02616                     break;
02617                 i2 -= 2; rr2 += 2;
02618             }
02619             if ( i2 > 0 ) break;
02620         }
02621         else break;
02622         rr2 = r2+*r2-1;
02623         i2 = (ABS(*rr2)-1)/2;
02624         while ( i2 > 1 ) {
02625             if ( rr2[-1] ) break;
02626             i2--; rr2--;
02627         }
02628         if ( i2 > 1 || rr2[-1] != 1 ) break;
02629         r2 += *r2;
02630     }
02631     else break;
02632 }
02633 if ( r2 < t2stop ) { t2 = t2 + t2[1]; goto retry2; }
02634 #endif
02635 /*
02636 OK, now we got two objects that can be used.
02637 */
02638 *AN.RepPoint = 1;
02639
02640 SHback = AN.SHvar;
02641 SH->finishuf = &FinishStuffle;
02642 SH->do_uffle = &DoStuffle;
02643 SH->outterm = AT.WorkPointer;
02644 AT.WorkPointer += *term;
02645 SH->ststop1 = t1 + t1[1];
02646 SH->ststop2 = t2 + t2[1];
02647 SH->thefunction = n;
02648 SH->option = option;
02649 SH->level = level;
02650 SH->incoef = tstop;
02651 SH->nincoef = ncoef;

```

```

02652     if ( AN.SHcombi == 0 || AN.SHcombisize == 0 ) {
02653         AN.SHcombisize = 200;
02654         AN.SHcombi = (UWORD *)Malloc1(AN.SHcombisize*sizeof(UWORD), "AN.SHcombi");
02655         SH->combilast = 0;
02656         SHback.combilast = 0;
02657     }
02658     else {
02659         SH->combilast += AN.SHcombi[SH->combilast]+1;
02660         if ( SH->combilast >= AN.SHcombisize - 100 ) {
02661             newcombi = (UWORD *)Malloc1(2*AN.SHcombisize*sizeof(UWORD), "AN.SHcombi");
02662             for ( k = 0; k < AN.SHcombisize; k++ ) newcombi[k] = AN.SHcombi[k];
02663             M_free(AN.SHcombi, "AN.SHcombi");
02664             AN.SHcombi = newcombi;
02665             AN.SHcombisize *= 2;
02666         }
02667     }
02668     AN.SHcombi[SH->combilast] = 1;
02669     AN.SHcombi[SH->combilast+1] = 1;
02670
02671     i = t1-term; to = SH->outterm; from = term;
02672     NCOPY(to, from, i)
02673     SH->outfun = to;
02674     for ( i = 0; i < FUNHEAD; i++ ) { *to++ = t1[i]; }
02675
02676     error = Stuff1e(t1+FUNHEAD, t2+FUNHEAD, to);
02677
02678     AT.WorkPointer = SH->outterm;
02679     AN.SHvar = SHback;
02680     if ( error ) {
02681         MesCall("DoStuff1e");
02682         return(-1);
02683     }
02684     return(0);
02685 }
02686
02687 /*
02688 #] DoStuff1e :
02689 #[ Stuff1e :
02690
02691 The way to generate the stuff1es
02692 1: select an argument in the first list (for(j1=0;j1<last;j1++))
02693 2: select an argument in the second list (for(j2=0;j2<last;j2++))
02694 3: put values for SH->ststop1 and SH->ststop2 at these arguments.
02695 4: generate all shuff1es of the arguments in front.
02696 5: Then put the stuff1e sum of arg(j1) and arg(j2)
02697 6: Then continue calling Stuff1e
02698 7: Once one gets exhausted, we can clean up the list and call FinishShuff1e
02699 8: if ( ( SH->option & 2 ) != 0 ) the stuff1e sum is negative.
02700 */
02701
02702 int Stuff1e(WORD *from1, WORD *from2, WORD *to)
02703 {
02704     GETIDENTITY
02705     WORD *t, *tf, *next1, *next2, *st1, *st2, *savel, *save2;
02706     SHvariables *SH = &(AN.SHvar);
02707     int i, retval;
02708
02709     /*
02710     First the special cases (exhausted list(s)):
02711     */
02712     savel = SH->ststop1; save2 = SH->ststop2;
02713     if ( from1 >= SH->ststop1 && from2 == SH->ststop2 ) {
02714         SH->ststop1 = SH->ststop1;
02715         SH->ststop2 = SH->ststop2;
02716         retval = FinishShuff1e(to);
02717         SH->ststop1 = savel; SH->ststop2 = save2;
02718         return(retval);
02719     }
02720     else if ( from1 >= SH->ststop1 ) {
02721         i = SH->ststop2 - from2; t = to; tf = from2; NCOPY(t, tf, i)
02722         SH->ststop1 = SH->ststop1;
02723         SH->ststop2 = SH->ststop2;
02724         retval = FinishShuff1e(t);
02725         SH->ststop1 = savel; SH->ststop2 = save2;
02726         return(retval);
02727     }
02728     else if ( from2 >= SH->ststop2 ) {
02729         i = SH->ststop1 - from1; t = to; tf = from1; NCOPY(t, tf, i)
02730         SH->ststop1 = SH->ststop1;
02731         SH->ststop2 = SH->ststop2;
02732         retval = FinishShuff1e(t);
02733         SH->ststop1 = savel; SH->ststop2 = save2;
02734         return(retval);
02735     }
02736
02737     /*
02738     Now the case that we have no stuff1e sums.
02739     */
02740     SH->ststop1 = SH->ststop1;

```



```

02739     SH->stop2 = SH->ststop2;
02740     SH->finishuf = &FinishShuffle;
02741     if ( Shuffle(from1,from2,to) ) goto stuffcall;
02742     SH->finishuf = &FinishStuffle;
02743     /*
02744     Now we have to select a pair, one from 1 and one from 2.
02745     */
02746     #ifndef NEWCODE
02747     st1 = from1; next1 = st1+2;          /* <----- */
02748     #else
02749     st1 = next1 = from1;
02750     NEXTARG(next1)
02751     #endif
02752     while ( next1 <= SH->ststop1 ) {
02753     #ifndef NEWCODE
02754     st2 = from2; next2 = st2+2;          /* <----- */
02755     #else
02756     next2 = st2 = from2;
02757     NEXTARG(next2)
02758     #endif
02759     while ( next2 <= SH->ststop2 ) {
02760         SH->stop1 = st1;
02761         SH->stop2 = st2;
02762         if ( st1 == from1 && st2 == from2 ) {
02763             t = to;
02764             #ifndef NEWCODE
02765             *t++ = -SNUMBER; *t++ = StuffAdd(st1[1],st2[1]);
02766             #else
02767             t = StuffRootAdd(st1,st2,t);
02768             #endif
02769             SH->option ^= 256;
02770             if ( Stuffle(next1,next2,t) ) goto stuffcall;
02771             SH->option ^= 256;
02772         }
02773         else if ( st1 == from1 ) {
02774             i = st2-from2;
02775             t = to; tf = from2; NCOPY(t,tf,i)
02776             #ifndef NEWCODE
02777             *t++ = -SNUMBER; *t++ = StuffAdd(st1[1],st2[1]);
02778             #else
02779             t = StuffRootAdd(st1,st2,t);
02780             #endif
02781             SH->option ^= 256;
02782             if ( Stuffle(next1,next2,t) ) goto stuffcall;
02783             SH->option ^= 256;
02784         }
02785         else if ( st2 == from2 ) {
02786             i = st1-from1;
02787             t = to; tf = from1; NCOPY(t,tf,i)
02788             #ifndef NEWCODE
02789             *t++ = -SNUMBER; *t++ = StuffAdd(st1[1],st2[1]);
02790             #else
02791             t = StuffRootAdd(st1,st2,t);
02792             #endif
02793             SH->option ^= 256;
02794             if ( Stuffle(next1,next2,t) ) goto stuffcall;
02795             SH->option ^= 256;
02796         }
02797         else {
02798             if ( Shuffle(from1,from2,to) ) goto stuffcall;
02799         }
02800     #ifndef NEWCODE
02801     st2 = next2; next2 += 2;          /* <----- */
02802     #else
02803     st2 = next2;
02804     NEXTARG(next2)
02805     #endif
02806     }
02807     #ifndef NEWCODE
02808     st1 = next1; next1 += 2;          /* <----- */
02809     #else
02810     st1 = next1;
02811     NEXTARG(next1)
02812     #endif
02813     }
02814     SH->stop1 = save1; SH->stop2 = save2;
02815     return(0);
02816 stuffcall:;
02817     MesCall("Stuffle");
02818     return(-1);
02819 }
02820
02821 /*
02822 #] Stuffle :
02823 #[ FinishStuffle :
02824
02825 The program only comes here from the Shuffle routine.

```

```

02826     It should add the stuffle sum and then call Stuffle again.
02827 */
02828
02829 int FinishStuffle(WORD *fini)
02830 {
02831     GETIDENTITY
02832     SHvariables *SH = &(AN.SHvar);
02833     #ifndef NEWCODE
02834     WORD *next1 = SH->stop1, *next2 = SH->stop2;
02835     fini = StuffRootAdd(next1,next2,fini);
02836     #else
02837     *fini++ = -SNUMBER; *fini++ = StuffAdd(SH->stop1[1],SH->stop2[1]);
02838     #endif
02839     SH->option ^= 256;
02840     #ifndef NEWCODE
02841     NEXTARG(next1)
02842     NEXTARG(next2)
02843     if ( Stuffle(next1,next2,fini) ) goto stuffcall;
02844     #else
02845     if ( Stuffle(SH->stop1+2,SH->stop2+2,fini) ) goto stuffcall;
02846     #endif
02847     SH->option ^= 256;
02848     return(0);
02849 stuffcall:;
02850     MesCall("FinishStuffle");
02851     return(-1);
02852 }
02853
02854 /*
02855 #] FinishStuffle :
02856 #[ StuffRootAdd :
02857
02858 Makes the stuffle sum of two arguments.
02859 The arguments can be of one of three types:
02860 1: -SNUMBER,num
02861 2: -SYMBOL,symbol
02862 3: Numerical (long) argument.
02863 4: Generic argument with (only) symbols that are roots of unity and
02864    a coefficient.
02865 We have excluded the case that both t1 and t2 are of type 1:
02866 The output should be written to 'to' and the new fill position should
02867 be the return value.
02868 `to' is inside the workspace.
02869
02870 The stuffle sum is sig_(t2)*t1+sig_(t1)*t2
02871 or sig_(t1)*sig_(t2)*(abs_(t1)+abs_(t2))
02872 */
02873
02874 #ifndef NEWCODE
02875
02876 WORD *StuffRootAdd(WORD *t1, WORD *t2, WORD *to)
02877 {
02878     int type1, type2, type3, sgn, sgn1, sgn2, sgn3, pow, root, nosymbols, i;
02879     WORD *tt1, *tt2, it1, it2, *t3, *r, size1, size2, size3;
02880     WORD scratch[2];
02881     LONG x;
02882     if ( *t1 == -SNUMBER ) { type1 = 1; if ( t1[1] < 0 ) sgn1 = -1; else sgn1 = 1; }
02883     else if ( *t1 == -SYMBOL ) { type1 = 2; sgn1 = 1; }
02884     else if ( ABS(t1[*t1-1]) == *t1-ARGHEAD-1 ) {
02885         type1 = 3; if ( t1[*t1-1] < 0 ) sgn1 = -1; else sgn1 = 1; }
02886     else { type1 = 4; if ( t1[*t1-1] < 0 ) sgn1 = -1; else sgn1 = 1; }
02887     if ( *t2 == -SNUMBER ) { type2 = 1; if ( t2[1] < 0 ) sgn2 = -1; else sgn2 = 1; }
02888     else if ( *t2 == -SYMBOL ) { type2 = 2; sgn2 = 1; }
02889     else if ( ABS(t2[*t2-1]) == *t2-ARGHEAD-1 ) {
02890         type2 = 3; if ( t2[*t2-1] < 0 ) sgn2 = -1; else sgn2 = 1; }
02891     else { type2 = 4; if ( t2[*t2-1] < 0 ) sgn2 = -1; else sgn2 = 1; }
02892     if ( type1 > type2 ) {
02893         t3 = t1; t1 = t2; t2 = t3;
02894         type3 = type1; type1 = type2; type2 = type3;
02895         sgn3 = sgn1; sgn1 = sgn2; sgn2 = sgn3;
02896     }
02897     nosymbols = 1; sgn3 = 1;
02898     switch ( type1 ) {
02899     case 1:
02900         if ( type2 == 1 ) {
02901             x = sgn2 * t1[1];
02902             x += sgn1 * t2[1];
02903             if ( x > MAXPOSITIVE || x < -(MAXPOSITIVE+1) ) {
02904                 if ( x < 0 ) { sgn1 = -3; x = -x; }
02905                 else sgn1 = 3;
02906                 *to++ = ARGHEAD+4;
02907                 *to++ = 0;
02908                 FILLARG(to)
02909                 *to++ = 4; *to++ = (UWORD)x; *to++ = 1; *to++ = sgn1;
02910             }
02911             else { *to++ = -SNUMBER; *to++ = (WORD)x; }
02912         }

```

```

02913     else if ( type2 == 2 ) {
02914         *to++ = ARGHEAD+8; *to++ = 0; FILLARG(to)
02915         *to++ = 8; *to++ = SYMBOL; *to++ = 4; *to++ = t2[1]; *to++ = 1;
02916         *to++ = ABS(t1[1])+1;
02917         *to++ = 1;
02918         *to++ = 3*sgn1;
02919     }
02920     else if ( type2 == 3 ) {
02921         tt1 = (WORD *)scratch; tt1[0] = ABS(t1[1]); size1 = 1;
02922         tt2 = t2+ARGHEAD+1; size2 = (ABS(t2[*t2-1])-1)/2;
02923         t3 = to;
02924         *to++ = 0; *to++ = 0; FILLARG(to) *to++ = 0;
02925         goto DoCoeffi;
02926     }
02927     else {
02928         /*
02929         t1 is (short) numeric, t2 has the symbol(s).
02930         */
02931         tt1 = (WORD *)scratch; tt1[0] = ABS(t1[1]); size1 = 1;
02932         tt2 = t2+ARGHEAD+1; tt2 += tt2[1]; size2 = (ABS(t2[*t2-1])-1)/2;
02933         t3 = to; i = tt2 - t2; r = t2;
02934         NCOPY(to,r,i)
02935         nosymbols = 0;
02936         goto DoCoeffi;
02937     }
02938     break;
02939 case 2:
02940     if ( type2 == 2 ) {
02941         if ( t1[1] == t2[1] ) {
02942             if ( ( symbols[t1[1]].maxpower == 4 )
02943                 && ( ( symbols[t1[1]].complex & VARTYPEMINUS ) == VARTYPEMINUS ) ) {
02944                 *to++ = -SNUMBER; *to++ = -2;
02945             }
02946             else if ( symbols[t1[1]].maxpower == 2 ) {
02947                 *to++ = -SNUMBER; *to++ = 2;
02948             }
02949             else {
02950                 *to++ = ARGHEAD+8; *to++ = 0; FILLARG(to)
02951                 *to++ = 8; *to++ = SYMBOL; *to++ = 4;
02952                 *to++ = t1[1]; *to++ = 2;
02953                 *to++ = 2; *to++ = 1; *to++ = 3;
02954             }
02955         }
02956         else {
02957             *to++ = ARGHEAD+10; *to++ = 0; FILLARG(to)
02958             *to++ = 10; *to++ = SYMBOL; *to++ = 6;
02959             if ( t1[1] < t2[1] ) {
02960                 *to++ = t1[1]; *to++ = 1; *to++ = t2[1]; *to++ = 1;
02961             }
02962             else {
02963                 *to++ = t2[1]; *to++ = 1; *to++ = t1[1]; *to++ = 1;
02964             }
02965             *to++ = 2; *to++ = 1; *to++ = 3;
02966         }
02967     }
02968     else if ( type2 == 3 ) {
02969         t3 = to;
02970         *to++ = 0; *to++ = 0; FILLARG(to) *to++ = 0;
02971         *to++ = SYMBOL; *to++ = 4; *to++ = t1[1]; *to++ = 1;
02972         tt1 = scratch; tt1[1] = 1; size1 = 1;
02973         tt2 = t2+ARGHEAD+1; size2 = (ABS(t2[*t2-1])-1)/2;
02974         nosymbols = 0;
02975         goto DoCoeffi;
02976     }
02977     else {
02978         tt1 = scratch; tt1[0] = 1; size1 = 1;
02979         t3 = to;
02980         *to++ = 0; *to++ = 0; FILLARG(to) *to++ = 0;
02981         *to++ = SYMBOL; *to++ = 0;
02982         tt2 = t2 + ARGHEAD+3; it2 = tt2[-1]-2;
02983         while ( it2 > 0 ) {
02984             if ( *tt2 == t1[1] ) {
02985                 pow = tt2[1]+1;
02986                 root = symbols[*tt2].maxpower;
02987                 if ( pow >= root ) pow -= root;
02988                 if ( ( symbols[*tt2].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
02989                     if ( ( root & 1 ) == 0 && pow >= root/2 ) {
02990                         pow -= root/2; sgn3 = -sgn3;
02991                     }
02992                 }
02993                 if ( pow != 0 ) {
02994                     *to++ = *tt2; *to++ = pow;
02995                 }
02996                 tt2 += 2; it2 -= 2;
02997                 break;
02998             }
02999             else if ( t1[1] < *tt2 ) {

```

```

03000          *to++ = t1[1]; *to++ = 1; break;
03001      }
03002      else {
03003          *to++ = *tt2++; *to++ = *tt2++; it2 -= 2;
03004          if ( it2 <= 0 ) { *to++ = t1[1]; *to++ = 1; }
03005      }
03006  }
03007  while ( it2 > 0 ) { *to++ = *tt2++; *to++ = *tt2++; it2 -= 2; }
03008  if ( (to - t3) > ARGHEAD+3 ) {
03009      t3[ARGHEAD+2] = (to-t3)-ARGHEAD-1; /* size of the SYMBOL field */
03010      nosymbols = 0;
03011  }
03012  else {
03013      to = t3+ARGHEAD+1; /* no SYMBOL field */
03014  }
03015  size2 = (ABS(t2[*t2-1])-1)/2;
03016  goto DoCoeffi;
03017  }
03018  break;
03019  case 3:
03020      if ( type2 == 3 ) {
03021  /*
03022          Both are numeric
03023  */
03024          tt1 = t1+ARGHEAD+1; size1 = (ABS(t1[*t1-1])-1)/2;
03025          tt2 = t2+ARGHEAD+1; size2 = (ABS(t2[*t2-1])-1)/2;
03026          t3 = to;
03027          *to++ = 0; *to++ = 0; FILLARG(to) *to++ = 0;
03028          goto DoCoeffi;
03029      }
03030      else {
03031  /*
03032          t1 is (long) numeric, t2 has the symbol(s).
03033  */
03034          tt1 = t1+ARGHEAD+1; size1 = (ABS(t1[*t1-1])-1)/2;
03035          tt2 = t2+ARGHEAD+1; tt2 += tt2[1]; size2 = (ABS(t2[*t2-1])-1)/2;
03036          t3 = to; i = tt2 - t2; r = t2;
03037          NCOPY(to,r,i)
03038          nosymbols = 0;
03039          goto DoCoeffi;
03040      }
03041  break;
03042  case 4:
03043  /*
03044          Both have roots of unity
03045          1: Merge the lists and simplify if possible
03046  */
03047          tt1 = t1+ARGHEAD+3; it1 = tt1[-1]-2;
03048          tt2 = t2+ARGHEAD+3; it2 = tt2[-1]-2;
03049          t3 = to;
03050          *to++ = 0; *to++ = 0; FILLARG(to)
03051          *to++ = 0; *to++ = SYMBOL; *to++ = 0;
03052          while ( it1 > 0 && it2 > 0 ) {
03053              if ( *tt1 == *tt2 ) {
03054                  pow = tt1[1]+tt2[1];
03055                  root = symbols[*tt1].maxpower;
03056                  if ( pow >= root ) pow -= root;
03057                  if ( ( symbols[*tt1].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
03058                      if ( ( root & 1 ) == 0 && pow >= root/2 ) {
03059                          pow -= root/2; sgn3 = -sgn3;
03060                      }
03061                  }
03062                  if ( pow != 0 ) {
03063                      *to++ = *tt1; *to++ = pow;
03064                  }
03065                  tt1 += 2; tt2 += 2; it1 -= 2; it2 -= 2;
03066              }
03067              else if ( *tt1 < *tt2 ) {
03068                  *to++ = *tt1++; *to++ = *tt1++; it1 -= 2;
03069              }
03070              else {
03071                  *to++ = *tt2++; *to++ = *tt2++; it2 -= 2;
03072              }
03073          }
03074          while ( it1 > 0 ) { *to++ = *tt1++; *to++ = *tt1++; it1 -= 2; }
03075          while ( it2 > 0 ) { *to++ = *tt2++; *to++ = *tt2++; it2 -= 2; }
03076          if ( (to - t3) > ARGHEAD+3 ) {
03077              t3[ARGHEAD+2] = (to-t3)-ARGHEAD-1; /* size of the SYMBOL field */
03078              nosymbols = 0;
03079          }
03080          else {
03081              to = t3+ARGHEAD+1; /* no SYMBOL field */
03082          }
03083          size1 = (ABS(t1[*t1-1])-1)/2;
03084          size2 = (ABS(t2[*t2-1])-1)/2;
03085  /*
03086          Now tt1 and tt2 are pointing at their coefficients.

```

```

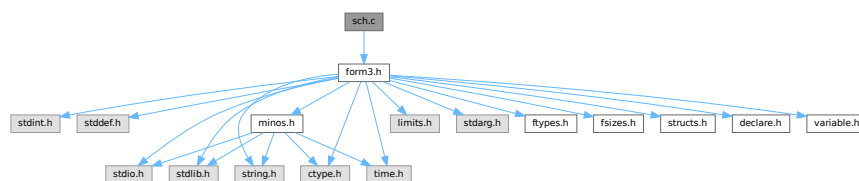
03087         sgn1 is the sign of 1, sgn2 is the sign of 2 and sgn3 is an extra
03088         overall sign.
03089     */
03090 DoCoeffi:
03091     if ( AddLong((UWORD *)tt1,size1,(UWORD *)tt2,size2,(UWORD *)to,&size3) ) {
03092         MLOCK(ErrorMessageLock);
03093         MesPrint("Called from StuffRootAdd");
03094         MUNLOCK(ErrorMessageLock);
03095         Terminate(-1);
03096     }
03097     sgn = sgn1*sgn2*sgn3;
03098     if ( nosymbols && size3 == 1 ) {
03099         if ( (UWORD)(to[0]) <= MAXPOSITIVE && sgn > 0 ) {
03100             sgn1 = to[0];
03101             to = t3; *to++ = -SNUMBER; *to++ = sgn1;
03102         }
03103         else if ( (UWORD)(to[0]) <= (MAXPOSITIVE+1) && sgn < 0 ) {
03104             sgn1 = to[0];
03105             to = t3; *to++ = -SNUMBER; *to++ = -sgn1;
03106         }
03107         else goto genericcoef;
03108     }
03109     else {
03110 genericcoef:
03111         to += size3;
03112         sgn = sgn*(2*size3+1);
03113         *to++ = 1;
03114         while ( size3 > 1 ) { *to++ = 0; size3--; }
03115         *to++ = sgn;
03116         t3[0] = to - t3;
03117         t3[ARGHEAD] = t3[0] - ARGHEAD;
03118     }
03119     break;
03120 }
03121 return(to);
03122 }
03123 #endif
03124
03125 /*
03126 #] StuffRootAdd :
03127 */

```

4.123 sch.c File Reference

```
#include "form3.h"
```

Include dependency graph for sch.c:



Functions

- UBYTE * [StrCopy](#) (UBYTE *from, UBYTE *to)
- void [AddToLine](#) (UBYTE *s)
- void [FiniLine](#) (void)
- void [IniLine](#) (WORD extrablank)
- void [LongToLine](#) (UWORD *a, WORD na)
- void [RatToLine](#) (UWORD *a, WORD na)
- void [TailToLine](#) (UWORD x)
- void [TokenToLine](#) (UBYTE *s)

- UBYTE * [CodeToLine](#) (WORD number, UBYTE *Out)
- void [MultiplyToLine](#) (void)
- UBYTE * [AddArrayIndex](#) (WORD num, UBYTE *out)
- void [PrtTerms](#) (void)
- UBYTE * [WrtPower](#) (UBYTE *Out, WORD Power)
- void [PrintTime](#) (UBYTE *mess)
- void [WriteLists](#) (void)
- void [WriteDictionary](#) (DICTIONARY *dict)
- void [WriteArgument](#) (WORD *t)
- int [WriteSubTerm](#) (WORD *sterm, WORD first)
- int [WriteInnerTerm](#) (WORD *term, WORD first)
- int [WriteTerm](#) (WORD *term, WORD *lbrac, WORD first, WORD prtf, WORD br)
- int [WriteExpression](#) (WORD *terms, LONG ltot)
- int [WriteAll](#) (void)
- int [WriteOne](#) (UBYTE *name, int alreadyinline, int nosemi, WORD plus)

4.123.1 Detailed Description

Contains the functions that deal with the writing of expressions/terms in a textual representation. (Dutch schrijven = to write)

Definition in file [sch.c](#).

4.123.2 Function Documentation

4.123.2.1 StrCopy()

```
UBYTE * StrCopy (
    UBYTE * from,
    UBYTE * to )
```

Definition at line [49](#) of file [sch.c](#).

4.123.2.2 AddToLine()

```
void AddToLine (
    UBYTE * s )
```

Definition at line [64](#) of file [sch.c](#).

4.123.2.3 FiniLine()

```
void FiniLine (
    void )
```

Definition at line [164](#) of file [sch.c](#).

4.123.2.4 IniLine()

```
void IniLine (
    WORD extrablk )
```

Definition at line 249 of file [sch.c](#).

4.123.2.5 LongToLine()

```
void LongToLine (
    UWORD * a,
    WORD na )
```

Definition at line 285 of file [sch.c](#).

4.123.2.6 RatToLine()

```
void RatToLine (
    UWORD * a,
    WORD na )
```

Definition at line 319 of file [sch.c](#).

4.123.2.7 TalToLine()

```
void TalToLine (
    UWORD x )
```

Definition at line 504 of file [sch.c](#).

4.123.2.8 TokenToLine()

```
void TokenToLine (
    UBYTE * s )
```

Definition at line 533 of file [sch.c](#).

4.123.2.9 CodeToLine()

```
UBYTE * CodeToLine (
    WORD number,
    UBYTE * Out )
```

Definition at line 640 of file [sch.c](#).

4.123.2.10 MultiplyToLine()

```
void MultiplyToLine (
    void )
```

Definition at line 653 of file [sch.c](#).

4.123.2.11 AddArrayIndex()

```
UBYTE * AddArrayIndex (
    WORD num,
    UBYTE * out )
```

Definition at line 678 of file [sch.c](#).

4.123.2.12 PrtTerms()

```
void PrtTerms (
    void )
```

Definition at line 698 of file [sch.c](#).

4.123.2.13 WrtPower()

```
UBYTE * WrtPower (
    UBYTE * Out,
    WORD Power )
```

Definition at line 720 of file [sch.c](#).

4.123.2.14 PrintTime()

```
void PrintTime (
    UBYTE * mess )
```

Definition at line 766 of file [sch.c](#).

4.123.2.15 WriteLists()

```
void WriteLists (
    void )
```

Definition at line 792 of file [sch.c](#).

4.123.2.16 WriteDictionary()

```
void WriteDictionary (
    DICTIONARY * dict )
```

Definition at line 1289 of file [sch.c](#).

4.123.2.17 WriteArgument()

```
void WriteArgument (
    WORD * t )
```

Definition at line 1406 of file [sch.c](#).

4.123.2.18 WriteSubTerm()

```
int WriteSubTerm (
    WORD * sterm,
    WORD first )
```

Definition at line 1524 of file [sch.c](#).

4.123.2.19 WriteInnerTerm()

```
int WriteInnerTerm (
    WORD * term,
    WORD first )
```

Definition at line 1933 of file [sch.c](#).

4.123.2.20 WriteTerm()

```
int WriteTerm (
    WORD * term,
    WORD * lbrac,
    WORD first,
    WORD prtft,
    WORD br )
```

Definition at line 2133 of file [sch.c](#).

4.123.2.21 WriteExpression()

```
int WriteExpression (
    WORD * terms,
    LONG ltot )
```

Definition at line 2455 of file [sch.c](#).

4.123.2.22 WriteAll()

```
int WriteAll (
    void )
```

Definition at line 2486 of file [sch.c](#).

4.123.2.23 WriteOne()

```
int WriteOne (
    UBYTE * name,
    int alreadyinline,
    int nosemi,
    WORD plus )
```

Definition at line 2712 of file [sch.c](#).

4.124 sch.c

[Go to the documentation of this file.](#)

```
00001
00006 /* #[] License : */
00007 /*
00008 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009 *   When using this file you are requested to refer to the publication
00010 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011 *   This is considered a matter of courtesy as the development was paid
00012 *   for by FOM the Dutch physics granting agency and we would like to
00013 *   be able to track its scientific use to convince FOM of its value
00014 *   for the community.
00015 *
00016 *   This file is part of FORM.
00017 *
00018 *   FORM is free software: you can redistribute it and/or modify it under the
00019 *   terms of the GNU General Public License as published by the Free Software
00020 *   Foundation, either version 3 of the License, or (at your option) any later
00021 *   version.
00022 *
00023 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00026 *   details.
00027 *
00028 *   You should have received a copy of the GNU General Public License along
00029 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00030 */
00031 /* #[] License : */
00032 /*
00033 *   #[] Includes : sch.c
00034 */
00035
00036 #include "form3.h"
00037
00038 static int startinline = 0;
00039 static char fcontchar = '&';
00040 static int noextralinefeed = 0;
00041 static int lowestlevel = 1;
00042
00043 /*
00044 *   #[] Includes :
00045 *   #[] schryf-Utilities :
00046 *   #[] StrCopy :          UBYTE *StrCopy(from,to)
00047 */
00048
00049 UBYTE *StrCopy(UBYTE *from, UBYTE *to)
00050 {
00051     while( ( *to++ = *from++ ) != 0 );
00052     return(to-1);
00053 }
00054
00055 /*
00056 *   #[] StrCopy :
00057 *   #[] AddToLine :      void AddToLine(s)
00058
00059 *   Puts the characters of s in the outputline. If the line becomes
00060 *   filled it is written.
00061
00062 */
00063
00064 void AddToLine(UBYTE *s)
00065 {
00066     UBYTE *Out;
00067     LONG num;
```

```

00068     int i;
00069     if ( AO.OutInBuffer ) { AddToDollarBuffer(s); return; }
00070     Out = AO.OutFill;
00071     while ( *s ) {
00072         if ( Out >= AO.OutStop ) {
00073             if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
00074                 *Out++ = fcontchar;
00075             }
00076 #ifdef WITHRETURN
00077             *Out++ = CARRIAGERETURN;
00078 #endif
00079             *Out++ = LINEFEED;
00080             AO.FortFirst = 0;
00081             num = Out - AO.OutputLine;
00082
00083             if ( AC.LogHandle >= 0 ) {
00084                 if ( WriteFile(AC.LogHandle,AO.OutputLine+startinline
00085                             ,num-startinline) != (num-startinline) ) {
00086 /*
00087                 We cannot write to an otherwise open log file.
00088                 The disk could be full of course.
00089 */
00090 #ifdef DEBUGGER
00091                 if ( BUG.logfileflag == 0 ) {
00092                     fprintf(stderr,"Panic: Cannot write to log file! Disk full?\n");
00093                     BUG.logfileflag = 1;
00094                 }
00095                 BUG.eflag = 1; BUG.printflag = 1;
00096 #else
00097                 Terminate(-1);
00098 #endif
00099             }
00100         }
00101
00102         if ( ( AO.PrintType & PRINTLFILE ) == 0 ) {
00103 #ifdef WITHRETURN
00104             if ( num > 1 && AO.OutputLine[num-2] == CARRIAGERETURN ) {
00105                 AO.OutputLine[num-2] = LINEFEED;
00106                 num--;
00107             }
00108 #endif
00109             if ( WriteFile(AM.Stdout,AO.OutputLine+startinline
00110                             ,num-startinline) != (num-startinline) ) {
00111 #ifdef DEBUGGER
00112                 if ( BUG.stdoutflag == 0 ) {
00113                     fprintf(stderr,"Panic: Cannot write to standard output!\n");
00114                     BUG.stdoutflag = 1;
00115                 }
00116                 BUG.eflag = 1; BUG.printflag = 1;
00117 #else
00118                 Terminate(-1);
00119 #endif
00120             }
00121         }
00122         /* thomasr 23/04/09: A continuation line has been started.
00123          * In Fortran90 we do not want a space after the initial
00124          * '&' character otherwise we might end up with something
00125          * like:
00126          *   ... 2.&
00127          *   & 0 ...
00128          */
00129         startinline = 0;
00130         for ( i = 0; i < AO.OutSkip; i++ ) AO.OutputLine[i] = ' ';
00131         Out = AO.OutputLine + AO.OutSkip;
00132         if ( ( AC.OutputMode == FORTRANMODE
00133             || AC.OutputMode == PFORTRANMODE ) && AO.OutSkip == 7 ) {
00134             /* thomasr 23/04/09: fix leading blank in fortran90 mode */
00135             if(AC.IsFortran90 == ISFORTRAN90) {
00136                 Out[-1] = fcontchar;
00137             }
00138             else {
00139                 Out[-2] = fcontchar;
00140                 Out[-1] = ' ';
00141             }
00142         }
00143         if ( AO.IsBracket ) { *Out++ = ' '; }
00144         if ( AC.OutputSpaces == NORMALFORMAT ) {
00145             *Out++ = ' '; *Out++ = ' '; }
00146     }
00147     *Out = '\0';
00148     if ( AC.OutputMode == FORTRANMODE
00149         || ( AC.OutputMode == CMODE && AO.FactorMode == 0 )
00150         || AC.OutputMode == PFORTRANMODE )
00151         AO.InFbrack++;
00152     }
00153     *Out++ = *s++;
00154 }

```

```

00155     *Out = '\0';
00156     AO.OutFill = Out;
00157 }
00158
00159 /*
00160     #] AddToLine :
00161     #[ FiniLine :      void FiniLine()
00162 */
00163
00164 void FiniLine(void)
00165 {
00166     UBYTE *Out;
00167     WORD i;
00168     LONG num;
00169     if ( AO.OutInBuffer ) return;
00170     Out = AO.OutFill;
00171     while ( Out > AO.OutputLine ) {
00172         if ( Out[-1] == ' ' ) Out--;
00173         else break;
00174     }
00175     i = (WORD)(Out-AO.OutputLine);
00176     if ( noextralinefeed == 0 ) {
00177         if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90
00178             && Out > AO.OutputLine ) {
00179             /*
00180                 *Out++ = fcontchar;
00181             */
00182         }
00183         #ifdef WITHRETURN
00184         *Out++ = CARRIAGERETURN;
00185         #endif
00186         *Out++ = LINEFEED;
00187         AO.FortFirst = 0;
00188     }
00189     num = Out - AO.OutputLine;
00190
00191     if ( AC.LogHandle >= 0 ) {
00192         if ( WriteFile(AC.LogHandle,AO.OutputLine+startinline
00193             ,num-startinline) != (num-startinline) ) {
00194             #ifdef DEBUGGER
00195                 if ( BUG.logfileflag == 0 ) {
00196                     fprintf(stderr,"Panic: Cannot write to log file! Disk full?\n");
00197                     BUG.logfileflag = 1;
00198                 }
00199                 BUG.eflag = 1; BUG.printflag = 1;
00200             #else
00201                 Terminate(-1);
00202             #endif
00203         }
00204     }
00205
00206     if ( ( AO.PrintType & PRINTLFILE ) == 0 ) {
00207         #ifdef WITHRETURN
00208         if ( num > 1 && AO.OutputLine[num-2] == CARRIAGERETURN ) {
00209             AO.OutputLine[num-2] = LINEFEED;
00210             num--;
00211         }
00212         #endif
00213         if ( WriteFile(AM.Stdout,AO.OutputLine+startinline,
00214             num-startinline) != (num-startinline) ) {
00215             #ifdef DEBUGGER
00216                 if ( BUG.stdoutflag == 0 ) {
00217                     fprintf(stderr,"Panic: Cannot write to standard output!\n");
00218                     BUG.stdoutflag = 1;
00219                 }
00220                 BUG.eflag = 1; BUG.printflag = 1;
00221             #else
00222                 Terminate(-1);
00223             #endif
00224         }
00225     }
00226     startinline = 0;
00227     if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
00228         || ( AC.OutputMode == CMODE && AO.FactorMode == 0 ) ) AO.InFbrack++;
00229     Out = AO.OutputLine;
00230     AO.OutStop = Out + AC.LineLength;
00231     i = AO.OutSkip;
00232     while ( --i >= 0 ) *Out++ = ' ';
00233     if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
00234         && AO.OutSkip == 7 ) {
00235         Out[-2] = fcontchar;
00236         Out[-1] = ' ';
00237     }
00238     AO.OutFill = Out;
00239 }
00240
00241 /*

```

```

00242         #] FiniLine :
00243         #[ IniLine :             void IniLine(extrablank)
00244
00245         Initializes the output line for the type of output
00246
00247     */
00248
00249     void IniLine(WORD extrablank)
00250     {
00251         UBYTE *Out;
00252         Out = AO.OutputLine;
00253         AO.OutStop = Out + AC.LineLength;
00254         *Out++ = ' ';
00255         *Out++ = ' ';
00256         *Out++ = ' ';
00257         *Out++ = ' ';
00258         *Out++ = ' ';
00259         if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {
00260             *Out++ = fcontchar;
00261             AO.OutSkip = 7;
00262         }
00263         else
00264             AO.OutSkip = 6;
00265         *Out++ = ' ';
00266         while ( extrablank > 0 ) {
00267             *Out++ = ' ';
00268             extrablank--;
00269         }
00270         AO.OutFill = Out;
00271     }
00272
00273     /*
00274         #] IniLine :
00275         #[ LongToLine :             void LongToLine(a,na)
00276
00277         Puts a Long integer in the output line. If it is only a single
00278         word long it is put in the line as a single token.
00279         The sign of a is ignored.
00280
00281     */
00282
00283     static UBYTE *LLscratch = 0;
00284
00285     void LongToLine(UWORD *a, WORD na)
00286     {
00287         UBYTE *OutScratch;
00288         if ( LLscratch == 0 ) {
00289             LLscratch = (UBYTE *)Malloca(4*(AM.MaxTal*sizeof(WORD)+2)*sizeof(UBYTE), "LongToLine");
00290         }
00291         OutScratch = LLscratch;
00292         if ( na < 0 ) na = -na;
00293         if ( na > 1 ) {
00294             PrtLong(a,na,OutScratch);
00295             if ( AO.NoSpacesInNumbers || AC.OutputMode == REDUCEMODE ) {
00296                 AO.BlockSpaces = 1;
00297                 TokenToLine(OutScratch);
00298                 AO.BlockSpaces = 0;
00299             }
00300             else {
00301                 TokenToLine(OutScratch);
00302             }
00303         }
00304         else if ( !na ) TokenToLine((UBYTE *)"0");
00305         else TalToLine(*a);
00306     }
00307
00308     /*
00309         #] LongToLine :
00310         #[ RatToLine :             void RatToLine(a,na)
00311
00312         Puts a rational number in the output line. The sign is ignored.
00313
00314     */
00315
00316     static UBYTE *RLscratch = 0;
00317     static UWORD *RLscratE = 0;
00318
00319     void RatToLine(UWORD *a, WORD na)
00320     {
00321         GETIDENTITY
00322         WORD adenom, anumer;
00323         ULONG maxInt;
00324
00325         if ( AC.OutputMode == CMODE ) {
00326             // In C, integer literals over 2^32-1 are automatically promoted to longer types up to
00327             // unsigned long long, however FORM has always given integers over 2^32-1 a floating suffix
00328             // in C mode. Retain that behaviour here for now. In the future, maxInt could be a user-

```

```

00329         // configurable parameter.
00330         maxInt = 4294967295;
00331     }
00332     else {
00333         // In Fortran modes, literals over 2^31-1 should be printed with a float suffix
00334         maxInt = 2147483647;
00335     }
00336
00337     if ( na < 0 ) na = -na;
00338
00339     if ( AC.OutNumberType == RATIONALMODE ) {
00340
00341         // Work out what is to be printed:
00342         WORD isOne = 0, isOneOver = 0, isIntegral = 0;
00343         WORD isLongNum = 0, isLongDen = 0;
00344         UnPack(a,na,&adenom,&anumer);
00345         if ( na == 1 && a[0] == 1 && a[1] == 1 ) { isOne = 1; isIntegral = 1; }
00346         else if ( adenom == 1 && a[na] == 1 ) { isIntegral = 1; }
00347         else if ( anumer == 1 && a[0] == 1 ) { isOneOver = 1; }
00348         if ( anumer > 1 || ( anumer == 1 && a[0] > maxInt ) ) { isLongNum = 1; };
00349         if ( adenom > 1 || ( adenom == 1 && a[na] > maxInt ) ) { isLongDen = 1; };
00350
00351         // Now sort out the float suffix for the numerator:
00352         UBYTE* suffNum = (UBYTE*)"";
00353         UBYTE* suffDen = (UBYTE*)"";
00354         if ( isLongNum || !isIntegral || AC.Fortran90Kind || ( AO.DoubleFlag & 4 ) == 4 ) {
00355             if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
00356                 if ( AC.Fortran90Kind ) { suffNum = AC.Fortran90Kind; }
00357                 else { suffNum = (UBYTE*)"."; }
00358             }
00359             else if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == CMODE ) {
00360                 if ( ( AO.DoubleFlag & 2 ) == 2 ) { suffNum = (UBYTE*)".Q0"; }
00361                 else if ( ( AO.DoubleFlag & 1 ) == 1 ) { suffNum = (UBYTE*)".D0"; }
00362                 else { suffNum = (UBYTE*)"."; }
00363             }
00364         }
00365         // The same again, for the denominator:
00366         if ( isLongDen || !isIntegral || AC.Fortran90Kind || ( AO.DoubleFlag & 4 ) == 4 ) {
00367             if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
00368                 if ( AC.Fortran90Kind ) { suffDen = AC.Fortran90Kind; }
00369                 else { suffDen = (UBYTE*)"."; }
00370             }
00371             else if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == CMODE ) {
00372                 if ( ( AO.DoubleFlag & 2 ) == 2 ) { suffDen = (UBYTE*)".Q0"; }
00373                 else if ( ( AO.DoubleFlag & 1 ) == 1 ) { suffDen = (UBYTE*)".D0"; }
00374                 else { suffDen = (UBYTE*)"."; }
00375             }
00376         }
00377         // In PFORTTRAN mode, rationals don't get a suffix if the integers are not long
00378         // Also the suffix is at least ".D0" (and never ".").
00379         if ( AC.OutputMode == PFORTTRANMODE ) {
00380             if ( isLongNum ) {
00381                 if ( ( AO.DoubleFlag & 2 ) == 2 ) { suffNum = (UBYTE*)".Q0"; }
00382                 else { suffNum = (UBYTE*)".D0"; }
00383             }
00384             if ( isLongDen ) {
00385                 if ( ( AO.DoubleFlag & 2 ) == 2 ) { suffDen = (UBYTE*)".Q0"; }
00386                 else { suffDen = (UBYTE*)".D0"; }
00387             }
00388         }
00389
00390         // Finally, print the number:
00391         // In PFORTTRAN we use
00392         //   one      if denom = numerator = 1
00393         //   integer   if denom = 1
00394         //   (one/integer) if numerator = 1
00395         //   ((one*integer)/integer) in the general case
00396         if ( AC.OutputMode == PFORTTRANMODE ) {
00397             if ( isOne ) {
00398                 AddToLine((UBYTE *) "one");
00399             }
00400             else if ( isOneOver ) {
00401                 AddToLine((UBYTE *) "(one/");
00402                 LongToLine(a+na,adenom);
00403                 AddToLine(suffDen);
00404                 AddToLine((UBYTE *) ")");
00405             }
00406             else if ( isIntegral ) {
00407                 LongToLine(a,anumer);
00408                 AddToLine(suffNum);
00409             }
00410             else {
00411                 // rational
00412                 AddToLine((UBYTE *) "( (one*");
00413                 LongToLine(a,anumer);
00414                 AddToLine(suffNum);
00415                 AddToLine((UBYTE *) ") / ");

```

```

00416         LongToLine(a+na,adenom);
00417         AddToLine(suffDen);
00418         AddToLine((UBYTE*)"");
00419     }
00420 }
00421 // All other modes use the same printing code:
00422 else {
00423     // Numerator:
00424     LongToLine(a,anumer);
00425     AddToLine(suffNum);
00426     // Denominator, if it is not 1:
00427     if (!isIntegral) {
00428         AddToLine((UBYTE*)"/");
00429         LongToLine(a+na,adenom);
00430         AddToLine(suffDen);
00431     }
00432 }
00433 }
00434
00435 else {
00436 /*
00437     This is the float mode
00438 */
00439     UBYTE *OutScratch;
00440     WORD exponent = 0, i, ndig, newl;
00441     UWORD *c, *den, b = 10, dig[10];
00442     UBYTE *o, *out, cc;
00443 /*
00444     First we have to adjust the numerator and denominator
00445 */
00446     if ( RLscratch == 0 ) {
00447         RLscratch = (UBYTE *)Malloca(4*(AM.MaxTal+2)*sizeof(UBYTE),"RatToLine");
00448         RLscratE = (UWORD *)Malloca(2*(AM.MaxTal+2)*sizeof(UWORD),"RatToLine");
00449     }
00450     out = OutScratch = RLscratch;
00451     c = RLscratE; for ( i = 0; i < 2*na; i++ ) c[i] = a[i];
00452     UnPack(c,na,&adenom,&anumer);
00453     while ( BigLong(c,anumer,c+na,adenom) >= 0 ) {
00454         Divvy(BHEAD c,&na,&b,1);
00455         UnPack(c,na,&adenom,&anumer);
00456         exponent++;
00457     }
00458     while ( BigLong(c,anumer,c+na,adenom) < 0 ) {
00459         Mully(BHEAD c,&na,&b,1);
00460         UnPack(c,na,&adenom,&anumer);
00461         exponent--;
00462     }
00463 /*
00464     Now division will give a number between 1 and 9
00465 */
00466     den = c + na; i = 1;
00467     DivLong(c,anumer,den,adenom,dig,&ndig,c,&newl);
00468     *out++ = (UBYTE)(dig[0]+'0'); *out++ = '.';
00469     while ( newl && i < AC.OutNumberType ) {
00470         Pack(c,&newl,den,adenom);
00471         Mully(BHEAD c,&newl,&b,1);
00472         na = newl;
00473         UnPack(c,na,&adenom,&anumer);
00474         den = c + na;
00475         DivLong(c,anumer,den,adenom,dig,&ndig,c,&newl);
00476         if ( ndig == 0 ) *out++ = '0';
00477         else *out++ = (UBYTE)(dig[0]+'0');
00478         i++;
00479     }
00480     *out++ = 'E';
00481     if ( exponent < 0 ) { exponent = -exponent; *out++ = '-'; }
00482     else { *out++ = '+'; }
00483     o = out;
00484     do {
00485         *out++ = (UBYTE)((exponent % 10)+'0');
00486         exponent /= 10;
00487     } while ( exponent );
00488     *out = 0; out--;
00489     while ( o < out ) { cc = *o; *o = *out; *out = cc; o++; out--; }
00490     TokenToLine(OutScratch);
00491 }
00492 }
00493
00494 /*
00495     #] RatToLine :
00496     #[ TalToLine :         void TalToLine(x)
00497
00498     Writes the unsigned number x to the output as a single token.
00499     Par indicates the number of leading blanks in the line.
00500     This parameter is needed here for the WriteLists routine.
00501
00502 */

```

```

00503
00504 void TalToLine(UWORD x)
00505 {
00506     UBYTE t[BITSINWORD/3+1];
00507     UBYTE *s;
00508     WORD i = 0, j;
00509     s = t;
00510     do { *s++ = (UBYTE)((x % 10)+'0'); i++; } while ( ( x /= 10 ) != 0 );
00511     *s-- = '\0';
00512     j = ( i - 1 ) >> 1;
00513     while ( j >= 0 ) {
00514         i = t[j]; t[j] = s[-j]; s[-j] = (UBYTE)i; j--;
00515     }
00516     TokenToLine(t);
00517 }
00518
00519 /*
00520     #] TalToLine :
00521     #[ TokenToLine :      void TokenToLine(s)
00522
00523     Puts s in the output buffer. If it doesn't fit the buffer is
00524     flushed first. This routine keeps tokens as one unit.
00525     Par indicates the number of leading blanks in the line.
00526     This parameter is needed here for the WriteLists routine.
00527
00528     Remark (27-oct-2007): i and j must be longer than WORD!
00529     It can happen that a number is so long that it has more than 2^15 or 2^31
00530     digits!
00531 */
00532
00533 void TokenToLine(UBYTE *s)
00534 {
00535     UBYTE *t, *Out;
00536     LONG num, i = 0, j;
00537     if ( AO.OutInBuffer ) { AddToDollarBuffer(s); return; }
00538     t = s; Out = AO.OutFill;
00539     while ( *t++ ) i++;
00540     while ( i > 0 ) {
00541         if ( ( Out + i ) >= AO.OutStop && ( ( i < ((AC.LineLength-AO.OutSkip)>>1) )
00542             || ( AO.OutStop-Out ) < (i>>2) ) ) ) {
00543             if ( AC.OutputMode == FORTRANMODE && AC.IsFortran90 == ISFORTRAN90 ) {
00544                 *Out++ = fcontchar;
00545             }
00546             #ifdef WITHRETURN
00547                 *Out++ = CARRIAGERETURN;
00548             #endif
00549             *Out++ = LINEFEED;
00550             AO.FortFirst = 0;
00551             num = Out - AO.OutputLine;
00552             if ( AC.LogHandle >= 0 ) {
00553                 if ( WriteFile(AC.LogHandle, AO.OutputLine+startinline,
00554                     num-startinline) != (num-startinline) ) {
00555                     #ifdef DEBUGGER
00556                         if ( BUG.logfileflag == 0 ) {
00557                             fprintf(stderr, "Panic: Cannot write to log file! Disk full?\n");
00558                             BUG.logfileflag = 1;
00559                         }
00560                         BUG.eflag = 1; BUG.printflag = 1;
00561                     #else
00562                         Terminate(-1);
00563                     #endif
00564                 }
00565             }
00566             if ( ( AO.PrintType & PRINTLFILE ) == 0 ) {
00567                 #ifdef WITHRETURN
00568                     if ( num > 1 && AO.OutputLine[num-2] == CARRIAGERETURN ) {
00569                         AO.OutputLine[num-2] = LINEFEED;
00570                         num--;
00571                     }
00572                 #endif
00573                 if ( WriteFile(AM.Stdout, AO.OutputLine+startinline,
00574                     num-startinline) != (num-startinline) ) {
00575                     #ifdef DEBUGGER
00576                         if ( BUG.stdoutflag == 0 ) {
00577                             fprintf(stderr, "Panic: Cannot write to standard output!\n");
00578                             BUG.stdoutflag = 1;
00579                         }
00580                         BUG.eflag = 1; BUG.printflag = 1;
00581                     #else
00582                         Terminate(-1);
00583                     #endif
00584                 }
00585             }
00586             startinline = 0;
00587             Out = AO.OutputLine;
00588             if ( AO.BlockSpaces == 0 ) {
00589                 for ( j = 0; j < AO.OutSkip; j++ ) { *Out++ = ' '; }

```



```

00590         if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) ) {
00591             if ( AO.OutSkip == 7 ) {
00592                 Out[-2] = fcontchar;
00593                 Out[-1] = ' ';
00594             }
00595         }
00596     }
00597 /*
00598     Out = AO.OutputLine + AO.OutSkip;
00599     if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
00600         && AO.OutSkip == 7 ) {
00601         Out[-2] = fcontchar;
00602         Out[-1] = ' ';
00603     }
00604     else {
00605         for ( j = 0; j < AO.OutSkip; j++ ) { AO.OutputLine[j] = ' '; }
00606     }
00607 */
00608     if ( AO.IsBracket ) { *Out++ = ' '; *Out++ = ' '; *Out++ = ' '; }
00609     *Out = '\0';
00610     if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
00611         || ( AC.OutputMode == CMODE && AO.FactorMode == 0 ) ) AO.InFbrack++;
00612 }
00613 if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE ) {
00614     /* Very long numbers */
00615     if ( i > (WORD)(AO.OutStop-Out) ) j = (WORD)(AO.OutStop - Out);
00616     else j = i;
00617     i -= j;
00618     NCOPYB(Out,s,j);
00619 }
00620 else {
00621     if ( i > (WORD)(AO.OutStop-Out) ) j = (WORD)(AO.OutStop - Out - 1);
00622     else j = i;
00623     i -= j;
00624     NCOPYB(Out,s,j);
00625     if ( i > 0 ) *Out++ = '\\';
00626 }
00627 }
00628 *Out = '\0';
00629 AO.OutFill = Out;
00630 }
00631
00632 /*
00633     #] TokenToLine :
00634     #[ CodeToLine :          void CodeToLine(name,number,mode)
00635
00636     Writes a name and possibly its number to output as a single token.
00637
00638 */
00639
00640 UBYTE *CodeToLine(WORD number, UBYTE *Out)
00641 {
00642     Out = StrCopy((UBYTE *) "(", Out);
00643     Out = NumCopy(number, Out);
00644     Out = StrCopy((UBYTE *) ")", Out);
00645     return(Out);
00646 }
00647
00648 /*
00649     #] CodeToLine :
00650     #[ MultiplyToLine :
00651 */
00652
00653 void MultiplyToLine(void)
00654 {
00655     int i;
00656     if ( AO.CurrentDictionary > 0 && AO.CurDictSpecials > 0
00657         && AO.CurDictSpecials == DICT_DOSPECIALS ) {
00658         DICTIONARY *dict = AO.Dictionaries[AO.CurrentDictionary-1];
00659     /*
00660         Find the star:
00661     */
00662     for ( i = 0; i < dict->numelements; i++ ) {
00663         if ( dict->elements[i]->type != DICT_SPECIALCHARACTER ) continue;
00664         if ( (UBYTE)dict->elements[i]->lhs[0] == (UBYTE)('*') ) {
00665             TokenToLine((UBYTE *) (dict->elements[i]->rhs));
00666             return;
00667         }
00668     }
00669 }
00670 TokenToLine((UBYTE *) "*");
00671 }
00672
00673 /*
00674     #] MultiplyToLine :
00675     #[ AddArrayIndex :
00676 */

```

```

00677
00678 UBYTE *AddArrayIndex(WORD num, UBYTE *out)
00679 {
00680     if ( AC.OutputMode == CMODE ) {
00681         out = StrCopy((UBYTE *) "[", out);
00682         out = NumCopy(num, out);
00683         out = StrCopy((UBYTE *) "]", out);
00684     }
00685     else {
00686         out = StrCopy((UBYTE *) "(", out);
00687         out = NumCopy(num, out);
00688         out = StrCopy((UBYTE *) ")", out);
00689     }
00690     return(out);
00691 }
00692
00693 /*
00694     #] AddArrayIndex :
00695     #[ PrtTerms :      void PrtTerms()
00696 */
00697
00698 void PrtTerms(void)
00699 {
00700     UWORD a[2];
00701     WORD na;
00702     a[0] = (UWORD)AO.NumInBrack;
00703     a[1] = (UWORD)(AO.NumInBrack >> BITSINWORD);
00704     if ( a[1] ) na = 2;
00705     else na = 1;
00706     TokenToLine((UBYTE *) " ");
00707     LongToLine(a, na);
00708     if ( a[0] == 1 && na == 1 ) {
00709         TokenToLine((UBYTE *) " term");
00710     }
00711     else TokenToLine((UBYTE *) " terms");
00712     AO.NumInBrack = 0;
00713 }
00714
00715 /*
00716     #] PrtTerms :
00717     #[ WrtPower :
00718 */
00719
00720 UBYTE *WrtPower(UBYTE *Out, WORD Power)
00721 {
00722     if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
00723         || AC.OutputMode == REDUCEMODE ) {
00724         *Out++ = '*'; *Out++ = '*';
00725     }
00726     else if ( AC.OutputMode == CMODE ) *Out++ = ',';
00727     else {
00728         UBYTE *Out1 = IsExponentSign();
00729         if ( Out1 == 0 ) *Out++ = '^';
00730         else {
00731             while ( *Out1 ) *Out++ = *Out1++;
00732             *Out = 0;
00733         }
00734     }
00735     if ( Power >= 0 ) {
00736         if ( Power < 2*MAXPOWER )
00737             Out = NumCopy(Power, Out);
00738         else
00739             Out = StrCopy(FindSymbol((WORD)((LONG)Power-2*MAXPOWER)), Out);
00740     /*
00741         Out = StrCopy(VARNAME(symbols, (LONG)Power-2*MAXPOWER), Out); */
00742     if ( AC.OutputMode == CMODE ) *Out++ = ')';
00743     *Out = 0;
00744 }
00745     else {
00746         if ( ( AC.OutputMode >= FORTRANMODE || AC.OutputMode >= PFORTRANMODE
00747             || AC.OutputMode >= REDUCEMODE ) && AC.OutputMode != CMODE )
00748             *Out++ = '(';
00749         *Out++ = '-';
00750         if ( Power > -2*MAXPOWER )
00751             Out = NumCopy(-Power, Out);
00752         else
00753             Out = StrCopy(FindSymbol((WORD)((LONG)Power-2*MAXPOWER)), Out);
00754     /*
00755         Out = StrCopy(VARNAME(symbols, (LONG)(-Power)-2*MAXPOWER), Out); */
00756     if ( AC.OutputMode >= FORTRANMODE || AC.OutputMode >= PFORTRANMODE
00757         || AC.OutputMode >= REDUCEMODE ) *Out++ = ')';
00758     *Out = 0;
00759 }
00760     return(Out);
00761 }
00762
00763 /*
00764     #] WrtPower :
00765     #[ PrintTime :

```

```

00764 */
00765
00766 void PrintTime(UBYTE *mess)
00767 {
00768     LONG millitime = TimeCPU(1);
00769     WORD timepart = (WORD)(millitime%1000);
00770     millitime /= 1000;
00771     timepart /= 10;
00772     MesPrint("At %s: Time = %7l.%2i sec",mess,millitime,timepart);
00773 }
00774
00775 /*
00776     #] PrintTime :
00777     #] schryf-Utilities :
00778     #[ schryf-Writes :
00779     #[ WriteLists :         void WriteLists()
00780
00781     Writes the namelists. If mode > 0 also the internal codes are given.
00782
00783 */
00784
00785 static UBYTE *symname[] = {
00786     (UBYTE *)"(cyclic)", (UBYTE *)"(reversecyclic)"
00787     , (UBYTE *)"(symmetric)", (UBYTE *)"(antisymmetric)" };
00788 static UBYTE *rsymname[] = {
00789     (UBYTE *)"(-cyclic)", (UBYTE *)"(-reversecyclic)"
00790     , (UBYTE *)"(-symmetric)", (UBYTE *)"(-antisymmetric)" };
00791
00792 void WriteLists(void)
00793 {
00794     GETIDENTITY
00795     WORD i, j, k, *skip;
00796     int first, startvalue;
00797     UBYTE *OutScr, *Out;
00798     EXPRESSIONS e;
00799     CBUF *C = cbuf+AC.cbunum;
00800     int olddict = AO.CurrentDictionary;
00801     skip = &AO.OutSkip;
00802     *skip = 0;
00803     AO.OutputLine = AO.OutFill = (UBYTE *)AT.WorkPointer;
00804     AO.CurrentDictionary = 0;
00805     FiniLine();
00806     OutScr = (UBYTE *)AT.WorkPointer + ( TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer) ) /2;
00807     if ( AC.CodesFlag || AC.NamesFlag > 1 ) startvalue = 0;
00808     else startvalue = FIRSTUSERSYMBOL;
00809 /*
00810     #[ Symbols :
00811 */
00812     if ( ( j = NumSymbols ) > startvalue ) {
00813         TokenToLine((UBYTE *)" Symbols");
00814         *skip = 3;
00815         FiniLine();
00816         for ( i = startvalue; i < j; i++ ) {
00817             if ( i >= BUILTINSYMBOLS && i < FIRSTUSERSYMBOL ) continue;
00818             Out = StrCopy(VARNAME(symbols,i),OutScr);
00819             if ( symbols[i].minpower > -MAXPOWER || symbols[i].maxpower < MAXPOWER ) {
00820                 Out = StrCopy((UBYTE *)"(",Out);
00821                 if ( symbols[i].minpower > -MAXPOWER )
00822                     Out = NumCopy(symbols[i].minpower,Out);
00823                 Out = StrCopy((UBYTE *)":",Out);
00824                 if ( symbols[i].maxpower < MAXPOWER )
00825                     Out = NumCopy(symbols[i].maxpower,Out);
00826                 Out = StrCopy((UBYTE *)")",Out);
00827             }
00828             if ( ( symbols[i].complex & VARTYPEIMAGINARY ) == VARTYPEIMAGINARY ) {
00829                 Out = StrCopy((UBYTE *)"#i",Out);
00830             }
00831             else if ( ( symbols[i].complex & VARTYPECOMPLEX ) == VARTYPECOMPLEX ) {
00832                 Out = StrCopy((UBYTE *)"#c",Out);
00833             }
00834             else if ( ( symbols[i].complex & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY ) {
00835                 Out = StrCopy((UBYTE *)"#",Out);
00836                 if ( ( symbols[i].complex & VARTYPEMINUS ) == VARTYPEMINUS ) {
00837                     Out = StrCopy((UBYTE *)"-",Out);
00838                 }
00839                 else {
00840                     Out = StrCopy((UBYTE *)"+",Out);
00841                 }
00842                 Out = NumCopy(symbols[i].maxpower,Out);
00843             }
00844             if ( AC.CodesFlag ) Out = CodeToLine(i,Out);
00845             if ( ( symbols[i].complex & VARTYPECOMPLEX ) == VARTYPECOMPLEX ) i++;
00846             StrCopy((UBYTE *)" ",Out);
00847             TokenToLine(OutScr);
00848         }
00849         *skip = 0;
00850         FiniLine();

```

```

00851     }
00852  /*
00853     #] Symbols :
00854     #[ Indices :
00855  */
00856     if ( AC.CodesFlag || AC.NamesFlag > 1 ) startvalue = 0;
00857     else startvalue = BUILTININDICES;
00858     if ( ( j = NumIndices ) > startvalue ) {
00859         TokenToLine((UBYTE *) " Indices");
00860         *skip = 3;
00861         FiniLine();
00862         for ( i = startvalue; i < j; i++ ) {
00863             Out = StrCopy(FindIndex(i+AM.OffsetIndex),OutScr);
00864             Out = StrCopy(VARNAME(indices,i),OutScr);
00865             if ( indices[i].dimension >= 0 ) {
00866                 if ( indices[i].dimension != AC.lDefDim ) {
00867                     Out = StrCopy((UBYTE *) "=",Out);
00868                     Out = NumCopy(indices[i].dimension,Out);
00869                 }
00870             }
00871             else if ( indices[i].dimension < 0 ) {
00872                 Out = StrCopy((UBYTE *) "=",Out);
00873                 Out = StrCopy(VARNAME(symbols,-indices[i].dimension),Out);
00874                 if ( indices[i].nmin4 < -NMIN4SHIFT ) {
00875                     Out = StrCopy((UBYTE *) ":",Out);
00876                     Out = StrCopy(VARNAME(symbols,-indices[i].nmin4-NMIN4SHIFT),Out);
00877                 }
00878             }
00879             if ( AC.CodesFlag ) Out = CodeToLine(i+AM.OffsetIndex,Out);
00880             StrCopy((UBYTE *) " ",Out);
00881             TokenToLine(OutScr);
00882         }
00883         *skip = 0;
00884         FiniLine();
00885     }
00886  /*
00887     #] Indices :
00888     #[ Vectors :
00889  */
00890     if ( AC.CodesFlag || AC.NamesFlag > 1 ) startvalue = 0;
00891     else startvalue = BUILTINVECTORS;
00892     if ( ( j = NumVectors ) > startvalue ) {
00893         TokenToLine((UBYTE *) " Vectors");
00894         *skip = 3;
00895         FiniLine();
00896         for ( i = startvalue; i < j; i++ ) {
00897             Out = StrCopy(VARNAME(vectors,i),OutScr);
00898             if ( AC.CodesFlag ) Out = CodeToLine(i+AM.OffsetVector,Out);
00899             StrCopy((UBYTE *) " ",Out);
00900             TokenToLine(OutScr);
00901         }
00902         *skip = 0;
00903         FiniLine();
00904     }
00905  /*
00906     #] Vectors :
00907     #[ Functions :
00908  */
00909     if ( AC.CodesFlag || AC.NamesFlag > 1 ) startvalue = 0;
00910     else startvalue = AM.NumFixedFunctions;
00911     for ( k = 0; k < 2; k++ ) {
00912         first = 1;
00913         j = NumFunctions;
00914         for ( i = startvalue; i < j; i++ ) {
00915             if ( i > MAXBUILTINFUNCTION-FUNCTION
00916                 && i < FIRSTUSERFUNCTION-FUNCTION ) continue;
00917             if ( ( k == 0 && functions[i].commute )
00918                 || ( k != 0 && !functions[i].commute ) ) {
00919                 if ( first ) {
00920                     TokenToLine((UBYTE *) (FG.FunNam[k]));
00921                     *skip = 3;
00922                     FiniLine();
00923                     first = 0;
00924                 }
00925                 Out = StrCopy(VARNAME(functions,i),OutScr);
00926                 if ( ( functions[i].complex & VARTYPEIMAGINARY ) == VARTYPEIMAGINARY ) {
00927                     Out = StrCopy((UBYTE *) "#i",Out);
00928                 }
00929                 else if ( ( functions[i].complex & VARTYPECOMPLEX ) == VARTYPECOMPLEX ) {
00930                     Out = StrCopy((UBYTE *) "#c",Out);
00931                 }
00932                 if ( functions[i].spec == VERTEXFUNCTION ) {
00933                     Out = StrCopy((UBYTE *) "(Particle)",Out);
00934                 }
00935                 else if ( functions[i].spec >= TENSORFUNCTION ) {
00936                     Out = StrCopy((UBYTE *) "(Tensor)",Out);
00937                 }
00938             }

```

```

00938         if ( functions[i].symmetric > 0 ) {
00939             if ( ( functions[i].symmetric & REVERSEORDER ) != 0 ) {
00940                 Out = StrCopy((UBYTE *) (rsymname[ (functions[i].symmetric &
~REVERSEORDER)-1]), Out);
00941             }
00942             else {
00943                 Out = StrCopy((UBYTE *) (symname[functions[i].symmetric-1]), Out);
00944             }
00945         }
00946         if ( AC.CodesFlag ) Out = CodeToLine(i+FUNCTION, Out);
00947         if ( ( functions[i].complex & VARTYPECOMPLEX ) == VARTYPECOMPLEX ) i++;
00948         StrCopy((UBYTE *) " ", Out);
00949         TokenToLine(OutScr);
00950     }
00951 }
00952 *skip = 0;
00953 if ( first == 0 ) FiniLine();
00954 }
00955 /*
00956 #] Functions :
00957 #[ Sets :
00958 */
00959 if ( AC.CodesFlag || AC.NamesFlag > 1 ) startvalue = 0;
00960 else startvalue = AM.NumFixedSets;
00961 if ( ( j = AC.SetList.num ) > startvalue ) {
00962     WORD element, LastElement, type, number;
00963     TokenToLine((UBYTE *) " Sets");
00964     for ( i = startvalue; i < j; i++ ) {
00965         *skip = 3;
00966         FiniLine();
00967         if ( Sets[i].name < 0 ) {
00968             Out = StrCopy((UBYTE *) "{}", OutScr);
00969         }
00970         else {
00971             Out = StrCopy(VARNAME(Sets, i), OutScr);
00972         }
00973         if ( AC.CodesFlag ) Out = CodeToLine(i, Out);
00974         StrCopy((UBYTE *) ":", Out);
00975         TokenToLine(OutScr);
00976         if ( i < AM.NumFixedSets ) {
00977             TokenToLine((UBYTE *) " ");
00978             TokenToLine((UBYTE *) fixedsets[i].description);
00979         }
00980         else if ( Sets[i].type == CRANGE ) {
00981             int iflag = 0;
00982             if ( Sets[i].first == 3*MAXPOWER ) {
00983             }
00984             else if ( Sets[i].first >= MAXPOWER ) {
00985                 TokenToLine((UBYTE *) "<=");
00986                 NumCopy(Sets[i].first-2*MAXPOWER, OutScr);
00987                 TokenToLine(OutScr);
00988                 iflag = 1;
00989             }
00990             else {
00991                 TokenToLine((UBYTE *) "<");
00992                 NumCopy(Sets[i].first, OutScr);
00993                 TokenToLine(OutScr);
00994                 iflag = 1;
00995             }
00996             if ( Sets[i].last == -3*MAXPOWER ) {
00997             }
00998             else if ( Sets[i].last <= -MAXPOWER ) {
00999                 if ( iflag ) TokenToLine((UBYTE *) ",");
01000                 TokenToLine((UBYTE *) ">=");
01001                 NumCopy(Sets[i].last+2*MAXPOWER, OutScr);
01002                 TokenToLine(OutScr);
01003             }
01004             else {
01005                 if ( iflag ) TokenToLine((UBYTE *) ",");
01006                 TokenToLine((UBYTE *) ">");
01007                 NumCopy(Sets[i].last, OutScr);
01008                 TokenToLine(OutScr);
01009             }
01010         }
01011         else {
01012             element = Sets[i].first;
01013             LastElement = Sets[i].last;
01014             type = Sets[i].type;
01015             while ( element < LastElement ) {
01016                 TokenToLine((UBYTE *) " ");
01017                 number = SetElements[element++];
01018                 switch ( type ) {
01019                     case CSYMBOL:
01020                         if ( number < 0 ) {
01021                             StrCopy(VARNAME(symbols, -number), OutScr);
01022                             StrCopy((UBYTE *) "?", Out);
01023                             TokenToLine(OutScr);

```

```

01024         }
01025         else if ( number < MAXPOWER )
01026             TokenToLine(VARNAME(symbols,number));
01027         else {
01028             NumCopy(number-2*MAXPOWER,OutScr);
01029             TokenToLine(OutScr);
01030         }
01031         break;
01032     case CINDEX:
01033         if ( number >= AM.IndDum ) {
01034             Out = StrCopy((UBYTE *) "N",OutScr);
01035             Out = NumCopy(number-(AM.IndDum),Out);
01036             StrCopy((UBYTE *) "_?",Out);
01037             TokenToLine(OutScr);
01038         }
01039         else if ( number >= AM.OffsetIndex + (WORD)WILDMASK ) {
01040             Out = StrCopy(VARNAME(indices,number
01041                             -AM.OffsetIndex-WILDMASK),OutScr);
01042             StrCopy((UBYTE *) "?",Out);
01043             TokenToLine(OutScr);
01044         }
01045         else if ( number >= AM.OffsetIndex ) {
01046             TokenToLine(VARNAME(indices,number-AM.OffsetIndex));
01047         }
01048         else {
01049             NumCopy(number,OutScr);
01050             TokenToLine(OutScr);
01051         }
01052         break;
01053     case CVECTOR:
01054         Out = OutScr;
01055         if ( number < AM.OffsetVector ) {
01056             number += WILDMASK;
01057             Out = StrCopy((UBYTE *) "-",Out);
01058         }
01059         if ( number >= AM.OffsetVector + WILDOFFSET ) {
01060             Out = StrCopy(VARNAME(vectors,number
01061                             -AM.OffsetVector-WILDOFFSET),Out);
01062             StrCopy((UBYTE *) "?",Out);
01063         }
01064         else {
01065             Out = StrCopy(VARNAME(vectors,number-AM.OffsetVector),Out);
01066         }
01067         TokenToLine(OutScr);
01068         break;
01069     case CFUNCTION:
01070         if ( number >= FUNCTION + (WORD)WILDMASK ) {
01071             Out = StrCopy(VARNAME(functions,number
01072                             -FUNCTION-WILDMASK),OutScr);
01073             StrCopy((UBYTE *) "?",Out);
01074             TokenToLine(OutScr);
01075         }
01076         TokenToLine(VARNAME(functions,number-FUNCTION));
01077         break;
01078     default:
01079         NumCopy(number,OutScr);
01080         TokenToLine(OutScr);
01081         break;
01082     }
01083 }
01084 }
01085 }
01086 *skip = 0;
01087 FiniLine();
01088 }
01089 /*
01090     #] Sets :
01091     #[ Expressions :
01092 */
01093 if ( AS.ExecMode ) {
01094     e = Expressions;
01095     j = NumExpressions;
01096     first = 1;
01097     for ( i = 0; i < j; i++, e++ ) {
01098         if ( e->status >= 0 ) {
01099             if ( first ) {
01100                 TokenToLine((UBYTE *) " Expressions");
01101                 *skip = 3;
01102                 FiniLine();
01103                 first = 0;
01104             }
01105             Out = StrCopy(AC.exprnames->namebuffer+e->name,OutScr);
01106             Out = StrCopy((UBYTE *) (FG.ExprStat[e->status]),Out);
01107             if ( AC.CodesFlag ) Out = CodeToLine(i,Out);
01108             StrCopy((UBYTE *) " ",Out);
01109             TokenToLine(OutScr);
01110         }
01111     }

```

```

01111     }
01112     if ( !first ) {
01113         *skip = 0;
01114         FiniLine();
01115     }
01116 }
01117 e = Expressions;
01118 j = NumExpressions;
01119 first = 1;
01120 for ( i = 0; i < j; i++ ) {
01121     if ( e->printflag && ( e->status == LOCALEXPRESSION ||
01122         e->status == GLOBALEXPRESSION || e->status == UNHIDELEXPRESSION
01123         || e->status == UNHIDEGEXPRESSION ) ) {
01124         if ( first ) {
01125             TokenToLine((UBYTE *) " Expressions to be printed");
01126             *skip = 3;
01127             FiniLine();
01128             first = 0;
01129         }
01130         Out = StrCopy(AC.exprnames->namebuffer+e->name,OutScr);
01131         StrCopy((UBYTE *) " ",Out);
01132         TokenToLine(OutScr);
01133     }
01134     e++;
01135 }
01136 if ( !first ) {
01137     *skip = 0;
01138     FiniLine();
01139 }
01140 /*
01141     #] Expressions :
01142     #[ Dollars :
01143 */
01144
01145 if ( AC.CodesFlag || AC.NamesFlag > 1 ) startvalue = 0;
01146 else startvalue = BUILTINDOLLARS;
01147 if ( ( j = NumDollars ) > startvalue ) {
01148     TokenToLine((UBYTE *) " Dollar variables");
01149     *skip = 3;
01150     FiniLine();
01151     for ( i = startvalue; i < j; i++ ) {
01152         Out = StrCopy((UBYTE *) "$", OutScr);
01153         Out = StrCopy(DOLLARNAME(Dollars, i), Out);
01154         if ( AC.CodesFlag ) Out = CodeToLine(i, Out);
01155         StrCopy((UBYTE *) " ", Out);
01156         TokenToLine(OutScr);
01157     }
01158     *skip = 0;
01159     FiniLine();
01160 }
01161
01162 if ( ( j = NumPotModdollars ) > 0 ) {
01163     TokenToLine((UBYTE *) " Dollar variables to be modified");
01164     *skip = 3;
01165     FiniLine();
01166     for ( i = 0; i < j; i++ ) {
01167         Out = StrCopy((UBYTE *) "$", OutScr);
01168         Out = StrCopy(DOLLARNAME(Dollars, PotModdollars[i]), Out);
01169         for ( k = 0; k < NumModOptdollars; k++ )
01170             if ( ModOptdollars[k].number == PotModdollars[i] ) break;
01171         if ( k < NumModOptdollars ) {
01172             switch ( ModOptdollars[k].type ) {
01173                 case MODSUM:
01174                     Out = StrCopy((UBYTE *) "(sum)", Out);
01175                     break;
01176                 case MODMAX:
01177                     Out = StrCopy((UBYTE *) "(maximum)", Out);
01178                     break;
01179                 case MODMIN:
01180                     Out = StrCopy((UBYTE *) "(minimum)", Out);
01181                     break;
01182                 case MODLOCAL:
01183                     Out = StrCopy((UBYTE *) "(local)", Out);
01184                     break;
01185                 default:
01186                     Out = StrCopy((UBYTE *) "(?)", Out);
01187                     break;
01188             }
01189         }
01190         StrCopy((UBYTE *) " ", Out);
01191         TokenToLine(OutScr);
01192     }
01193     *skip = 0;
01194     FiniLine();
01195 }
01196 /*
01197     #] Dollars :

```

```

01198 */
01199
01200     if ( AC.ncmod != 0 ) {
01201         TokenToLine((UBYTE *)"All arithmetic is modulus ");
01202         LongToLine((UWORD *)AC.cmod,ABS(AC.ncmod));
01203         if ( AC.ncmod > 0 ) TokenToLine((UBYTE *)" with powerreduction");
01204         else                 TokenToLine((UBYTE *)" without powerreduction");
01205         if ( ( AC.modmode & POSNEG ) != 0 ) TokenToLine((UBYTE *)" centered around 0");
01206         else                 TokenToLine((UBYTE *)" positive numbers only");
01207         FiniLine();
01208     }
01209     if ( AC.lDefDim != 4 ) {
01210         TokenToLine((UBYTE *)"The default dimension is ");
01211         if ( AC.lDefDim >= 0 ) {
01212             NumCopy(AC.lDefDim,OutScr);
01213             TokenToLine(OutScr);
01214         }
01215         else {
01216             TokenToLine(VARNAME(symbols,-AC.lDefDim));
01217             if ( AC.lDefDim4 != -NMIN4SHIFT ) {
01218                 TokenToLine((UBYTE *)"");
01219                 if ( AC.lDefDim4 >= -NMIN4SHIFT ) {
01220                     NumCopy(AC.lDefDim4,OutScr);
01221                     TokenToLine(OutScr);
01222                 }
01223                 else {
01224                     TokenToLine(VARNAME(symbols,-AC.lDefDim4-NMIN4SHIFT));
01225                 }
01226             }
01227         }
01228         FiniLine();
01229     }
01230     if ( AC.lUnitTrace != 4 ) {
01231         TokenToLine((UBYTE *)"The trace of the unit matrix is ");
01232         if ( AC.lUnitTrace >= 0 ) {
01233             NumCopy(AC.lUnitTrace,OutScr);
01234             TokenToLine(OutScr);
01235         }
01236         else {
01237             TokenToLine(VARNAME(symbols,-AC.lUnitTrace));
01238         }
01239         FiniLine();
01240     }
01241     if ( AO.NumDictionaries > 0 ) {
01242         for ( i = 0; i < AO.NumDictionaries; i++ ) {
01243             WriteDictionary(AO.Dictionaries[i]);
01244         }
01245         if ( olddict > 0 )
01246             MesPrint("\nCurrently dictionary %s is active\n",
01247                     AO.Dictionaries[olddict-1]->name);
01248         else
01249             MesPrint("\nCurrently there is no active dictionary\n");
01250     }
01251     if ( AC.CodesFlag ) {
01252         if ( C->numlhs > 0 ) {
01253             TokenToLine((UBYTE *)" Left Hand Sides:");
01254             AO.OutSkip = 3;
01255             for ( i = 1; i <= C->numlhs; i++ ) {
01256                 FiniLine();
01257                 skip = C->lhs[i];
01258                 j = skip[1];
01259                 while ( --j >= 0 ) { TalToLine((UWORD)(*skip++)); TokenToLine((UBYTE *)" "); }
01260             }
01261             AO.OutSkip = 0;
01262             FiniLine();
01263         }
01264         if ( C->numrhs > 0 ) {
01265             TokenToLine((UBYTE *)" Right Hand Sides:");
01266             AO.OutSkip = 3;
01267             for ( i = 1; i <= C->numrhs; i++ ) {
01268                 FiniLine();
01269                 skip = C->rhs[i];
01270                 while ( ( j = skip[0] ) != 0 ) {
01271                     while ( --j >= 0 ) { TalToLine((UWORD)(*skip++)); TokenToLine((UBYTE *)" "); }
01272                 }
01273                 FiniLine();
01274             }
01275             AO.OutSkip = 0;
01276             FiniLine();
01277         }
01278     }
01279     AO.CurrentDictionary = olddict;
01280 }
01281
01282 /*
01283     #] WriteLists :
01284     #[ WriteDictionary :

```



```

01285
01286     This routine is part of WriteLists and should be called from there.
01287 */
01288
01289 void WriteDictionary(DICTIONARY *dict)
01290 {
01291     GETIDENTITY
01292     int i, first;
01293     WORD *skip, na, *a, spec, *t, *tstop, j;
01294     UBYTE str[2], *OutScr, *Out;
01295     WORD oldoutputmode = AC.OutputMode, oldoutputsaces = AC.OutputSpaces;
01296     WORD oldoutskip = AO.OutSkip;
01297     AC.OutputMode = NORMALFORMAT;
01298     AC.OutputSpaces = NOSPACEFORMAT;
01299     MesPrint("===Contents of dictionary %s===", dict->name);
01300     skip = &AO.OutSkip;
01301     *skip = 3;
01302     AO.OutputLine = AO.OutFill = (UBYTE *)AT.WorkPointer;
01303     for ( j = 0; j < *skip; j++ ) *(AO.OutFill)++ = ' ';
01304
01305     OutScr = (UBYTE *)AT.WorkPointer + ( TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer) ) /2;
01306     for ( i = 0; i < dict->numelements; i++ ) {
01307         switch ( dict->elements[i]->type ) {
01308             case DICT_INTEGERNUMBER:
01309                 LongToLine((UWORD *) (dict->elements[i]->lhs), dict->elements[i]->size);
01310                 Out = OutScr; *Out = 0;
01311                 break;
01312             case DICT_RATIONALNUMBER:
01313                 a = dict->elements[i]->lhs;
01314                 na = a[a[0]-1]; na = (ABS(na)-1)/2;
01315                 RatToLine((UWORD *) (a+1), na);
01316                 Out = OutScr; *Out = 0;
01317                 break;
01318             case DICT_SYMBOL:
01319                 na = dict->elements[i]->lhs[0];
01320                 Out = StrCopy(VARNAME(symbols, na), OutScr);
01321                 break;
01322             case DICT_VECTOR:
01323                 na = dict->elements[i]->lhs[0]-AM.OffsetVector;
01324                 Out = StrCopy(VARNAME(vectors, na), OutScr);
01325                 break;
01326             case DICT_INDEX:
01327                 na = dict->elements[i]->lhs[0]-AM.OffsetIndex;
01328                 Out = StrCopy(VARNAME(indices, na), OutScr);
01329                 break;
01330             case DICT_FUNCTION:
01331                 na = dict->elements[i]->lhs[0]-FUNCTION;
01332                 Out = StrCopy(VARNAME(functions, na), OutScr);
01333                 break;
01334             case DICT_FUNCTION_WITH_ARGUMENTS:
01335                 t = dict->elements[i]->lhs;
01336                 na = *t-FUNCTION;
01337                 Out = StrCopy(VARNAME(functions, na), OutScr);
01338                 spec = functions[*t - FUNCTION].spec;
01339                 tstop = t + t[1];
01340                 first = 1;
01341                 if ( t[1] <= FUNHEAD ) {}
01342                 else if ( spec >= TENSORFUNCTION ) {
01343                     t += FUNHEAD; *Out++ = (UBYTE) '(';
01344                     while ( t < tstop ) {
01345                         if ( first == 0 ) *Out++ = (UBYTE) (',' );
01346                         else first = 0;
01347                         j = *t++;
01348                         if ( j >= 0 ) {
01349                             if ( j < AM.OffsetIndex ) { Out = NumCopy(j, Out); }
01350                             else if ( j < AM.IndDum ) {
01351                                 Out = StrCopy(VARNAME(indices, j-AM.OffsetIndex), Out);
01352                             }
01353                             else {
01354                                 MesPrint("Currently wildcards are not allowed in dictionary
01355 elements");
01356                                 Terminate(-1);
01357                             }
01358                         }
01359                         else {
01360                             Out = StrCopy(VARNAME(vectors, j-AM.OffsetVector), Out);
01361                         }
01362                     }
01363                     *Out++ = (UBYTE) ')'; *Out = 0;
01364                 }
01365             else {
01366                 t += FUNHEAD; *Out++ = (UBYTE) '('; *Out = 0;
01367                 TokenToLine(OutScr);
01368                 while ( t < tstop ) {
01369                     if ( !first ) TokenToLine((UBYTE *) "", "");
01370                     WriteArgument(t);
01371                     NEXTARG(t)

```

```

01371             first = 0;
01372         }
01373         Out = OutScr;
01374         *Out++ = (UBYTE)'\''; *Out = 0;
01375     }
01376     break;
01377     case DICT_SPECIALCHARACTER:
01378         str[0] = (UBYTE)(dict->elements[i]->lhs[0]);
01379         str[1] = 0;
01380         Out = StrCopy(str, OutScr);
01381         break;
01382     default:
01383         Out = OutScr; *Out = 0;
01384         break;
01385 }
01386 Out = StrCopy((UBYTE *)": \"", Out);
01387 Out = StrCopy((UBYTE *) (dict->elements[i]->rhs), Out);
01388 Out = StrCopy((UBYTE *) "\"", Out);
01389 TokenToLine(OutScr);
01390 FiniLine();
01391 }
01392 MesPrint("====End of dictionary %s====", dict->name);
01393 AC.OutputMode = oldoutputmode;
01394 AC.OutputSpaces = oldoutputspaces;
01395 AO.OutSkip = oldoutskip;
01396 }
01397
01398 /*
01399     #] WriteDictionary :
01400     #[ WriteArgument :      void WriteArgument(WORD *t)
01401
01402     Write a single argument field. The general field goes to
01403     WriteExpression and the fast field is dealt with here.
01404 */
01405 void WriteArgument(WORD *t)
01406 {
01407     UBYTE buffer[180];
01408     UBYTE *Out;
01409     WORD i;
01410     int oldoutsidfun, oldlowestlevel = lowestlevel;
01411     lowestlevel = 0;
01412     if ( *t > 0 ) {
01413         oldoutsidfun = AC.outsidfun; AC.outsidfun = 0;
01414         WriteExpression(t+ARGHEAD, (LONG)(t-ARGHEAD));
01415         AC.outsidfun = oldoutsidfun;
01416         goto CleanUp;
01417     }
01418     Out = buffer;
01419     if ( *t == -SNUMBER ) {
01420         NumCopy(t[1], Out);
01421     }
01422     else if ( *t == -SYMBOL ) {
01423         if ( t[1] >= MAXVARIABLES-cbuf[AM.sbufnum].numrhs ) {
01424             Out = StrCopy(FindExtraSymbol(MAXVARIABLES-t[1]), Out);
01425         }
01426         /*
01427             Out = StrCopy((UBYTE *)AC.extrasym, Out);
01428             if ( AC.extrasymbols == 0 ) {
01429                 Out = NumCopy((MAXVARIABLES-t[1]), Out);
01430                 Out = StrCopy((UBYTE *) "_", Out);
01431             }
01432             else if ( AC.extrasymbols == 1 ) {
01433                 Out = AddArrayIndex((MAXVARIABLES-t[1]), Out);
01434             }
01435         */
01436         /*
01437             else if ( AC.extrasymbols == 2 ) {
01438                 Out = NumCopy((MAXVARIABLES-t[1]), Out);
01439             }
01440         */
01441     }
01442     else {
01443         StrCopy(FindSymbol(t[1]), Out);
01444         /*
01445             StrCopy(VARNAME(symbols, t[1]), Out); */
01446     }
01447     else if ( *t == -VECTOR ) {
01448         if ( t[1] == FUNNYVEC ) { *Out++ = '?'; *Out = 0; }
01449         else
01450             StrCopy(FindVector(t[1]), Out);
01451         /*
01452             StrCopy(VARNAME(vectors, t[1] - AM.OffsetVector), Out); */
01453     }
01454     else if ( *t == -MINVECTOR ) {
01455         *Out++ = '-';
01456         StrCopy(FindVector(t[1]), Out);
01457         /*
01458             StrCopy(VARNAME(vectors, t[1] - AM.OffsetVector), Out); */
01459     }
01460 }

```

```

01458     else if ( *t == -INDEX ) {
01459         if ( t[1] >= 0 ) {
01460             if ( t[1] < AM.OffsetIndex ) { NumCopy(t[1],Out); }
01461             else {
01462                 i = t[1];
01463                 if ( i >= AM.IndDum ) {
01464                     i -= AM.IndDum;
01465                     *Out++ = 'N';
01466                     Out = NumCopy(i,Out);
01467                     *Out++ = '-';
01468                     *Out++ = '?';
01469                     *Out = 0;
01470                 }
01471                 else {
01472                     i -= AM.OffsetIndex;
01473                     Out = StrCopy(FindIndex(i%WILDOFFSET+AM.OffsetIndex),Out);
01474                     /* Out = StrCopy(VARNAME(indices,i%WILDOFFSET),Out); */
01475                     if ( i >= WILDOFFSET ) { *Out++ = '?'; *Out = 0; }
01476                 }
01477             }
01478         }
01479         else if ( t[1] == FUNNYVEC ) { *Out++ = '?'; *Out = 0; }
01480         else
01481             StrCopy(FindVector(t[1]),Out);
01482         /* StrCopy(VARNAME(vectors,t[1] - AM.OffsetVector),Out); */
01483     }
01484     else if ( *t == -DOLLAREXPRESSION ) {
01485         DOLLARS d = Dollars + t[1];
01486         *Out++ = '$';
01487         StrCopy(AC.dollarnames->namebuffer+d->name,Out);
01488     }
01489     else if ( *t == -EXPRESSION ) {
01490         StrCopy(EXPRNAME(t[1]),Out);
01491     }
01492     else if ( *t == -SETSET ) {
01493         StrCopy(VARNAME(Sets,t[1]),Out);
01494     }
01495     else if ( *t <= -FUNCTION ) {
01496         StrCopy(FindFunction(-*t),Out);
01497         /* StrCopy(VARNAME(functions,-*t-FUNCTION),Out); */
01498     }
01499     else {
01500         MesPrint("Illegal function argument while writing");
01501         goto CleanUp;
01502     }
01503     TokenToLine(buffer);
01504 CleanUp:
01505     lowestlevel = oldlowestlevel;
01506     return;
01507 }
01508
01509 /*
01510     #] WriteArgument :
01511     #[ WriteSubTerm :      WORD WriteSubTerm(sterm,first)
01512
01513     Writes a single subterm field to the output line.
01514     There is a recursion for functions.
01515
01516 #define NUMSPECS 8
01517 UBYTE *specfunnames[NUMSPECS] = {
01518     (UBYTE *)"fac", (UBYTE *)"nargs", (UBYTE *)"binom"
01519     , (UBYTE *)"sign", (UBYTE *)"mod", (UBYTE *)"min", (UBYTE *)"max"
01520     , (UBYTE *)"invfac" };
01521 */
01522
01523 int WriteSubTerm(WORD *sterm, WORD first)
01524 {
01525     UBYTE buffer[80];
01526     UBYTE *Out, closepar[2] = { (UBYTE)')', 0};
01527     WORD *stopper, *tt, *t, i, j, po = 0;
01528     int oldoutsidefun;
01529     stopper = sterm + sterm[1];
01530     t = sterm + 2;
01531     switch ( *sterm ) {
01532     case SYMBOL :
01533         while ( t < stopper ) {
01534             if ( lowestlevel && ( ( AO.PrintType & PRINTALL ) != 0 ) ) {
01535                 FiniLine();
01536                 if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
01537                 else IniLine(3);
01538                 if ( first ) TokenToLine((UBYTE *)" ");
01539             }
01540             if ( !first ) MultiplyToLine();
01541             if ( AC.OutputMode == CMODE && t[1] != 1 ) {
01542                 if ( AC.Cnumpows >= t[1] && t[1] > 0 ) {
01543                     po = t[1];
01544

```

```

01545         Out = StrCopy((UBYTE *)"POW",buffer);
01546         Out = NumCopy(po,Out);
01547         Out = StrCopy((UBYTE *)"(",Out);
01548         TokenToLine(buffer);
01549     }
01550     else {
01551         TokenToLine((UBYTE *)"pow(");
01552     }
01553 }
01554 if ( *t < NumSymbols ) {
01555     Out = StrCopy(FindSymbol(*t),buffer); t++;
01556 /*     Out = StrCopy(VARNAME(symbols,*t),buffer); t++; */
01557 }
01558 else {
01559 /*
01560     see also routine PrintSubtermList.
01561 */
01562     Out = StrCopy(FindExtraSymbol(MAXVARIABLES-*t),buffer);
01563 /*
01564     Out = StrCopy((UBYTE *)AC.extrasym,buffer);
01565     if ( AC.extrasymbols == 0 ) {
01566         Out = NumCopy((MAXVARIABLES-*t),Out);
01567         Out = StrCopy((UBYTE *)"_",Out);
01568     }
01569     else if ( AC.extrasymbols == 1 ) {
01570         Out = AddArrayIndex((MAXVARIABLES-*t),Out);
01571     }
01572 */
01573 /*
01574     else if ( AC.extrasymbols == 2 ) {
01575         Out = NumCopy((MAXVARIABLES-*t),Out);
01576     }
01577 */
01578     t++;
01579 }
01580 if ( AC.OutputMode == CMODE && po > 1
01581     && AC.Cnumpows >= po ) {
01582     Out = StrCopy((UBYTE *)"") ,Out);
01583     po = 0;
01584 }
01585 else if ( *t != 1 ) WrtPower(Out,*t);
01586 TokenToLine(buffer);
01587 t++;
01588 first = 0;
01589 }
01590 break;
01591 case VECTOR :
01592     while ( t < stopper ) {
01593         if ( lowestlevel && ( ( AO.PrintType & PRINTALL ) != 0 ) ) {
01594             FiniLine();
01595             if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
01596             else IniLine(3);
01597             if ( first ) TokenToLine((UBYTE *)" ");
01598         }
01599         if ( !first ) MultiplyToLine();
01600
01601         Out = StrCopy(FindVector(*t),buffer);
01602 /*     Out = StrCopy(VARNAME(vectors,*t - AM.OffsetVector),buffer); */
01603         t++;
01604         if ( AC.OutputMode == MATHEMATICAMODE ) *Out++ = '[';
01605         else *Out++ = '(';
01606         if ( *t >= AM.OffsetIndex ) {
01607             i = *t++;
01608             if ( i >= AM.IndDum ) {
01609                 i -= AM.IndDum;
01610                 *Out++ = 'N';
01611                 Out = NumCopy(i,Out);
01612                 *Out++ = '_';
01613                 *Out++ = '?';
01614                 *Out = 0;
01615             }
01616             else
01617                 Out = StrCopy(FindIndex(i),Out);
01618 /*     Out = StrCopy(VARNAME(indices,i - AM.OffsetIndex),Out); */
01619         }
01620         else if ( *t == FUNNYVEC ) { *Out++ = '?'; *Out = 0; }
01621         else {
01622             Out = NumCopy(*t++,Out);
01623         }
01624         if ( AC.OutputMode == MATHEMATICAMODE ) *Out++ = ']';
01625         else *Out++ = ')';
01626         *Out = 0;
01627         TokenToLine(buffer);
01628         first = 0;
01629     }
01630     break;
01631 case INDEX :

```

```

01632         while ( t < stopper ) {
01633             if ( lowestlevel && ( ( AO.PrintType & PRINTALL ) != 0 ) ) {
01634                 FiniLine();
01635                 if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
01636                 else IniLine(3);
01637                 if ( first ) TokenToLine((UBYTE *) " ");
01638             }
01639             if ( !first ) MultiplyToLine();
01640             if ( *t >= 0 ) {
01641                 if ( *t < AM.OffsetIndex ) {
01642                     TalToLine((UWORD) (*t++));
01643                 }
01644                 else {
01645                     i = *t++;
01646                     if ( i >= AM.IndDum ) {
01647                         i -= AM.IndDum;
01648                         Out = buffer;
01649                         *Out++ = 'N';
01650                         Out = NumCopy(i, Out);
01651                         *Out++ = '_';
01652                         *Out++ = '?';
01653                         *Out = 0;
01654                     }
01655                     else {
01656                         i -= AM.OffsetIndex;
01657                         Out = StrCopy(FindIndex(i%WILDOFFSET+AM.OffsetIndex), buffer);
01658                         Out = StrCopy(VARNAME(indices, i%WILDOFFSET), buffer); /*
01659                         if ( i >= WILDOFFSET ) { *Out++ = '?'; *Out = 0; }
01660                     }
01661                     TokenToLine(buffer);
01662                 }
01663             }
01664             else {
01665                 TokenToLine(FindVector(*t)); t++;
01666                 /* TokenToLine(VARNAME(vectors, *t - AM.OffsetVector)); t++; */
01667             }
01668             first = 0;
01669         }
01670         break;
01671     case DOLLAREXPRESSION:
01672     {
01673         DOLLARS d = Dollars + sterm[2];
01674         Out = StrCopy((UBYTE *) "$", buffer);
01675         Out = StrCopy(AC.dollarnames->namebuffer+d->name, Out);
01676         if ( sterm[3] != 1 ) WrtPower(Out, sterm[3]);
01677         TokenToLine(buffer);
01678     }
01679     first = 0;
01680     break;
01681     case DELTA :
01682         while ( t < stopper ) {
01683             if ( lowestlevel && ( ( AO.PrintType & PRINTALL ) != 0 ) ) {
01684                 FiniLine();
01685                 if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
01686                 else IniLine(3);
01687                 if ( first ) TokenToLine((UBYTE *) " ");
01688             }
01689             if ( !first ) MultiplyToLine();
01690             Out = StrCopy((UBYTE *) "d_", buffer);
01691             if ( *t >= AM.OffsetIndex ) {
01692                 if ( *t < AM.IndDum ) {
01693                     Out = StrCopy(FindIndex(*t), Out);
01694                     Out = StrCopy(VARNAME(indices, *t - AM.OffsetIndex), Out); /*
01695                     t++;
01696                 }
01697                 else {
01698                     *Out++ = 'N';
01699                     Out = NumCopy( *t++ - AM.IndDum, Out);
01700                     *Out++ = '_';
01701                     *Out++ = '?';
01702                     *Out = 0;
01703                 }
01704             }
01705             else if ( *t == FUNNYVEC ) { *Out++ = '?'; *Out = 0; }
01706             else {
01707                 Out = NumCopy(*t++, Out);
01708             }
01709             *Out++ = ',';
01710             if ( *t >= AM.OffsetIndex ) {
01711                 if ( *t < AM.IndDum ) {
01712                     Out = StrCopy(FindIndex(*t), Out);
01713                     Out = StrCopy(VARNAME(indices, *t - AM.OffsetIndex), Out); /*
01714                     t++;
01715                 }
01716                 else {
01717                     *Out++ = 'N';
01718                     Out = NumCopy(*t++ - AM.IndDum, Out);

```

```

01719             *Out++ = ' _';
01720             *Out++ = ' ?';
01721         }
01722     }
01723     else {
01724         Out = NumCopy(*t++,Out);
01725     }
01726     *Out++ = ')';
01727     *Out = 0;
01728     TokenToLine(buffer);
01729     first = 0;
01730 }
01731 break;
01732 case DOTPRODUCT :
01733     while ( t < stopper ) {
01734         if ( lowestlevel && ( ( AO.PrintType & PRINTALL ) != 0 ) ) {
01735             FiniLine();
01736             if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
01737             else IniLine(3);
01738             if ( first ) TokenToLine((UBYTE *) " ");
01739         }
01740         if ( !first ) MultiplyToLine();
01741         if ( AC.OutputMode == CMODE && t[2] != 1 )
01742             TokenToLine((UBYTE *) "pow(");
01743         if ( AC.OutputMode == MATHEMATICAMODE )
01744             TokenToLine((UBYTE *) "(");
01745         Out = StrCopy(FindVector(*t),buffer);
01746         /* Out = StrCopy(VARNAME(vectors,*t - AM.OffsetVector),buffer); */
01747         t++;
01748         if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
01749             || AC.OutputMode == CMODE )
01750             *Out++ = AO.FortDotChar;
01751         else *Out++ = ' .';
01752         Out = StrCopy(FindVector(*t),Out);
01753         /* Out = StrCopy(VARNAME(vectors,*t - AM.OffsetVector),Out); */
01754         if ( AC.OutputMode == MATHEMATICAMODE ) {
01755             *Out++ = ')';
01756             *Out = 0;
01757         }
01758         t++;
01759         if ( *t != 1 ) WrtPower(Out,*t);
01760         t++;
01761         TokenToLine(buffer);
01762         first = 0;
01763     }
01764     break;
01765 case EXPONENT :
01766     #if FUNHEAD != 2
01767         t += FUNHEAD - 2;
01768     #endif
01769     if ( !first ) MultiplyToLine();
01770     if ( AC.OutputMode == CMODE ) TokenToLine((UBYTE *) "pow(");
01771     else TokenToLine((UBYTE *) "(");
01772     WriteArgument(t);
01773     if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
01774         || AC.OutputMode == REDUCEMODE )
01775         TokenToLine((UBYTE *) ")**(");
01776     else if ( AC.OutputMode == CMODE ) TokenToLine((UBYTE *) ",");
01777     else {
01778         UBYTE *Out1 = IsExponentSign();
01779         if ( Out1 ) {
01780             TokenToLine((UBYTE *) ")");
01781             TokenToLine(Out1);
01782             TokenToLine((UBYTE *) "(");
01783         }
01784         else TokenToLine((UBYTE *) ")^(");
01785     }
01786     NEXTARG(t)
01787     WriteArgument(t);
01788     TokenToLine((UBYTE *) ")");
01789     break;
01790 case DENOMINATOR :
01791     #if FUNHEAD != 2
01792         t += FUNHEAD - 2;
01793     #endif
01794     if ( first ) TokenToLine((UBYTE *) "1/(");
01795     else TokenToLine((UBYTE *) "/"");
01796     WriteArgument(t);
01797     TokenToLine((UBYTE *) ")");
01798     break;
01799 case SUBEXPRESSION:
01800     if ( !first ) MultiplyToLine();
01801     TokenToLine((UBYTE *) "(");
01802     t = cbuf[sterm[4]].rhs[sterm[2]];
01803     tt = t;
01804     while ( *tt ) tt += *tt;
01805     oldoutsidefun = AC.outsidefun; AC.outsidefun = 0;

```

```

01806         if ( *t ) {
01807             WriteExpression(t, (LONG)(tt-t));
01808         }
01809         else {
01810             TokenToLine((UBYTE *)"0");
01811         }
01812         AC.outsidefun = oldoutsidefun;
01813         TokenToLine((UBYTE *)"");
01814         if ( sterm[3] != 1 ) {
01815             UBYTE *Out1 = IsExponentSign();
01816             if ( Out1 ) TokenToLine(Out1);
01817             else TokenToLine((UBYTE *)"^");
01818             Out = buffer;
01819             NumCopy(sterm[3], Out);
01820             TokenToLine(buffer);
01821         }
01822         break;
01823     default :
01824         if ( lowestlevel && ( ( AO.PrintType & PRINTALL ) != 0 ) ) {
01825             FiniLine();
01826             if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
01827             else IniLine(3);
01828             if ( first ) TokenToLine((UBYTE *)" ");
01829         }
01830         if ( *sterm < FUNCTION ) {
01831             return(MesPrint("Illegal subterm while writing"));
01832         }
01833         if ( !first ) MultiplyToLine();
01834         first = 1;
01835         { UBYTE *tmp;
01836             if ( ( tmp = FindFunWithArgs(sterm) ) != 0 ) {
01837                 TokenToLine(tmp);
01838                 break;
01839             }
01840         }
01841         t += FUNHEAD-2;
01842
01843         if ( *sterm == GAMMA && t[-FUNHEAD+1] == FUNHEAD+1 ) {
01844             TokenToLine((UBYTE *)"g1_(");
01845         }
01846         else {
01847             if ( *sterm != DUMFUN ) {
01848                 Out = StrCopy(FindFunction(*sterm), buffer);
01849                 /* Out = StrCopy(VARNAME(functions, *sterm - FUNCTION), buffer); */
01850             }
01851             else { Out = buffer; *Out = 0; }
01852             if ( t >= stopper ) {
01853                 TokenToLine(buffer);
01854                 break;
01855             }
01856             if ( AC.OutputMode == MATHEMATICAMODE ) { *Out++ = '['; closepar[0] = (UBYTE)']'; }
01857             else { *Out++ = '('; }
01858             *Out = 0;
01859             TokenToLine(buffer);
01860         }
01861         i = functions[*sterm - FUNCTION].spec;
01862         if ( i >= TENSORFUNCTION ) {
01863             int curdict = AO.CurrentDictionary;
01864             if ( AO.CurrentDictionary && AO.CurDictNotInFunctions > 0 )
01865                 AO.CurrentDictionary = 0;
01866             t = sterm + FUNHEAD;
01867             while ( t < stopper ) {
01868                 if ( !first ) TokenToLine((UBYTE *)",");
01869                 else first = 0;
01870                 j = *t++;
01871                 if ( j >= 0 ) {
01872                     if ( j < AM.OffsetIndex ) TalToLine((UWORD)(j));
01873                     else if ( j < AM.IndDum ) {
01874                         i = j - AM.OffsetIndex;
01875                         Out = StrCopy(FindIndex(i%WILDOFFSET+AM.OffsetIndex), buffer);
01876                         /* Out = StrCopy(VARNAME(indices, i%WILDOFFSET), buffer); */
01877                         if ( i >= WILDOFFSET ) { *Out++ = '?'; *Out = 0; }
01878                         TokenToLine(buffer);
01879                     }
01880                     else {
01881                         Out = buffer;
01882                         *Out++ = 'N';
01883                         Out = NumCopy(j - AM.IndDum, Out);
01884                         *Out++ = '_';
01885                         *Out++ = '?';
01886                         *Out = 0;
01887                         TokenToLine(buffer);
01888                     }
01889                 }
01890                 else if ( j == FUNNYVEC ) { TokenToLine((UBYTE *)"?"); }
01891                 else if ( j > -WILDOFFSET ) {
01892                     Out = buffer;

```

```

01893             Out = NumCopy((UWORD)(-j + 4),Out);
01894             *Out++ = ' ';
01895             *Out = 0;
01896             TokenToLine(buffer);
01897         }
01898         else {
01899             TokenToLine(FindVector(j));
01900             /* TokenToLine(VARNAME(vectors,j - AM.OffsetVector)); */
01901         }
01902     }
01903     AO.CurrentDictionary = curdict;
01904 }
01905 else {
01906     int curdict = AO.CurrentDictionary;
01907     if ( AO.CurrentDictionary && AO.CurDictNotInFunctions > 0 )
01908         AO.CurrentDictionary = 0;
01909     while ( t < stopper ) {
01910         if ( !first ) TokenToLine((UBYTE *)",");
01911         WriteArgument(t);
01912         NEXTARG(t);
01913         first = 0;
01914     }
01915     AO.CurrentDictionary = curdict;
01916 }
01917 TokenToLine(closepar);
01918 closepar[0] = (UBYTE)';
01919 break;
01920 }
01921 return(0);
01922 }
01923
01924 /*
01925     #] WriteSubTerm :
01926     #[ WriteInnerTerm :      WORD WriteInnerTerm(term,first)
01927
01928     Writes the contents of term to the output.
01929     Only the part that is inside parentheses is written.
01930
01931 */
01932
01933 int WriteInnerTerm(WORD *term, WORD first)
01934 {
01935     WORD *t, *s, *s1, *s2, n, i, pow;
01936     #ifdef WITHFLOAT
01937     int FloatChars = 0;
01938     GETIDENTITY
01939     #endif
01940     t = term;
01941     s = t+1;
01942     GETCOEF(t,n);
01943     while ( s < t ) {
01944         if ( *s == HAAKJE ) break;
01945         s += s[1];
01946     }
01947     if ( s < t ) { s += s[1]; }
01948     else { s = term+1; }
01949
01950     if ( n < 0 || !first ) {
01951         if ( n > 0 ) { TOKENTOLINE(" + ","+") }
01952         else if ( n < 0 ) { n = -n; TOKENTOLINE(" - ","-") }
01953     }
01954     if ( AC.modpowers ) {
01955         if ( n == 1 && *t == 1 && t > s ) first = 1;
01956         else if ( ABS(AC.ncmod) == 1 ) {
01957             UBYTE *Out1 = IsExponentSign();
01958             LongToLine((UWORD *)AC.powmod,AC.npowmod);
01959             if ( Out1 ) TokenToLine(Out1);
01960             else TokenToLine((UBYTE *)"^");
01961             TalToLine(AC.modpowers[(LONG)((UWORD)*t)]);
01962             first = 0;
01963         }
01964         else {
01965             LONG jj;
01966             UBYTE *Out1 = IsExponentSign();
01967             LongToLine((UWORD *)AC.powmod,AC.npowmod);
01968             if ( Out1 ) TokenToLine(Out1);
01969             else TokenToLine((UBYTE *)"^");
01970             jj = (UWORD)*t;
01971             if ( n == 2 ) jj += ((LONG)t[1])<<BITSINWORD;
01972             if ( AC.modpowers[jj+1] == 0 ) {
01973                 TalToLine(AC.modpowers[jj]);
01974             }
01975             else {
01976                 LongToLine(AC.modpowers+jj,2);
01977             }
01978             first = 0;
01979         }
01980     }

```



```

01980     }
01981     else if ( n != 1 || *t != 1 || t[1] != 1 || t <= s ) {
01982         if ( lowestlevel && ( ( AO.PrintType & PRINONEFUNCTION ) != 0 ) ) {
01983             FiniLine();
01984             if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
01985             else IniLine(3);
01986         }
01987         if ( AO.CurrentDictionary > 0 ) TransformRational((UWORD *)t,n);
01988         else RatToLine((UWORD *)t,n);
01989         first = 0;
01990     }
01991 #ifdef WITHFLOAT
01992 /*
01993     Check whether there is a 'proper' float_ function and no raw mode.
01994     If so, print as float.
01995     Raw mode is indicated as AO.FloatPrec < 0.
01996     AO.FloatPrec == 0 indicated as many digits as the precision allows.
01997 */
01998     else if ( AO.FloatPrec >= 0 && AT.aux_ != 0 ) {
01999         WORD *ss = s;
02000         while ( ss < t ) {
02001             if ( *ss == FLOATFUN ) {
02002                 if ( ( FloatChars = PrintFloat(ss,AO.FloatPrec) ) != 0 ) {
02003                     TokenToLine(AO.floatspace);
02004                     if ( AC.IsFortran90 == ISFORTRAN90 && AC.Fortran90Kind ) {
02005                         AddToLine(AC.Fortran90Kind);
02006                     }
02007                     first = 0;
02008                 }
02009                 break;
02010             }
02011             ss += ss[1];
02012         }
02013         if ( ss >= t ) first = 1;
02014     }
02015 #endif
02016     else first = 1;
02017     while ( s < t ) {
02018         if ( lowestlevel && ( (AO.PrintType & (PRINONEFUNCTION | PRINTALL)) == PRINONEFUNCTION ) ) {
02019             FiniLine();
02020             if ( AC.OutputSpaces == NOSPACEFORMAT ) IniLine(1);
02021             else IniLine(3);
02022         }
02023     }
02024 /*
02025     #[ NEWGAMMA :
02026 */
02027 #ifdef NEWGAMMA
02028     if ( *s == GAMMA ) { /* String them up */
02029         WORD *tt,*ss;
02030         ss = AT.WorkPointer;
02031         *ss++ = GAMMA;
02032         *ss++ = s[1];
02033         FILLFUN(ss)
02034         *ss++ = s[FUNHEAD];
02035         tt = s + FUNHEAD + 1;
02036         n = s[1] - FUNHEAD-1;
02037         do {
02038             while ( --n >= 0 ) *ss++ = *tt++;
02039             tt = s + s[1];
02040             while ( *tt == GAMMA && tt[FUNHEAD] == s[FUNHEAD] && tt < t ) {
02041                 s = tt;
02042                 tt += FUNHEAD + 1;
02043                 n = s[1] - FUNHEAD-1;
02044                 if ( n > 0 ) break;
02045             }
02046             while ( n > 0 );
02047             tt = AT.WorkPointer;
02048             AT.WorkPointer = ss;
02049             tt[1] = WORDDIF(ss,tt);
02050             if ( WriteSubTerm(tt,first) ) {
02051                 MesCall("WriteInnerTerm");
02052                 SETERROR(-1)
02053             }
02054             AT.WorkPointer = tt;
02055         }
02056     else
02057 #endif
02058 /*
02059     #] NEWGAMMA :
02060 */
02061 #ifdef WITHFLOAT
02062     if ( *s == FLOATFUN && AO.FloatPrec >= 0 && AT.aux_ != 0 ) {
02063     }
02064     else
02065 #endif
02066     {

```

```

02067         if ( *s >= FUNCTION && AC.funpowers > 0
02068         && functions[*s-FUNCTION].spec == 0 && ( AC.funpowers == ALLFUNPOWERS ||
02069         ( AC.funpowers == COMFUNPOWERS && functions[*s-FUNCTION].commute == 0 ) ) ) {
02070             pow = 1;
02071             for(;;) {
02072                 s1 = s; s2 = s + s[1]; i = s[1];
02073                 if ( s2 < t ) {
02074                     while ( --i >= 0 && *s1 == *s2 ) { s1++; s2++; }
02075                     if ( i < 0 ) {
02076                         pow++; s = s+s[1];
02077                     }
02078                     else break;
02079                 }
02080                 else break;
02081             }
02082             if ( pow > 1 ) {
02083                 if ( AC.OutputMode == CMODE ) {
02084                     if ( !first ) MultiplyToLine();
02085                     TokenToLine((UBYTE *) "pow(");
02086                     first = 1;
02087                 }
02088                 if ( WriteSubTerm(s,first) ) {
02089                     MesCall("WriteInnerTerm");
02090                     SETERROR(-1)
02091                 }
02092                 if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE
02093                 || AC.OutputMode == REDUCEMODE ) { TokenToLine((UBYTE *) "***"); }
02094                 else if ( AC.OutputMode == CMODE ) { TokenToLine((UBYTE *) ","); }
02095                 else {
02096                     UBYTE *Out1 = IsExponentSign();
02097                     if ( Out1 ) TokenToLine(Out1);
02098                     else TokenToLine((UBYTE *) "^");
02099                 }
02100                 TalToLine(pow);
02101                 if ( AC.OutputMode == CMODE ) TokenToLine((UBYTE *) ")");
02102             }
02103             else if ( WriteSubTerm(s,first) ) {
02104                 MesCall("WriteInnerTerm");
02105                 SETERROR(-1)
02106             }
02107         }
02108         else if ( WriteSubTerm(s,first) ) {
02109             MesCall("WriteInnerTerm");
02110             SETERROR(-1)
02111         }
02112     }
02113     first = 0;
02114     s += s[1];
02115 }
02116 return(0);
02117 }
02118
02119 /*
02120     #] WriteInnerTerm :
02121     #[ WriteTerm :      WORD WriteTerm(term,lbrac,first,prtf,br)
02122
02123     Writes a term to output. It tests the bracket information first.
02124     If there are no brackets or the bracket is the same all is passed
02125     to WriteInnerTerm. If there are brackets and the bracket is not
02126     the same as for the predecessor the old bracket is closed and
02127     a new one is opened.
02128     br indicates whether we are in a subexpression, barring zeroing
02129     AO.IsBracket
02130
02131 */
02132
02133 int WriteTerm(WORD *term, WORD *lbrac, WORD first, WORD prtf, WORD br)
02134 {
02135     WORD *t, *stopper, *b, n;
02136     int oldIsFortran90 = AC.IsFortran90, i;
02137     if ( *lbrac >= 0 ) {
02138         t = term + 1;
02139         stopper = (term + *term - 1);
02140         stopper -= ABS(*stopper) - 1;
02141         while ( t < stopper ) {
02142             if ( *t == HAAKJE ) {
02143                 stopper = t;
02144                 t = term+1;
02145             }
02146             if ( *lbrac == ( n = WORDDIF(stopper,t) ) ) {
02147                 b = AO.bracket + 1;
02148                 t = term + 1;
02149                 while ( n > 0 && ( *b++ == *t++ ) ) { n--; }
02150                 if ( n <= 0 && ( ( AM.FortranCont <= 0 || AO.InFbrack < AM.FortranCont )
02151                 || ( lowestlevel == 0 ) ) ) {
02152                     We continue inside a bracket.
02153                 }

```

```

02154         AO.IsBracket = 1;
02155         if ( ( prtf & PRINTCONTENTS ) != 0 ) {
02156             AO.NumInBrack++;
02157         }
02158         else {
02159             if ( WriteInnerTerm(term,0) ) goto WrtTmes;
02160             if ( ( AO.PrintType & PRINTONETERM ) != 0 ) {
02161                 FiniLine();
02162                 TokenToLine((UBYTE *) "    ");
02163             }
02164         }
02165         return(0);
02166     }
02167     t = term + 1;
02168     n = WORDDIF(stopper,t);
02169 }
02170 /*
02171 Close the bracket
02172 */
02173 if ( *lbrac ) {
02174     if ( ( prtf & PRINTCONTENTS ) ) PrtTerms();
02175     TOKENTOLINE(" ",")");
02176     if ( AC.OutputMode == CMODE && AO.FactorMode == 0 )
02177         TokenToLine((UBYTE *) ";");
02178     else if ( AO.FactorMode && ( n == 0 ) ) {
02179         /*
02180         This should not happen.
02181         */
02182         return(0);
02183     }
02184     AC.IsFortran90 = ISNOTFORTRAN90;
02185     FiniLine();
02186     AC.IsFortran90 = oldIsFortran90;
02187     if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE
02188         && AC.OutputSpaces == NORMALFORMAT
02189         && AO.FactorMode == 0 ) FiniLine();
02190 }
02191 else {
02192     if ( AC.OutputMode == CMODE && AO.FactorMode == 0 )
02193         TokenToLine((UBYTE *) ";");
02194     if ( AO.FortFirst == 0 ) {
02195         if ( !first ) {
02196             AC.IsFortran90 = ISNOTFORTRAN90;
02197             FiniLine();
02198             AC.IsFortran90 = oldIsFortran90;
02199         }
02200     }
02201 }
02202 if ( AO.FactorMode == 0 ) {
02203     if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
02204         && !first ) {
02205         WORD oldmode = AC.OutputMode;
02206         AC.OutputMode = 0;
02207         IniLine(0);
02208         AC.OutputMode = oldmode;
02209         AO.OutSkip = 7;
02210
02211         if ( AO.FortFirst == 0 ) {
02212             TokenToLine(AO.CurBufWrt);
02213             TOKENTOLINE(" = ","=")
02214             TokenToLine(AO.CurBufWrt);
02215         }
02216         else {
02217             AO.FortFirst = 0;
02218             TokenToLine(AO.CurBufWrt);
02219             TOKENTOLINE(" = ","=")
02220         }
02221     }
02222     else if ( AC.OutputMode == CMODE && !first ) {
02223         IniLine(0);
02224         if ( AO.FortFirst == 0 ) {
02225             TokenToLine(AO.CurBufWrt);
02226             TOKENTOLINE(" += ","+=")
02227         }
02228         else {
02229             AO.FortFirst = 0;
02230             TokenToLine(AO.CurBufWrt);
02231             TOKENTOLINE(" = ","=")
02232         }
02233     }
02234     else if ( startinline == 0 ) {
02235         IniLine(0);
02236     }
02237     AO.InFbrack = 0;
02238     if ( ( *lbrac = n ) > 0 ) {
02239         b = AO.bracket;
02240         *b++ = n + 4;

```

```

02241         while ( --n >= 0 ) *b++ = *t++;
02242         *b++ = 1; *b++ = 1; *b = 3;
02243         AO.IsBracket = 0;
02244         if ( WriteInnerTerm(AO.bracket,0) ) {
02245             /* Error message */
02246             WORD i;
02247 WrtTmes:         t = term;
02248                 AO.OutSkip = 3;
02249                 FiniLine();
02250                 i = *t;
02251                 while ( --i >= 0 ) { TalToLine((UWORD) (*t++));
02252                     if ( AC.OutputSpaces == NORMALFORMAT )
02253                         TokenToLine((UBYTE *) " "); }
02254                 AO.OutSkip = 0;
02255                 FiniLine();
02256                 MesCall("WriteTerm");
02257                 SETERROR(-1)
02258             }
02259             TOKENTOLINE(" * ( ", "*(")
02260             AO.NumInBrack = 0;
02261             AO.IsBracket = 1;
02262             if ( ( prtft & PRINTONETERM ) != 0 ) {
02263                 first = 0;
02264                 FiniLine();
02265                 TokenToLine((UBYTE *) " ");
02266             }
02267             else first = 1;
02268         }
02269         else {
02270             AO.IsBracket = 0;
02271             first = 0;
02272         }
02273     }
02274     else {
02275         /*
02276         Here is the code that writes the glue between two factors.
02277         We should not forget factors that are zero!
02278         */
02279         if ( ( *lbrac = n ) > 0 ) {
02280             b = AO.bracket;
02281             *b++ = n + 4;
02282             while ( --n >= 0 ) *b++ = *t++;
02283             *b++ = 1; *b++ = 1; *b = 3;
02284             for ( i = AO.FactorNum+1; i < AO.bracket[4]; i++ ) {
02285                 if ( first ) {
02286                     TOKENTOLINE(" ( 0 )", " (0)")
02287                     first = 0;
02288                 }
02289                 else {
02290                     TOKENTOLINE(" * ( 0 )", "*(0)")
02291                 }
02292                 FiniLine();
02293                 IniLine(0);
02294             }
02295             AO.FactorNum = AO.bracket[4];
02296         }
02297         else {
02298             AO.NumInBrack = 0;
02299             return(0);
02300         }
02301         if ( first == 0 ) { TOKENTOLINE(" * ( ", "*(") }
02302         else { TOKENTOLINE(" ( ", "(" ) }
02303         AO.NumInBrack = 0;
02304         first = 1;
02305     }
02306     if ( ( prtft & PRINTCONTENTS ) != 0 ) AO.NumInBrack++;
02307     else if ( WriteInnerTerm(term,first) ) goto WrtTmes;
02308     if ( ( AO.PrintType & PRINTONETERM ) != 0 ) {
02309         FiniLine();
02310         TokenToLine((UBYTE *) " ");
02311     }
02312     return(0);
02313 }
02314 else t += t[1];
02315 }
02316 if ( *lbrac > 0 ) {
02317     if ( ( prtft & PRINTCONTENTS ) != 0 ) PrtTerms();
02318     TokenToLine((UBYTE *) " )");
02319     if ( AC.OutputMode == CMODE ) TokenToLine((UBYTE *) ";");
02320     if ( AO.FortFirst == 0 ) {
02321         AC.IsFortran90 = ISNOTFORTRAN90;
02322         FiniLine();
02323         AC.IsFortran90 = oldIsFortran90;
02324     }
02325     if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE
02326         && AC.OutputSpaces == NORMALFORMAT ) FiniLine();
02327     if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )

```

```

02328         && !first ) {
02329             WORD oldmode = AC.OutputMode;
02330             AC.OutputMode = 0;
02331             IniLine(0);
02332             AC.OutputMode = oldmode;
02333             AO.OutSkip = 7;
02334             if ( AO.FortFirst == 0 ) {
02335                 TokenToLine(AO.CurBufWrt);
02336                 TOKENTOLINE(" = ", "=")
02337                 TokenToLine(AO.CurBufWrt);
02338             }
02339             else {
02340                 AO.FortFirst = 0;
02341                 TokenToLine(AO.CurBufWrt);
02342                 TOKENTOLINE(" = ", "=")
02343             }
02344             /*
02345                 TokenToLine(AO.CurBufWrt);
02346                 TOKENTOLINE(" = ", "=")
02347                 if ( AO.FortFirst == 0 )
02348                     TokenToLine(AO.CurBufWrt);
02349                 else AO.FortFirst = 0;
02350             */
02351         }
02352         else if ( AC.OutputMode == CMODE && !first ) {
02353             IniLine(0);
02354             if ( AO.FortFirst == 0 ) {
02355                 TokenToLine(AO.CurBufWrt);
02356                 TOKENTOLINE(" += ", "+=")
02357             }
02358             else {
02359                 AO.FortFirst = 0;
02360                 TokenToLine(AO.CurBufWrt);
02361                 TOKENTOLINE(" = ", "=")
02362             }
02363             /*
02364                 TokenToLine(AO.CurBufWrt);
02365                 if ( AO.FortFirst == 0 ) { TOKENTOLINE(" += ", "+=") }
02366                 else {
02367                     TOKENTOLINE(" = ", "=")
02368                     AO.FortFirst = 0;
02369                 }
02370             */
02371         }
02372         else IniLine(0);
02373         *lbrac = 0;
02374         first = 1;
02375     }
02376 }
02377 if ( !br ) AO.IsBracket = 0;
02378 if ( ( AM.FortranCont > 0 && AO.InFbrack >= AM.FortranCont ) && lowestlevel ) {
02379     if ( AC.OutputMode == CMODE ) TokenToLine((UBYTE *)";");
02380     if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
02381         && !first ) {
02382         WORD oldmode = AC.OutputMode;
02383         if ( AO.FortFirst == 0 ) {
02384             AC.IsFortran90 = ISNOTFORTRAN90;
02385             FiniLine();
02386             AC.IsFortran90 = oldIsFortran90;
02387             AC.OutputMode = 0;
02388             IniLine(0);
02389             AC.OutputMode = oldmode;
02390             AO.OutSkip = 7;
02391             TokenToLine(AO.CurBufWrt);
02392             TOKENTOLINE(" = ", "=")
02393             TokenToLine(AO.CurBufWrt);
02394         }
02395         else {
02396             AO.FortFirst = 0;
02397             /*
02398                 TokenToLine(AO.CurBufWrt);
02399                 TOKENTOLINE(" = ", "=")
02400             */
02401         }
02402         /*
02403             TokenToLine(AO.CurBufWrt);
02404             TOKENTOLINE(" = ", "=")
02405             if ( AO.FortFirst == 0 )
02406                 TokenToLine(AO.CurBufWrt);
02407             else AO.FortFirst = 0;
02408         */
02409     }
02410     else if ( AC.OutputMode == CMODE && !first ) {
02411         FiniLine();
02412         IniLine(0);
02413         if ( AO.FortFirst == 0 ) {
02414             TokenToLine(AO.CurBufWrt);

```

```

02415         TOKENTOLINE(" += ", "+=")
02416     }
02417     else {
02418         AO.FortFirst = 0;
02419         TokenToLine(AO.CurBufWrt);
02420         TOKENTOLINE(" = ", "=")
02421     }
02422 /*
02423     TokenToLine(AO.CurBufWrt);
02424     if ( AO.FortFirst == 0 ) { TOKENTOLINE(" += ", "+=") }
02425     else {
02426         TOKENTOLINE(" = ", "=")
02427         AO.FortFirst = 0;
02428     }
02429 */
02430 }
02431 else {
02432     FiniLine();
02433     IniLine(0);
02434 }
02435 AO.InFbrack = 0;
02436 }
02437 if ( WriteInnerTerm(term,first) ) goto WrtTmes;
02438 if ( ( AO.PrintType & PRINTONETERM ) != 0 ) {
02439     FiniLine();
02440     IniLine(0);
02441 }
02442 return(0);
02443 }
02444
02445 /*
02446     #] WriteTerm :
02447     #[ WriteExpression :    WORD WriteExpression(terms,ltot)
02448
02449     Writes a subexpression to output.
02450     The subexpression is in terms and contains ltot words.
02451     This is only used for function arguments.
02452 */
02453 */
02454
02455 int WriteExpression(WORD *terms, LONG ltot)
02456 {
02457     WORD *stopper;
02458     WORD first, btot;
02459     WORD OldIsBracket = AO.IsBracket, OldPrintType = AO.PrintType;
02460     if ( !AC.outsidefun ) { AO.PrintType &= ~PRINTONETERM; first = 1; }
02461     else first = 0;
02462     stopper = terms + ltot;
02463     btot = -1;
02464     while ( terms < stopper ) {
02465         AO.IsBracket = OldIsBracket;
02466         if ( WriteTerm(terms,&btot,first,0,1) ) {
02467             MesCall("WriteExpression");
02468             SETERROR(-1)
02469         }
02470         first = 0;
02471         terms += *terms;
02472     }
02473 /* AO.IsBracket = 0; */
02474 AO.IsBracket = OldIsBracket;
02475 AO.PrintType = OldPrintType;
02476 return(0);
02477 }
02478
02479 /*
02480     #] WriteExpression :
02481     #[ WriteAll :          WORD WriteAll()
02482
02483     Writes all expressions that should be written
02484 */
02485 */
02486 int WriteAll(void)
02487 {
02488     GETIDENTITY
02489     WORD lbrac, first;
02490     WORD *t, *stopper, n, prtf;
02491     int oldIsFortran90 = AC.IsFortran90, i;
02492     POSITION pos;
02493     FILEHANDLE *f;
02494     EXPRESSIONS e;
02495     if ( AM.exitflag ) return(0);
02496 #ifdef WITHMPI
02497     if ( PF.me != MASTER ) {
02498         /*
02499          * For the slaves, we need to call Optimize() the same number of times
02500          * as the master. The first argument doesn't have any important role.
02501          */

```

```

02502     for ( n = 0; n < NumExpressions; n++ ) {
02503         e = &Expressions[n];
02504         if ( (!e->printflag) & PRINTON ) continue;
02505         switch ( e->status ) {
02506             case LOCALEXPRESSION:
02507             case GLOBALEXPRESSION:
02508             case UNHIDELEXPRESSION:
02509             case UNHIDEGEXPRESSION:
02510                 break;
02511             default:
02512                 continue;
02513         }
02514         e->printflag = 0;
02515         PutPreVar(AM.oldnumextrasympols, GetPreVar((UBYTE *)"EXTRASYNBOLS_", 0), 0, 1);
02516         if ( AO.OptimizationLevel > 0 ) {
02517             if ( Optimize(0, 1) ) return(-1);
02518         }
02519     }
02520     return(0);
02521 }
02522 #endif
02523 SeekScratch(AR.outfile,&pos);
02524 if ( ResetScratch() ) {
02525     MesCall("WriteAll");
02526     SETERROR(-1)
02527 }
02528 AO.termbuf = AT.WorkPointer;
02529 AO.bracket = (WORD *)(((UBYTE *) (AT.WorkPointer)) + AM.MaxTer);
02530 AT.WorkPointer = (WORD *)(((UBYTE *) (AT.WorkPointer)) + AM.MaxTer*2);
02531 AO.OutFill = AO.OutputLine = (UBYTE *)AT.WorkPointer;
02532 AT.WorkPointer += 2*AC.LineLength;
02533 *(AR.CompressBuffer) = 0;
02534 first = 0;
02535 for ( n = 0; n < NumExpressions; n++ ) {
02536     if ( ( Expressions[n].printflag & PRINTON ) != 0 ) { first = 1; break; }
02537 }
02538 if ( !first ) goto EndWrite;
02539 AO.IsBracket = 0;
02540 AO.OutSkip = 3;
02541 AR.DeferFlag = 0;
02542 while ( GetTerm(BHEAD AO.termbuf) ) {
02543     t = AO.termbuf + 1;
02544     e = Expressions + AO.termbuf[3];
02545     n = e->status;
02546     if ( ( n == LOCALEXPRESSION || n == GLOBALEXPRESSION
02547         || n == UNHIDELEXPRESSION || n == UNHIDEGEXPRESSION ) &&
02548         ( ( prtft = e->printflag ) & PRINTON ) != 0 ) {
02549         e->printflag = 0;
02550         AO.NumInBrack = 0;
02551         PutPreVar(AM.oldnumextrasympols,
02552             GetPreVar((UBYTE *)"EXTRASYNBOLS_",0),0,1);
02553         if ( ( prtft & PRINTLFILE ) != 0 ) {
02554             if ( AC.LogHandle < 0 ) prtft &= ~PRINTLFILE;
02555         }
02556         AO.PrintType = prtft;
02557 /*
02558         if ( AC.OutputMode == VORTRANMODE ) {
02559             UBYTE *oldOutFill = AO.OutFill, *oldOutputLine = AO.OutputLine;
02560             AO.OutSkip = 6;
02561             if ( Optimize(AO.termbuf[3], 1) ) goto AboWrite;
02562             AO.OutSkip = 3;
02563             AO.OutFill = oldOutFill; AO.OutputLine = oldOutputLine;
02564             FiniLine();
02565             continue;
02566         }
02567         else
02568 */
02569         if ( AO.OptimizationLevel > 0 ) {
02570             UBYTE *oldOutFill = AO.OutFill, *oldOutputLine = AO.OutputLine;
02571             AO.OutSkip = 6;
02572             if ( Optimize(AO.termbuf[3], 1) ) goto AboWrite;
02573             AO.OutSkip = 3;
02574             AO.OutFill = oldOutFill; AO.OutputLine = oldOutputLine;
02575             FiniLine();
02576             continue;
02577         }
02578         if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
02579             AO.OutSkip = 6;
02580         FiniLine();
02581         AO.CurBufWrt = EXPRNAME(AO.termbuf[3]);
02582         TokenToLine(AO.CurBufWrt);
02583         stopper = t + t[1];
02584         t += SUBEXPSIZE;
02585         if ( t < stopper ) {
02586             TokenToLine((UBYTE *) "(" );
02587             first = 1;
02588             while ( t < stopper ) {

```

```

02589         n = *t;
02590         if ( !first ) TokenToLine((UBYTE *)",");
02591         switch ( n ) {
02592             case SYMTOSYM :
02593                 TokenToLine(FindSymbol(t[2]));
02594             /* TokenToLine(VARNAME(symbols,t[2])); */
02595                 break;
02596             case VECTOVEC :
02597                 TokenToLine(FindVector(t[2]));
02598             /* TokenToLine(VARNAME(vectors,t[2] - AM.OffsetVector)); */
02599                 break;
02600             case INDTOIND :
02601                 TokenToLine(FindIndex(t[2]));
02602             /* TokenToLine(VARNAME(indices,t[2] - AM.OffsetIndex)); */
02603                 break;
02604             default :
02605                 TokenToLine(FindFunction(t[2]));
02606             /* TokenToLine(VARNAME(functions,t[2] - FUNCTION)); */
02607                 break;
02608         }
02609         t += t[1];
02610         first = 0;
02611     }
02612     TokenToLine((UBYTE *)");");
02613 }
02614 TOKENTOLINE(" =", "=");
02615 if ( AC.OutputMode == MATHEMATICAMODE ) {
02616     TOKENTOLINE(" (","(");
02617 }
02618 lbrac = 0;
02619 AO.InFbrack = 0;
02620 if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
02621     AO.FortFirst = 1;
02622 else
02623     AO.FortFirst = 0;
02624 first = 1;
02625 if ( ( e->vflags & ISFACTORIZED ) != 0 ) {
02626     AO.FactorMode = 1+e->numfactors;
02627     AO.FactorNum = 0; /* Which factor are we doing. For factors that are zero */
02628 }
02629 else {
02630     AO.FactorMode = 0;
02631 }
02632 while ( GetTerm(BHEAD AO.termbuf) ) {
02633     WORD *m;
02634     GETSTOP(AO.termbuf,m);
02635     if ( ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
02636         && ( ( prt & PRINTONETERM ) != 0 ) ) {}
02637     else {
02638         if ( first ) {
02639             FiniLine();
02640             IniLine(0);
02641         }
02642     }
02643     if ( ( prt & PRINTONETERM ) != 0 ) first = 0;
02644     if ( WriteTerm(AO.termbuf,&lbrac,first,prt,0) )
02645         goto AboWrite;
02646     first = 0;
02647 }
02648 if ( AO.FactorMode ) {
02649     if ( first ) { AO.FactorNum = 1; TOKENTOLINE(" ( 0)"," (0)"); }
02650     else TOKENTOLINE(" )","");
02651     for ( i = AO.FactorNum+1; i <= e->numfactors; i++ ) {
02652         FiniLine();
02653         IniLine(0);
02654         TOKENTOLINE(" * ( 0)","*(0)");
02655     }
02656     AO.FactorNum = e->numfactors;
02657     if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE )
02658         TokenToLine((UBYTE *)",");
02659 }
02660 else if ( AO.FactorMode == 0 || first ) {
02661     if ( first ) { TOKENTOLINE(" 0","0"); }
02662     else if ( lbrac ) {
02663         if ( ( prt & PRINTCONTENTS ) != 0 ) PrtTerms();
02664         TOKENTOLINE(" )","");
02665     }
02666     else if ( ( prt & PRINTCONTENTS ) != 0 ) {
02667         TOKENTOLINE(" + 1 * ( ","+1*(")
02668         PrtTerms();
02669         TOKENTOLINE(" )","");
02670     }
02671     if ( AC.OutputMode == MATHEMATICAMODE ) {
02672         TokenToLine((UBYTE *)",");
02673     }
02674     if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE )
02675         TokenToLine((UBYTE *)",");

```



```

02676         }
02677         AO.OutSkip = 3;
02678         AC.IsFortran90 = ISNOTFORTHAN90;
02679         FiniLine();
02680         AC.IsFortran90 = oldIsFortran90;
02681         AO.FactorMode = 0;
02682     }
02683     else {
02684         do { } while ( GetTerm(BHEAD AO.termbuf) );
02685     }
02686 }
02687 if ( AC.OutputSpaces == NORMALFORMAT ) FiniLine();
02688 EndWrite:
02689 if ( AR.infile->handle >= 0 ) {
02690     SeekFile(AR.infile->handle, &(AR.infile->filesize), SEEK_SET);
02691 }
02692 AO.IsBracket = 0;
02693 AT.WorkPointer = AO.termbuf;
02694 SetScratch(AR.infile, &pos);
02695 f = AR.outfile; AR.outfile = AR.infile; AR.infile = f;
02696 return(0);
02697 AboWrite:
02698 SetScratch(AR.infile, &pos);
02699 f = AR.outfile; AR.outfile = AR.infile; AR.infile = f;
02700 MesCall("WriteAll");
02701 Terminate(-1);
02702 return(-1);
02703 }
02704
02705 /*
02706     #] WriteAll :
02707     #[ WriteOne :      WORD WriteOne(name,alreadyinline)
02708
02709     Writes one expression from the preprocessor
02710 */
02711
02712 int WriteOne(UBYTE *name, int alreadyinline, int nosemi, WORD plus)
02713 {
02714     GETIDENTITY
02715     WORD number;
02716     WORD lbrac, first;
02717     POSITION pos;
02718     FILEHANDLE *f;
02719     WORD prf;
02720
02721     if ( GetName(AC.exprnames, name, &number, NOAUTO) != CEXPRESSION ) {
02722         MesPrint("@%s is not an expression", name);
02723         return(-1);
02724     }
02725     switch ( Expressions[number].status ) {
02726     case HIDDENLEXPRESSION:
02727     case HIDDENGEXPRESSION:
02728     case HIDELEXPRESSION:
02729     case HIDEGEXPRESSION:
02730     case UNHIDELEXPRESSION:
02731     case UNHIDEGEXPRESSION:
02732     /*
02733         case DROPHLEXPRESSION:
02734         case DROPHGEXPRESSION:
02735     */
02736         AR.GetFile = 2;
02737         break;
02738     case LOCALEXPRESSION:
02739     case GLOBALEXPRESSION:
02740     case SKIPLEXPRESSION:
02741     case SKIPGEXPRESSION:
02742     /*
02743         case DROPLEXPRESSION:
02744         case DROPGEXPRESSION:
02745     */
02746         AR.GetFile = 0;
02747         break;
02748     default:
02749         MesPrint("@expressions %s is not active. It cannot be written", name);
02750         return(-1);
02751     }
02752     SeekScratch(AR.outfile, &pos);
02753     f = AR.outfile; AR.outfile = AR.infile; AR.infile = f;
02754
02755     /*
02756     if ( ResetScratch() ) {
02757         MesCall("WriteOne");
02758         SETERROR(-1)
02759     }
02760     */
02761     if ( AR.GetFile == 2 ) f = AR.hidefile;
02762     else f = AR.infile;

```

```

02763     prf = Expressions[number].printflag;
02764     if ( plus ) prf |= PRINTONETERM;
02765  /*
02766         Now position the file
02767  */
02768     if ( f->handle >= 0 ) {
02769         SetScratch(f,&(Expressions[number].onfile));
02770     }
02771     else {
02772         f->POfill = (WORD *) ((UBYTE *) (f->PObuffer)
02773             + BASEPOSITION(Expressions[number].onfile));
02774     }
02775     AO.termbuf = AT.WorkPointer;
02776     AO.bracket = (WORD *) ((UBYTE *) (AT.WorkPointer)) + AM.MaxTer;
02777     AT.WorkPointer = (WORD *) ((UBYTE *) (AT.WorkPointer)) + AM.MaxTer*2;
02778
02779     AO.OutFill = AO.OutputLine = (UBYTE *) AT.WorkPointer;
02780     AT.WorkPointer += 2*AC.LineLength;
02781     *(AR.CompressBuffer) = 0;
02782
02783     AO.IsBracket = 0;
02784     AO.OutSkip = 3;
02785     AR.DeferFlag = 0;
02786
02787     if ( AC.OutputMode == FORTRANMODE || AC.OutputMode == PFORTRANMODE )
02788         AO.OutSkip = 6;
02789     if ( GetTerm(BHEAD AO.termbuf) <= 0 ) {
02790         MesPrint("@ReadError in expression %s",name);
02791         goto AboWrite;
02792     }
02793  /*
02794     PutPreVar(AM.oldnumextrasymbols,
02795         GetPreVar((UBYTE *) "EXTRASYMBOLS_",0),0,1);
02796  */
02797  /*
02798     * Currently WriteOne() is called only from writeToChannel() with setting
02799     * AO.OptimizationLevel = 0, which means Optimize() is never called here.
02800     * So we don't need to think about how to ensure that the master and the
02801     * slaves call Optimize() at the same time. (TU 26 Jul 2013)
02802  */
02803     if ( AO.OptimizationLevel > 0 ) {
02804         AO.OutSkip = 6;
02805         if ( Optimize(AO.termbuf[3], 1) ) goto AboWrite;
02806         AO.OutSkip = 3;
02807         FiniLine();
02808     }
02809     else {
02810         lbrac = 0;
02811         AO.InFbrack = 0;
02812         AO.FortFirst = 0;
02813         first = 1;
02814         while ( GetTerm(BHEAD AO.termbuf) ) {
02815             WORD *m;
02816             GETSTOP(AO.termbuf,m);
02817             if ( first ) {
02818                 IniLine(0);
02819                 startinline = alreadyinline;
02820                 AO.OutFill = AO.OutputLine + startinline;
02821                 if ( WriteTerm(AO.termbuf,&lbrac,first,0,0) )
02822                     goto AboWrite;
02823                 first = 0;
02824             }
02825             else {
02826                 if ( ( prf & PRINTONETERM ) != 0 ) first = 1;
02827                 if ( first ) {
02828                     FiniLine();
02829                     IniLine(0);
02830                 }
02831                 first = 0;
02832                 if ( WriteTerm(AO.termbuf,&lbrac,first,0,0) )
02833                     goto AboWrite;
02834             }
02835         }
02836         if ( first ) {
02837             IniLine(0);
02838             startinline = alreadyinline;
02839             AO.OutFill = AO.OutputLine + startinline;
02840             TOKENTOLINE(" 0","0");
02841         }
02842         else if ( lbrac ) {
02843             TOKENTOLINE(" )","");
02844         }
02845         if ( AC.OutputMode != FORTRANMODE && AC.OutputMode != PFORTRANMODE
02846             && nosemi == 0 ) TokenToLine((UBYTE *) "");
02847         AO.OutSkip = 3;
02848         if ( AC.OutputSpaces == NORMALFORMAT && nosemi == 0 ) {
02849             FiniLine();

```

```

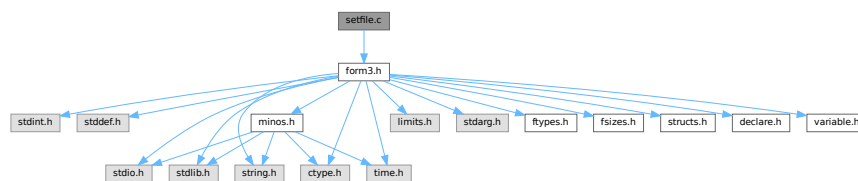
02850     }
02851     else {
02852         noextralinefeed = 1;
02853         FiniLine();
02854         noextralinefeed = 0;
02855     }
02856 }
02857 AO.IsBracket = 0;
02858 AT.WorkPointer = AO.termbuf;
02859 SetScratch(f, &pos);
02860 f = AR.outfile; AR.outfile = AR.infile; AR.infile = f;
02861 AO.InFbrack = 0;
02862 return(0);
02863 AboWrite:
02864 SetScratch(AR.infile, &pos);
02865 f->POposition = pos;
02866 f = AR.outfile; AR.outfile = AR.infile; AR.infile = f;
02867 MesCall("WriteOne");
02868 Terminate(-1);
02869 return(-1);
02870 }
02871
02872 /*
02873     #] WriteOne :
02874     #] schryf-Writes :
02875 */

```

4.125 setfile.c File Reference

```
#include "form3.h"
```

Include dependency graph for setfile.c:



Macros

- #define [NUMERICALVALUE](#) 0
- #define [STRINGVALUE](#) 1
- #define [PATHVALUE](#) 2
- #define [ONOFFVALUE](#) 3
- #define [DEFINEVALUE](#) 4
- #define [SETBUFSIZE](#) 257

Functions

- int [DoSetups](#) (void)
- int [ProcessOption](#) (UBYTE *s1, UBYTE *s2, int filetype)
- [SETUPPARAMETERS](#) * [GetSetupPar](#) (UBYTE *s)
- int [RecalcSetups](#) (void)
- int [AllocSetups](#) (void)
- void [WriteSetup](#) (void)
- [SORTING](#) * [AllocSort](#) (LONG inLargeSize, LONG inSmallSize, LONG inSmallEsize, LONG inTermsInSmall, int inMaxPatches, int inMaxFpatches, LONG inIsize, int level)

- void [AllocSortFileName](#) ([SORTING](#) *sort)
- [FILEHANDLE](#) * [AllocFileHandle](#) ([WORD](#) par, char *name)
- void [DeAllocFileHandle](#) ([FILEHANDLE](#) *fh)
- int [MakeSetupAllocs](#) (void)
- int [TryFileSetups](#) (void)
- int [TryEnvironment](#) (void)

Variables

- char [curdirp](#) [] = "."
- char [cursortdirp](#) [] = "."
- char [commentchar](#) [] = "*"
- char [dotchar](#) [] = "_"
- char [highfirst](#) [] = "highfirst"
- char [lowfirst](#) [] = "lowfirst"
- char [procedureextension](#) [] = "prc"
- [SETUPPARAMETERS](#) [setupparameters](#) []

4.125.1 Detailed Description

The routines that deal with the setup parameters.

Definition in file [setfile.c](#).

4.125.2 Macro Definition Documentation

4.125.2.1 NUMERICALVALUE

```
#define NUMERICALVALUE 0
```

Definition at line 47 of file [setfile.c](#).

4.125.2.2 STRINGVALUE

```
#define STRINGVALUE 1
```

Definition at line 48 of file [setfile.c](#).

4.125.2.3 PATHVALUE

```
#define PATHVALUE 2
```

Definition at line 49 of file [setfile.c](#).

4.125.2.4 ONOFFVALUE

```
#define ONOFFVALUE 3
```

Definition at line 50 of file [setfile.c](#).

4.125.2.5 DEFINEVALUE

```
#define DEFINEVALUE 4
```

Definition at line 51 of file [setfile.c](#).

4.125.2.6 SETBUFSIZE

```
#define SETBUFSIZE 257
```

Definition at line 1129 of file [setfile.c](#).

4.125.3 Function Documentation

4.125.3.1 DoSetups()

```
int DoSetups (  
    void )
```

Definition at line 132 of file [setfile.c](#).

4.125.3.2 ProcessOption()

```
int ProcessOption (  
    UBYTE * s1,  
    UBYTE * s2,  
    int filetype )
```

Definition at line 182 of file [setfile.c](#).

4.125.3.3 GetSetupPar()

```
SETUPPARAMETERS * GetSetupPar (  
    UBYTE * s )
```

Definition at line 337 of file [setfile.c](#).

4.125.3.4 RecalcSetups()

```
int RecalcSetups (  
    void )
```

Definition at line 358 of file [setfile.c](#).

4.125.3.5 AllocSetups()

```
int AllocSetups (
    void )
```

Definition at line 405 of file [setfile.c](#).

4.125.3.6 WriteSetup()

```
void WriteSetup (
    void )
```

Definition at line 800 of file [setfile.c](#).

4.125.3.7 AllocSort()

```
SORTING * AllocSort (
    LONG inLargeSize,
    LONG inSmallSize,
    LONG inSmallEsize,
    LONG inTermsInSmall,
    int inMaxPatches,
    int inMaxFpatches,
    LONG inIOSize,
    int level )
```

Definition at line 858 of file [setfile.c](#).

4.125.3.8 AllocSortFileName()

```
void AllocSortFileName (
    SORTING * sort )
```

Definition at line 1008 of file [setfile.c](#).

4.125.3.9 AllocFileHandle()

```
FILEHANDLE * AllocFileHandle (
    WORD par,
    char * name )
```

Definition at line 1033 of file [setfile.c](#).

4.125.3.10 DeAllocFileHandle()

```
void DeAllocFileHandle (
    FILEHANDLE * fh )
```

Definition at line 1094 of file [setfile.c](#).

4.125.3.11 MakeSetupAllocs()

```
int MakeSetupAllocs (  
    void )
```

Definition at line 1111 of file [setfile.c](#).

4.125.3.12 TryFileSetups()

```
int TryFileSetups (  
    void )
```

Definition at line 1131 of file [setfile.c](#).

4.125.3.13 TryEnvironment()

```
int TryEnvironment (  
    void )
```

Definition at line 1217 of file [setfile.c](#).

4.125.4 Variable Documentation

4.125.4.1 curdirp

```
char curdirp[] = "."
```

Definition at line 39 of file [setfile.c](#).

4.125.4.2 cursortdirp

```
char cursortdirp[] = "."
```

Definition at line 40 of file [setfile.c](#).

4.125.4.3 commentchar

```
char commentchar[] = "*"
```

Definition at line 41 of file [setfile.c](#).

4.125.4.4 dotchar

```
char dotchar[] = "_"
```

Definition at line 42 of file [setfile.c](#).

4.125.4.5 highfirst

```
char highfirst[] = "highfirst"
```

Definition at line 43 of file [setfile.c](#).

4.125.4.6 lowfirst

```
char lowfirst[] = "lowfirst"
```

Definition at line 44 of file [setfile.c](#).

4.125.4.7 procedureextension

```
char procedureextension[] = "prc"
```

Definition at line 45 of file [setfile.c](#).

4.125.4.8 setupparameters

```
SETUPPARAMETERS setupparameters[ ]
```

Definition at line 53 of file [setfile.c](#).

4.126 setfile.c

[Go to the documentation of this file.](#)

```
00001
00005 /* # [ License : */
00006 /*
00007 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00008 * When using this file you are requested to refer to the publication
00009 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010 * This is considered a matter of courtesy as the development was paid
00011 * for by FOM the Dutch physics granting agency and we would like to
00012 * be able to track its scientific use to convince FOM of its value
00013 * for the community.
00014 *
00015 * This file is part of FORM.
00016 *
00017 * FORM is free software: you can redistribute it and/or modify it under the
00018 * terms of the GNU General Public License as published by the Free Software
00019 * Foundation, either version 3 of the License, or (at your option) any later
00020 * version.
00021 *
00022 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00025 * details.
00026 *
00027 * You should have received a copy of the GNU General Public License along
00028 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00029 */
00030 /* # [ License : */
00031 /*
00032 * # [ Includes :
00033 *
00034 * Routines that deal with settings and the setup file
00035 */
00036
00037 #include "form3.h"
00038
```



```

00039 char curdirp[] = ".";
00040 char cursortdirp[] = ".";
00041 char commentchar[] = "*";
00042 char dotchar[] = "_";
00043 char highfirst[] = "highfirst";
00044 char lowfirst[] = "lowfirst";
00045 char procedureextension[] = "prc";
00046
00047 #define NUMERICALVALUE 0
00048 #define STRINGVALUE 1
00049 #define PATHVALUE 2
00050 #define ONOFFVALUE 3
00051 #define DEFINEVALUE 4
00052
00053 SETUPPARAMETERS setupparameters[] =
00054 {
00055     {(UBYTE *) "bracketindexsize", NUMERICALVALUE, 0, (LONG) MAXBRACKETBUFFERSIZE}
00056     , {(UBYTE *) "commentchar", STRINGVALUE, 0, (LONG) commentchar}
00057     , {(UBYTE *) "compresssize", NUMERICALVALUE, 0, (LONG) COMPRESSBUFFER}
00058     , {(UBYTE *) "constindex", NUMERICALVALUE, 0, (LONG) NUMFIXED}
00059     , {(UBYTE *) "continuationlines", NUMERICALVALUE, 0, (LONG) FORTRANCONTINUATIONLINES}
00060 #ifdef WITHFLOAT
00061     , {(UBYTE *) "defaultprecision", NUMERICALVALUE, 0, (LONG) DEFAULTPRECISION}
00062 #endif
00063     , {(UBYTE *) "define", DEFINEVALUE, 0, (LONG) 0}
00064     , {(UBYTE *) "dotchar", STRINGVALUE, 0, (LONG) dotchar}
00065     , {(UBYTE *) "factorizationcache", NUMERICALVALUE, 0, (LONG) FBUFFERSIZE}
00066     , {(UBYTE *) "filepatches", NUMERICALVALUE, 0, (LONG) MAXFPATCHES}
00067     , {(UBYTE *) "functionlevels", NUMERICALVALUE, 0, (LONG) MAXFLEVELS}
00068     , {(UBYTE *) "hidesize", NUMERICALVALUE, 0, (LONG) 0}
00069     , {(UBYTE *) "incdir", PATHVALUE, 0, (LONG) curdirp}
00070     , {(UBYTE *) "indentspace", NUMERICALVALUE, 0, (LONG) INDENTSPACE}
00071     , {(UBYTE *) "insidefirst", ONOFFVALUE, 0, (LONG) 1}
00072     , {(UBYTE *) "jumpratio", NUMERICALVALUE, 0, (LONG) JUMPRATIO}
00073     , {(UBYTE *) "largepatches", NUMERICALVALUE, 0, (LONG) MAXPATCHES}
00074     , {(UBYTE *) "largesize", NUMERICALVALUE, 0, (LONG) LARGEBUFFER}
00075     , {(UBYTE *) "maxnumbersize", NUMERICALVALUE, 0, (LONG) 0}
00076     /* , {(UBYTE *) "maxnumbersize", NUMERICALVALUE, 0, (LONG) MAXNUMBERSIZE} */
00077     , {(UBYTE *) "maxtermsize", NUMERICALVALUE, 0, (LONG) MAXTER}
00078 #ifdef WITHFLOAT
00079     , {(UBYTE *) "maxweight", NUMERICALVALUE, 0, (LONG) MAXWEIGHT}
00080 #endif
00081     , {(UBYTE *) "maxwildcards", NUMERICALVALUE, 0, (LONG) MAXWILDC}
00082     , {(UBYTE *) "nospacesinnumbers", ONOFFVALUE, 0, (LONG) 0}
00083     , {(UBYTE *) "numstorecaches", NUMERICALVALUE, 0, (LONG) NUMSTORECACHES}
00084     , {(UBYTE *) "nwritefinalstatistics", ONOFFVALUE, 0, (LONG) 0}
00085     , {(UBYTE *) "nwriteprocessstatistics", ONOFFVALUE, 0, (LONG) 0}
00086     , {(UBYTE *) "nwritestatistics", ONOFFVALUE, 0, (LONG) 0}
00087     , {(UBYTE *) "nwritethreadstatistics", ONOFFVALUE, 0, (LONG) 0}
00088     , {(UBYTE *) "oldfactarg", ONOFFVALUE, 0, (LONG) NEWFACTARG}
00089     , {(UBYTE *) "oldgcd", ONOFFVALUE, 0, (LONG) 1}
00090     , {(UBYTE *) "oldorder", ONOFFVALUE, 0, (LONG) 0}
00091     , {(UBYTE *) "oldparallelstatistics", ONOFFVALUE, 0, (LONG) 0}
00092     , {(UBYTE *) "parentheses", NUMERICALVALUE, 0, (LONG) MAXPARLEVEL}
00093     , {(UBYTE *) "path", PATHVALUE, 0, (LONG) curdirp}
00094     , {(UBYTE *) "procedureextension", STRINGVALUE, 0, (LONG) procedureextension}
00095     , {(UBYTE *) "processbucketsize", NUMERICALVALUE, 0, (LONG) DEFAULTPROCESSBUCKETSIZE}
00096     , {(UBYTE *) "resettimeonclear", ONOFFVALUE, 0, (LONG) 1}
00097     , {(UBYTE *) "scratchsize", NUMERICALVALUE, 0, (LONG) SCRATCHSIZE}
00098     , {(UBYTE *) "shmwinsize", NUMERICALVALUE, 0, (LONG) SHMWINSIZE}
00099     , {(UBYTE *) "sizestorecache", NUMERICALVALUE, 0, (LONG) SIZESTORECACHE}
00100     , {(UBYTE *) "smallestextension", NUMERICALVALUE, 0, (LONG) SMALLOVERFLOW}
00101     , {(UBYTE *) "smallsize", NUMERICALVALUE, 0, (LONG) SMALLBUFFER}
00102     , {(UBYTE *) "sortiosize", NUMERICALVALUE, 0, (LONG) SORTIOSIZE}
00103     , {(UBYTE *) "sorttype", STRINGVALUE, 0, (LONG) lowfirst}
00104     , {(UBYTE *) "spectatorsize", NUMERICALVALUE, 0, (LONG) SPECTATORSIZE}
00105     , {(UBYTE *) "subfilepatches", NUMERICALVALUE, 0, (LONG) SMAXFPATCHES}
00106     , {(UBYTE *) "sublargepatches", NUMERICALVALUE, 0, (LONG) SMAXPATCHES}
00107     , {(UBYTE *) "sublargesize", NUMERICALVALUE, 0, (LONG) SLARGEBUFFER}
00108     , {(UBYTE *) "subsmallestextension", NUMERICALVALUE, 0, (LONG) SSMALLOVERFLOW}
00109     , {(UBYTE *) "subsmallsize", NUMERICALVALUE, 0, (LONG) SMALLBUFFER}
00110     , {(UBYTE *) "subsortiosize", NUMERICALVALUE, 0, (LONG) SSORTIOSIZE}
00111     , {(UBYTE *) "subtermsinsmall", NUMERICALVALUE, 0, (LONG) STERMSSMALL}
00112     , {(UBYTE *) "tempdir", STRINGVALUE, 0, (LONG) curdirp}
00113     , {(UBYTE *) "tempsortdir", STRINGVALUE, 0, (LONG) cursortdirp}
00114     , {(UBYTE *) "termsinsmall", NUMERICALVALUE, 0, (LONG) TERMSMALL}
00115     , {(UBYTE *) "threadbucketsize", NUMERICALVALUE, 0, (LONG) DEFAULTTHREADBUCKETSIZE}
00116     , {(UBYTE *) "threadloadbalancing", ONOFFVALUE, 0, (LONG) DEFAULTTHREADLOADBALANCING}
00117     , {(UBYTE *) "threads", NUMERICALVALUE, 0, (LONG) DEFAULTTHREADS}
00118     , {(UBYTE *) "threadscratchoutsize", NUMERICALVALUE, 0, (LONG) THREADSCRATCHOUTSIZE}
00119     , {(UBYTE *) "threadscratchsize", NUMERICALVALUE, 0, (LONG) THREADSCRATCHSIZE}
00120     , {(UBYTE *) "theadsortfilesynch", ONOFFVALUE, 0, (LONG) 0}
00121     , {(UBYTE *) "totalsize", ONOFFVALUE, 0, (LONG) 2}
00122     , {(UBYTE *) "workspace", NUMERICALVALUE, 0, (LONG) WORKBUFFER}
00123     , {(UBYTE *) "wtimestats", ONOFFVALUE, 0, (LONG) 2}
00124 };
00125

```

```

00126 /*
00127     #] Includes :
00128     #[ Setups :
00129         #[ DoSetups :
00130 */
00131
00132 int DoSetups(void)
00133 {
00134     UBYTE *setbuffer, *s, *t, *u /*, c */;
00135     int errors = 0;
00136     setbuffer = LoadInputFile((UBYTE *)setupfilename,SETUPFILE);
00137     if ( setbuffer ) {
00138         /*
00139             The contents of the file are now in setbuffer.
00140             Each line is commentary or a single command.
00141             The buffer is terminated with a zero.
00142         */
00143         s = setbuffer;
00144         while ( *s ) {
00145             if ( *s == ' ' || *s == '\t' || *s == '*' || *s == '#' || *s == '\n' ) {
00146                 while ( *s && *s != '\n' ) s++;
00147             }
00148             else if ( tolower(*s) < 'a' || tolower(*s) > 'z' ) {
00149                 t = s;
00150                 while ( *s && *s != '\n' ) s++;
00151                 /*
00152                     c = *s; *s = 0;
00153                     Error1("Setup file: Illegal statement: ",t);
00154                     errors++; *s = c;
00155                 */
00156             }
00157             else {
00158                 t = s; /* name of the option */
00159                 while ( tolower(*s) >= 'a' && tolower(*s) <= 'z' ) s++;
00160                 *s++ = 0;
00161                 while ( *s == ' ' || *s == '\t' ) s++;
00162                 u = s; /* 'value' of the option */
00163                 while ( *s && *s != '\n' && *s != '\r' ) s++;
00164                 if ( *s ) *s++ = 0;
00165                 errors += ProcessOption(t,u,0);
00166             }
00167             while ( *s == '\n' || *s == '\r' ) s++;
00168         }
00169         M_free(setbuffer,"setup file buffer");
00170     }
00171     if ( errors ) return(1);
00172     else return(0);
00173 }
00174
00175 /*
00176     #] DoSetups :
00177     #[ ProcessOption :
00178 */
00179
00180 static char *proopl[3] = { "Setup file", "Setups in .frm file", "Setup in environment" };
00181
00182 int ProcessOption(UBYTE *s1, UBYTE *s2, int filetype)
00183 {
00184     SETUPPARAMETERS *sp;
00185     int n, giveback = 0, error = 0;
00186     UBYTE *s, *t, *s2ret;
00187     LONG x;
00188     sp = GetSetupPar(s1);
00189     if ( sp ) {
00190         /*
00191             We check now whether there are ` variables to be looked up in the
00192             environment. This is new (30-may-2008). This is only allowed in s2.
00193         */
00194         restart;;
00195         {
00196             UBYTE *s3,*s4,*s5,*s6, c, *start;
00197             int n1,n2,n3;
00198             s = s2;
00199             while ( *s ) {
00200                 if ( *s == '\\\\' ) s += 2;
00201                 else if ( *s == '\\' ) {
00202                     start = s; s++;
00203                     while ( *s && *s != '\\' ) {
00204                         if ( *s == '\\\\' ) s++;
00205                         s++;
00206                     }
00207                     if ( *s == 0 ) {
00208                         MesPrint("%s: Illegal use of ` character for parameter %s"
00209                             ,proopl[filetype],s1);
00210                         return(1);
00211                     }
00212                     c = *s; *s = 0;

```

```

00213         s3 = (UBYTE *)getenv((char *) (start+1));
00214         if ( s3 == 0 ) {
00215             MesPrint("%s: Cannot find environment variable %s for parameter %s"
00216                 ,proopl[filetype],start+1,s1);
00217             return(1);
00218         }
00219     }
00220     *s = c; s++;
00221     n1 = start - s2; s4 = s3; n2 = 0;
00222     while ( *s4 ) {
00223         if ( *s4 == '\\ ' ) { s4++; n2++; }
00224         s4++; n2++;
00225     }
00226     s4 = s; n3 = 0;
00227     while ( *s4 ) {
00228         if ( *s4 == '\\ ' ) { s4++; n3++; }
00229         s4++; n3++;
00230     }
00231     s4 = (UBYTE *)Malloc1((n1+n2+n3+1)*sizeof(UBYTE),"environment in setup");
00232     s5 = s2; s6 = s4;
00233     while ( n1-- > 0 ) *s6++ = *s5++;
00234     s5 = s3;
00235     while ( n2-- > 0 ) *s6++ = *s5++;
00236     s5 = s;
00237     while ( n3-- > 0 ) *s6++ = *s5++;
00238     *s6 = 0;
00239     if ( giveback ) M_free(s2,"environment in setup");
00240     s2 = s4;
00241     giveback = 1;
00242     goto restart;
00243 }
00244     else s++;
00245 }
00246 }
00247 n = sp->type;
00248 s2ret = s2;
00249 switch ( n ) {
00250     case NUMERICALVALUE:
00251         ParseNumber(x,s2);
00252         if ( *s2 == 'K' ) { x = x * 1000; s2++; }
00253         else if ( *s2 == 'M' ) { x = x * 1000000; s2++; }
00254         else if ( *s2 == 'G' ) { x = x * 1000000000; s2++; }
00255         else if ( *s2 == 'T' ) { x = x * 1000000000000; s2++; }
00256         if ( *s2 && *s2 != ' ' && *s2 != '\\t' ) {
00257             MesPrint("%s: Numerical value expected for parameter %s"
00258                 ,proopl[filetype],s1);
00259             error = 1; break;
00260         }
00261         sp->value = x;
00262         sp->flags = USEDFLAG;
00263         break;
00264     case STRINGVALUE:
00265         if ( StrICmp(s1,(UBYTE *)"tempsortdir") == 0 ) AM.havesortdir = 1;
00266         s = s2; t = s2;
00267         while ( *s ) {
00268             if ( *s == ' ' || *s == '\\t' ) break;
00269             if ( *s == '\\ ' ) s++;
00270             *t++ = *s++;
00271         }
00272         *t = 0;
00273         if ( sp->flags == USEDFLAG && sp->value != 0 )
00274             M_free((void *) (sp->value),"Process option");
00275         sp->value = (LONG)strDup1(s2,"Process option");
00276         sp->flags = USEDFLAG;
00277         break;
00278     case PATHVALUE:
00279         if ( StrICmp(s1,(UBYTE *)"incdir") == 0 ) {
00280             AM.IncDir = 0;
00281         }
00282         else if ( StrICmp(s1,(UBYTE *)"path") == 0 ) {
00283             if ( AM.Path ) M_free(AM.Path,"path");
00284             AM.Path = 0;
00285         }
00286         else {
00287             MesPrint("Setups: %s not yet implemented",s1);
00288             error = 1;
00289             break;
00290         }
00291         if ( sp->flags == USEDFLAG && sp->value != 0 )
00292             M_free((void *) (sp->value),"Process option");
00293         sp->value = (LONG)strDup1(s2,"Process option");
00294         sp->flags = USEDFLAG;
00295         break;
00296     case ONOFFVALUE:
00297         if ( tolower(*s2) == 'o' && tolower(s2[1]) == 'n'
00298             && ( s2[2] == 0 || s2[2] == ' ' || s2[2] == '\\t' ) )
00299             sp->value = 1;

```

```

00300         else if ( tolower(*s2) == 'o' && tolower(s2[1]) == 'f'
00301             && tolower(s2[2]) == 'f'
00302             && ( s2[3] == 0 || s2[3] == ' ' || s2[3] == '\\t' ) )
00303             sp->value = 0;
00304         else {
00305             MesPrint("%s: Unrecognized option for parameter %s: %s"
00306                 ,proopl[filetype],s1,s2);
00307             error = 1; break;
00308         }
00309         sp->flags = USEDFLAG;
00310         break;
00311     case DEFINEVALUE:
00312     /*
00313         if ( sp->value ) M_free((UBYTE *) (sp->value), "Process option");
00314         sp->value = (LONG)strDup1(s2, "Process option");
00315     */
00316         if ( TheDefine(s2,2) ) error = 1;
00317         break;
00318     default:
00319         Error1("Error in setupparameter table for:",s1);
00320         error = 1;
00321         break;
00322     }
00323 }
00324 else {
00325     MesPrint("%s: Keyword not recognized: %s",proopl[filetype],s1);
00326     error = 1;
00327 }
00328 if ( giveback ) M_free(s2ret, "environment in setup");
00329 return(error);
00330 }
00331
00332 /*
00333     #] ProcessOption :
00334     #[ GetSetupPar :
00335 */
00336
00337 SETUPPARAMETERS *GetSetupPar(UBYTE *s)
00338 {
00339     int hi, med, lo, i;
00340     lo = 0;
00341     // -1: remove possibility of out-of-bounds read in StrICmp
00342     hi = sizeof(setupparameters)/sizeof(SETUPPARAMETERS) - 1;
00343     do {
00344         med = ( hi + lo ) / 2;
00345         i = StrICmp(s, (UBYTE *)setupparameters[med].parameter);
00346         if ( i == 0 ) return(setupparameters+med);
00347         if ( i < 0 ) hi = med-1;
00348         else lo = med+1;
00349     } while ( hi >= lo );
00350     return(0);
00351 }
00352
00353 /*
00354     #] GetSetupPar :
00355     #[ RecalcSetups :
00356 */
00357
00358 int RecalcSetups(void)
00359 {
00360     SETUPPARAMETERS *sp, *spl;
00361
00362     spl = GetSetupPar((UBYTE *) "threads");
00363     if ( AM.totalnumberofthreads > 1 ) spl->value = AM.totalnumberofthreads - 1;
00364     else spl->value = 0;
00365     /*
00366     if ( spl->value > 0 ) AM.totalnumberofthreads = spl->value+1;
00367     if ( AM.totalnumberofthreads == 0 ) AM.totalnumberofthreads = 1;
00368     */
00369     sp = GetSetupPar((UBYTE *) "filepatches");
00370     if ( sp->value < AM.totalnumberofthreads-1 )
00371         sp->value = AM.totalnumberofthreads - 1;
00372
00373     sp = GetSetupPar((UBYTE *) "smallsize");
00374     spl = GetSetupPar((UBYTE *) "smallextension");
00375     if ( 6*spl->value < 7*sp->value ) spl->value = (7*sp->value)/6;
00376     sp = GetSetupPar((UBYTE *) "termsinsmall");
00377     sp->value = ( sp->value + 15 ) & (-16L);
00378 #ifdef WITHPTHREADS
00379 {
00380     SETUPPARAMETERS *sp2;
00381     LONG totalsize, minimumsize;
00382     sp = GetSetupPar((UBYTE *) "largesize");
00383     totalsize = spl->value+sp->value;
00384     sp2 = GetSetupPar((UBYTE *) "maxtermsize");
00385     AM.MaxTer = sp2->value*sizeof(WORD);
00386     if ( AM.MaxTer < 200*(LONG) (sizeof(WORD)) ) AM.MaxTer = 200*(LONG) (sizeof(WORD));

```

```

00387     if ( AM.MaxTer > MAXPOSITIVE - 200*(LONG)(sizeof(WORD)) ) AM.MaxTer = MAXPOSITIVE -
200*(LONG)(sizeof(WORD));
00388     AM.MaxTer /= sizeof(WORD);
00389     AM.MaxTer *= sizeof(WORD);
00390     minimumsize = (AM.totalnumberofthreads-1)*(AM.MaxTer+
00391         NUMBEROFBLOCKSINSORT*MINIMUMNUMBEROFTERMS*AM.MaxTer);
00392     if ( totalsize < minimumsize ) {
00393         sp->value = minimumsize - sp1->value;
00394     }
00395 }
00396 #endif
00397 return(0);
00398 }
00399
00400 /*
00401     #] RecalcSetups :
00402     #[ AllocSetups :
00403 */
00404
00405 int AllocSetups(void)
00406 {
00407     SETUPPARAMETERS *sp;
00408     LONG LargeSize, SmallSize, SmallEsize, TermsInSmall, IOSize;
00409     int MaxPatches, MaxFpatches, error = 0, i, size;
00410     UBYTE *s;
00411 #ifndef WITHPTHREADS
00412     int j;
00413 #endif
00414     sp = GetSetupPar((UBYTE *)"threads");
00415     if ( sp->value > 0 ) AM.totalnumberofthreads = sp->value+1;
00416
00417     AM.OutBuffer = (UBYTE *)Malloc1(AM.OutBufSize+1,"OutputBuffer");
00418     AP.PreAssignStack = (LONG *)Malloc1(AP.MaxPreAssignLevel*sizeof(LONG *),"PreAssignStack");
00419     for ( i = 0; i < AP.MaxPreAssignLevel; i++ ) AP.PreAssignStack[i] = 0;
00420     AC.iBuffer = (UBYTE *)Malloc1(AC.iBufferSize+1,"statement buffer");
00421     AC.iStop = AC.iBuffer + AC.iBufferSize-2;
00422     AP.preStart = (UBYTE *)Malloc1(AP.pSize,"instruction buffer");
00423     AP.preStop = AP.preStart + AP.pSize - 3;
00424     /* AP.PreIfStack is already allocated in StartPrepro(), but to be sure we
00425        "if" the freeing */
00426     if ( AP.PreIfStack ) M_free(AP.PreIfStack,"PreIfStack");
00427     AP.PreIfStack = (int *)Malloc1(AP.MaxPreIfLevel*sizeof(int),
00428         "Preprocessor if stack");
00429     AP.PreIfStack[0] = EXECUTINGIF;
00430     sp = GetSetupPar((UBYTE *)"insidefirst");
00431     AM.ginsidefirst = AC.minsidefirst = AC.insidefirst = sp->value;
00432     /*
00433         We need to consider eliminating this variable
00434     */
00435     sp = GetSetupPar((UBYTE *)"maxtermsize");
00436     AM.MaxTer = sp->value*sizeof(WORD);
00437     if ( AM.MaxTer < 200*(LONG)(sizeof(WORD)) ) AM.MaxTer = 200*(LONG)(sizeof(WORD));
00438     if ( AM.MaxTer > MAXPOSITIVE - 200*(LONG)(sizeof(WORD)) ) AM.MaxTer = MAXPOSITIVE -
200*(LONG)(sizeof(WORD));
00439     AM.MaxTer /= (LONG)sizeof(WORD);
00440     AM.MaxTer *= (LONG)sizeof(WORD);
00441     /*
00442         Allocate workspace.
00443     */
00444     sp = GetSetupPar((UBYTE *)"workspace");
00445     AM.WorkSize = sp->value;
00446 #ifdef WITHPTHREADS
00447 #else
00448     AT.WorkSpace = (WORD *)Malloc1(AM.WorkSize*sizeof(WORD), (char *) (sp->parameter));
00449     AT.WorkTop = AT.WorkSpace + AM.WorkSize;
00450     AT.WorkPointer = AT.WorkSpace;
00451 #endif
00452     /*
00453         Fixed indices
00454     */
00455     sp = GetSetupPar((UBYTE *)"constindex");
00456     if ( ( sp->value+100+5*WILDOFFSET ) > MAXPOSITIVE ) {
00457         MesPrint("Setting of %s in setupfile too large","constindex");
00458         AM.OffsetIndex = MAXPOSITIVE - 5*WILDOFFSET - 100;
00459         MesPrint("value corrected to maximum allowed: %d",AM.OffsetIndex);
00460     }
00461     else AM.OffsetIndex = sp->value + 1;
00462     AC.FixIndices = (WORD *)Malloc1((AM.OffsetIndex)*sizeof(WORD), (char *) (sp->parameter));
00463     AM.WilInd = AM.OffsetIndex + WILDOFFSET;
00464     AM.DumInd = AM.OffsetIndex + 2*WILDOFFSET;
00465     AM.IndDum = AM.DumInd + WILDOFFSET;
00466 #ifndef WITHPTHREADS
00467     AR.CurDum = AN.IndDum = AM.IndDum;
00468 #endif
00469     AM.mTraceDum = AM.IndDum + 2*WILDOFFSET;
00470
00471     sp = GetSetupPar((UBYTE *)"parentheses");

```

```

00472     AM.MaxParLevel = sp->value+1;
00473     AC.tokenarglevel = (WORD *)Malloca1((sp->value+1)*sizeof(WORD), (char *) (sp->parameter));
00474 /*
00475     Space during calculations
00476 */
00477     sp = GetSetupPar((UBYTE *) "maxnumbersize");
00478 /*
00479     size = ( sp->value + 11 ) & (-4);
00480     AM.MaxTal = size - 2;
00481     if ( AM.MaxTal > (AM.MaxTer/sizeof(WORD)-2)/2 )
00482         AM.MaxTal = (AM.MaxTer/sizeof(WORD)-2)/2;
00483     if ( AM.MaxTal < (AM.MaxTer/sizeof(WORD)-2)/4 )
00484         AM.MaxTal = (AM.MaxTer/sizeof(WORD)-2)/4;
00485 */
00486 /*
00487     There is too much confusion about MaxTal cq maxnumbersize.
00488     It seems better to fix it at its maximum value. This way we only worry
00489     about maxtermsize. This can be understood better by the 'innocent' user.
00490 */
00491     if ( sp->value == 0 ) {
00492         AM.MaxTal = (AM.MaxTer/sizeof(WORD)-2)/2;
00493     }
00494     else {
00495         size = ( sp->value + 11 ) & (-4);
00496         AM.MaxTal = size - 2;
00497         if ( (size_t)AM.MaxTal > (size_t)((AM.MaxTer/sizeof(WORD)-2)/2) )
00498             AM.MaxTal = (AM.MaxTer/sizeof(WORD)-2)/2;
00499     }
00500     AM.MaxTal &= -sizeof(WORD)*2;
00501
00502     sp->value = AM.MaxTal;
00503     AC.cmod = (UWORD *)Malloca1(AM.MaxTal*4*sizeof(UWORD), (char *) (sp->parameter));
00504     AM.gcmmod = AC.cmod + AM.MaxTal;
00505     AC.powmod = AM.gcmmod + AM.MaxTal;
00506     AM.gpowmod = AC.powmod + AM.MaxTal;
00507 /*
00508     The IO buffers for the input and output expressions.
00509     Fscr[2] will be assigned in a later stage for hiding expressions from
00510     the regular action. That will make the program faster.
00511 */
00512     sp = GetSetupPar((UBYTE *) "scratchsize");
00513     AM.ScratSize = sp->value/sizeof(WORD);
00514     if ( AM.ScratSize < 4*AM.MaxTer ) AM.ScratSize = 4*AM.MaxTer;
00515     AM.HideSize = AM.ScratSize;
00516     sp = GetSetupPar((UBYTE *) "hidesize");
00517     if ( sp->value > 0 ) {
00518         AM.HideSize = sp->value/sizeof(WORD);
00519         if ( AM.HideSize < 4*AM.MaxTer ) AM.HideSize = 4*AM.MaxTer;
00520     }
00521     sp = GetSetupPar((UBYTE *) "factorizationcache");
00522     AM.fbuffersize = sp->value;
00523 #ifdef WITHPTHREADS
00524     sp = GetSetupPar((UBYTE *) "threadscratchsize");
00525     AM.ThreadScratSize = sp->value/sizeof(WORD);
00526     sp = GetSetupPar((UBYTE *) "threadscratchoutsize");
00527     AM.ThreadScratOutSize = sp->value/sizeof(WORD);
00528 #endif
00529 #ifndef WITHPTHREADS
00530     for ( j = 0; j < 2; j++ ) {
00531         WORD *ScratchBuf;
00532         ScratchBuf = (WORD *)Malloca1(AM.ScratSize*sizeof(WORD), "scratchsize");
00533         AR.Fscr[j].POsize = AM.ScratSize * sizeof(WORD);
00534         AR.Fscr[j].POfull = AR.Fscr[j].POfill = AR.Fscr[j].PObuffer = ScratchBuf;
00535         AR.Fscr[j].POstop = AR.Fscr[j].PObuffer + AM.ScratSize;
00536         PUTZERO(AR.Fscr[j].POposition);
00537     }
00538     AR.Fscr[2].PObuffer = 0;
00539 #endif
00540     sp = GetSetupPar((UBYTE *) "threadbucketsize");
00541     AC.ThreadBucketSize = AM.gThreadBucketSize = sp->value;
00542     sp = GetSetupPar((UBYTE *) "threadloadbalancing");
00543     AC.ThreadBalancing = AM.gThreadBalancing = sp->value;
00544     sp = GetSetupPar((UBYTE *) "threadsortfilesynch");
00545     AC.ThreadSortFileSynch = AM.gThreadSortFileSynch = sp->value;
00546 /*
00547     The size for shared memory window for oneside MPI2 communications
00548 */
00549     sp = GetSetupPar((UBYTE *) "shmwinsize");
00550     AM.shmWinSize = sp->value/sizeof(WORD);
00551     if ( AM.shmWinSize < 4*AM.MaxTer ) AM.shmWinSize = 4*AM.MaxTer;
00552 /*
00553     The sort buffer
00554 */
00555     sp = GetSetupPar((UBYTE *) "smallsize");
00556     SmallSize = sp->value;
00557     sp = GetSetupPar((UBYTE *) "smallextension");
00558     SmallEsize = sp->value;

```

```

00559     sp = GetSetupPar((UBYTE *)"largesize");
00560     LargeSize = sp->value;
00561     sp = GetSetupPar((UBYTE *)"termsinsmall");
00562     TermsInSmall = sp->value;
00563     sp = GetSetupPar((UBYTE *)"largepatches");
00564     MaxPatches = sp->value;
00565     sp = GetSetupPar((UBYTE *)"filepatches");
00566     MaxFpatches = sp->value;
00567     sp = GetSetupPar((UBYTE *)"sortiosize");
00568     IOsize = sp->value;
00569     if ( IOsize < AM.MaxTer ) { IOsize = AM.MaxTer; sp->value = IOsize; }
00570 #ifndef WITHPTHREADS
00571 #ifdef WITHZLIB
00572     for ( j = 0; j < 2; j++ ) { AR.Fscr[j].ziosize = IOsize; }
00573 #endif
00574 #endif
00575     AM.S0 = 0;
00576     AM.S0 = AllocSort(LargeSize,SmallSize,SmallEsize,TermsInSmall
00577                       ,MaxPatches,MaxFpatches,IOsize,0);
00578     /* AM.S0->file.ziosize was already set to a (larger) value by AllocSort, here it is re-set. */
00579 #ifdef WITHZLIB
00580     AM.S0->file.ziosize = IOsize;
00581 #ifndef WITHPTHREADS
00582     AR.FoStage4[0].ziosize = IOsize;
00583     AR.FoStage4[1].ziosize = IOsize;
00584     AT.S0 = AM.S0;
00585 #endif
00586 #else
00587 #ifndef WITHPTHREADS
00588     AT.S0 = AM.S0;
00589 #endif
00590 #endif
00591 #ifndef WITHPTHREADS
00592     AR.FoStage4[0].POsize = ((IOsize+sizeof(WORD)-1)/sizeof(WORD))*sizeof(WORD);
00593     AR.FoStage4[1].POsize = ((IOsize+sizeof(WORD)-1)/sizeof(WORD))*sizeof(WORD);
00594 #endif
00595     sp = GetSetupPar((UBYTE *)"subsmallsize");
00596     AM.SSmallSize = sp->value;
00597     sp = GetSetupPar((UBYTE *)"subsmallexension");
00598     AM.SSmallEsize = sp->value;
00599     sp = GetSetupPar((UBYTE *)"sublargesize");
00600     AM.SLargeSize = sp->value;
00601     sp = GetSetupPar((UBYTE *)"subtermsinsmall");
00602     AM.STermsInSmall = sp->value;
00603     sp = GetSetupPar((UBYTE *)"sublargepatches");
00604     AM.SMaxPatches = sp->value;
00605     sp = GetSetupPar((UBYTE *)"subfilepatches");
00606     AM.SMaxFpatches = sp->value;
00607     sp = GetSetupPar((UBYTE *)"subsortiosize");
00608     AM.SIOsize = sp->value;
00609     /* As for IOsize, make sure SIOsize is at least as large as AM.MaxTer. */
00610     if ( AM.SIOsize < AM.MaxTer ) { AM.SIOsize = AM.MaxTer; sp->value = AM.SIOsize; }
00611     sp = GetSetupPar((UBYTE *)"spectatorsize");
00612     AM.SpectatorSize = sp->value;
00613 /*
00614     The next code is just for the moment (26-jan-1997) because we have
00615     the new parts combined with the old. Once the old parts are gone
00616     from the program, we can eliminate this code too.
00617 */
00618     sp = GetSetupPar((UBYTE *)"functionlevels");
00619     AM.maxFlevels = sp->value + 1;
00620 #ifdef WITHPTHREADS
00621 #else
00622     AT.Nest = (NESTING)Malloc1((LONG)sizeof(struct NeStInG)*AM.maxFlevels,"functionlevels");
00623     AT.NestStop = AT.Nest + AM.maxFlevels;
00624     AT.NestPoin = AT.Nest;
00625 #endif
00626
00627     sp = GetSetupPar((UBYTE *)"maxwildcards");
00628     AM.MaxWildcards = sp->value;
00629 #ifdef WITHPTHREADS
00630 #else
00631     AT.WildMask = (WORD *)Malloc1((LONG)AM.MaxWildcards*sizeof(WORD),"maxwildcards");
00632 #endif
00633
00634     sp = GetSetupPar((UBYTE *)"compresssize");
00635     if ( sp->value < 2*AM.MaxTer ) sp->value = 2*AM.MaxTer;
00636     AM.CompressSize = sp->value;
00637 #ifndef WITHPTHREADS
00638     AR.CompressBuffer = (WORD *)Malloc1((AM.CompressSize+10)*sizeof(WORD),"compresssize");
00639     AR.CompressPointer = AR.CompressBuffer;
00640     AR.ComprTop = AR.CompressBuffer + AM.CompressSize;
00641 #endif
00642     sp = GetSetupPar((UBYTE *)"bracketindexsize");
00643     if ( sp->value < 20*AM.MaxTer ) sp->value = 20*AM.MaxTer;
00644     AM.MaxBracketBufferSize = sp->value/sizeof(WORD);
00645

```

```

00646     sp = GetSetupPar((UBYTE *)"dotchar");
00647     AO.FortDotChar = ((UBYTE *) (sp->value))[0];
00648     sp = GetSetupPar((UBYTE *)"commentchar");
00649     AP.cComChar = AP.ComChar = ((UBYTE *) (sp->value))[0];
00650     sp = GetSetupPar((UBYTE *)"procedureextension");
00651 /*
00652     Check validity first.
00653 */
00654     s = (UBYTE *) (sp->value);
00655     if ( FG.cTable[*s] != 0 ) {
00656         MesPrint(" Illegal string for procedure extension %s", (UBYTE *)sp->value);
00657         error = -2;
00658     }
00659     else {
00660         s++;
00661         while ( *s ) {
00662             if ( *s == ' ' || *s == '\t' || *s == '\n' ) {
00663                 MesPrint(" Illegal string for procedure extension %s", (UBYTE *)sp->value);
00664                 error = -2;
00665                 break;
00666             }
00667             s++;
00668         }
00669     }
00670     AP.cprocedureExtension = strDupl((UBYTE *) (sp->value), "procedureExtension");
00671     AP.procedureExtension = strDupl(AP.cprocedureExtension, "procedureExtension");
00672
00673     sp = GetSetupPar((UBYTE *)"totalsize");
00674     if ( sp->value != 2 ) AM.PrintTotalSize = sp->value;
00675
00676     sp = GetSetupPar((UBYTE *)"continuationlines");
00677     AM.FortranCont = sp->value;
00678     sp = GetSetupPar((UBYTE *)"oldorder");
00679     AM.OldOrderFlag = sp->value;
00680     sp = GetSetupPar((UBYTE *)"resettimeonclear");
00681     AM.resetTimeOnClear = sp->value;
00682     sp = GetSetupPar((UBYTE *)"nospacesinnumbers");
00683     AO.NoSpacesInNumbers = AM.gNoSpacesInNumbers = AM.ggNoSpacesInNumbers = sp->value;
00684     sp = GetSetupPar((UBYTE *)"indentspace");
00685     AO.IndentSpace = AM.gIndentSpace = AM.ggIndentSpace = sp->value;
00686     sp = GetSetupPar((UBYTE *)"jumpratio");
00687     AM.jumpratio = sp->value;
00688     sp = GetSetupPar((UBYTE *)"nwritestatistics");
00689     AC.StatsFlag = AM.gStatsFlag = AM.ggStatsFlag = 1-sp->value;
00690     sp = GetSetupPar((UBYTE *)"nwritefinalstatistics");
00691     AC.FinalStats = AM.gFinalStats = AM.ggFinalStats = 1-sp->value;
00692     sp = GetSetupPar((UBYTE *)"nwritethreadstatistics");
00693     AC.ThreadStats = AM.gThreadStats = AM.ggThreadStats = 1-sp->value;
00694     sp = GetSetupPar((UBYTE *)"nwriteprocessstatistics");
00695     AC.ProcessStats = AM.gProcessStats = AM.ggProcessStats = 1-sp->value;
00696     sp = GetSetupPar((UBYTE *)"oldparallelstatistics");
00697     AC.OldParallelStats = AM.gOldParallelStats = AM.ggOldParallelStats = sp->value;
00698     sp = GetSetupPar((UBYTE *)"oldfactarg");
00699     AC.OldFactArgFlag = AM.gOldFactArgFlag = AM.ggOldFactArgFlag = sp->value;
00700     sp = GetSetupPar((UBYTE *)"oldgcd");
00701     AC.OldGCDflag = AM.gOldGCDflag = AM.ggOldGCDflag = sp->value;
00702     sp = GetSetupPar((UBYTE *)"wtimestats");
00703     if ( sp->value == 2 ) sp->value = AM.ggWTimeStatsFlag;
00704     AC.WTimeStatsFlag = AM.gWTimeStatsFlag = AM.ggWTimeStatsFlag = sp->value;
00705 #ifdef WITHFLOAT
00706     sp = GetSetupPar((UBYTE *)"maxweight");
00707     AC.MaxWeight = AM.gMaxWeight = AM.ggMaxWeight = sp->value;
00708     sp = GetSetupPar((UBYTE *)"defaultprecision");
00709     AC.DefaultPrecision = AM.gDefaultPrecision = AM.ggDefaultPrecision = sp->value;
00710 #endif
00711     sp = GetSetupPar((UBYTE *)"sorttype");
00712     if ( StrICmp((UBYTE *)"lowfirst", (UBYTE *)sp->value) == 0 ) {
00713         AC.lSortType = SORTLOWFIRST;
00714     }
00715     else if ( StrICmp((UBYTE *)"highfirst", (UBYTE *)sp->value) == 0 ) {
00716         AC.lSortType = SORTHIGHFIRST;
00717     }
00718     else {
00719         MesPrint(" Illegal SortType specification: %s", (UBYTE *)sp->value);
00720         error = -2;
00721     }
00722
00723     sp = GetSetupPar((UBYTE *)"processbucketsize");
00724     AM.hProcessBucketSize = AM.gProcessBucketSize =
00725     AC.ProcessBucketSize = AC.mProcessBucketSize = sp->value;
00726 /*
00727     The store caches (code installed 15-aug-2006 JV)
00728 */
00729     sp = GetSetupPar((UBYTE *)"numstorecaches");
00730     AM.NumStoreCaches = sp->value;
00731     sp = GetSetupPar((UBYTE *)"sizestorecache");
00732     AM.SizeStoreCache = sp->value;

```



```

00733      /* Make sure this is a multiple of sizeof(WORD). */
00734      AM.SizeStoreCache = ((AM.SizeStoreCache+sizeof(WORD)-1)/sizeof(WORD))*sizeof(WORD);
00735 #ifndef WITHPTHREADS
00736 /*
00737      Install the store caches (15-aug-2006 JV)
00738      Note that in the case of PTHREADS this is done in InitializeOneThread
00739 */
00740      AT.StoreCache = AT.StoreCacheAlloc = 0;
00741      if ( AM.NumStoreCaches > 0 ) {
00742          STORECACHE sa, sb;
00743          size = sizeof(struct StOrEcAcHe)+AM.SizeStoreCache;
00744          size = ((size-1)/sizeof(size_t)+1)*sizeof(size_t);
00745          AT.StoreCacheAlloc = (STORECACHE)Malloc1(size*AM.NumStoreCaches,"StoreCaches");
00746          AT.StoreCache = AT.StoreCacheAlloc;
00747          sa = AT.StoreCache;
00748          for ( j = 0; j < AM.NumStoreCaches; j++ ) {
00749              sb = (STORECACHE)(void *)((UBYTE *)sa+size);
00750              if ( j == AM.NumStoreCaches-1 ) {
00751                  sa->next = 0;
00752              }
00753              else {
00754                  sa->next = sb;
00755              }
00756              SETBASEPOSITION(sa->position,-1);
00757              SETBASEPOSITION(sa->toppos,-1);
00758              sa = sb;
00759          }
00760      }
00761 #endif
00762
00763 /*
00764      And now some order sensitive things
00765 */
00766      if ( AM.Path == 0 ) {
00767          sp = GetSetupPar((UBYTE *)"path");
00768          AM.Path = strDup1((UBYTE *) (sp->value), "path");
00769      }
00770      if ( AM.IncDir == 0 ) {
00771          sp = GetSetupPar((UBYTE *)"incdir");
00772          AM.IncDir = strDup1((UBYTE *) (sp->value), "incdir");
00773      }
00774 /*
00775      if ( AM.TempDir == 0 ) {
00776          sp = GetSetupPar((UBYTE *)"tempdir");
00777          AM.TempDir = strDup1((UBYTE *) (sp->value), "tempdir");
00778      }
00779 */
00780      return(error);
00781 }
00782
00783 /*
00784      #] AllocSetups :
00785      #[ WriteSetup :
00786
00787      The routine writes the values of the setup parameters.
00788      We should do this better. (JV, 21-may-2008)
00789      The way it should be done is:
00790          a: write the raw values.
00791          b: give readjusted values.
00792          c: give derived values.
00793      Because this is a difficult subject, it would be nice to have a LaTeX
00794      document that explains this all exactly. There should then be a
00795      mechanism to poke the values of the setup into the LaTeX document.
00796      probably the easiest way is to make a file with lots of \def definitions
00797      and have that included into the LaTeX file.
00798 */
00799
00800 void WriteSetup(void)
00801 {
00802     int n = sizeof(setupparameters)/sizeof(SETUPPARAMETERS);
00803     SETUPPARAMETERS *sp;
00804     MesPrint(" The setup parameters are:");
00805     for ( sp = setupparameters; n > 0; n--, sp++ ) {
00806         switch(sp->type){
00807             case NUMERICALVALUE:
00808                 MesPrint(" %s: %1", sp->parameter, sp->value);
00809                 break;
00810             case PATHVALUE:
00811                 if ( StrICmp(sp->parameter, (UBYTE *)"path") == 0 && AM.Path ) {
00812                     MesPrint(" %s: '%s'", sp->parameter, (UBYTE *) (AM.Path));
00813                     break;
00814                 }
00815                 if ( StrICmp(sp->parameter, (UBYTE *)"incdir") == 0 && AM.IncDir ) {
00816                     MesPrint(" %s: '%s'", sp->parameter, (UBYTE *) (AM.IncDir));
00817                     break;
00818                 }
00819                 /* fall through */

```

```

00820         case STRINGVALUE:
00821             if ( StrICmp(sp->parameter, (UBYTE *) "tempdir") == 0 && AM.TempDir ) {
00822                 MesPrint("    %s: '%s'", sp->parameter, (UBYTE *) (AM.TempDir));
00823             }
00824             else if ( StrICmp(sp->parameter, (UBYTE *) "tempSortDir") == 0 && AM.TempSortDir ) {
00825                 MesPrint("    %s: '%s'", sp->parameter, (UBYTE *) (AM.TempSortDir));
00826             }
00827             else {
00828                 MesPrint("    %s: '%s'", sp->parameter, (UBYTE *) (sp->value));
00829             }
00830             break;
00831         case ONOFFVALUE:
00832             if ( sp->value == 0 )
00833                 MesPrint("    %s: OFF", sp->parameter);
00834             else if ( sp->value == 1 )
00835                 MesPrint("    %s: ON", sp->parameter);
00836             break;
00837         case DEFINEVALUE:
00838             /*
00839                 MesPrint("    %s: '%s'", sp->parameter, (UBYTE *) (sp->value));
00840             */
00841             break;
00842     }
00843 }
00844 AC.SetupFlag = 0;
00845 }
00846
00847 /*
00848     #] WriteSetup :
00849     #[ AllocSort :
00850
00851     Routine allocates a complete struct for sorting.
00852     To be used for the main allocation of the sort buffers, and
00853     in a later stage for the function and subroutine sort buffers.
00854     The level arg denotes a main buffer allocation (0) or a sub-buffer
00855     allocation (1), used only for printing warning messages for buffer
00856     size adjustments in DEBUGGING mode.
00857 */
00858 SORTING *AllocSort(LONG inLargeSize, LONG inSmallSize, LONG inSmallEsize, LONG inTermsInSmall,
00859                    int inMaxPatches, int inMaxFpatches, LONG inIOsize, int level)
00860 {
00861     DUMMYUSE(level); /* This is only used in DEBUGGING mode */
00862     LONG LargeSize = inLargeSize;
00863     LONG SmallSize = inSmallSize;
00864     LONG SmallEsize = inSmallEsize;
00865     LONG TermsInSmall = inTermsInSmall;
00866     int MaxPatches = inMaxPatches;
00867     int MaxFpatches = inMaxFpatches;
00868     LONG IOsize = inIOsize;
00869
00870     LONG longer, terms2insmall, sortsize, longerp;
00871     LONG IObuffersize = IOsize;
00872     LONG IOtry;
00873     SORTING *sort;
00874     int fname2Size = 0, j = 0;
00875     char *s;
00876     if ( AM.S0 != 0 ) {
00877         s = FG.fname2; fname2Size = 0;
00878         while ( *s ) { s++; fname2Size++; }
00879         fname2Size += 16;
00880     }
00881     if ( MaxFpatches < 4 ) MaxFpatches = 4;
00882     longer = MaxPatches > MaxFpatches ? MaxPatches : MaxFpatches;
00883     longerp = longer;
00884     while ( (1 << j) < longerp ) j++;
00885     longerp = (1 << j) + 1;
00886     longerp += sizeof(WORD*) - (longerp*sizeof(WORD*));
00887     longer++;
00888     longer += sizeof(WORD*) - (longer*sizeof(WORD*));
00889     if ( SmallSize < 16*AM.MaxTer ) SmallSize = 16*AM.MaxTer+16;
00890     TermsInSmall = (TermsInSmall+15) & (-16L);
00891     terms2insmall = 2*TermsInSmall; /* Used to be just + 100 rather than *2 */
00892     if ( SmallEsize < (SmallSize+3)/2 ) SmallEsize = (SmallSize+3)/2;
00893     if ( LargeSize > 0 && LargeSize < 2*SmallSize ) LargeSize = 2*SmallSize;
00894     SmallEsize = (SmallEsize+15) & (-16L);
00895     if ( LargeSize < 0 ) LargeSize = 0;
00896     sortsize = sizeof(SORTING);
00897     sortsize = (sortsize+15) & (-16L);
00898     IObuffersize = (IObuffersize+sizeof(WORD)-1)/sizeof(WORD);
00899     /*
00900     The next statement fixes a bug. In the rare case that we have a
00901     problem here, we expand the size of the large buffer or the
00902     small extension
00903     */
00904     if ( (ULONG) ( LargeSize+SmallEsize ) < MaxFpatches*((IObuffersize
00905         +COMPINC)*sizeof(WORD)+2*AM.MaxTer) ) {
00906         if ( LargeSize == 0 )

```

```

00907         SmallSize = MaxFpatches*((IObuffersize+COMPINC)*sizeof(WORD)+2*AM.MaxTer);
00908     else
00909         LargeSize = MaxFpatches*((IObuffersize+COMPINC)*sizeof(WORD)+2*AM.MaxTer)
00910             - SmallSize;
00911     }
00912
00913     IOtry = ((LargeSize+SmallSize)/MaxFpatches-2*AM.MaxTer)/sizeof(WORD)-COMPINC;
00914
00915     /* Here both IObuffersize and IOtry are in units of sizeof(WORD). */
00916     if ( IObuffersize < IOtry ) {
00917         IObuffersize = IOtry;
00918     }
00919
00920 #if DEBUGGING
00921     char *prefix;
00922     if ( level == 0 ) { prefix = ""; }
00923     else { prefix = "Sub"; }
00924     if ( LargeSize != inLargeSize ) { MesPrint("Warning: %sLargeSize adjusted: %l -> %l", prefix,
inLargeSize, LargeSize); }
00925     if ( SmallSize != inSmallSize ) { MesPrint("Warning: %sSmallSize adjusted: %l -> %l", prefix,
inSmallSize, SmallSize); }
00926     if ( SmallEsize != inSmallEsize ) {MesPrint("Warning: %sSmallEsize adjusted: %l -> %l", prefix,
inSmallEsize, SmallEsize); }
00927     if ( TermsInSmall != inTermsInSmall ) { MesPrint("Warning: %sTermsInSmall adjusted: %l -> %l",
prefix, inTermsInSmall, TermsInSmall); }
00928     if ( MaxPatches != inMaxPatches ) { MesPrint("Warning: MaxPatches adjusted: %d -> %d",
inMaxPatches, MaxPatches); }
00929     if ( MaxFpatches != inMaxFpatches ) {MesPrint("Warning: MaxFPatches adjusted: %d -> %d",
inMaxFpatches, MaxFpatches); }
00930     /* This one is always changed if the LargeSize has not been... */
00931     /* if ( IObuffersize != inIOsize/(LONG)sizeof(WORD) ) { MesPrint("Warning: IOsize adjusted: %l ->
%l", inIOsize/sizeof(WORD), IObuffersize); } */
00932 #endif
00933
00934     /* Allocate separate buffers for most struct members, for better testing and debugging with
valgrind. */
00935     sort = Malloc1(sizeof(*sort), "AllocSort: sorting struct");
00936
00937     sort->LargeSize = LargeSize/sizeof(WORD);
00938     sort->SmallSize = SmallSize/sizeof(WORD);
00939     sort->SmallEsize = SmallEsize/sizeof(WORD);
00940     sort->MaxPatches = MaxPatches;
00941     sort->MaxFpatches = MaxFpatches;
00942     sort->TermsInSmall = TermsInSmall;
00943     sort->Terms2InSmall = terms2insmall;
00944
00945     sort->sPointer = Malloc1(sizeof(*(sort->sPointer )))*terms2insmall, "AllocSort: sPointer");
00946     sort->Patches = Malloc1(sizeof(*(sort->Patches )))*longer, "AllocSort: Patches");
00947     sort->pStop = Malloc1(sizeof(*(sort->pStop )))*longer, "AllocSort: pStop");
00948     sort->poina = Malloc1(sizeof(*(sort->poina )))*longerp, "AllocSort: poina");
00949     sort->poin2a = Malloc1(sizeof(*(sort->poin2a )))*longerp, "AllocSort: poin2a");
00950
00951     sort->fPatches = Malloc1(sizeof(*(sort->fPatches )))*longer, "AllocSort: fPatches");
00952     sort->fPatchesStop = Malloc1(sizeof(*(sort->fPatchesStop))*longer, "AllocSort: fPatchesStop");
00953     sort->inPatches = Malloc1(sizeof(*(sort->inPatches )))*longer, "AllocSort: inPatches");
00954     sort->tree = Malloc1(sizeof(*(sort->tree )))*longerp, "AllocSort: tree");
00955     sort->used = Malloc1(sizeof(*(sort->used )))*longerp, "AllocSort: used");
00956
00957 #ifdef WITHZLIB
00958     sort->fpcompressed = Malloc1(sizeof(*(sort->fpcompressed ))*(longerp+2), "AllocSort:
fpcompressed");
00959     sort->fpincompressed = Malloc1(sizeof(*(sort->fpincompressed))*(longerp+2), "AllocSort:
fpincompressed");
00960     sort->zsparray = 0;
00961 #endif
00962
00963     sort->ktoi = Malloc1(sizeof(*(sort->ktoi))*(longerp+2), "AllocSort: ktoi");
00964
00965     // The combined Large buffer and Small buffer (+ extension) are used.
00966     // They must be allocated together.
00967     sort->lBuffer = Malloc1(sizeof(*(sort->lBuffer))*(sort->LargeSize+sort->SmallEsize),
"AllocSort: lBuffer+sBuffer");
00968     sort->lTop = sort->lBuffer+sort->LargeSize;
00969
00970     sort->sBuffer = sort->lTop;
00971     if ( sort->LargeSize == 0 ) { sort->lBuffer = 0; sort->lTop = 0; }
00972     sort->sTop = sort->sBuffer + sort->SmallSize;
00973     sort->sTop2 = sort->sBuffer + sort->SmallEsize;
00974     sort->sHalf = sort->sBuffer + (LONG)((sort->SmallSize+sort->SmallEsize)>1);
00975
00976     sort->file.PObuffer = Malloc1(IObuffersize*sizeof(*(sort->file.PObuffer))+fname2Size+16,
"AllocSort: PObuffer");
00977     sort->file.POstop = sort->file.PObuffer+IObuffersize;
00978     sort->file.POsize = IObuffersize * sizeof(WORD);
00979     sort->file.POfill = sort->file.POfull = sort->file.PObuffer;
00980     sort->file.active = 0;
00981     sort->file.handle = -1;

```

```

00982     PUTZERO(sort->file.POposition);
00983 #ifdef WITHPTHREADS
00984     sort->file.pthreadslck = dummylock;
00985 #endif
00986 #ifdef WITHZLIB
00987     sort->file.ziosize = IObuffersize*sizeof(WORD);
00988     sort->file.ziobuffer = 0;
00989 #endif
00990     if ( AM.S0 != 0 ) {
00991         sort->file.name = (char *) (sort->file.PObuffer + IObuffersize);
00992         AllocSortFileName(sort);
00993     }
00994     else sort->file.name = 0;
00995     sort->cBuffer = 0;
00996     sort->cBufferSize = 0;
00997     sort->f = 0;
00998     sort->PolyWise = 0;
00999
01000     return(sort);
01001 }
01002
01003 /*
01004     #] AllocSort :
01005     #[ AllocSortFileName :
01006 */
01007
01008 void AllocSortFileName(SORTING *sort)
01009 {
01010     GETIDENTITY
01011     char *s, *t;
01012     /*
01013         This is not the allocation before the tempfiles are determined.
01014         Hence we can use the name in FG.fname2 and modify the tail
01015     */
01016     s = FG.fname2; t = sort->file.name;
01017     while ( *s ) *t++ = *s++;
01018 #ifdef WITHPTHREADS
01019     t[-2] = 'F';
01020     snprintf(t-1,FG.fname2size-(t-1-sort->file.name),"%d.%d",identity,AN.filename);
01021 #else
01022     t[-2] = 'f';
01023     snprintf(t-1,FG.fname2size-(t-1-sort->file.name),"%d",AN.filename);
01024 #endif
01025     AN.filename++;
01026 }
01027
01028 /*
01029     #] AllocSortFileName :
01030     #[ AllocFileHandle :
01031 */
01032
01033 FILEHANDLE *AllocFileHandle(WORD par,char *name)
01034 {
01035     GETIDENTITY
01036     LONG allocation, Ssize;
01037     FILEHANDLE *fh;
01038     int i = 0;
01039     char *s, *t;
01040
01041     s = FG.fname2; i = 0;
01042     while ( *s ) { s++; i++; }
01043     if ( par == 0 ) { i += 16; Ssize = AM.SIOsize; }
01044     else { s = name; while ( *s ) { i++; s++; } i += 2; Ssize = AM.SpectatorSize; }
01045
01046     allocation = sizeof(FILEHANDLE) + (Ssize+1)*sizeof(WORD) + i*sizeof(char)
01047                 +FG.fname2size+20;
01048     fh = (FILEHANDLE *)Malloc1(allocation,"FileHandle");
01049
01050     fh->PObuffer = (WORD *) (fh+1);
01051     fh->POstop = fh->PObuffer+Ssize;
01052     fh->POsize = Ssize * sizeof(WORD);
01053     fh->active = 0;
01054     fh->handle = -1;
01055     PUTZERO(fh->POposition);
01056 #ifdef WITHPTHREADS
01057     fh->pthreadslck = dummylock;
01058 #endif
01059     if ( par == 0 ) { /* sort file */
01060         if ( AM.S0 != 0 ) {
01061             fh->name = (char *) (fh->POstop + 1);
01062             s = FG.fname2; t = fh->name;
01063             while ( *s ) *t++ = *s++;
01064 #ifdef WITHPTHREADS
01065             t[-2] = 'F';
01066             snprintf(t-1,FG.fname2size-(t-1-fh->name),"%d-%d",identity,AN.filename);
01067 #else
01068             t[-2] = 'f';

```

```

01069     snprintf(t-1,FG.fname2size-(t-1-fh->name),"%d",AN.filenum);
01070 #endif
01071     AN.filenum++;
01072 }
01073     else fh->name = 0;
01074 }
01075     else { /* Spectator file */
01076         fh->name = (char *) (fh->POstop + 1);
01077         s = FG.fname; t = fh->name;
01078         for ( i = 0; i < FG.fnamebase; i++ ) *t++ = *s++;
01079         s = name;
01080         while ( *s ) *t++ = *s++;
01081         *t = 0;
01082     }
01083     fh->POfill = fh->POfull = fh->PObuffer;
01084     return(fh);
01085 }
01086
01087 /*
01088     #] AllocFileHandle :
01089     #[ DeAllocFileHandle :
01090
01091     Made to repair deallocation of AN.filenum. 21-sep-2000
01092 */
01093
01094 void DeAllocFileHandle(FILEHANDLE *fh)
01095 {
01096     GETIDENTITY
01097     if ( fh->handle >= 0 ) {
01098         CloseFile(fh->handle);
01099         fh->handle = -1;
01100         remove(fh->name);
01101     }
01102     AN.filenum--; /* free namespace. was forgotten in first reading */
01103     M_free(fh,"Temporary FileHandle");
01104 }
01105
01106 /*
01107     #] DeAllocFileHandle :
01108     #[ MakeSetupAllocs :
01109 */
01110
01111 int MakeSetupAllocs(void)
01112 {
01113     if ( RecalcSetups() || AllocSetups() ) return(1);
01114     else return(0);
01115 }
01116
01117 /*
01118     #] MakeSetupAllocs :
01119     #[ TryFileSetups :
01120
01121     Routine looks in the input file for a start of the type
01122     [#-]
01123     #: setupparameter value
01124     It keeps looking until the first line that does not start with
01125     #-, #+ or #:
01126     Then it rewinds the input.
01127 */
01128
01129 #define SETBUFSIZE 257
01130
01131 int TryFileSetups(void)
01132 {
01133     LONG oldstreamposition;
01134     int oldstream;
01135     int error = 0, eqnum;
01136     int oldNoShowInput = AC.NoShowInput;
01137     UBYTE buff[SETBUFSIZE+1], *s, *t, *u, *settop, c;
01138     LONG linenum, prevline;
01139
01140     if ( AC.CurrentStream == 0 ) return(error);
01141     oldstream = AC.CurrentStream - AC.Streams;
01142     oldstreamposition = GetStreamPosition(AC.CurrentStream);
01143     linenum = AC.CurrentStream->linenumber;
01144     prevline = AC.CurrentStream->prevline;
01145     eqnum = AC.CurrentStream->eqnum;
01146     AC.NoShowInput = 1;
01147     settop = buff + SETBUFSIZE;
01148     if ( AC.CurrentStream->type == INPUTSTREAM && oldstreamposition == 0 )
01149         AC.CurrentStream->fileposition = 0;
01149     for(;;) {
01150         c = GetInput();
01151         if ( c == '*' || c == '\n' ) {
01152             while ( c != '\n' && c != ENDOFINPUT ) c = GetInput();
01153             if ( c == ENDOFINPUT ) goto eoi;
01154             continue;

```

```

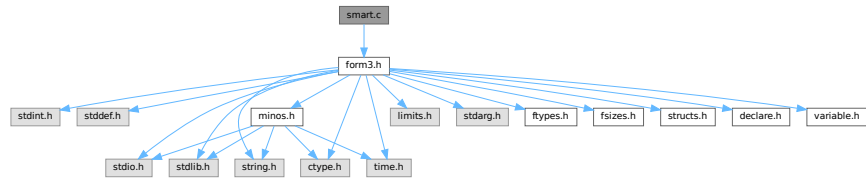
01155     }
01156     if ( c == ENDOFINPUT ) goto eoi;
01157     if ( c != '#' ) break;
01158     c = GetInput();
01159     if ( c == ENDOFINPUT ) goto eoi;
01160     if ( c != '-' && c != '+' && c != ':' ) break;
01161     if ( c != ':' ) {
01162         while ( c != '\n' && c != ENDOFINPUT ) c = GetInput();
01163         continue;
01164     }
01165     s = buff;
01166     while ( ( c = GetInput() ) == ' ' || c == '\t' || c == '\r' ) {}
01167     if ( c == ENDOFINPUT ) break;
01168     if ( c == LINEFEED ) continue;
01169     if ( c == 0 || c == ENDOFINPUT ) break;
01170     while ( c != LINEFEED && c != ENDOFINPUT ) {
01171         *s++ = c;
01172         c = GetInput();
01173         if ( c != LINEFEED && c != '\r' ) continue;
01174         if ( s >= settop ) {
01175             while ( c != '\n' && c != ENDOFINPUT ) c = GetInput();
01176             MesPrint("Setups in .frm file: Line too long. setup ignored");
01177             error++; goto nextline;
01178         }
01179     }
01180     *s++ = '\n';
01181     t = s = buff; /* name of the option */
01182     while ( tolower(*s) >= 'a' && tolower(*s) <= 'z' ) s++;
01183     if ( *s != '\n' ) {
01184         *s++ = 0;
01185         while ( *s == ' ' || *s == '\t' ) s++;
01186         u = s; /* 'value' of the option */
01187         while ( *s && *s != '\n' && *s != '\r' ) s++;
01188         if ( *s ) *s++ = 0;
01189     }
01190     else {
01191         /* The value is empty. */
01192         u = s;
01193         *s++ = 0;
01194     }
01195     error += ProcessOption(t,u,1);
01196 nextline:;
01197 }
01198 AC.NoShowInput = oldNoShowInput;
01199 AC.CurrentStream = AC.Streams + oldstream;
01200 PositionStream(AC.CurrentStream,oldstreamposition);
01201 AC.CurrentStream->linenum = linenum;
01202 AC.CurrentStream->prevline = prevline;
01203 AC.CurrentStream->eqnum = eqnum;
01204 ClearPushback();
01205 if ( AC.CurrentStream->type == INPUTSTREAM ) AC.CurrentStream->fileposition = -1;
01206 return(error);
01207 eoi:
01208 MesPrint("Input file without a program.");
01209 return(-1);
01210 }
01211
01212 /*
01213     #] TryFileSetups :
01214     #[ TryEnvironment :
01215 */
01216
01217 int TryEnvironment(void)
01218 {
01219     char *s, *t, *u, varname[100];
01220     int i,imax = sizeof(setupparameters)/sizeof(SETUPPARAMETERS);
01221     int error = 0;
01222     varname[0] = 'F'; varname[1] = 'O'; varname[2] = 'R'; varname[3] = 'M';
01223     varname[4] = '_'; varname[5] = 0;
01224     for ( i = 0; i < imax; i++ ) {
01225         t = s = (char *) (setupparameters[i].parameter);
01226         u = varname+5;
01227         while ( *s ) { *u++ = (char) (toupper((unsigned char)*s)); s++; }
01228         *u = 0;
01229         s = (char *) (getenv(varname));
01230         if ( s ) {
01231             error += ProcessOption((UBYTE *)t, (UBYTE *)s,2);
01232         }
01233     }
01234     return(error);
01235 }
01236
01237 /*
01238     #] TryEnvironment :
01239     #] Setups :
01240 */

```

4.127 smart.c File Reference

```
#include "form3.h"
```

Include dependency graph for smart.c:



Functions

- int [StudyPattern](#) (WORD *lhs)
- int [MatchIsPossible](#) (WORD *pattern, WORD *term)

4.127.1 Detailed Description

The functions for smart pattern searches in combinations of functions. When many wildcards are involved and the functions are (anti)symmetric an exhaustive search for all possibilities may take very much time (like factorial in the number of wildcards) while a human can often see immediately that there cannot be a match. The routines here try to make FORM a bit smarter in this respect.

This is just the beginning. It still needs lots of work!

Definition in file [smart.c](#).

4.127.2 Function Documentation

4.127.2.1 StudyPattern()

```
int StudyPattern (
    WORD * lhs )
```

Definition at line 62 of file [smart.c](#).

4.127.2.2 MatchIsPossible()

```
int MatchIsPossible (
    WORD * pattern,
    WORD * term )
```

Definition at line 299 of file [smart.c](#).

4.128 smart.c

[Go to the documentation of this file.](#)

```

00001
00012 /* #[ License : */
00013 /*
00014 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00015 *   When using this file you are requested to refer to the publication
00016 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00017 *   This is considered a matter of courtesy as the development was paid
00018 *   for by FOM the Dutch physics granting agency and we would like to
00019 *   be able to track its scientific use to convince FOM of its value
00020 *   for the community.
00021 *
00022 *   This file is part of FORM.
00023 *
00024 *   FORM is free software: you can redistribute it and/or modify it under the
00025 *   terms of the GNU General Public License as published by the Free Software
00026 *   Foundation, either version 3 of the License, or (at your option) any later
00027 *   version.
00028 *
00029 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00030 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00031 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00032 *   details.
00033 *
00034 *   You should have received a copy of the GNU General Public License along
00035 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00036 */
00037 /* #[ License : */
00038 /*
00039     #[ Includes : function.c
00040 */
00041
00042 #include "form3.h"
00043
00044 /*
00045     #[ Includes :
00046     #[ StudyPattern :
00047
00048     Argument is a complete lhs of an id-statement
00049     Its last word is a zero (new convention(18-may-1997)) to indicate
00050     that no extra information is following. We can add there information
00051     about the pattern that will help during the actual pattern matching.
00052     Currently the WorkPointer points to just after this lhs.
00053
00054     Task of this routine: To study the functions, their symmetry properties
00055     and their wildcards to determine in which order the functions can
00056     be matched best. If the order should be different we can change it here.
00057
00058     Problem encountered 22-dec-2008 (JV): we don't take noncommuting
00059     functions into account.
00060 */
00061
00062 int StudyPattern(WORD *lhs)
00063 {
00064     GETIDENTITY
00065     WORD *fullproto, *pat, *p, *p1, *p2, *pstop, *info, f, nn;
00066     int numfun = 0, numsym = 0, allwilds = 0, i, j, k, nc;
00067     FUN_INFO *finf, *fmin, *f1, *f2, funscratch;
00068
00069     fullproto = lhs + IDHEAD;
00070 /* if ( *lhs == TYPEIF ) fullproto--; */
00071     pat = fullproto + fullproto[1];
00072     info = pat + *pat;
00073
00074     p = pat + 1;
00075     while ( p < info ) {
00076         if ( *p >= FUNCTION ) {
00077             numfun++;
00078             nn = *p - FUNCTION;
00079             if ( nn >= WILDOFFSET ) nn -= WILDOFFSET;
00080 /*
00081         We check here for cases that are not allowed like ?a inside
00082         symmetric functions or tensors.
00083 */
00084             if ( ( functions[nn].symmetric == SYMMETRIC ) ||
00085                 ( functions[nn].symmetric == ANTISYMMETRIC ) ) {
00086                 p2 = p+p[1]; p1 = p+FUNHEAD;
00087                 if ( functions[nn].spec > 0 ) {
00088                     while ( p1 < p2 ) {
00089                         if ( *p1 == FUNNYWILD ) {
00090                             MesPrint("&Argument field wildcards are not allowed inside (anti)symmetric
functions or tensors");
00091                             return(1);

```



```

00092         }
00093         p1++;
00094     }
00095 }
00096 else {
00097     while ( p1 < p2 ) {
00098         if ( *p1 == -ARGWILD ) {
00099             MesPrint("&Argument field wildcards are not allowed inside (anti)symmetric
functions or tensors");
00100             return(1);
00101         }
00102         NEXTARG(p1);
00103     }
00104 }
00105 }
00106 }
00107 p += p[1];
00108 }
00109 if ( numfun == 0 ) return(0);
00110 if ( ( lhs[2] & SUBMASK ) == SUBALL ) {
00111     p = pat + 1;
00112     while ( p < info ) {
00113         if ( *p == SYMBOL || *p == VECTOR || *p == DOTPRODUCT || *p == INDEX ) {
00114             MesPrint("&id,all can have only functions and/or tensors in the lhs.");
00115             return(1);
00116         }
00117         p += p[1];
00118     }
00119 }
00120 /*
00121 We need now some room for the information about the functions
00122 */
00123 if ( numfun > AN.numfuninfo ) {
00124     if ( AN.FunInfo ) M_free(AN.FunInfo,"funinfo");
00125     AN.numfuninfo = numfun + 10;
00126     AN.FunInfo = (FUN_INFO *)Malloc1(AN.numfuninfo*sizeof(FUN_INFO),"funinfo");
00127 }
00128 /*
00129 Now collect the information. First the locations.
00130 */
00131 p = pat + 1; i = 0;
00132 while ( p < info ) {
00133     if ( *p >= FUNCTION ) AN.FunInfo[i++].location = p;
00134     p += p[1];
00135 }
00136 for ( i = 0, finf = AN.FunInfo; i < numfun; i++, finf++ ) {
00137     p = finf->location;
00138     pstop = p + p[1];
00139     f = *p;
00140     if ( f > FUNCTION+WILDOFFSET ) f -= WILDOFFSET;
00141     finf->numargs = finf->numfunnies = finf->numwildcards = 0;
00142     finf->symmet = functions[f-FUNCTION].symmetric;
00143     finf->tensor = functions[f-FUNCTION].spec;
00144     finf->commute = functions[f-FUNCTION].commute;
00145     if ( finf->tensor >= TENSORFUNCTION ) {
00146         p += FUNHEAD;
00147         while ( p < pstop ) {
00148             if ( *p == FUNNYWILD ) {
00149                 finf->numfunnies++; p+= 2; continue;
00150             }
00151             else if ( *p < 0 ) {
00152                 if ( *p >= AM.OffsetVector + WILDOFFSET && *p < MINSPEC ) {
00153                     finf->numwildcards++;
00154                 }
00155             }
00156             else {
00157                 if ( *p >= AM.OffsetIndex + WILDOFFSET &&
00158                     *p <= AM.OffsetIndex + 2*WILDOFFSET ) finf->numwildcards++;
00159             }
00160             finf->numargs++;
00161             p++;
00162         }
00163     }
00164     else {
00165         p += FUNHEAD;
00166         while ( p < pstop ) {
00167             if ( *p > 0 ) { finf->numargs++; p += *p; continue; }
00168             if ( *p <= -FUNCTION ) {
00169                 if ( *p <= -FUNCTION - WILDOFFSET ) finf->numwildcards++;
00170                 p++;
00171             }
00172             else if ( *p == -SYMBOL ) {
00173                 if ( p[1] >= 2*MAXPOWER ) finf->numwildcards++;
00174                 p += 2;
00175             }
00176             else if ( *p == -INDEX ) {
00177                 if ( p[1] >= AM.OffsetIndex + WILDOFFSET &&

```

```

00178         p[1] <= AM.OffsetIndex + 2*WILDOFFSET ) finf->numwildcards++;
00179         p += 2;
00180     }
00181     else if ( *p == -VECTOR || *p == -MINVECTOR ) {
00182         if ( p[1] >= AM.OffsetVector + WILDOFFSET && p[1] < MINSPEC ) {
00183             finf->numwildcards++;
00184         }
00185         p += 2;
00186     }
00187     else if ( *p == -ARGWILD ) {
00188         finf->numfunnies++;
00189         p += 2;
00190     }
00191     else { p += 2; }
00192     finf->numargs++;
00193 }
00194 }
00195 if ( finf->symmet ) {
00196     numsym++;
00197     allwilds += finf->numwildcards + finf->numfunnies;
00198 }
00199 }
00200 if ( numsym == 0 ) return(0);
00201 if ( allwilds == 0 ) return(0);
00202 /*
00203 We have the information in the array AN.FunInfo.
00204 We sort things and then write the sorted pattern.
00205 Of course we may not play with the order of the noncommuting functions.
00206 Of course we have to become even smarter in the future and look during
00207 the sorting which wildcards are assigned when.
00208 But for now this should do.
00209 */
00210 for ( nc = numfun-1; nc >= 0; nc-- ) { if ( AN.FunInfo[nc].commute ) break; }
00211
00212 finf = AN.FunInfo;
00213 for ( i = nc+2; i < numfun; i++ ) {
00214     fmin = finf; finf++;
00215     if ( ( finf->symmet < fmin->symmet ) || (
00216         ( finf->symmet == fmin->symmet ) &&
00217         ( ( finf->numwildcards+finf->numfunnies < fmin->numwildcards+fmin->numfunnies )
00218         || ( ( finf->numwildcards+finf->numfunnies == fmin->numwildcards+fmin->numfunnies )
00219         && ( finf->numwildcards < fmin->numfunnies ) ) ) ) ) {
00220         funscratch = AN.FunInfo[i];
00221         AN.FunInfo[i] = AN.FunInfo[i-1];
00222         AN.FunInfo[i-1] = funscratch;
00223         for ( j = i-1; j > nc && j > 0; j-- ) {
00224             f1 = AN.FunInfo+j;
00225             f2 = f1-1;
00226             if ( ( f1->symmet < f2->symmet ) || (
00227                 ( f1->symmet == f2->symmet ) &&
00228                 ( ( f1->numwildcards+f1->numfunnies < f2->numwildcards+f2->numfunnies )
00229                 || ( ( f1->numwildcards+f1->numfunnies == f2->numwildcards+f2->numfunnies )
00230                 && ( f1->numwildcards < f2->numfunnies ) ) ) ) ) {
00231                 funscratch = AN.FunInfo[j];
00232                 AN.FunInfo[j] = AN.FunInfo[j-1];
00233                 AN.FunInfo[j-1] = funscratch;
00234             }
00235             else break;
00236         }
00237     }
00238 }
00239 /*
00240 Now we rewrite the pattern. First into the space after it and then we
00241 copy it back. Be careful with the non-commutative functions. There the
00242 worst one should decide.
00243 */
00244 p = pat + 1;
00245 p2 = info;
00246 for ( i = 0; i < numfun; i++ ) {
00247     if ( i == nc ) {
00248         for ( k = 0; k <= nc; k++ ) {
00249             if ( AN.FunInfo[k].commute ) {
00250                 p1 = AN.FunInfo[k].location; j = p1[1];
00251                 NCOPY(p2,p1,j)
00252             }
00253         }
00254     }
00255     else if ( AN.FunInfo[i].commute == 0 ) {
00256         p1 = AN.FunInfo[i].location; j = p1[1];
00257         NCOPY(p2,p1,j)
00258     }
00259 }
00260 p = pat + 1; p1 = info;
00261 while ( p1 < p2 ) *p++ = *p1++;
00262 /*
00263 And now we place the relevant information in the info part
00264 */

```

```

00265     p2 = info+1;
00266     for ( i = 0; i < numfun; i++ ) {
00267         if ( i == nc ) {
00268             for ( k = 0; k <= nc; k++ ) {
00269                 if ( AN.FunInfo[k].commute ) {
00270                     finf = AN.FunInfo + k;
00271                     *p2++ = finf->numargs;
00272                     *p2++ = finf->numwildcards;
00273                     *p2++ = finf->numfunnies;
00274                     *p2++ = finf->symmet;
00275                 }
00276             }
00277         }
00278         else if ( AN.FunInfo[i].commute == 0 ) {
00279             finf = AN.FunInfo + i;
00280             *p2++ = finf->numargs;
00281             *p2++ = finf->numwildcards;
00282             *p2++ = finf->numfunnies;
00283             *p2++ = finf->symmet;
00284         }
00285     }
00286     *info = p2-info;
00287     lhs[1] = p2-lhs;
00288     return(0);
00289 }
00290
00291 /*
00292  #] StudyPattern :
00293  #[ MatchIsPossible :
00294
00295  We come here when there are functions and there is nontrivial
00296  symmetry related wildcarding.
00297 */
00298
00299 int MatchIsPossible(WORD *pattern, WORD *term)
00300 {
00301     GETIDENTITY
00302     WORD *info = pattern + *pattern;
00303     WORD *t, *tstop, *tt, *inf, *p;
00304     int numfun = 0, inpat, i, j, numargs;
00305     FUN_INFO *finf;
00306 /*
00307  We need a list of functions and their number of arguments
00308 */
00309     GETSTOP(term,tstop);
00310     t = term + 1;
00311     while ( t < tstop ) {
00312         if ( *t >= FUNCTION ) numfun++;
00313         t += t[1];
00314     }
00315     if ( numfun == 0 ) goto NotPossible;
00316     if ( *info == SETSET ) info += info[1];
00317     inpat = (*info-1)/4;
00318     if ( inpat > numfun ) goto NotPossible;
00319 /*
00320  We need now some room for the information about the functions
00321 */
00322     if ( numfun > AN.numfuninfo ) {
00323         if ( AN.FunInfo ) M_free(AN.FunInfo,"funinfo");
00324         AN.numfuninfo = numfun + 10;
00325         AN.FunInfo = (FUN_INFO *)Malloc1(AN.numfuninfo*sizeof(FUN_INFO),"funinfo");
00326     }
00327     t = term + 1; finf = AN.FunInfo;
00328     while ( t < tstop ) {
00329         if ( *t >= FUNCTION ) {
00330             finf->location = t;
00331             if ( functions[*t-FUNCTION].spec >= TENSORFUNCTION ) {
00332                 numargs = t[1]-FUNHEAD;
00333                 t += t[1];
00334             }
00335             else {
00336                 numargs = 0;
00337                 tt = t + t[1];
00338                 t += FUNHEAD;
00339                 while ( t < tt ) {
00340                     numargs++;
00341                     NEXTARG(t)
00342                 }
00343             }
00344             finf->numargs = numargs;
00345             finf++;
00346         }
00347         else t += t[1];
00348     }
00349 /*
00350  Now we first find a partner for each function in the pattern
00351  with a fixed number of arguments

```

```

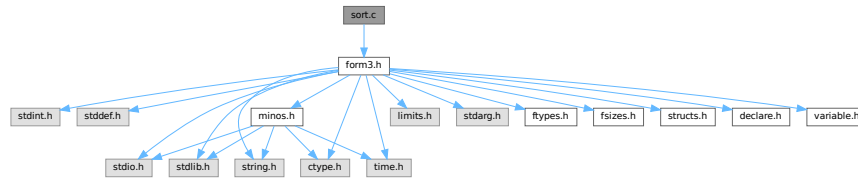
00352 */
00353     for ( i = 0, inf = info+1, p = pattern+1; i < inpat; i++, inf += 4, p+=p[1] ) {
00354         if ( inf[2] ) continue;
00355         if ( *p >= (FUNCTION+WILDOFFSET) ) continue;
00356         for ( j = 0, finf = AN.FunInfo; j < numfun; j++, finf++ ) {
00357             if ( *p == *(finf->location) && *inf == finf->numargs ) {
00358                 finf->numargs = -finf->numargs-1;
00359                 break;
00360             }
00361         }
00362         if ( j >= numfun ) goto NotPossible;
00363     }
00364     for ( i = 0, inf = info+1, p = pattern+1; i < inpat; i++, inf += 4, p+=p[1] ) {
00365         if ( inf[2] ) continue;
00366         if ( *p < (FUNCTION+WILDOFFSET) ) continue;
00367         for ( j = 0, finf = AN.FunInfo; j < numfun; j++, finf++ ) {
00368             if ( *inf == finf->numargs ) {
00369                 finf->numargs = -finf->numargs-1;
00370                 break;
00371             }
00372         }
00373         if ( j >= numfun ) goto NotPossible;
00374     }
00375     for ( i = 0, inf = info+1, p = pattern+1; i < inpat; i++, inf += 4, p+=p[1] ) {
00376         if ( inf[2] == 0 ) continue;
00377         if ( *p >= (FUNCTION+WILDOFFSET) ) continue;
00378         for ( j = 0, finf = AN.FunInfo; j < numfun; j++, finf++ ) {
00379             if ( *p == *(finf->location) && (*inf-inf[2]) <= finf->numargs ) {
00380                 finf->numargs = -finf->numargs-1;
00381                 break;
00382             }
00383         }
00384         if ( j >= numfun ) goto NotPossible;
00385     }
00386     for ( i = 0, inf = info+1, p = pattern+1; i < inpat; i++, inf += 4, p+=p[1] ) {
00387         if ( inf[2] == 0 ) continue;
00388         if ( *p < (FUNCTION+WILDOFFSET) ) continue;
00389         for ( j = 0, finf = AN.FunInfo; j < numfun; j++, finf++ ) {
00390             if ( (*inf-inf[2]) <= finf->numargs ) {
00391                 finf->numargs = -finf->numargs-1;
00392                 break;
00393             }
00394         }
00395         if ( j >= numfun ) goto NotPossible;
00396     }
00397 /*
00398     Thus far we have determined that for each function in the pattern
00399     there is a potential partner with enough arguments.
00400     For now the rest is up to the real pattern matcher.
00401
00402     To undo the disabling of the number of arguments we need this code
00403     for ( i = 0, finf = AN.FunInfo; i < numfun; i++, finf++ ) {
00404         if ( finf->numargs < 0 ) finf->numargs = -finf->numargs-1;
00405     }
00406 */
00407     return(1);
00408 NotPossible:
00409 /*
00410     PrintTerm(pattern,"p");
00411     PrintTerm(term,"t");
00412 */
00413     return(0);
00414 }
00415
00416 /*
00417 #] MatchIsPossible :
00418 */

```

4.129 sort.c File Reference

```
#include "form3.h"
```

Include dependency graph for sort.c:



Macros

- `#define NEWSPLITMERGE`
- `#define NEWSPLITMERGETIMSORT`
- `#define HUMANSTRLEN 12`
- `#define HUMANSUFFLEN 4`
- `#define INSLENGTH(x) w[1] = FUNHEAD+ARGHEAD+x; w[FUNHEAD] = ARGHEAD+x;`

Functions

- void [HumanString](#) (char *string, float input, const char suffix[HUMANSUFFLEN][4])
- void [WriteStats](#) (POSITION *plspace, WORD par, WORD checkLogType)
- int [NewSort](#) (PHEAD0)
- LONG [EndSort](#) (PHEAD WORD *buffer, int par)
- LONG [PutIn](#) (FILEHANDLE *file, POSITION *position, WORD *buffer, WORD **take, int npat)
- int [Sflush](#) (FILEHANDLE *fi)
- WORD [PutOut](#) (PHEAD WORD *term, POSITION *position, FILEHANDLE *fi, WORD ncomp)
- int [FlushOut](#) (POSITION *position, FILEHANDLE *fi, int compr)
- int [AddCoef](#) (PHEAD WORD **ps1, WORD **ps2)
- int [AddPoly](#) (PHEAD WORD **ps1, WORD **ps2)
- void [AddArgs](#) (PHEAD WORD *s1, WORD *s2, WORD *m)
- WORD [Compare1](#) (PHEAD WORD *term1, WORD *term2, WORD level)
- WORD [CompareSymbols](#) (PHEAD WORD *term1, WORD *term2, WORD par)
- WORD [CompareHSymbols](#) (PHEAD WORD *term1, WORD *term2, WORD par)
- LONG [ComPress](#) (WORD **ss, LONG *n)
- LONG [SplitMerge](#) (PHEAD WORD **Pointer, LONG number)
- void [GarbHand](#) (void)
- int [MergePatches](#) (WORD par)
- int [StoreTerm](#) (PHEAD WORD *term)
- void [StageSort](#) (FILEHANDLE *fout)
- int [SortWild](#) (WORD *w, WORD nw)
- void [CleanUpSort](#) (int num)
- void [LowerSortLevel](#) (void)
- WORD * [PolyRatFunSpecial](#) (PHEAD WORD *t1, WORD *t2)
- void [SimpleSplitMergeRec](#) (WORD *array, WORD num, WORD *auxarray)
- void [SimpleSplitMerge](#) (WORD *array, WORD num)
- WORD [BinarySearch](#) (WORD *array, WORD num, WORD x)

Variables

- char * [totterms](#) [] = { " ", ">>", "-->" }
- const char [humanTermsSuffix](#) [HUMANSUFFLEN][4] = {"K ", "M ", "B ", "T "}
- const char [humanBytesSuffix](#) [HUMANSUFFLEN][4] = {"KiB", "MiB", "GiB", "TiB"}

4.129.1 Detailed Description

Contains the sort routines. We distinguish levels of sorting. The ground level is the sorting of terms in an expression. When a term has functions, the arguments can contain terms that need sorting, which this then done by raising the level. This can happen recursively. NewSort and EndSort automatically raise and lower the level. Because the ground level does some special things, sometimes we have to raise immediately to the second level skipping the ground level.

Special routines for the parallel sorting are in the file [threads.c](#). Also the sorting of terms in polynomials is special but most of that is controlled by changing the address of the compare routine. Other routines relevant for adding rational polynomials are in the file [polynito.c](#).

Definition in file [sort.c](#).

4.129.2 Macro Definition Documentation

4.129.2.1 NEWSPLITMERGE

```
#define NEWSPLITMERGE
```

Definition at line 53 of file [sort.c](#).

4.129.2.2 NEWSPLITMERGETIMSORT

```
#define NEWSPLITMERGETIMSORT
```

Definition at line 55 of file [sort.c](#).

4.129.2.3 HUMANSTRLEN

```
#define HUMANSTRLEN 12
```

Definition at line 91 of file [sort.c](#).

4.129.2.4 HUMANSUFFLEN

```
#define HUMANSUFFLEN 4
```

Definition at line 92 of file [sort.c](#).

4.129.2.5 INSLENGTH

```
#define INSLENGTH(
    x ) w[1] = FUNHEAD+ARGHEAD+x; w[FUNHEAD] = ARGHEAD+x;
```

Definition at line 2338 of file [sort.c](#).

4.129.3 Function Documentation

4.129.3.1 HumanString()

```
void HumanString (
    char * string,
    float input,
    const char suffix[HUMANSUFFLEN][4] )
```

Definition at line 95 of file [sort.c](#).

4.129.3.2 WriteStats()

```
void WriteStats (
    POSITION * plspace,
    WORD par,
    WORD checkLogType )
```

Writes the statistics.

Parameters

<i>plspace</i>	The size in bytes currently occupied
<i>par</i>	par = 0 = STATSSPLITMERGE after a splitmerge. par = 1 = STATSMERGETOFILE after merge to sortfile. par = 2 = STATSPOSTSORT after the sort
<i>checkLogType</i>	== CHECKLOGTYPE: The output should not go to the log file if AM.LogType is 1.

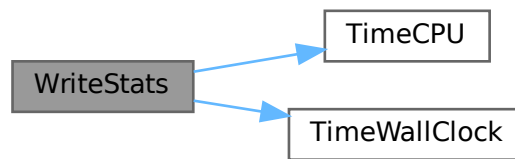
current expression is to be found in AR.CurExpr. terms are in S->TermsLeft. S->GenTerms.

Definition at line 127 of file [sort.c](#).

References [TimeCPU\(\)](#), and [TimeWallClock\(\)](#).

Referenced by [EndSort\(\)](#), [PF_Processor\(\)](#), and [StoreTerm\(\)](#).

Here is the call graph for this function:



4.129.3.3 NewSort()

```
int NewSort (
    PHEAD0 )
```

Starts a new sort. At the lowest level this is a 'main sort' with the struct according to the parameters in S0. At higher levels this is a sort for functions, subroutines or dollars. We prepare the arrays and structs.

Returns

Regular convention (OK -> 0)

Definition at line 649 of file [sort.c](#).

Referenced by [generate_expression\(\)](#), [get_expression\(\)](#), [InFunction\(\)](#), [MakeDollarInteger\(\)](#), [MakeDollarMod\(\)](#), [PF_CollectModifiedDollars\(\)](#), [PF_Processor\(\)](#), [poly_factorize_expression\(\)](#), [poly_sort\(\)](#), [poly_unfactorize_expression\(\)](#), [PolyFunMul\(\)](#), [Processor\(\)](#), [TakeArgContent\(\)](#), and [TestMatch\(\)](#).

4.129.3.4 EndSort()

```
LONG EndSort (
    PHEAD WORD * buffer,
    int par )
```

Finishes a sort. At AR.sLevel == 0 the output is to the regular output stream. When AR.sLevel > 0, the parameter par determines the actual output. The AR.sLevel will be popped. All ongoing stages are finished and if the sortfile is open it is closed. The statistics are printed when AR.sLevel == 0 par == 0 [Output](#) to the buffer. par == 1 Sort for function arguments. The output will be copied into the buffer. It is assumed that this is in the Workspace. par == 2 Sort for \$-variable. We return the address of the buffer that contains the output in buffer (treated like WORD **). We first catch the output in a file (unless we can intercept things after the small buffer has been sorted) Then we read from the file into a buffer. Only when par == 0 data compression can be attempted at AT.SS==AT.S0.

Parameters

<i>buffer</i>	buffer for output when needed
<i>par</i>	See above

Returns

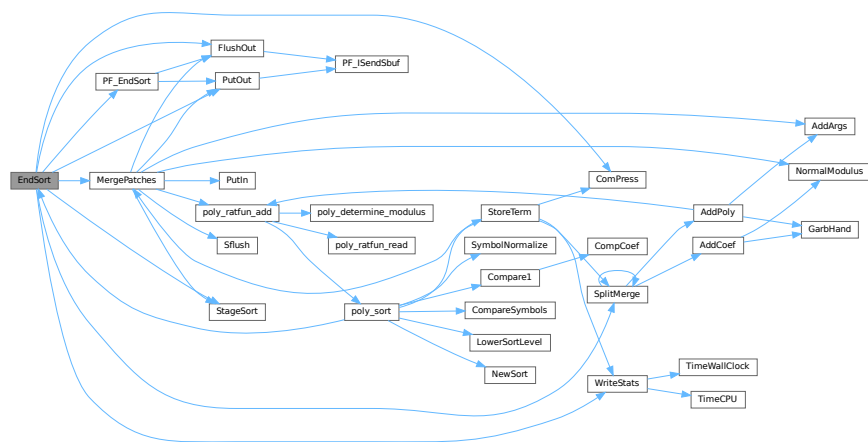
If negative: error. If positive: number of words in output.

Definition at line 745 of file [sort.c](#).

References [ComPress\(\)](#), [FlushOut\(\)](#), [MergePatches\(\)](#), [PF_EndSort\(\)](#), [PutOut\(\)](#), [SplitMerge\(\)](#), [StageSort\(\)](#), and [WriteStats\(\)](#).

Referenced by [generate_expression\(\)](#), [get_expression\(\)](#), [InFunction\(\)](#), [MakeDollarInteger\(\)](#), [MakeDollarMod\(\)](#), [PF_CollectModifiedDollars\(\)](#), [PF_Processor\(\)](#), [poly_factorize_expression\(\)](#), [poly_sort\(\)](#), [poly_unfactorize_expression\(\)](#), [PolyFunMul\(\)](#), [Processor\(\)](#), [TakeArgContent\(\)](#), and [TestMatch\(\)](#).

Here is the call graph for this function:



4.129.3.5 PutIn()

```

LONG PutIn (
    FILEHANDLE * file,
    POSITION * position,
    WORD * buffer,
    WORD ** take,
    int npat )

```

Reads a new patch from position in file handle. It is put at buffer, anything after take is moved forward. This would be part of a term that hasn't been used yet. Because of this there should be some space before the start of the buffer

Parameters

<i>file</i>	The file system from which to read
<i>position</i>	The position from which to read
<i>buffer</i>	The buffer into which to read
<i>take</i>	The unused tail should be moved before the buffer
<i>npat</i>	The number of the patch. Is needed if the information was compressed with gzip, because each patch has its own independent gzip encoding.

Definition at line 1324 of file [sort.c](#).

References [FiLe::handle](#).

Referenced by [MergePatches\(\)](#).

4.129.3.6 Sflush()

```
int Sflush (
    FILEHANDLE * fi )
```

Puts the contents of a buffer to output Only to be used when there is a single patch in the large buffer.

Parameters

<i>fi</i>	The filesystem (or its cache) to which the patch should be written

Definition at line 1384 of file [sort.c](#).

References [FiLe::handle](#).

Referenced by [MergePatches\(\)](#).

4.129.3.7 PutOut()

```
WORD PutOut (
    PHEAD WORD * term,
    POSITION * position,
    FILEHANDLE * fi,
    WORD ncomp )
```

Routine writes one term to file handle at position. It returns the new value of the position.

NOTE: For 'final output' we may have to index the brackets. See the struct BRACKETINDEX. We should maintain:
 1: a list with brackets array with the brackets
 2: a list of objects of type BRACKETINDEX. It contains array with either pointers or offsets to the list of brackets. starting positions in the file. The index may be tied to a maximum size. In that case we may have to prune the list occasionally.

Parameters

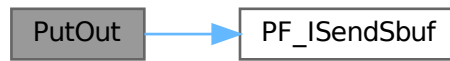
<i>term</i>	The term to be written
<i>position</i>	The position in the file. Afterwards it is updated
<i>fi</i>	The file (or its cache) to which should be written
<i>ncomp</i>	Information about what type of compression should be used

Definition at line 1470 of file [sort.c](#).

References [FiLe::handle](#), and [PF_ISendSbuf\(\)](#).

Referenced by [EndSort\(\)](#), [generate_expression\(\)](#), [MergePatches\(\)](#), [PF_EndSort\(\)](#), [PF_Processor\(\)](#), and [Processor\(\)](#).

Here is the call graph for this function:



4.129.3.8 FlushOut()

```
int FlushOut (
    POSITION * position,
    FILEHANDLE * fi,
    int compr )
```

Completes output to an output file and writes the trailing zero.

Parameters

<i>position</i>	The position in the file after writing
<i>fi</i>	The file (or its cache)
<i>compr</i>	Indicates whether there should be compression with gzip.

Returns

Regular conventions (OK -> 0).

Definition at line 1832 of file [sort.c](#).

References [FiLe::handle](#), [BrAcKeTiNfO::indexbuffer](#), and [PF_ISendSbuf\(\)](#).

Referenced by [EndSort\(\)](#), [MergePatches\(\)](#), [PF_EndSort\(\)](#), and [Processor\(\)](#).

Here is the call graph for this function:



4.129.3.9 AddCoef()

```
int AddCoef (
    PHEAD WORD ** ps1,
    WORD ** ps2 )
```

Adds the coefficients of the terms *ps1 and *ps2. The problem comes when there is not enough space for a new longer coefficient. First a local solution is tried. If this is not successful we need to move terms around. The possibility of a garbage collection should not be ignored, as avoiding this costs very much extra space which is nearly wasted otherwise.

If the return value is zero the terms cancelled.

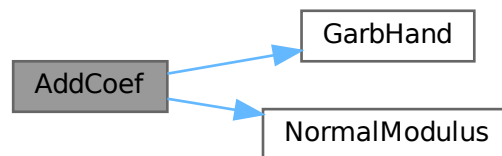
The resulting term is left in *ps1.

Definition at line 2046 of file [sort.c](#).

References [GarbHand\(\)](#), and [NormalModulus\(\)](#).

Referenced by [SplitMerge\(\)](#).

Here is the call graph for this function:



4.129.3.10 AddPoly()

```
int AddPoly (
    PHEAD WORD ** ps1,
    WORD ** ps2 )
```

Routine should be called when `S->PolyWise != 0`. It points then to the position of `AR.PolyFun` in both terms.

We add the contents of the arguments of the two polynomials. Special attention has to be given to special arguments. We have to reserve a space equal to the size of one term + the size of the argument of the other. The addition has to be done in this routine because not all objects are reentrant.

Newer addition (12-nov-2007). The `PolyFun` can have two arguments. In that case `S->PolyFlag` is 2 and we have to call the routine for adding rational polynomials. We have to be rather careful what happens with: The location of the output The order of the terms in the arguments At first we allow only univariate polynomials in the `PolyFun`. This restriction will be lifted a.s.a.p.

Parameters

<i>ps1</i>	A pointer to the position of the first term
<i>ps2</i>	A pointer to the position of the second term

Returns

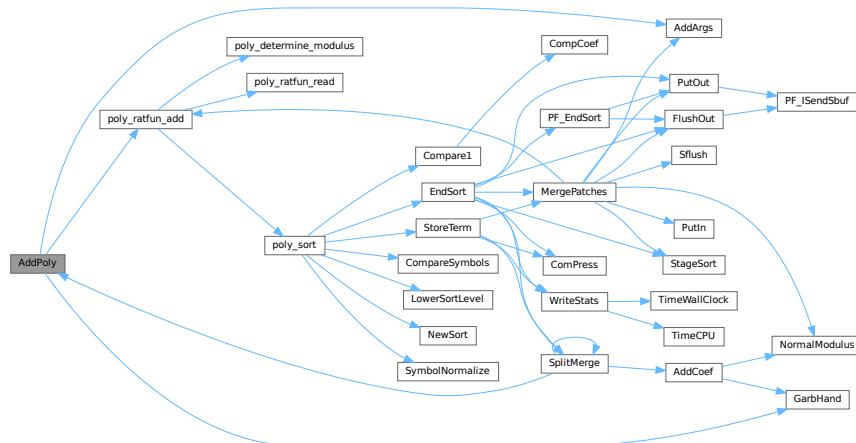
If zero the terms cancel. Otherwise the new term is in *ps1.

Definition at line 2175 of file [sort.c](#).

References [AddArgs\(\)](#), [GarbHand\(\)](#), and [poly_ratfun_add\(\)](#).

Referenced by [SplitMerge\(\)](#).

Here is the call graph for this function:



4.129.3.11 AddArgs()

```
void AddArgs (
    PHEAD WORD * s1,
    WORD * s2,
    WORD * m )
```

Adds the arguments of two occurrences of the PolyFun.

Parameters

<i>s1</i>	Pointer to the first occurrence.
<i>s2</i>	Pointer to the second occurrence.
<i>m</i>	Pointer to where the answer should be.

Definition at line 2347 of file [sort.c](#).

Referenced by [AddPoly\(\)](#), and [MergePatches\(\)](#).

4.129.3.12 Compare1()

```
WORD Compare1 (
    PHEAD WORD * term1,
    WORD * term2,
    WORD level )
```

Compares two terms. The answer is: 0 equal (with exception of the coefficient if level == 0.) >0 term1 comes first. <0 term2 comes first. Some special precautions may be needed to keep the CompCoef routine from generating overflows, although this is very unlikely in subterms. This routine should not return an error condition.

Originally this routine was called Compare. With the treatment of special polynomials with terms that contain only symbols and the need for extreme speed for the polynomial routines we made a special compare routine and now we store the address of the current compare routine in AR.CompareRoutine and have a macro Compare which makes all existing code work properly and we can just replace the routine on a thread by thread basis (each thread has its own AR struct).

Parameters

<i>term1</i>	First input term
<i>term2</i>	Second input term
<i>level</i>	The sorting level (may influence on the result)

Returns

0 equal (with exception of the coefficient if level == 0.) >0 term1 comes first. <0 term2 comes first.

When there are floating point numbers active (float_ = FLOATFUN) the presence of one or more float_ functions is returned in AT.SortFloatMode: 0: no float_ 1: float_ in term1 only 2: float_ in term2 only 3: float_ in both terms

Definition at line [2640](#) of file [sort.c](#).

References [CompCoef\(\)](#).

Referenced by [InFunction\(\)](#), [poly_sort\(\)](#), and [StartVariables\(\)](#).

Here is the call graph for this function:



4.129.3.13 CompareSymbols()

```
WORD CompareSymbols (
    PHEAD WORD * term1,
    WORD * term2,
    WORD par )
```

Compares the terms, based on the value of AN.polysortflag. If $\text{term1} < \text{term2}$ the return value is -1 If $\text{term1} > \text{term2}$ the return value is 1 If $\text{term1} = \text{term2}$ the return value is 0 The coefficients may differ. The terms contain only a single subterm of type SYMBOL. If AN.polysortflag = 0 it is a 'regular' compare. If AN.polysortflag = 1 the sum of the powers is more important par is a dummy parameter to make the parameter field identical to that of Compare1 which is the regular compare routine in [sort.c](#)

Definition at line 3110 of file [sort.c](#).

Referenced by [InFunction\(\)](#), and [poly_sort\(\)](#).

4.129.3.14 CompareHSymbols()

```
WORD CompareHSymbols (
    PHEAD WORD * term1,
    WORD * term2,
    WORD par )
```

Compares terms that can have only SYMBOL and HAAKJE subterms. If $\text{term1} < \text{term2}$ the return value is -1 If $\text{term1} > \text{term2}$ the return value is 1 If $\text{term1} = \text{term2}$ the return value is 0 par is a dummy parameter to make the parameter field identical to that of Compare1 which is the regular compare routine in [sort.c](#)

Definition at line 3153 of file [sort.c](#).

4.129.3.15 ComPress()

```
LONG ComPress (
    WORD ** ss,
    LONG * n )
```

Gets a list of pointers to terms and compresses the terms. In n it collects the number of terms and the return value of the function is the space that is occupied.

We have to pay some special attention to the compression of terms with a PolyFun. This PolyFun should occur only straight before the coefficient, so we can use the same trick as for the coefficient to sabotage compression of this object (Replace in the history the function pointer by zero. This is safe, because terms that would be identical otherwise would have been added).

Parameters

<i>ss</i>	Array of pointers to terms to be compressed.
<i>n</i>	Number of pointers in ss.

Returns

Total number of words needed for the compressed result.

Definition at line 3206 of file [sort.c](#).

Referenced by [EndSort\(\)](#), and [StoreTerm\(\)](#).

4.129.3.16 SplitMerge()

```
LONG SplitMerge (
    PHEAD WORD ** Pointer,
    LONG number )
```

Algorithm by J.A.M.Vermaseren (31-7-1988)

Note that AN.SplitScratch and AN.InScratch are used also in GarbHand

Merge sort in memory. The input is an array of pointers. Sorting is done recursively by dividing the array in two equal parts and calling SplitMerge for each. When the parts are small enough we can do the compare and take the appropriate action. An addition is that we look for 'runs'. Sequences that are already ordered. This happens a lot when there is very little action in a module. This made FORM faster by a few percent.

Parameters

<i>Pointer</i>	The array of pointers to the terms to be sorted.
<i>number</i>	The number of pointers in Pointer.

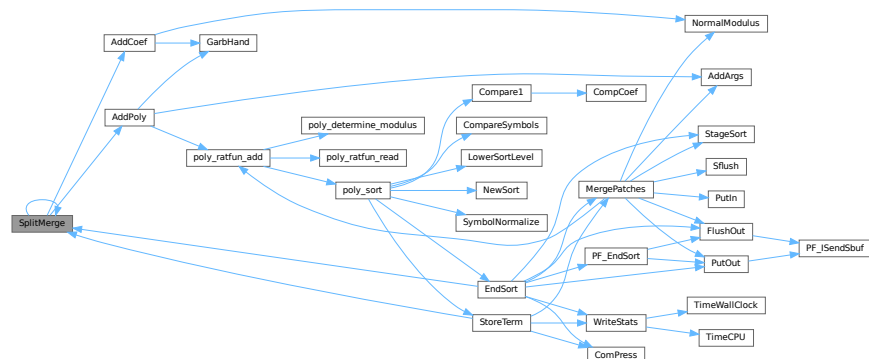
The terms are supposed to be sitting in the small buffer and there is supposed to be an extension to this buffer for when there are two terms that should be added and the result takes more space than each of the original terms. The notation guarantees that the result never needs more space than the sum of the spaces of the original terms.

Definition at line 3372 of file [sort.c](#).

References [AddCoef\(\)](#), [AddPoly\(\)](#), and [SplitMerge\(\)](#).

Referenced by [EndSort\(\)](#), [SplitMerge\(\)](#), and [StoreTerm\(\)](#).

Here is the call graph for this function:



4.129.3.17 GarbHand()

```
void GarbHand (
    void )
```

Garbage collection that takes place when the small extension is full and we need to place more terms there. When this is the case there are many holes in the small buffer and the whole can be compactified. The major complication is the buffer for SplitMerge. There are two options for temporary memory: 1: find some buffer that has enough space (maybe in the large buffer). 2: allocate a buffer. Give it back afterwards of course. If the small extension is properly dimensioned this routine should be called very rarely. Most of the time it will be called when the polyfun or polyratfun is active.

Definition at line 3652 of file [sort.c](#).

Referenced by [AddCoef\(\)](#), and [AddPoly\(\)](#).

4.129.3.18 MergePatches()

```
int MergePatches (
    WORD par )
```

The general merge routine. Can be used for the large buffer and the file merging. The array S->Patches tells where the patches start S->pStop tells where they end (has to be computed first). The end of a 'line to be merged' is indicated by a zero. If the end is reached without running into a zero or a term runs over the boundary of a patch it is a file merging operation and a new piece from the file is read in.

Parameters

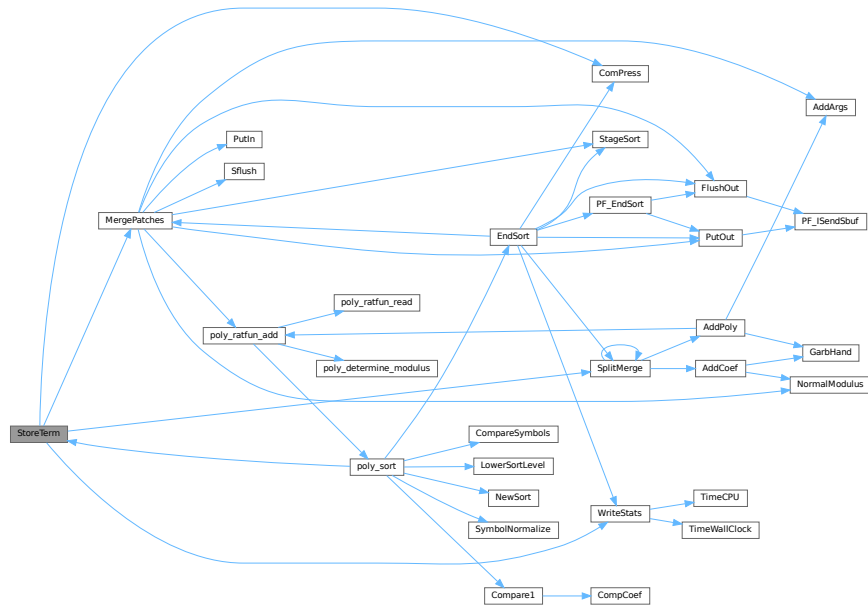
<i>par</i>	If <i>par</i> == 0 the sort is for file -> outputfile. If <i>par</i> == 1 the sort is for large buffer -> sortfile. If <i>par</i> == 2 the sort is for large buffer -> outputfile.

Definition at line 3767 of file [sort.c](#).

References [AddArgs\(\)](#), [FlushOut\(\)](#), [FiLe::handle](#), [NormalModulus\(\)](#), [poly_ratfun_add\(\)](#), [PutIn\(\)](#), [PutOut\(\)](#), [Sflush\(\)](#), and [StageSort\(\)](#).

Referenced by [EndSort\(\)](#), and [StoreTerm\(\)](#).

Here is the call graph for this function:



4.129.3.20 StageSort()

```
void StageSort (
    FILEHANDLE * fout )
```

Prepares a stage 4 or higher sort. Stage 4 sorts occur when the sort file contains more patches than can be merged in one pass.

Definition at line 4664 of file [sort.c](#).

References [FiLe::handle](#).

Referenced by [EndSort\(\)](#), and [MergePatches\(\)](#).

4.129.3.21 SortWild()

```
int SortWild (
    WORD * w,
    WORD nw )
```

Sorts the wildcard entries in the parameter w. Double entries are removed. Full space taken is nw words. Routine serves for the reading of wildcards in the compiler. The entries come in the format: (type,4,number,0) in which the zero is reserved for the future replacement of 'number'.

Parameters

<i>w</i>	buffer with wildcard entries.
<i>nw</i>	number of wildcard entries.

Returns

Normal conventions (OK -> 0)

Definition at line [4763](#) of file [sort.c](#).

4.129.3.22 CleanUpSort()

```
void CleanUpSort (
    int num )
```

Partially or completely frees function sort buffers.

Definition at line [4856](#) of file [sort.c](#).

4.129.3.23 LowerSortLevel()

```
void LowerSortLevel (
    void )
```

Lowers the level in the sort system.

Definition at line [4956](#) of file [sort.c](#).

Referenced by [generate_expression\(\)](#), [get_expression\(\)](#), [InFunction\(\)](#), [PF_CollectModifiedDollars\(\)](#), [PF_Processor\(\)](#), [poly_factorize_expression\(\)](#), [poly_sort\(\)](#), [poly_unfactorize_expression\(\)](#), [PolyFunMul\(\)](#), [Processor\(\)](#), and [TestMatch\(\)](#).

4.129.3.24 PolyRatFunSpecial()

```
WORD * PolyRatFunSpecial (
    PHEAD WORD * t1,
    WORD * t2 )
```

Definition at line [4973](#) of file [sort.c](#).

4.129.3.25 SimpleSplitMergeRec()

```
void SimpleSplitMergeRec (
    WORD * array,
    WORD num,
    WORD * auxarray )
```

Definition at line [5075](#) of file [sort.c](#).

4.129.3.26 SimpleSplitMerge()

```
void SimpleSplitMerge (
    WORD * array,
    WORD num )
```

Definition at line [5103](#) of file [sort.c](#).

4.129.3.27 BinarySearch()

```
WORD BinarySearch (
    WORD * array,
    WORD num,
    WORD x )
```

Definition at line [5121](#) of file [sort.c](#).

4.129.4 Variable Documentation**4.129.4.1 toterms**

```
char* toterms[] = { " ", " >>", "-->" }
```

Definition at line [89](#) of file [sort.c](#).

4.129.4.2 humanTermsSuffix

```
const char humanTermsSuffix[HUMANSUFFLEN][4] = {"K ", "M ", "B ", "T "}
```

Definition at line [93](#) of file [sort.c](#).

4.129.4.3 humanBytesSuffix

```
const char humanBytesSuffix[HUMANSUFFLEN][4] = {"KiB", "MiB", "GiB", "TiB"}
```

Definition at line [94](#) of file [sort.c](#).

4.130 sort.c

[Go to the documentation of this file.](#)

```
00001
00017 /* # [ License : */
00018 /*
00019 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00020 * When using this file you are requested to refer to the publication
00021 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00022 * This is considered a matter of courtesy as the development was paid
00023 * for by FOM the Dutch physics granting agency and we would like to
00024 * be able to track its scientific use to convince FOM of its value
00025 * for the community.
00026 *
00027 * This file is part of FORM.
00028 *
00029 * FORM is free software: you can redistribute it and/or modify it under the
00030 * terms of the GNU General Public License as published by the Free Software
00031 * Foundation, either version 3 of the License, or (at your option) any later
00032 * version.
00033 *
00034 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00035 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00036 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00037 * details.
00038 *
00039 * You should have received a copy of the GNU General Public License along
00040 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00041 */
```

```

00042 /* #] License : */
00043 /*
00044     #[ Includes : sort.c
00045
00046     Sort routines according to new conventions (25-jun-1997).
00047     This is more object oriented.
00048     The active sort is indicated by AT.SS which should agree with
00049     AN.FunSorts[AR.sLevel];
00050
00051 #define GZIPDEBUG
00052 */
00053 #define NEWSPLITMERGE
00054 /* Comment to turn off Timsort in SplitMerge for debugging: */
00055 #define NEWSPLITMERGETIMSORT
00056 /* During SplitMerge, print pointer array state on entry. Very spammy, for debugging. */
00057 /* #define SPLITMERGEDEBUG */
00058 /* Debug printing for GarbHand */
00059 /* #define TESTGARB */
00060
00061 #include "form3.h"
00062
00063 #ifdef WITHPTHREADS
00064 UBYTE THRbuf[100];
00065 #endif
00066
00067 #ifdef WITHSTATS
00068 extern LONG numwrites;
00069 extern LONG numreads;
00070 extern LONG numseeks;
00071 extern LONG nummallocs;
00072 extern LONG numfrees;
00073 #endif
00074
00075 // #define COUNTCOMPARES
00076 #ifdef COUNTCOMPARES
00077     // This needs to be large enough for the number of threads.
00078     // It is hardcoded here, but 1024 should be enough.
00079     // Enabling this has a performance impact.
00080     LONG numcompares[1024];
00081 #endif
00082
00083 /*
00084     #] Includes :
00085     #[ SortUtilities :
00086         #[ WriteStats :                void WriteStats(lspace,par,checkLogType)
00087 */
00088
00089 char *totterms[] = { " ", " ", " »", "-->" };
00090
00091 #define HUMANSTRLEN 12
00092 #define HUMANSUFFLEN 4
00093 const char humanTermsSuffix[HUMANSUFFLEN][4] = {"K ", "M ", "B ", "T "};
00094 const char humanBytesSuffix[HUMANSUFFLEN][4] = {"KiB", "MiB", "GiB", "TiB"};
00095 void HumanString(char* string, float input, const char suffix[HUMANSUFFLEN][4]) {
00096     int ind = -1;
00097     while (ind < 0 || (input >= 1000.0f && ind < HUMANSUFFLEN) ) {
00098         input /= 1000.0f;
00099         ind++;
00100     }
00101     if ( input <= 0.5f ) {
00102         snprintf(string, HUMANSTRLEN,
00103             " ( <1 %s)", suffix[ind]);
00104     }
00105     else {
00106         snprintf(string, HUMANSTRLEN,
00107             " (%3.f %s)", input, suffix[ind]);
00108     }
00109 }
00110
00127 void WriteStats(POSITION *plspace, WORD par, WORD checkLogType)
00128 {
00129     GETIDENTITY
00130     LONG millitime, y = 0x7FFFFFFFL » 1;
00131     WORD timepart;
00132     SORTING *S;
00133     POSITION pp;
00134     int use_wtime;
00135     if ( AT.SS == AT.S0 && AC.StatsFlag ) {
00136 #ifdef WITHPTHREADS
00137         if ( AC.ThreadStats == 0 && identity > 0 ) return;
00138 #elif defined(WITHMPI)
00139         if ( AC.OldParallelStats ) return;
00140         if ( ! AC.ProcessStats && PF.me != MASTER ) return;
00141 #endif
00142         if ( Expressions == 0 ) return;
00143
00144         if ( par == STATSSPLITMERGE ) {

```

```

00145         if ( AC.ShortStatsMax == 0 ) return;
00146         AR.ShortSortCount++;
00147         if ( AR.ShortSortCount < AC.ShortStatsMax ) return;
00148     }
00149     AR.ShortSortCount = 0;
00150
00151     S = AT.SS;
00152
00153     char humanGenTermsText[HUMANSTRLEN] = "";
00154     char humanTermsLeftText[HUMANSTRLEN] = "";
00155     char humanBytesText[HUMANSTRLEN] = "";
00156     if ( AC.HumanStatsFlag ) {
00157         HumanString(humanGenTermsText, (float)(S->GenTerms), humanTermsSuffix);
00158         HumanString(humanTermsLeftText, (float)(S->TermsLeft), humanTermsSuffix);
00159         HumanString(humanBytesText, (float)(BASEPOSITION(*plspace)), humanBytesSuffix);
00160     }
00161
00162     MLOCK(ErrorMessageLock);
00163
00164     /* If the statistics should not go to the log file, temporarily hide the
00165      * LogHandle. This must be done within the ErrorMessageLock region to
00166      * avoid a data race between threads. */
00167     const WORD oldLogHandle = AC.LogHandle;
00168     if ( checkLogType && AM.LogType ) {
00169         AC.LogHandle = -1;
00170     }
00171
00172     if ( AC.ShortStats ) {}
00173     else {
00174 #ifdef WITHPTHREADS
00175         if ( identity > 0 ) {
00176             MesPrint("          Thread %d reporting",identity);
00177         }
00178         else {
00179             MesPrint("");
00180         }
00181 #elif defined(WITHMPI)
00182         if ( PF.me != MASTER ) {
00183             MesPrint("          Process %d reporting",PF.me);
00184         }
00185         else {
00186             MesPrint("");
00187         }
00188 #else
00189         MesPrint("");
00190 #endif
00191     }
00192     /*
00193      * We define WTimeStatsFlag as a flag to print the wall-clock time on
00194      * the *master*, not in workers. This can be confusing in thread
00195      * statistics when short statistics is used. Technically,
00196      * TimeWallClock() is not thread-safe in TFORM.
00197      */
00198     use_wtime = AC.WTimeStatsFlag;
00199 #if defined(WITHPTHREADS)
00200     if ( use_wtime && identity > 0 ) use_wtime = 0;
00201 #elif defined(WITHMPI)
00202     if ( use_wtime && PF.me != MASTER ) use_wtime = 0;
00203 #endif
00204     millitime = use_wtime ? TimeWallClock(1) * 10 : TimeCPU(1);
00205     timepart = (WORD)(millitime%1000);
00206     millitime /= 1000;
00207     timepart /= 10;
00208     if ( AC.ShortStats ) {
00209 #if defined(WITHPTHREADS) || defined(WITHMPI)
00210 #ifdef WITHPTHREADS
00211         if ( identity > 0 ) {
00212 #else
00213         if ( PF.me != MASTER ) {
00214             const int identity = PF.me;
00215 #endif
00216         if ( par == STATSSPLITMERGE || par == STATSPOSTSORT ) {
00217             SETBASEPOSITION(pp,y);
00218             if ( ISLESSPOS(*plspace,pp) ) {
00219                 MesPrint("%d: %7l.%2is %8l>%10l%3s%10l:%10p %s %s",identity,
00220                     millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
00221                     S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00222             /*
00223              * MesPrint("%d: %14s %7l.%2is %8l>%10l%3s%10l:%10p",identity,
00224              * EXPRNAME(AR.CurExpr),AC.Commercial,millitime,timepart,
00225              * AN.ninterms,S->GenTerms,toterms[par],S->TermsLeft,plspace);
00226              */
00227             }
00228             else {
00229                 y = 1000000000L;
00230                 SETBASEPOSITION(pp,y);
00231                 MULPOS(pp,100);

```

```

00232         if ( ISLESSPOS(*plspace,pp) ) {
00233             MesPrint("%d: %7l.%2is %8l>%10l%3s%10l:%11p %s %s",identity,
00234                 millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
00235                 S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00236         }
00237         else {
00238             MULPOS(pp,10);
00239             if ( ISLESSPOS(*plspace,pp) ) {
00240                 MesPrint("%d: %7l.%2is %8l>%10l%3s%10l:%12p %s %s",identity,
00241                     millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
00242                     S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00243             }
00244             else {
00245                 MULPOS(pp,10);
00246                 if ( ISLESSPOS(*plspace,pp) ) {
00247                     MesPrint("%d: %7l.%2is %8l>%10l%3s%10l:%13p %s %s",identity,
00248                         millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
00249                         S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00250                 }
00251                 else {
00252                     MULPOS(pp,10);
00253                     if ( ISLESSPOS(*plspace,pp) ) {
00254                         MesPrint("%d: %7l.%2is %8l>%10l%3s%10l:%14p %s %s",identity,
00255                             millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
00256                             S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00257                     }
00258                     else {
00259                         MULPOS(pp,10);
00260                         if ( ISLESSPOS(*plspace,pp) ) {
00261                             MesPrint("%d: %7l.%2is %8l>%10l%3s%10l:%15p %s %s",identity,
00262                                 millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
00263                                 S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00264                         }
00265                         else {
00266                             MULPOS(pp,10);
00267                             if ( ISLESSPOS(*plspace,pp) ) {
00268                                 MesPrint("%d: %7l.%2is %8l>%10l%3s%10l:%16p %s %s",identity,
00269                                     millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
00270                                     S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00271                             }
00272                             else {
00273                                 MULPOS(pp,10);
00274                                 if ( ISLESSPOS(*plspace,pp) ) {
00275                                     MesPrint("%d: %7l.%2is %8l>%10l%3s%10l:%17p %s %s",identity,
00276                                         millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
00277                                         S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00278                                 }
00279                                 } } } } }
00280         }
00281     }
00282 }
00283 else if ( par == STATSMERGETOFILE ) {
00284     SETBASEPOSITION(pp,y);
00285     if ( ISLESSPOS(*plspace,pp) ) {
00286         MesPrint("%d: %7l.%2is %10l:%10p",identity,millitime,timepart,
00287             S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00288     }
00289     else {
00290         y = 1000000000L;
00291         SETBASEPOSITION(pp,y);
00292         MULPOS(pp,100);
00293         if ( ISLESSPOS(*plspace,pp) ) {
00294             MesPrint("%d: %7l.%2is %10l:%11p",identity,millitime,timepart,
00295                 S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00296         }
00297         else {
00298             MULPOS(pp,10);
00299             if ( ISLESSPOS(*plspace,pp) ) {
00300                 MesPrint("%d: %7l.%2is %10l:%12p",identity,millitime,timepart,
00301                     S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00302             }
00303             else {
00304                 MULPOS(pp,10);
00305                 if ( ISLESSPOS(*plspace,pp) ) {
00306                     MesPrint("%d: %7l.%2is %10l:%13p",identity,millitime,timepart,
00307                         S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00308                 }
00309                 else {
00310                     MULPOS(pp,10);
00311                     if ( ISLESSPOS(*plspace,pp) ) {
00312                         MesPrint("%d: %7l.%2is %10l:%14p",identity,millitime,timepart,
00313                             S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00314                     }
00315                     else {
00316                         MULPOS(pp,10);
00317                         if ( ISLESSPOS(*plspace,pp) ) {
00318                             MesPrint("%d: %7l.%2is %10l:%15p",identity,millitime,timepart,

```



```

00319             S->TermsLeft,plspace,EXPRNAME (AR.CurExpr),AC.Commercial);
00320         }
00321         else {
00322             MULPOS (pp,10);
00323             if ( ISLESSPOS(*plspace,pp) ) {
00324                 MesPrint ("%d: %7l.%2is %10l:%16p",identity,millitime,timepart,
00325                     S->TermsLeft,plspace,EXPRNAME (AR.CurExpr),AC.Commercial);
00326             }
00327             else {
00328                 MULPOS (pp,10);
00329                 if ( ISLESSPOS(*plspace,pp) ) {
00330                     MesPrint ("%d: %7l.%2is %10l:%17p",identity,millitime,timepart,
00331                         S->TermsLeft,plspace,EXPRNAME (AR.CurExpr),AC.Commercial);
00332                 }
00333             } } } }
00334     }
00335 }
00336 } } else
00337 #endif
00338 {
00339     if ( par == STATSSPLITMERGE || par == STATSPOSTSORT ) {
00340         SETBASEPOSITION (pp,y);
00341         if ( ISLESSPOS(*plspace,pp) ) {
00342             MesPrint ("%7l.%2is %8l>%10l%3s%10l:%10p %s %s",
00343                 millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
00344                 S->TermsLeft,plspace,EXPRNAME (AR.CurExpr),AC.Commercial);
00345             /*
00346             MesPrint ("%14s %17s %7l.%2is %8l>%10l%3s%10l:%10p",
00347                 EXPRNAME (AR.CurExpr),AC.Commercial,millitime,timepart,
00348                 AN.ninterms,S->GenTerms,toterms[par],S->TermsLeft,plspace);
00349             */
00350         }
00351         else {
00352             y = 1000000000L;
00353             SETBASEPOSITION (pp,y);
00354             MULPOS (pp,100);
00355             if ( ISLESSPOS(*plspace,pp) ) {
00356                 MesPrint ("%7l.%2is %8l>%10l%3s%10l:%11p %s %s",
00357                     millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
00358                     S->TermsLeft,plspace,EXPRNAME (AR.CurExpr),AC.Commercial);
00359             }
00360             else {
00361                 MULPOS (pp,10);
00362                 if ( ISLESSPOS(*plspace,pp) ) {
00363                     MesPrint ("%7l.%2is %8l>%10l%3s%10l:%12p %s %s",
00364                         millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
00365                         S->TermsLeft,plspace,EXPRNAME (AR.CurExpr),AC.Commercial);
00366                 }
00367                 else {
00368                     MULPOS (pp,10);
00369                     if ( ISLESSPOS(*plspace,pp) ) {
00370                         MesPrint ("%7l.%2is %8l>%10l%3s%10l:%13p %s %s",
00371                             millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
00372                             S->TermsLeft,plspace,EXPRNAME (AR.CurExpr),AC.Commercial);
00373                     }
00374                     else {
00375                         MULPOS (pp,10);
00376                         if ( ISLESSPOS(*plspace,pp) ) {
00377                             MesPrint ("%7l.%2is %8l>%10l%3s%10l:%14p %s %s",
00378                                 millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
00379                                 S->TermsLeft,plspace,EXPRNAME (AR.CurExpr),AC.Commercial);
00380                         }
00381                         else {
00382                             MULPOS (pp,10);
00383                             if ( ISLESSPOS(*plspace,pp) ) {
00384                                 MesPrint ("%7l.%2is %8l>%10l%3s%10l:%15p %s %s",
00385                                     millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
00386                                     S->TermsLeft,plspace,EXPRNAME (AR.CurExpr),AC.Commercial);
00387                             }
00388                             else {
00389                                 MULPOS (pp,10);
00390                                 if ( ISLESSPOS(*plspace,pp) ) {
00391                                     MesPrint ("%7l.%2is %8l>%10l%3s%10l:%16p %s %s",
00392                                         millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
00393                                         S->TermsLeft,plspace,EXPRNAME (AR.CurExpr),AC.Commercial);
00394                                 }
00395                                 else {
00396                                     MULPOS (pp,10);
00397                                     if ( ISLESSPOS(*plspace,pp) ) {
00398                                         MesPrint ("%7l.%2is %8l>%10l%3s%10l:%17p %s %s",
00399                                             millitime,timepart,AN.ninterms,S->GenTerms,toterms[par],
00400                                             S->TermsLeft,plspace,EXPRNAME (AR.CurExpr),AC.Commercial);
00401                                     }
00402                                 } } } }
00403                             }
00404                         }
00405                     }

```

```

00406     else if ( par == STATSMERGETOFILE ) {
00407         SETBASEPOSITION(pp,y);
00408         if ( ISLESSPOS(*plspace,pp) ) {
00409             MesPrint("%7l.%2is %10l:%10p",millitime,timepart,
00410                 S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00411         }
00412         else {
00413             y = 1000000000L;
00414             SETBASEPOSITION(pp,y);
00415             MULPOS(pp,100);
00416             if ( ISLESSPOS(*plspace,pp) ) {
00417                 MesPrint("%7l.%2is %10l:%11p",millitime,timepart,
00418                     S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00419             }
00420             else {
00421                 MULPOS(pp,10);
00422                 if ( ISLESSPOS(*plspace,pp) ) {
00423                     MesPrint("%7l.%2is %10l:%12p",millitime,timepart,
00424                         S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00425                 }
00426                 else {
00427                     MULPOS(pp,10);
00428                     if ( ISLESSPOS(*plspace,pp) ) {
00429                         MesPrint("%7l.%2is %10l:%13p",millitime,timepart,
00430                             S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00431                     }
00432                     else {
00433                         MULPOS(pp,10);
00434                         if ( ISLESSPOS(*plspace,pp) ) {
00435                             MesPrint("%7l.%2is %10l:%14p",millitime,timepart,
00436                                 S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00437                         }
00438                         else {
00439                             MULPOS(pp,10);
00440                             if ( ISLESSPOS(*plspace,pp) ) {
00441                                 MesPrint("%7l.%2is %10l:%15p",millitime,timepart,
00442                                     S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00443                             }
00444                             else {
00445                                 MULPOS(pp,10);
00446                                 if ( ISLESSPOS(*plspace,pp) ) {
00447                                     MesPrint("%7l.%2is %10l:%16p",millitime,timepart,
00448                                         S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00449                                 }
00450                                 else {
00451                                     MULPOS(pp,10);
00452                                     if ( ISLESSPOS(*plspace,pp) ) {
00453                                         MesPrint("%7l.%2is %10l:%17p",millitime,timepart,
00454                                             S->TermsLeft,plspace,EXPRNAME(AR.CurExpr),AC.Commercial);
00455                                     }
00456                                     } } } } }
00457             }
00458         }
00459     }
00460 }
00461 }
00462 else {
00463     if ( par == STATSMERGETOFILE ) {
00464         if ( use_wtime ) {
00465             MesPrint("WTime = %7l.%2i sec",millitime,timepart);
00466         }
00467         else {
00468             MesPrint("Time = %7l.%2i sec",millitime,timepart);
00469         }
00470     }
00471     else {
00472         #if ( BITSINLONG > 32 )
00473             if ( S->GenTerms >= 1000000000L ) {
00474                 if ( use_wtime ) {
00475                     MesPrint("WTime = %7l.%2i sec    Generated terms = %16l%s",
00476                         millitime,timepart,S->GenTerms,humanGenTermsText);
00477                 }
00478                 else {
00479                     MesPrint("Time = %7l.%2i sec    Generated terms = %16l%s",
00480                         millitime,timepart,S->GenTerms,humanGenTermsText);
00481                 }
00482             }
00483             else {
00484                 if ( use_wtime ) {
00485                     MesPrint("WTime = %7l.%2i sec    Generated terms = %10l%s",
00486                         millitime,timepart,S->GenTerms,humanGenTermsText);
00487                 }
00488                 else {
00489                     MesPrint("Time = %7l.%2i sec    Generated terms = %10l%s",
00490                         millitime,timepart,S->GenTerms,humanGenTermsText);
00491                 }
00492             }
00493         }
00494     }
00495 }
00496 #else

```

```

00493         if ( use_wtime ) {
00494             MesPrint("WTime = %7l.%2i sec    Generated terms = %10l%s",
00495                     millitime,timepart,S->GenTerms,humanGenTermsText);
00496         }
00497         else {
00498             MesPrint("Time = %7l.%2i sec    Generated terms = %10l%s",
00499                     millitime,timepart,S->GenTerms,humanGenTermsText);
00500         }
00501     #endif
00502 }
00503 #if ( BITSINLONG > 32 )
00504     if ( par == STATSSPLITMERGE )
00505         if ( S->TermsLeft >= 10000000000L ) {
00506             MesPrint("%16s%8l Terms %s = %16l%s",EXPRNAME(AR.CurExpr),
00507                     AN.ninterms,FG.swmes[par],S->TermsLeft,humanTermsLeftText);
00508         }
00509         else {
00510             MesPrint("%16s%8l Terms %s = %10l%s",EXPRNAME(AR.CurExpr),
00511                     AN.ninterms,FG.swmes[par],S->TermsLeft,humanTermsLeftText);
00512         }
00513     else {
00514         if ( S->TermsLeft >= 10000000000L ) {
00515             #ifdef WITHPTHREADS
00516                 if ( identity > 0 && par == STATSPOSTSORT ) {
00517                     MesPrint("%16s          Terms in thread = %16l%s",
00518                             EXPRNAME(AR.CurExpr),S->TermsLeft,humanTermsLeftText);
00519                 }
00520                 else
00521                     #elif defined(WITHMPI)
00522                     if ( PF.me != MASTER && par == STATSPOSTSORT ) {
00523                         MesPrint("%16s          Terms in process= %16l%s",
00524                                 EXPRNAME(AR.CurExpr),S->TermsLeft,humanTermsLeftText);
00525                     }
00526                     else
00527                         #endif
00528                         {
00529                             MesPrint("%16s          Terms %s = %16l%s",
00530                                     EXPRNAME(AR.CurExpr),FG.swmes[par],S->TermsLeft,humanTermsLeftText);
00531                         }
00532                 }
00533                 else {
00534                     #ifdef WITHPTHREADS
00535                         if ( identity > 0 && par == STATSPOSTSORT ) {
00536                             MesPrint("%16s          Terms in thread = %10l%s",
00537                                     EXPRNAME(AR.CurExpr),S->TermsLeft,humanTermsLeftText);
00538                         }
00539                         else
00540                             #elif defined(WITHMPI)
00541                             if ( PF.me != MASTER && par == STATSPOSTSORT ) {
00542                                 MesPrint("%16s          Terms in process= %10l%s",
00543                                         EXPRNAME(AR.CurExpr),S->TermsLeft,humanTermsLeftText);
00544                             }
00545                             else
00546                                 #endif
00547                                 {
00548                                     MesPrint("%16s          Terms %s = %10l%s",
00549                                             EXPRNAME(AR.CurExpr),FG.swmes[par],S->TermsLeft,humanTermsLeftText);
00550                                 }
00551                         }
00552                 }
00553             #else
00554                 if ( par == STATSSPLITMERGE )
00555                     MesPrint("%16s%8l Terms %s = %10l%s",EXPRNAME(AR.CurExpr),
00556                             AN.ninterms,FG.swmes[par],S->TermsLeft,humanTermsLeftText);
00557                 else {
00558                     #ifdef WITHPTHREADS
00559                         if ( identity > 0 && par == STATSPOSTSORT ) {
00560                             MesPrint("%16s          Terms in thread = %10l%s",
00561                                     EXPRNAME(AR.CurExpr),S->TermsLeft,humanTermsLeftText);
00562                         }
00563                         else
00564                             #elif defined(WITHMPI)
00565                             if ( PF.me != MASTER && par == STATSPOSTSORT ) {
00566                                 MesPrint("%16s          Terms in process= %10l%s",
00567                                         EXPRNAME(AR.CurExpr),S->TermsLeft,humanTermsLeftText);
00568                             }
00569                             else
00570                                 #endif
00571                                 {
00572                                     MesPrint("%16s          Terms %s = %10l%s",
00573                                             EXPRNAME(AR.CurExpr),FG.swmes[par],S->TermsLeft,humanTermsLeftText);
00574                                 }
00575                         }
00576                     #endif
00577                     SETBASEPOSITION(pp,y);
00578                     if ( ISLESSPOS(*plspace,pp) ) {
00579                         MesPrint("%24s Bytes used          = %10p%s",AC.Commercial,plspace,humanBytesText);

```

```

00580     }
00581     else {
00582         y = 1000000000L;
00583         SETBASEPOSITION(pp,y);
00584         MULPOS(pp,100);
00585         if ( ISLESSPOS(*plspace,pp) ) {
00586             MesPrint("%24s Bytes used      =%11p%s",AC.Commercial,plspace,humanBytesText);
00587         }
00588         else {
00589             MULPOS(pp,10);
00590             if ( ISLESSPOS(*plspace,pp) ) {
00591                 MesPrint("%24s Bytes used      =%12p%s",AC.Commercial,plspace,humanBytesText);
00592             }
00593             else {
00594                 MULPOS(pp,10);
00595                 if ( ISLESSPOS(*plspace,pp) ) {
00596                     MesPrint("%24s Bytes used      =%13p%s",AC.Commercial,plspace,humanBytesText);
00597                 }
00598                 else {
00599                     MULPOS(pp,10);
00600                     if ( ISLESSPOS(*plspace,pp) ) {
00601                         MesPrint("%24s Bytes used      =%14p%s",AC.Commercial,plspace,humanBytesText);
00602                     }
00603                     else {
00604                         MULPOS(pp,10);
00605                         if ( ISLESSPOS(*plspace,pp) ) {
00606                             MesPrint("%24s Bytes used      =%15p%s",AC.Commercial,plspace,humanBytesText);
00607                         }
00608                         else {
00609                             MULPOS(pp,10);
00610                             if ( ISLESSPOS(*plspace,pp) ) {
00611                                 MesPrint("%24s Bytes used      =%16p%s",AC.Commercial,plspace,humanBytesText);
00612                             }
00613                             else {
00614                                 MULPOS(pp,10);
00615                                 if ( ISLESSPOS(*plspace,pp) ) {
00616                                     MesPrint("%24s Bytes used      =%17p%s",AC.Commercial,plspace,humanBytesText);
00617                                 }
00618                                 } } } } }
00619         }
00620     } }
00621 #ifdef WITHSTATS
00622     MesPrint("Total number of writes: %l, reads: %l, seeks, %l"
00623             ,numwrites,numreads,numseeks);
00624     MesPrint("Total number of mallocs: %l, frees: %l"
00625             ,nummallocs,numfrees);
00626 #endif
00627     /* Put back the original LogHandle, it was changed if the statistics were
00628      * not printed in the log file. */
00629     AC.LogHandle = oldLogHandle;
00630
00631     MUNLOCK(ErrorMessageLock);
00632 }
00633 }
00634
00635 /*
00636     #] WriteStats :
00637     #[ NewSort :          WORD NewSort()
00638 */
00649 int NewSort(PHEAD0)
00650 {
00651     GETBIDENTITY
00652     SORTING *S, **newFS;
00653     int i, newsize;
00654     if ( AN.SoScratC == 0 )
00655         AN.SoScratC = (UWORD *)Malloc1(2*(AM.MaxTal+2)*sizeof(UWORD),"NewSort");
00656     AR.sLevel++;
00657     if ( AR.sLevel >= AN.NumFunSorts ) {
00658         if ( AN.NumFunSorts == 0 ) newsize = 100;
00659         else newsize = 2*AN.NumFunSorts;
00660         newFS = (SORTING **)Malloc1((newsize+1)*sizeof(SORTING *),"FunSort pointers");
00661         for ( i = 0; i < AN.NumFunSorts; i++ ) newFS[i] = AN.FunSorts[i];
00662         for ( ; i <= newsize; i++ ) newFS[i] = 0;
00663         if ( AN.FunSorts ) M_free(AN.FunSorts,"FunSort pointers");
00664         AN.FunSorts = newFS; AN.NumFunSorts = newsize;
00665     }
00666     if ( AR.sLevel == 0 ) {
00667
00668 #ifdef COUNTCOMPARES
00669 #ifdef WITHPTHREADS
00670         numcompares[AT.identity] = 0;
00671 #else
00672         numcompares[0] = 0;
00673 #endif
00674 #endif
00675
00676         AN.FunSorts[0] = AT.S0;

```

```

00677         if ( AR.PolyFun == 0 ) { AT.S0->PolyFlag = 0; }
00678     else if ( AR.PolyFunType == 1 ) { AT.S0->PolyFlag = 1; }
00679     else if ( AR.PolyFunType == 2 ) {
00680         if ( AR.PolyFunExp == 2
00681             || AR.PolyFunExp == 3 ) AT.S0->PolyFlag = 1;
00682         else AT.S0->PolyFlag = 2;
00683     }
00684     AR.ShortSortCount = 0;
00685 }
00686 else {
00687     if ( AN.FunSorts[AR.sLevel] == 0 ) {
00688         AN.FunSorts[AR.sLevel] = AllocSort(
00689             AM.SLargeSize,AM.SSmallSize,AM.SSmallEsize,AM.STermsInSmall
00690             ,AM.SMaxPatches,AM.SMaxFpatches,AM.SIOsize,1);
00691     }
00692     AN.FunSorts[AR.sLevel]->PolyFlag = 0;
00693 }
00694 AT.SS = S = AN.FunSorts[AR.sLevel];
00695 S->sFill = S->sBuffer;
00696 S->lFill = S->lBuffer;
00697 S->lPatch = 0;
00698 S->fPatchN = 0;
00699 S->GenTerms = S->TermsLeft = S->GenSpace = S->SpaceLeft = 0;
00700 S->PoinFill = S->sPointer;
00701 *S->PoinFill = S->sFill;
00702 if ( AR.sLevel > 0 ) { S->PolyWise = 0; }
00703 PUTZERO(S->SizeInFile[0]); PUTZERO(S->SizeInFile[1]); PUTZERO(S->SizeInFile[2]);
00704 S->sTerms = 0;
00705 PUTZERO(S->file.POposition);
00706 S->stage4 = 0;
00707 if ( AR.sLevel > AN.MaxFunSorts ) AN.MaxFunSorts = AR.sLevel;
00708 /*
00709     The next variable is for the staged sort only.
00710     It should be treated differently
00711
00712     PUTZERO(AN.OldPosOut);
00713 */
00714 return(0);
00715 }
00716
00717 /*
00718     #] NewSort :
00719     #[ EndSort :                WORD EndSort(PHEAD buffer,par)
00720 */
00745 LONG EndSort(PHEAD WORD *buffer, int par)
00746 {
00747     GETBIDENTITY
00748     SORTING *S = AT.SS;
00749     WORD j, **ss, *to, *t;
00750     LONG sSpace, over, tover, spare, retval = 0;
00751     POSITION position, pp;
00752     off_t lSpace;
00753     FILEHANDLE *fout = 0, *oldoutfile = 0, *newout = 0;
00754
00755     if ( AM.exitflag && AR.sLevel == 0 ) return(0);
00756 #ifdef WITHMPI
00757     if( (retval = PF_EndSort()) > 0){
00758         oldoutfile = AR.outfile;
00759         retval = 0;
00760         goto RetRetval;
00761     }
00762     else if(retval < 0){
00763         retval = -1;
00764         goto RetRetval;
00765     }
00766     /* PF_EndSort returned 0: for S != AM.S0 and slaves still do the regular sort */
00767 #endif /* WITHMPI */
00768     oldoutfile = AR.outfile;
00769     /* PolyFlag repair action
00770     if ( S == AT.S0 ) {
00771         if ( AR.PolyFun == 0 ) { S->PolyFlag = 0; }
00772         else if ( AR.PolyFunType == 1 ) { S->PolyFlag = 1; }
00773         else if ( AR.PolyFunType == 2 ) {
00774             if ( AR.PolyFunExp == 2
00775                 || AR.PolyFunExp == 3 ) S->PolyFlag = 1;
00776             else S->PolyFlag = 2;
00777         }
00778         S->PolyWise = 0;
00779     }
00780     else {
00781         S->PolyFlag = S->PolyWise = 0;
00782     }
00783 */
00784     S->PolyWise = 0;
00785     *(S->PoinFill) = 0;
00786 #ifdef SPLITTIME
00787     PrintTime((UBYTE *)"EndSort, before SplitMerge");

```

```

00788 #endif
00789     S->sPointer[SplitMerge(BHEAD S->sPointer,S->sTerms)] = 0;
00790 #ifdef SPLITTIME
00791     PrintTime((UBYTE *)"Endsort,  after SplitMerge");
00792 #endif
00793     sSpace = 0;
00794     tover = over = S->sTerms;
00795     ss = S->sPointer;
00796     if ( over >= 0 ) {
00797         if ( S->lPatch > 0 || S->file.handle >= 0 ) {
00798             ss[over] = 0;
00799             sSpace = ComPress(ss,&spare);
00800             S->TermsLeft -= over - spare;
00801             if ( par == 1 ) { AR.outfile = newout = AllocFileHandle(0,(char *)0); }
00802         }
00803         else if ( S != AT.S0 ) {
00804             ss[over] = 0;
00805             if ( par == 2 ) {
00806                 sSpace = 3;
00807                 while ( ( t = *ss++ ) != 0 ) { sSpace += *t; }
00808                 if ( AN.tryterm > 0 && ( (sSpace+1)*sizeof(WORD) < (size_t)(AM.MaxTer) ) ) {
00809                     to = TermMalloc("$-sort space");
00810                 }
00811                 else {
00812                     LONG allocsp = sSpace+1;
00813                     if ( allocsp < MINALLOC ) allocsp = MINALLOC;
00814                     allocsp = ((allocsp+7)/8)*8;
00815                     to = (WORD *)Malloc1(allocsp*sizeof(WORD),"$-sort space");
00816                     if ( AN.tryterm > 0 ) AN.tryterm = 0;
00817                 }
00818                 *((WORD **)buffer) = to;
00819                 ss = S->sPointer;
00820                 while ( ( t = *ss++ ) != 0 ) {
00821                     j = *t; while ( --j >= 0 ) *to++ = *t++;
00822                 }
00823                 *to = 0;
00824                 retval = sSpace + 1;
00825             }
00826             else {
00827                 to = buffer;
00828                 sSpace = 0;
00829                 while ( ( t = *ss++ ) != 0 ) {
00830                     j = *t;
00831                     if ( ( sSpace += j ) > AM.MaxTer/((LONG)sizeof(WORD)) ) {
00832                         /* Too big! Get the total size for useful error message */
00833                         while ( ( t = *ss++ ) != 0 ) {
00834                             sSpace += *t;
00835                         }
00836                         MLOCK(ErrorMessageLock);
00837                         MesPrint("Sorted function argument too long (%d words). Increase MaxTermSize
(%d words).", sSpace, AM.MaxTer/((LONG)sizeof(WORD)));
00838                         MUNLOCK(ErrorMessageLock);
00839                         retval = -1; goto RetRetval;
00840                     }
00841                     while ( --j >= 0 ) *to++ = *t++;
00842                 }
00843                 *to = 0;
00844                 retval = to - buffer;
00845             }
00846             goto RetRetval;
00847         }
00848         else {
00849             POSITION oldpos;
00850             if ( S == AT.S0 ) {
00851                 fout = AR.outfile;
00852                 *AR.CompressPointer = 0;
00853                 SeekScratch(AR.outfile,&position);
00854             }
00855             else {
00856                 fout = &(S->file);
00857                 PUTZERO(position);
00858             }
00859             oldpos = position;
00860             S->TermsLeft = 0;
00861             /*
00862             Here we can go directly to the output.
00863             */
00864             #ifdef WITHZLIB
00865             { int oldgzipCompress = AR.gzipCompress;
00866               AR.gzipCompress = 0;
00867             }
00868             #endif
00869             if ( tover > 0 ) {
00870                 ss = S->sPointer;
00871                 while ( ( t = *ss++ ) != 0 ) {
00872                     if ( *t ) S->TermsLeft++;
00873                 }
00874             }
00875             #ifdef WITHPTHREADS
00876             if ( AS.MasterSort && ( fout == AR.outfile ) ) { PutToMaster(BHEAD t); }
00877         }
00878     }

```

```

00874             else
00875 #endif
00876             if ( PutOut(BHEAD t,&position,fout,1) < 0 ) {
00877                 retval = -1; goto RetRetval;
00878             }
00879         }
00880     }
00881 #ifdef WITHPTHREADS
00882     if ( AS.MasterSort && ( fout == AR.outfile ) ) { PutToMaster(BHEAD 0); }
00883     else
00884 #endif
00885     if ( FlushOut(&position,fout,1) ) {
00886         retval = -1; goto RetRetval;
00887     }
00888 #ifdef WITHZLIB
00889     AR.gzipCompress = oldgzipCompress;
00890 }
00891 #endif
00892 #ifdef WITHPTHREADS
00893     if ( AS.MasterSort && ( fout == AR.outfile ) ) goto RetRetval;
00894 #endif
00895 #ifdef WITHMPI
00896     if ( PF.me != MASTER && PF.exprtodo < 0 ) goto RetRetval;
00897 #endif
00898     DIFPOS(oldpos,position,oldpos);
00899     S->SpaceLeft = BASEPOSITION(oldpos);
00900     WriteStats(&oldpos,STATSPOSTSORT,NOCHECKLOGTYPE);
00901     pp = oldpos;
00902     goto RetRetval;
00903 }
00904 }
00905 else if ( par == 1 && newout == 0 ) { AR.outfile = newout = AllocFileHandle(0,(char *)0); }
00906 sSpace++;
00907 lSpace = sSpace + (S->lFill - S->lBuffer) - (LONG)S->lPatch*(AM.MaxTer/sizeof(WORD));
00908 /* Note wrt MaxTer and lPatch: each patch starts with space for decompression */
00909 /* Not needed if only large buffer, but needed when using files (?) */
00910 SETBASEPOSITION(pp,lSpace);
00911 MULPOS(pp,sizeof(WORD));
00912 if ( S->file.handle >= 0 ) {
00913     ADD2POS(pp,S->fPatches[S->fPatchN]);
00914 }
00915 if ( S == AT.S0 ) {
00916     if ( S->lPatch > 0 || S->file.handle >= 0 ) {
00917         WriteStats(&pp,STATSSPLITMERGE,CHECKLOGTYPE);
00918     }
00919 }
00920 if ( par == 2 ) { AR.outfile = newout = AllocFileHandle(0,(char *)0); }
00921 if ( S->lPatch > 0 ) {
00922     if ( ( S->lPatch >= S->MaxPatches ) ||
00923         ( ( WORD * )(((UBYTE *) (S->lFill + sSpace)) + 2*AM.MaxTer) ) >= S->lTop ) ) {
00924 /*
00925         The large buffer is too full. Merge and write it
00926 */
00927 #ifdef GZIPDEBUG
00928         MLOCK(ErrorMessageLock);
00929         MesPrint("%w EndSort: lPatch = %d, MaxPatches = %d,lFill = %x, sSpace = %ld, MaxTer = %d,
00930             lTop = %x"
00931             ,S->lPatch,S->MaxPatches,S->lFill,sSpace,AM.MaxTer/sizeof(WORD),S->lTop);
00932         MUNLOCK(ErrorMessageLock);
00933 #endif
00934         if ( MergePatches(1) ) {
00935             MLOCK(ErrorMessageLock);
00936             MesCall("EndSort");
00937             MUNLOCK(ErrorMessageLock);
00938             retval = -1; goto RetRetval;
00939         }
00940         S->lPatch = 0;
00941         pp = S->SizeInFile[1];
00942         MULPOS(pp,sizeof(WORD));
00943 #ifndef WITHPTHREADS
00944         if ( S == AT.S0 )
00945 #endif
00946         {
00947             POSITION pppp;
00948             SETBASEPOSITION(pppp,0);
00949             SeekFile(S->file.handle,&pppp,SEEK_CUR);
00950             SeekFile(S->file.handle,&pp,SEEK_END);
00951             SeekFile(S->file.handle,&pppp,SEEK_SET);
00952             WriteStats(&pp,STATSMERGETOFILE,CHECKLOGTYPE);
00953             UpdateMaxSize();
00954         }
00955     }
00956     else {
00957         S->Patches[S->lPatch++] = S->lFill;
00958         to = (WORD * )(((UBYTE *) (S->lFill)) + AM.MaxTer);
00959         if ( tover > 0 ) {

```

```

00960         ss = S->sPointer;
00961         while ( ( t = *ss++ ) != 0 ) {
00962             j = *t;
00963             if ( j < 0 ) j = t[1] + 2;
00964             while ( --j >= 0 ) *to++ = *t++;
00965         }
00966     }
00967     *to++ = 0;
00968     S->lFill = to;
00969     if ( S->file.handle < 0 ) {
00970         if ( MergePatches(2) ) {
00971             MLOCK(ErrorMessageLock);
00972             MesCall("EndSort");
00973             MUNLOCK(ErrorMessageLock);
00974             retval = -1; goto RetRetval;
00975         }
00976         if ( S == AT.S0 ) {
00977             pp = S->SizeInFile[2];
00978             MULPOS(pp, sizeof(WORD));
00979 #ifdef WITHPTHREADS
00980             if ( AS.MasterSort && ( fout == AR.outfile ) ) goto RetRetval;
00981 #endif
00982             WriteStats(&pp, STATSPOSTSORT, NOCHECKLOGTYPE);
00983             UpdateMaxSize();
00984         }
00985         else {
00986             if ( par == 2 && newout->handle >= 0 ) {
00987                 POSITION zeropos;
00988                 PUTZERO(zeropos);
00989 #ifdef ALLLOCK
00990                 LOCK(newout->pthreadlock);
00991 #endif
00992                 SeekFile(newout->handle, &zeropos, SEEK_SET);
00993                 to = (WORD *)Malloc1(BASEPOSITION(newout->filesize)+sizeof(WORD)*2
00994                                     , "$-buffer reading");
00995                 if ( AN.tryterm > 0 ) AN.tryterm = 0;
00996                 if ( ( retval = ReadFile(newout->handle, (UBYTE
00997 *)to, BASEPOSITION(newout->filesize)) ) !=
00997                     BASEPOSITION(newout->filesize) ) {
00998                     MLOCK(ErrorMessageLock);
00999                     MesPrint("Error reading information for $ variable");
01000                     MUNLOCK(ErrorMessageLock);
01001                     M_free(to, "$-buffer reading");
01002                     retval = -1;
01003                 }
01004                 else {
01005                     *((WORD **)buffer) = to;
01006                     retval /= sizeof(WORD);
01007                 }
01008 #ifdef ALLLOCK
01009                 UNLOCK(newout->pthreadlock);
01010 #endif
01011             }
01012             else if ( newout->handle >= 0 ) {
01013 /*
01014         We land here if par == 1 (function arg sort) and PutOut has created a file.
01015         This means that the term is larger than SIOsize, which we ensure is at least
01016         as large as MaxTermSize in setfile.c. Thus we know already that the term won't
01017         fit.
01018 */
01019         TooLarge:
01020             MLOCK(ErrorMessageLock);
01021             MesPrint("(1) Output should fit inside a single term. Increase MaxTermSize?");
01022             MesCall("EndSort");
01023             MUNLOCK(ErrorMessageLock);
01024             retval = -1; goto RetRetval;
01025         }
01026         else {
01027             t = newout->PObuffer;
01028             // We deal with the par == 2 case after RetRetval.
01029             if ( par != 2 ) {
01030                 j = newout->POfill - t;
01031                 to = buffer;
01032                 if ( to >= AT.WorkSpace && to < AT.WorkTop && to+j > AT.WorkTop )
01033                     goto WorkspaceError;
01034                 if ( j > AM.MaxTer ) {
01035                     MLOCK(ErrorMessageLock);
01036                     MesPrint("Encountered term of size: %d words.", j/(LONG)sizeof(WORD)
01037 );
01038                     MUNLOCK(ErrorMessageLock);
01039                     goto TooLarge;
01040                 }
01041                 NCOPY(to, t, j);
01042                 retval = to - buffer - 1;
01043             }
01044         }
01045     }
01046 }

```



```

01044         goto RetRetVal;
01045     }
01046     if ( MergePatches(1) ) { /* --> SortFile */
01047         MLOCK(ErrorMessageLock);
01048         MesCall("EndSort");
01049         MUNLOCK(ErrorMessageLock);
01050         retval = -1; goto RetRetVal;
01051     }
01052     UpdateMaxSize();
01053     pp = S->SizeInFile[1];
01054     MULPOS(pp, sizeof(WORD));
01055 #ifndef WITHPTHREADS
01056     if ( S == AT.S0 )
01057 #endif
01058     {
01059         POSITION pppp;
01060         SETBASEPOSITION(pppp, 0);
01061         SeekFile(S->file.handle, &pppp, SEEK_CUR);
01062         SeekFile(S->file.handle, &pp, SEEK_END);
01063         SeekFile(S->file.handle, &pppp, SEEK_SET);
01064         WriteStats(&pp, STATSMERGETOFILE, CHECKLOGTYPE);
01065     }
01066 #ifdef WITHERRORXXX
01067     if ( S != AT.S0 ) {
01068         /*
01069          * This is wrong! We have sorted to the sort file.
01070          * Things are not sitting in the output yet.
01071          */
01072         if ( newout->handle >= 0 ) goto TooLarge;
01073         t = newout->PObuffer;
01074         j = newout->POfill - t;
01075         to = buffer;
01076         if ( to >= AT.WorkSpace && to < AT.WorkTop && to+j > AT.WorkTop )
01077             goto WorkspaceError;
01078         if ( j > AM.MaxTer ) goto TooLarge;
01079         NCOPY(to, t, j);
01080         goto RetRetVal;
01081     }
01082 #endif
01083     }
01084     if ( S->file.handle >= 0 ) {
01085 #ifdef GZIPDEBUG
01086         MLOCK(ErrorMessageLock);
01087         MesPrint("%w EndSort: fPatchN = %d, lPatch = %d, position = %12p"
01088             , S->fPatchN, S->lPatch, &(S->fPatches[S->fPatchN]));
01089         MUNLOCK(ErrorMessageLock);
01090 #endif
01091         if ( S->lPatch <= 0 ) {
01092             StageSort(&(S->file));
01093             position = S->fPatches[S->fPatchN];
01094             ss = S->sPointer;
01095             if ( *ss ) {
01096                 *AR.CompressPointer = 0;
01097 #ifdef WITHZLIB
01098                 if ( S == AT.S0 && AR.NoCompress == 0 && AR.gzipCompress > 0 )
01099                     S->fpcompressed[S->fPatchN] = 1;
01100                 else
01101                     S->fpcompressed[S->fPatchN] = 0;
01102                 SetupOutputGZIP(&(S->file));
01103 #endif
01104             }
01105             while ( ( t = *ss++ ) != 0 ) {
01106                 if ( PutOut(BHEAD t, &position, &(S->file), 1) < 0 ) {
01107                     retval = -1; goto RetRetVal;
01108                 }
01109             }
01110             if ( FlushOut(&position, &(S->file), 1) ) {
01111                 retval = -1; goto RetRetVal;
01112             }
01113             ++(S->fPatchN);
01114             S->fPatches[S->fPatchN] = position;
01115             UpdateMaxSize();
01116 #ifdef GZIPDEBUG
01117             MLOCK(ErrorMessageLock);
01118             MesPrint("%w EndSort+: fPatchN = %d, lPatch = %d, position = %12p"
01119                 , S->fPatchN, S->lPatch, &(S->fPatches[S->fPatchN]));
01120             MUNLOCK(ErrorMessageLock);
01121 #endif
01122         }
01123     }
01124     AR.Stage4Name = 0;
01125 #ifdef WITHPTHREADS
01126     if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
01127         if ( S->file.handle >= 0 ) {
01128             SynchFile(S->file.handle);
01129         }
01130     }

```

```

01131 #endif
01132     UpdateMaxSize();
01133     if ( MergePatches(0) ) {
01134         MLOCK(ErrorMessageLock);
01135         MesCall("EndSort");
01136         MUNLOCK(ErrorMessageLock);
01137         retval = -1; goto RetRetval;
01138     }
01139     S->stage4 = 0;
01140 #ifdef WITHPTHREADS
01141     if ( AS.MasterSort && ( fout == AR.outfile ) ) goto RetRetval;
01142 #endif
01143     pp = S->SizeInFile[0];
01144     MULPOS(pp, sizeof(WORD));
01145     WriteStats(&pp, STATSPOSTSORT, NOCHECKLOGTYPE);
01146     UpdateMaxSize();
01147 }
01148 RetRetval:
01149
01150 #ifdef WITHMPI
01151 /* NOTE: PF_EndSort has been changed such that it sets S->TermsLeft. (TU 30 Jun 2011) */
01152 if ( AR.sLevel == 0 && (PF.me == MASTER || PF.exprtodo >= 0) ) {
01153     Expressions[AR.CurExpr].counter = S->TermsLeft;
01154     Expressions[AR.CurExpr].size = pp;
01155 }
01156 #else
01157 if ( AR.sLevel == 0 ) {
01158     Expressions[AR.CurExpr].counter = S->TermsLeft;
01159     Expressions[AR.CurExpr].size = pp;
01160 } /*if ( AR.sLevel == 0 )*/
01161 #endif
01162 /*:[25nov2003 mt]*/
01163 if ( S->file.handle >= 0 && ( par != 1 ) && ( par != 2 ) ) {
01164     /* sortfile is still open */
01165     UpdateMaxSize();
01166 #ifdef WITHZLIB
01167     ClearSortGZIP(&(S->file));
01168 #endif
01169     CloseFile(S->file.handle);
01170     S->file.handle = -1;
01171     remove(S->file.name);
01172 #ifdef GZIPDEBUG
01173     MLOCK(ErrorMessageLock);
01174     MesPrint("%wEndSort: sortfile %s removed", S->file.name);
01175     MUNLOCK(ErrorMessageLock);
01176 #endif
01177 }
01178 AR.outfile = oldoutfile;
01179 AR.sLevel--;
01180 if ( AR.sLevel >= 0 ) AT.SS = AN.FunSorts[AR.sLevel];
01181 if ( par == 1 ) {
01182     if ( retval < 0 ) {
01183         UpdateMaxSize();
01184         if ( newout ) {
01185             DeAllocFileHandle(newout);
01186             newout = 0;
01187         }
01188     }
01189     else if ( newout ) {
01190         if ( newout->handle >= 0 ) {
01191             MLOCK(ErrorMessageLock);
01192             MesPrint("(2)Output should fit inside a single term. Increase MaxTermSize?");
01193             MesCall("EndSort");
01194             MUNLOCK(ErrorMessageLock);
01195             Terminate(-1);
01196         }
01197         else if ( newout->POfill > newout->PObuffer ) {
01198 /*
01199             Here we have to copy the contents of the 'file' into
01200             the buffer. We assume that this buffer lies in the Workspace.
01201             Hence
01202 */
01203             j = newout->POfill - newout->PObuffer;
01204             if ( buffer >= AT.WorkSpace && buffer < AT.WorkTop && buffer+j > AT.WorkTop )
01205                 goto WorkspaceError;
01206             else {
01207                 to = buffer; t = newout->PObuffer;
01208                 while ( j-- > 0 ) *to++ = *t++;
01209             }
01210             UpdateMaxSize();
01211         }
01212         DeAllocFileHandle(newout);
01213         newout = 0;
01214     }
01215 }
01216 else if ( par == 2 ) {
01217     if ( newout ) {

```

```

01218         if ( retval == 0 ) {
01219             if ( newout->handle >= 0 ) {
01220                 /*
01221                     output resides at the moment in a file
01222                     Find the size, make a buffer, copy into the buffer and clean up.
01223                 */
01224                 POSITION zeropos;
01225                 PUTZERO(position);
01226 #ifdef ALLLOCK
01227                 LOCK(newout->pthreadslck);
01228 #endif
01229                 SeekFile(newout->handle,&position,SEEK_END);
01230                 PUTZERO(zeropos);
01231                 SeekFile(newout->handle,&zeropos,SEEK_SET);
01232                 to = (WORD *)Malloc1(BASEPOSITION(position)+sizeof(WORD)*3
01233                     ,"$-buffer reading");
01234                 if ( AN.tryterm > 0 ) AN.tryterm = 0;
01235                 if ( ( retval = ReadFile(newout->handle, (UBYTE *)to,BASEPOSITION(position)) ) !=
01236                     BASEPOSITION(position) ) {
01237                     MLOCK(ErrorMessageLock);
01238                     MesPrint("Error reading information for $ variable");
01239                     MUNLOCK(ErrorMessageLock);
01240                     M_free(to,"$-buffer reading");
01241                     retval = -1;
01242                 }
01243                 else {
01244                     *((WORD **)buffer) = to;
01245                     retval /= sizeof(WORD);
01246                 }
01247 #ifdef ALLLOCK
01248                 UNLOCK(newout->pthreadslck);
01249 #endif
01250             }
01251             else {
01252                 /*
01253                     output resides in the cache buffer and the file was never opened
01254                 */
01255                 LONG wsiz = newout->POfill - newout->PObuffer;
01256                 if ( AN.tryterm > 0 && ( (wsiz+2)*sizeof(WORD) < (size_t)(AM.MaxTer) ) ) {
01257                     to = TermMalloc("$-sort space");
01258                 }
01259                 else {
01260                     LONG allocsp = wsiz+2;
01261                     if ( allocsp < MINALLOC ) allocsp = MINALLOC;
01262                     allocsp = ((allocsp+7)/8)*8;
01263                     to = (WORD *)Malloc1(allocsp*sizeof(WORD),"$-buffer reading");
01264                     if ( AN.tryterm > 0 ) AN.tryterm = 0;
01265                 }
01266                 *((WORD **)buffer) = to; t = newout->PObuffer;
01267                 retval = wsiz;
01268                 NCOPY(to,t,wsiz);
01269             }
01270         }
01271         UpdateMaxSize();
01272         DeAllocFileHandle(newout);
01273         newout = 0;
01274     }
01275 }
01276 else {
01277     if ( newout ) {
01278         DeAllocFileHandle(newout);
01279         newout = 0;
01280     }
01281 }
01282
01283 #ifdef COUNTCOMPARES
01284     if ( AR.sLevel < 0 ) {
01285 #ifdef WITHPTHREADS
01286         MLOCK(ErrorMessageLock);
01287         MesPrint(">>number of calls to Compare: %l (tid %d)", numcompares[AT.identity], AT.identity);
01288         MUNLOCK(ErrorMessageLock);
01289 #else
01290         MesPrint(">>number of calls to Compare: %l", numcompares[0]);
01291 #endif
01292     }
01293 #endif
01294
01295     return(retval);
01296 WorkspaceError:
01297     MLOCK(ErrorMessageLock);
01298     MesWork();
01299     MesCall("EndSort");
01300     MUNLOCK(ErrorMessageLock);
01301     Terminate(-1);
01302     return(-1);
01303 }
01304

```

```

01305 /*
01306      #] EndSort :
01307      #[ PutIn :          LONG PutIn(handle,position,buffer,take,npat)
01308 */
01324 LONG PutIn(FILEHANDLE *file, POSITION *position, WORD *buffer, WORD **take, int npat)
01325 {
01326     LONG i, RetCode;
01327     WORD *from, *to;
01328     #ifndef WITHZLIB
01329         DUMMYUSE(npat);
01330     #endif
01331     from = buffer + ( file->PSize * sizeof(UBYTE) )/sizeof(WORD);
01332     i = from - *take;
01333     if ( i*((LONG)(sizeof(WORD))) > AM.MaxTer ) {
01334         MLOCK(ErrorMessageLock);
01335         MesPrint("Problems in PutIn");
01336         MUNLOCK(ErrorMessageLock);
01337         Terminate(-1);
01338     }
01339     to = buffer;
01340     while ( --i >= 0 ) ***to = ***from;
01341     *take = to;
01342     #ifndef WITHZLIB
01343     if ( ( RetCode = FillInputGZIP(file,position,(UBYTE *)buffer
01344                                     ,file->PSize,npat) ) < 0 ) {
01345         MLOCK(ErrorMessageLock);
01346         MesPrint("PutIn: We have RetCode = %x while reading %x bytes",
01347                 RetCode,file->PSize);
01348         MUNLOCK(ErrorMessageLock);
01349         Terminate(-1);
01350     }
01351     #else
01352     #ifdef ALLLOCK
01353         LOCK(file->pthreadlock);
01354     #endif
01355     SeekFile(file->handle,position,SEEK_SET);
01356     if ( ( RetCode = ReadFile(file->handle,(UBYTE *)buffer,file->PSize) ) < 0 ) {
01357     #ifdef ALLLOCK
01358         UNLOCK(file->pthreadlock);
01359     #endif
01360         MLOCK(ErrorMessageLock);
01361         MesPrint("PutIn: We have RetCode = %x while reading %x bytes",
01362                 RetCode,file->PSize);
01363         MUNLOCK(ErrorMessageLock);
01364         Terminate(-1);
01365     }
01366     #ifdef ALLLOCK
01367         UNLOCK(file->pthreadlock);
01368     #endif
01369     #endif
01370     return(RetCode);
01371 }
01372
01373 /*
01374      #] PutIn :
01375      #[ Sflush :          WORD Sflush(file)
01376 */
01384 int Sflush(FILEHANDLE *fi)
01385 {
01386     LONG size, RetCode;
01387     #ifndef WITHZLIB
01388         GETIDENTITY
01389         int dobracketindex = 0;
01390         if ( AR.sLevel <= 0 && Expressions[AR.CurExpr].newbracketinfo
01391             && ( fi == AR.outfile || fi == AR.hidefile ) ) dobracketindex = 1;
01392     #endif
01393     if ( fi->handle < 0 ) {
01394         if ( ( RetCode = CreateFile(fi->name) ) >= 0 ) {
01395     #ifdef GZIPDEBUG
01396         MLOCK(ErrorMessageLock);
01397         MesPrint("%w Sflush created scratch file %s",fi->name);
01398         MUNLOCK(ErrorMessageLock);
01399     #endif
01400         fi->handle = (WORD)RetCode;
01401         PUTZERO(fi->filesize);
01402         PUTZERO(fi->PPosition);
01403     }
01404     else {
01405         MLOCK(ErrorMessageLock);
01406         MesPrint("Cannot create scratch file %s",fi->name);
01407         MUNLOCK(ErrorMessageLock);
01408         return(-1);
01409     }
01410 }
01411 #ifndef WITHZLIB
01412 if ( AT.SS == AT.S0 && !AR.NoCompress && AR.gzipCompress > 0
01413     && dobracketindex == 0 ) {

```

```

01414         if ( FlushOutputGZIP(fi) ) return(-1);
01415         fi->POfill = fi->PObuffer;
01416     }
01417     else
01418     #endif
01419     {
01420     #ifdef ALLLOCK
01421         LOCK(fi->pthreadlock);
01422     #endif
01423         size = (fi->POfill-fi->PObuffer)*sizeof(WORD);
01424         SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
01425         if ( WriteFile(fi->handle,(UBYTE *) (fi->PObuffer),size) != size ) {
01426     #ifdef ALLLOCK
01427         UNLOCK(fi->pthreadlock);
01428     #endif
01429         MLOCK(ErrorMessageLock);
01430         MesPrint("Write error while finishing sort. Disk full?");
01431         MUNLOCK(ErrorMessageLock);
01432         return(-1);
01433     }
01434     ADDPOS(fi->filesize,size);
01435     ADDPOS(fi->POposition,size);
01436     fi->POfill = fi->PObuffer;
01437     #ifdef ALLLOCK
01438     UNLOCK(fi->pthreadlock);
01439     #endif
01440     }
01441     return(0);
01442 }
01443
01444 /*
01445     #] Sflush :
01446     #[ PutOut :                WORD PutOut(term,position,file,ncomp)
01447 */
01470 WORD PutOut(PHEAD WORD *term, POSITION *position, FILEHANDLE *fi, WORD ncomp)
01471 {
01472     GETBIDENTITY
01473     WORD i, *p, ret, *r, *rr, j, k, first;
01474     int dobracketindex = 0;
01475     LONG RetCode;
01476
01477     if ( AT.SS != AT.SO ) {
01478 /*
01479         For this case no compression should be used
01480 */
01481         if ( ( i = *term ) <= 0 ) return(0);
01482         ret = i;
01483         ADDPOS(*position,i*sizeof(WORD));
01484         p = fi->POfill;
01485         do {
01486             if ( p >= fi->POstop ) {
01487                 if ( fi->handle < 0 ) {
01488                     if ( ( RetCode = CreateFile(fi->name) ) >= 0 ) {
01489     #ifdef GZIPDEBUG
01490                         MLOCK(ErrorMessageLock);
01491                         MesPrint("%w PutOut created sortfile %s",fi->name);
01492                         MUNLOCK(ErrorMessageLock);
01493                     #endif
01494                     fi->handle = (WORD)RetCode;
01495                     PUTZERO(fi->filesize);
01496                     PUTZERO(fi->POposition);
01497 /*
01498                         Should not be here anymore?
01499     #ifdef WITHZLIB
01500                         fi->ziobuffer = 0;
01501                     #endif
01502 */
01503                 }
01504             }
01505             else {
01506                 MLOCK(ErrorMessageLock);
01507                 MesPrint("Cannot create scratch file %s",fi->name);
01508                 MUNLOCK(ErrorMessageLock);
01509                 return(-1);
01510             }
01511     #ifdef ALLLOCK
01512             LOCK(fi->pthreadlock);
01513     #endif
01514             if ( fi == AR.hidefile ) {
01515                 LOCK(AS.inputslock);
01516             }
01517             SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
01518             if ( ( RetCode = WriteFile(fi->handle,(UBYTE *) (fi->PObuffer),fi->POsize) ) !=
fi->POsize ) {
01519                 if ( fi == AR.hidefile ) {
01520                     UNLOCK(AS.inputslock);
01521                 }

```

```

01522 #ifdef ALLLOCK
01523         UNLOCK(fi->pthreadlock);
01524 #endif
01525         MLOCK(ErrorMessageLock);
01526         MesPrint("Write error during sort. Disk full?");
01527         MesPrint("Attempt to write %l bytes on file %d at position %l5p",
01528                 fi->POsize, fi->handle, &(fi->POposition));
01529         MesPrint("RetCode = %l, Buffer address = %l", RetCode, (LONG) (fi->PObuffer));
01530         MUNLOCK(ErrorMessageLock);
01531         return(-1);
01532     }
01533     ADDPOS(fi->filesize, fi->POsize);
01534     p = fi->PObuffer;
01535     ADDPOS(fi->POposition, fi->POsize);
01536     if ( fi == AR.hidefile ) {
01537         UNLOCK(AS.inputslock);
01538     }
01539 #ifdef ALLLOCK
01540     UNLOCK(fi->pthreadlock);
01541 #endif
01542 #ifdef WITHPTHREADS
01543     if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
01544         if ( fi->handle >= 0 ) SynchFile(fi->handle);
01545     }
01546 #endif
01547     }
01548     *p++ = *term++;
01549     } while ( --i > 0 );
01550     fi->POfull = fi->POfill = p;
01551     return(ret);
01552 }
01553 if ( ( AP.PreDebug & DUMPOUTTERMS ) == DUMPOUTTERMS ) {
01554     MLOCK(ErrorMessageLock);
01555 #ifdef WITHPTHREADS
01556     snprintf((char *) (THRbuf), 100, "PutOut(%d)", AT.identity);
01557     PrintTerm(term, (char *) (THRbuf));
01558 #else
01559     PrintTerm(term, "PutOut");
01560 #endif
01561     MesPrint("ncomp = %d, AR.NoCompress = %d, AR.sLevel = %d", ncomp, AR.NoCompress, AR.sLevel);
01562     MesPrint("File %s, position %p", fi->name, position);
01563     MUNLOCK(ErrorMessageLock);
01564 }
01565
01566 if ( AR.sLevel <= 0 && Expressions[AR.CurExpr].newbracketinfo
01567     && ( fi == AR.outfile || fi == AR.hidefile ) ) dobracketindex = 1;
01568 r = rr = AR.CompressPointer;
01569 first = j = k = ret = 0;
01570 if ( ( i = *term ) != 0 ) {
01571     if ( i < 0 ) { /* Compressed term */
01572         i = term[1] + 2;
01573         if ( fi == AR.outfile || fi == AR.hidefile ) {
01574             MLOCK(ErrorMessageLock);
01575             MesPrint("Ran into precompressed term");
01576             MUNLOCK(ErrorMessageLock);
01577             Terminate(-1);
01578             return(-1);
01579         }
01580     }
01581     else if ( !AR.NoCompress && ( ncomp > 0 ) && AR.sLevel <= 0 ) { /* Must compress */
01582         if ( dobracketindex ) {
01583             PutBracketInIndex(BHEAD term, position);
01584         }
01585         j = *r++ - 1;
01586         p = term + 1;
01587         i--;
01588         if ( AR.PolyFun ) {
01589             WORD *polystop, *sa;
01590             sa = p + i;
01591             sa -= ABS(sa[-1]);
01592             polystop = p;
01593             while ( polystop < sa && *polystop != AR.PolyFun ) {
01594                 polystop += polystop[1];
01595             }
01596             if ( polystop < sa ) {
01597                 if ( AR.PolyFunType == 2 ) polystop[2] &= ~MUSTCLEANPRF;
01598                 while ( i > 0 && j > 0 && *p == *r && p < polystop ) {
01599                     i--; j--; k--; p++; r++;
01600                 }
01601             }
01602             else {
01603                 while ( i > 0 && j > 0 && *p == *r && p < sa ) { i--; j--; k--; p++; r++; }
01604             }
01605         }
01606 #ifdef WITHFLOAT
01607         else if ( AC.DefaultPrecision ) {
01608             WORD *floatstop, *sa;

```

```

01609         sa = p + i;
01610         sa -= ABS(sa[-1]);
01611         floatstop = p;
01612         while ( floatstop < sa && *floatstop != FLOATFUN ) {
01613             floatstop += floatstop[1];
01614         }
01615         if ( floatstop < sa ) {
01616             while ( i > 0 && j > 0 && *p == *r && p < floatstop ) {
01617                 i--; j--; k--; p++; r++;
01618             }
01619         }
01620         else {
01621             while ( i > 0 && j > 0 && *p == *r && p < sa ) { i--; j--; k--; p++; r++; }
01622         }
01623     }
01624 #endif
01625     else {
01626         WORD *sa;
01627         sa = p + i;
01628         sa -= ABS(sa[-1]);
01629         while ( i > 0 && j > 0 && *p == *r && p < sa ) { i--; j--; k--; p++; r++; }
01630     }
01631     if ( k > -2 ) {
01632 nocompress:
01633         j = i = *term;
01634         k = 0;
01635         p = term;
01636         r = rr;
01637         NCOPY(r,p,j);
01638     }
01639     else {
01640         *rr = *term;
01641         term = p;
01642         j = i;
01643         NCOPY(r,p,j);
01644         j = i;
01645         i += 2;
01646         first = 2;
01647     }
01648 /*             Sabotage getting into the coefficient next time */
01649     r[-(ABS(r[-1]))] = 0;
01650     if ( r >= AR.ComprTop ) {
01651         MLOCK(ErrorMessageLock);
01652         MesPrint("CompressSize of %10l is insufficient",AM.CompressSize);
01653         MUNLOCK(ErrorMessageLock);
01654         Terminate(-1);
01655         return(-1);
01656     }
01657 }
01658 else if ( !AR.NoCompress && ( ncomp < 0 ) && AR.sLevel <= 0 ) {
01659     /* No compress but put in compress buffer anyway */
01660     if ( dobracketindex ) {
01661         PutBracketInIndex(BHEAD term,position);
01662     }
01663     j = *r++ - 1;
01664     p = term + 1;
01665     i--;
01666     if ( AR.PolyFun ) {
01667         WORD *polystop, *sa;
01668         sa = p + i;
01669         sa -= ABS(sa[-1]);
01670         polystop = p;
01671         while ( polystop < sa && *polystop != AR.PolyFun ) {
01672             polystop += polystop[1];
01673         }
01674         if ( polystop < sa ) {
01675             if ( AR.PolyFunType == 2 ) polystop[2] &= ~MUSTCLEANPRF;
01676             while ( i > 0 && j > 0 && *p == *r && p < polystop ) {
01677                 i--; j--; k--; p++; r++;
01678             }
01679         }
01680         else {
01681             while ( i > 0 && j > 0 && *p == *r ) { i--; j--; k--; p++; r++; }
01682         }
01683     }
01684     else {
01685         while ( i > 0 && j > 0 && *p == *r ) { i--; j--; k--; p++; r++; }
01686     }
01687     goto nocompress;
01688 }
01689 else {
01690     if ( AR.PolyFunType == 2 ) {
01691         WORD *t, *tstop;
01692         tstop = term + *term;
01693         tstop -= ABS(tstop[-1]);
01694         t = term+1;
01695         while ( t < tstop ) {

```

```

01696             if ( *t == AR.PolyFun ) {
01697                 t[2] &= ~MUSTCLEANPRF;
01698             }
01699             t += t[1];
01700         }
01701     }
01702     if ( dobracketindex ) {
01703         PutBracketInIndex(BHEAD term,position);
01704     }
01705 }
01706 ret = i;
01707 ADDPOS(*position,i*sizeof(WORD));
01708 p = fi->POfill;
01709 do {
01710     if ( p >= fi->POstop ) {
01711 #ifdef WITHMPI /* [16mar1998 ar] */
01712         if ( PF.me != MASTER && AR.sLevel <= 0 && (fi == AR.outfile || fi == AR.hidefile) &&
PF.parallel && PF.exprtoto < 0 ) {
01713             PF_BUFFER *sbuf = PF.sbuf;
01714             sbuf->fill[sbuf->active] = fi->POstop;
01715             PF_ISendSbuf(MASTER,PF_BUFFER_MSGTAG);
01716             p = fi->PObuffer = fi->POfill = fi->POfull =
01717                 sbuf->buff[sbuf->active];
01718             fi->POstop = sbuf->stop[sbuf->active];
01719         }
01720     } else
01721 #endif /* WITHMPI [16mar1998 ar] */
01722     {
01723         if ( fi->handle < 0 ) {
01724             if ( ( RetCode = CreateFile(fi->name) ) >= 0 ) {
01725 #ifdef GZIPDEBUG
01726                 MLOCK(ErrorMessageLock);
01727                 MesPrint("%w PutOut created sortfile %s",fi->name);
01728                 MUNLOCK(ErrorMessageLock);
01729 #endif
01730                 fi->handle = (WORD)RetCode;
01731                 PUTZERO(fi->filesize);
01732                 PUTZERO(fi->POposition);
01733             } /*
01734             Should not be here?
01735             #ifdef WITHZLIB
01736                 fi->ziobuffer = 0;
01737             #endif
01738             */
01739         }
01740         else {
01741             MLOCK(ErrorMessageLock);
01742             MesPrint("Cannot create scratch file %s",fi->name);
01743             MUNLOCK(ErrorMessageLock);
01744             return(-1);
01745         }
01746     }
01747 #ifdef WITHZLIB
01748     if ( !AR.NoCompress && ncomp > 0 && AR.gzipCompress > 0
&& dobracketindex == 0 && fi->zsp != 0 ) {
01749         fi->POfill = p;
01750         if ( PutOutputGZIP(fi) ) return(-1);
01751         p = fi->PObuffer;
01752     }
01753 } else
01754 #endif
01755 {
01756     #ifdef ALLLOCK
01757         LOCK(fi->pthreadlock);
01758     #endif
01759     if ( fi == AR.hidefile ) {
01760         LOCK(AS.inputslock);
01761     }
01762     SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
01763     if ( ( RetCode = WriteFile(fi->handle,(UBYTE *) (fi->PObuffer),fi->POsize) ) !=
fi->POsize ) {
01764         if ( fi == AR.hidefile ) {
01765             UNLOCK(AS.inputslock);
01766         }
01767         #ifdef ALLLOCK
01768             UNLOCK(fi->pthreadlock);
01769         #endif
01770         MLOCK(ErrorMessageLock);
01771         MesPrint("Write error during sort. Disk full?");
01772         MesPrint("Attempt to write %l bytes on file %d at position %15p",
01773             fi->POsize,fi->handle,&(fi->POposition));
01774         MesPrint("RetCode = %l, Buffer address = %l",RetCode,(LONG) (fi->PObuffer));
01775         MUNLOCK(ErrorMessageLock);
01776         return(-1);
01777     }
01778 }
01779 ADDPOS(fi->filesize,fi->POsize);
01780 p = fi->PObuffer;

```



```

01781             ADDPOS(fi->POposition,fi->POsize);
01782             if ( fi == AR.hidefile ) {
01783                 UNLOCK(AS.inputslock);
01784             }
01785 #ifdef ALLLOCK
01786             UNLOCK(fi->pthreadlock);
01787 #endif
01788 #ifdef WITHPTHREADS
01789             if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
01790                 if ( fi->handle >= 0 ) SynchFile(fi->handle);
01791             }
01792 #endif
01793         }
01794     }
01795 }
01796 if ( first ) {
01797     if ( first == 2 ) *p++ = k;
01798     else *p++ = j;
01799     first--;
01800 }
01801 else *p++ = *term++;
01802 /*
01803     if ( AP.DebugFlag ) {
01804         TailToLine((UWORD) (p[-1])); TokenToLine((UBYTE *) " ");
01805     }
01806 */
01807 } while ( --i > 0 );
01808 fi->POfull = fi->POfill = p;
01809 }
01810 /*
01811     if ( AP.DebugFlag ) {
01812         AO.OutSkip = 0;
01813         FiniLine();
01814     }
01815 */
01816 return(ret);
01817 }
01818
01819 /*
01820     #] PutOut :
01821     #[ FlushOut :                WORD FlushOut(position,file,compr)
01822 */
01832 int FlushOut(POSITION *position, FILEHANDLE *fi, int compr)
01833 {
01834     GETIDENTITY
01835     LONG size, RetCode;
01836     int dobracketindex = 0;
01837 #ifndef WITHZLIB
01838     DUMMYUSE(compr);
01839 #endif
01840     if ( AR.sLevel <= 0 && Expressions[AR.CurExpr].newbracketinfo
01841         && ( fi == AR.outfile || fi == AR.hidefile ) ) dobracketindex = 1;
01842 #ifndef WITHMPI /* [16mar1998 ar] */
01843     if ( PF.me != MASTER && AR.sLevel <= 0 && (fi == AR.outfile || fi == AR.hidefile) && PF.parallel
01844         && PF.exp Prado < 0 ) {
01845         PF_BUFFER *sbuf = PF.sbuf;
01846         if ( fi->POfill >= fi->POstop ) {
01847             sbuf->fill[sbuf->active] = fi->POstop;
01848             PF_ISendSbuf(MASTER,PF_BUFFER_MSTAG);
01849             fi->POfull = fi->POfill = fi->PObuffer = sbuf->buff[sbuf->active];
01850             fi->POstop = sbuf->stop[sbuf->active];
01851         }
01852         *(fi->POfill)++ = 0;
01853         sbuf->fill[sbuf->active] = fi->POfill;
01854         PF_ISendSbuf(MASTER,PF_ENDBUFFER_MSTAG);
01855         fi->PObuffer = fi->POfill = fi->POfull = sbuf->buff[sbuf->active];
01856         fi->POstop = sbuf->stop[sbuf->active];
01857         return(0);
01858     }
01859 #endif /* WITHMPI [16mar1998 ar] */
01860     if ( fi->POfill >= fi->POstop ) {
01861         if ( fi->handle < 0 ) {
01862             if ( ( RetCode = CreateFile(fi->name) ) >= 0 ) {
01863                 #ifdef GZIPDEBUG
01864                 MLOCK(ErrorMessageLock);
01865                 MesPrint("%w FlushOut created scratch file %s",fi->name);
01866                 MUNLOCK(ErrorMessageLock);
01867             #endif
01868             PUTZERO(fi->filesize);
01869             PUTZERO(fi->POposition);
01870             fi->handle = (WORD)RetCode;
01871             /*
01872             Should not be here?
01873             */
01874             fi->ziobuffer = 0;
01875         }
01876     }

```

```

01876         }
01877         else {
01878             MLOCK(ErrorMessageLock);
01879             MesPrint("Cannot create scratch file %s",fi->name);
01880             MUNLOCK(ErrorMessageLock);
01881             return(-1);
01882         }
01883     }
01884 #ifdef WITHZLIB
01885     if ( AT.SS == AT.S0 && !AR.NoCompress && AR.gzipCompress > 0
01886         && dobracketindex == 0 && ( compr > 0 ) && fi->zsp != 0 ) {
01887         if ( PutOutputGZIP(fi) ) return(-1);
01888         fi->POfill = fi->PObuffer;
01889     }
01890     else
01891 #endif
01892     {
01893 #ifdef ALLLOCK
01894         LOCK(fi->pthreadlock);
01895 #endif
01896         if ( fi == AR.hidefile ) {
01897             LOCK(AS.inputslock);
01898             SeekFile(fi->handle,&(fi->POposition),SEEK_SET);
01899             if ( ( RetCode = WriteFile(fi->handle,(UBYTE *) (fi->PObuffer),fi->POsize) ) != fi->POsize )
01900             {
01901 #ifdef ALLLOCK
01902                 UNLOCK(fi->pthreadlock);
01903 #endif
01904                 if ( fi == AR.hidefile ) {
01905                     UNLOCK(AS.inputslock);
01906                 }
01907                 MLOCK(ErrorMessageLock);
01908                 MesPrint("Write error while sorting. Disk full?");
01909                 MesPrint("Attempt to write %l bytes on file %d at position %l5p",
01910                     fi->POsize,fi->handle,&(fi->POposition));
01911                 MesPrint("RetCode = %l, Buffer address = %l",RetCode,(LONG) (fi->PObuffer));
01912                 MUNLOCK(ErrorMessageLock);
01913                 return(-1);
01914             }
01915             ADDPOS(fi->filesize,fi->POsize);
01916             fi->POfill = fi->PObuffer;
01917             ADDPOS(fi->POposition,fi->POsize);
01918             if ( fi == AR.hidefile ) {
01919                 UNLOCK(AS.inputslock);
01920             }
01921 #ifdef ALLLOCK
01922             UNLOCK(fi->pthreadlock);
01923 #endif
01924 #ifdef WITHPTHREADS
01925             if ( AS.MasterSort && AC.ThreadSortFileSynch && fi != AR.hidefile ) {
01926                 if ( fi->handle >= 0 ) SynchFile(fi->handle);
01927             }
01928 #endif
01929         }
01930     }
01931     *(fi->POfill)++ = 0;
01932     fi->POfull = fi->POfill;
01933     /*
01934     {
01935         UBYTE OutBuf[140];
01936         if ( AP.DebugFlag ) {
01937             AO.OutFill = AO.OutputLine = OutBuf;
01938             AO.OutSkip = 3;
01939             FiniLine();
01940             TokenToLine((UBYTE *) "End of expression written");
01941             FiniLine();
01942         }
01943     }
01944     */
01945     size = (fi->POfill-fi->PObuffer)*sizeof(WORD);
01946     if ( fi->handle >= 0 ) {
01947 #ifdef WITHZLIB
01948         if ( AT.SS == AT.S0 && !AR.NoCompress && AR.gzipCompress > 0
01949             && dobracketindex == 0 && ( compr > 0 ) && fi->zsp != 0 ) {
01950             if ( FlushOutputGZIP(fi) ) return(-1);
01951             fi->POfill = fi->PObuffer;
01952         }
01953         else
01954 #endif
01955         {
01956 #ifdef ALLLOCK
01957             LOCK(fi->pthreadlock);
01958 #endif
01959             if ( fi == AR.hidefile ) {
01960                 LOCK(AS.inputslock);
01961             }

```

```

01962         SeekFile(fi->handle,&(fi->PPosition),SEEK_SET);
01963  /*
01964         MesPrint("FlushOut: writing %l bytes to position %l2p",size,&(fi->PPosition));
01965  */
01966         if ( ( RetCode = WriteFile(fi->handle,(UBYTE *) (fi->PObuffer),size) ) != size ) {
01967 #ifdef ALLLOCK
01968             UNLOCK(fi->pthreadlock);
01969 #endif
01970             if ( fi == AR.hidefile ) {
01971                 UNLOCK(AS.inputslock);
01972             }
01973             MLOCK(ErrorMessageLock);
01974             MesPrint("Write error while finishing sorting. Disk full?");
01975             MesPrint("Attempt to write %l bytes on file %d at position %l5p",
01976                 size,fi->handle,&(fi->PPosition));
01977             MesPrint("RetCode = %l, Buffer address = %l",RetCode,(LONG) (fi->PObuffer));
01978             MUNLOCK(ErrorMessageLock);
01979             return(-1);
01980         }
01981         ADDPOS(fi->filesize,size);
01982         ADDPOS(fi->PPosition,size);
01983         fi->POfill = fi->PObuffer;
01984         if ( fi == AR.hidefile ) {
01985             UNLOCK(AS.inputslock);
01986         }
01987 #ifdef ALLLOCK
01988         UNLOCK(fi->pthreadlock);
01989 #endif
01990 #ifdef WITHPTHREADS
01991         if ( AS.MasterSort && AC.ThreadSortFileSynch ) {
01992             if ( fi->handle >= 0 ) SynchFile(fi->handle);
01993         }
01994 #endif
01995     }
01996     if ( dobracketindex ) {
01997         BRACKETINFO *b = Expressions[AR.CurExpr].newbracketinfo;
01998         if ( b->indexfill > 0 ) {
02000             DIFPOS(b->indexbuffer[b->indexfill-1].next,*position,Expressions[AR.CurExpr].onfile);
02001         }
02002     }
02003 #ifdef WITHZLIB
02004     if ( AT.SS == AT.S0 && !AR.NoCompress && AR.gzipCompress > 0
02005         && dobracketindex == 0 && ( compr > 0 ) && fi->zsp != 0 ) {
02006         PUTZERO(*position);
02007         if ( fi->handle >= 0 ) {
02008 #ifdef ALLLOCK
02009             LOCK(fi->pthreadlock);
02010 #endif
02011             SeekFile(fi->handle,position,SEEK_END);
02012 #ifdef ALLLOCK
02013             UNLOCK(fi->pthreadlock);
02014 #endif
02015         }
02016         else {
02017             ADDPOS(*position,((UBYTE *) fi->POfill-(UBYTE *) fi->PObuffer));
02018         }
02019     }
02020     else
02021 #endif
02022     {
02023         ADDPOS(*position,sizeof(WORD));
02024     }
02025     return(0);
02026 }
02027
02028 /*
02029     #] FlushOut :
02030     #[ AddCoef :                WORD AddCoef(pterm1,pterm2)
02031 */
02046 int AddCoef(PHEAD WORD **ps1, WORD **ps2)
02047 {
02048     GETBIDENTITY
02049     SORTING *S = AT.SS;
02050     WORD *s1, *s2;
02051     WORD l1, l2, i;
02052     WORD OutLen, *t, j;
02053     UWORD *OutCoef;
02054 #ifdef WITHFLOAT
02055     if ( AT.SortFloatMode ) return(AddWithFloat(BHEAD ps1,ps2));
02056 #endif
02057     OutCoef = AN.SoScratC;
02058     s1 = *ps1; s2 = *ps2;
02059     GETCOEF(s1,l1);
02060     GETCOEF(s2,l2);
02061     if ( AddRat(BHEAD (UWORD *)s1,l1,(UWORD *)s2,l2,OutCoef,&OutLen) ) {
02062         MLOCK(ErrorMessageLock);

```

```

02063     MesCall("AddCoef");
02064     MUNLOCK(ErrorMessageLock);
02065     Terminate(-1);
02066 }
02067 if ( AN.ncmod != 0 ) {
02068     if ( ( AC.modmode & POSNEG ) != 0 ) {
02069         NormalModulus(OutCoef,&OutLen);
02070     /*
02071         We had forgotten that this can also become smaller but the
02072         denominator isn't there. Correct in the other case
02073         17-may-2009 [JV]
02074     */
02075         j = ABS(OutLen); OutCoef[j] = 1;
02076         for ( i = 1; i < j; i++ ) OutCoef[j+i] = 0;
02077     }
02078     else if ( BigLong(OutCoef,OutLen, (UWORD *)AC.cmod,ABS(AN.ncmod)) >= 0 ) {
02079         SubPLon(OutCoef,OutLen, (UWORD *)AC.cmod,ABS(AN.ncmod),OutCoef,&OutLen);
02080         OutCoef[OutLen] = 1;
02081         for ( i = 1; i < OutLen; i++ ) OutCoef[OutLen+i] = 0;
02082     }
02083 }
02084 if ( !OutLen ) { *ps1 = *ps2 = 0; return(0); }
02085 OutLen *= 2;
02086 if ( OutLen < 0 ) i = - ( --OutLen );
02087 else i = ++OutLen;
02088 if ( l1 < 0 ) l1 = -l1;
02089 l1 *= 2; l1++;
02090 if ( i <= l1 ) { /* Fits in 1 */
02091     l1 -= i;
02092     **ps1 -= l1;
02093     s2 = (WORD *)OutCoef;
02094     while ( --i > 0 ) *s1++ = *s2++;
02095     *s1++ = OutLen;
02096     while ( --l1 >= 0 ) *s1++ = 0;
02097     goto RegEnd;
02098 }
02099 if ( l2 < 0 ) l2 = -l2;
02100 l2 *= 2; l2++;
02101 if ( i <= l2 ) { /* Fits in 2 */
02102     l2 -= i;
02103     **ps2 -= l2;
02104     s1 = (WORD *)OutCoef;
02105     while ( --i > 0 ) *s2++ = *s1++;
02106     *s2++ = OutLen;
02107     while ( --l2 >= 0 ) *s2++ = 0;
02108     *ps1 = *ps2;
02109     goto RegEnd;
02110 }
02111
02112 /* Doesn't fit. Make a new term. */
02113
02114 t = s1;
02115 s1 = *ps1;
02116 j = *s1++ + i - l1; /* Space needed */
02117 if ( (S->sFill + j) >= S->sTop2 ) {
02118     GarbHand();
02119     s1 = *ps1;
02120     t = s1 + *s1 - 1;
02121     j = *s1++ + i - l1; /* Space needed */
02122     l1 = *t;
02123     if ( l1 < 0 ) l1 = - l1;
02124     t -= l1-1;
02125 }
02126 s2 = S->sFill;
02127 *s2++ = j;
02128 while ( s1 < t ) *s2++ = *s1++;
02129 s1 = (WORD *)OutCoef;
02130 while ( --i > 0 ) *s2++ = *s1++;
02131 *s2++ = OutLen;
02132 *ps1 = S->sFill;
02133 S->sFill = s2;
02134 RegEnd:
02135 *ps2 = 0;
02136 if ( **ps1 > AM.MaxTer/((LONG)(sizeof(WORD))) ) {
02137     MLOCK(ErrorMessageLock);
02138     MesPrint("Term too complex after addition in sort. MaxTermSize = %10l",
02139         AM.MaxTer/sizeof(WORD));
02139     MUNLOCK(ErrorMessageLock);
02140     Terminate(-1);
02141 }
02142
02143 return(1);
02144 }
02145
02146 /*
02147     #] AddCoef :
02148     #[ AddPoly : WORD AddPoly(pterm1,pterm2)
02149 */

```

```

02175 int AddPoly(PHEAD WORD **ps1, WORD **ps2)
02176 {
02177     GETBIDENTITY
02178     SORTING *S = AT.SS;
02179     WORD i;
02180     WORD *s1, *s2, *m, *w, *t, oldpw = S->PolyWise;
02181     s1 = *ps1 + S->PolyWise;
02182     s2 = *ps2 + S->PolyWise;
02183     w = AT.WorkPointer;
02184     /*
02185     Add here the two arguments. Is a straight merge.
02186     */
02187     if ( S->PolyFlag == 2 && AR.PolyFunExp != 2 && AR.PolyFunExp != 3 ) {
02188         WORD **oldSplitScratch = AN.SplitScratch;
02189         LONG oldSplitScratchSize = AN.SplitScratchSize;
02190         LONG oldInScratch = AN.InScratch;
02191         WORD oldtype = AR.SortType;
02192         if ( (WORD *) ((UBYTE *)w + AM.MaxTer) >= AT.WorkTop ) {
02193             MLOCK(ErrorMessageLock);
02194             MesPrint("Program was adding polyratfun arguments");
02195             MesWork();
02196             MUNLOCK(ErrorMessageLock);
02197         }
02198         AR.SortType = SORTHIGHFIRST;
02199         S->PolyWise = 0;
02200         AN.SplitScratch = AN.SplitScratch1;
02201         AN.SplitScratchSize = AN.SplitScratchSize1;
02202         AN.InScratch = AN.InScratch1;
02203         poly_ratfun_add(BHEAD s1,s2);
02204         S->PolyWise = oldpw;
02205         AN.SplitScratch1 = AN.SplitScratch;
02206         AN.SplitScratchSize1 = AN.SplitScratchSize;
02207         AN.InScratch1 = AN.InScratch;
02208         AN.SplitScratch = oldSplitScratch;
02209         AN.SplitScratchSize = oldSplitScratchSize;
02210         AN.InScratch = oldInScratch;
02211         AT.WorkPointer = w;
02212         AR.SortType = oldtype;
02213         if ( w[1] <= FUNHEAD ||
02214             ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 ) ) {
02215             *ps1 = *ps2 = 0; return(0);
02216         }
02217     }
02218     else {
02219         if ( w + s1[1] + s2[1] + 12 + ARGHEAD >= AT.WorkTop ) {
02220             MLOCK(ErrorMessageLock);
02221             MesPrint("Program was adding polyfun arguments");
02222             MesWork();
02223             MUNLOCK(ErrorMessageLock);
02224         }
02225         AddArgs(BHEAD s1,s2,w);
02226     }
02227     /*
02228     Now we need to store the result in a convenient place.
02229     */
02230     if ( w[1] <= FUNHEAD ) { *ps1 = *ps2 = 0; return(0); }
02231     if ( w[1] <= s1[1] || w[1] <= s2[1] ) { /* Fits in place. */
02232         if ( w[1] > s1[1] ) {
02233             *ps1 = *ps2;
02234             s1 = s2;
02235         }
02236         t = s1 + s1[1];
02237         m = *ps1 + **ps1;
02238         i = w[1];
02239         NCOPY(s1,w,i);
02240         if ( s1 != t ) {
02241             while ( t < m ) *s1++ = *t++;
02242             **ps1 = WORDDIF(s1,(*ps1));
02243         }
02244         *ps2 = 0;
02245     }
02246     else { /* Make new term */
02247 #ifdef TESTGARB
02248         s2 = *ps2;
02249 #endif
02250         *ps2 = 0;
02251         if ( (S->sFill + (**ps1 + w[1] - s1[1])) >= S->sTop2 ) {
02252 #ifdef TESTGARB
02253             MesPrint("-----Garbage collection-----");
02254 #endif
02255             AT.WorkPointer += w[1];
02256             GarbHand();
02257             AT.WorkPointer = w;
02258             s1 = *ps1;
02259             if ( (S->sFill + (**ps1 + w[1] - s1[1])) >= S->sTop2 ) {
02260 #ifdef TESTGARB
02261                 UBYTE OutBuf[140];

```

```

02262         MLOCK(ErrorMessageLock);
02263         AO.OutFill = AO.OutputLine = OutBuf;
02264         AO.OutSkip = 3;
02265         FiniLine();
02266         i = *s2;
02267         while ( --i >= 0 ) {
02268             TalToLine((UWORD) (*s2++)); TokenToLine((UBYTE *) " ");
02269         }
02270         FiniLine();
02271         AO.OutFill = AO.OutputLine = OutBuf;
02272         AO.OutSkip = 3;
02273         FiniLine();
02274         s2 = *ps1;
02275         i = *s2;
02276         while ( --i >= 0 ) {
02277             TalToLine((UWORD) (*s2++)); TokenToLine((UBYTE *) " ");
02278         }
02279         FiniLine();
02280         AO.OutFill = AO.OutputLine = OutBuf;
02281         AO.OutSkip = 3;
02282         FiniLine();
02283         s2 = w;
02284         i = w[1];
02285         while ( --i >= 0 ) {
02286             TalToLine((UWORD) (*s2++)); TokenToLine((UBYTE *) " ");
02287         }
02288         FiniLine();
02289         if ( AR.sLevel > 0 ) {
02290             MesPrint("Please increase SubSmallExtension setup parameter.");
02291         }
02292         else {
02293             MesPrint("Please increase SmallExtension setup parameter.");
02294         }
02295         MUNLOCK(ErrorMessageLock);
02296 #else
02297         MLOCK(ErrorMessageLock);
02298         if ( AR.sLevel > 0 ) {
02299             MesPrint("Please increase SubSmallExtension setup parameter.");
02300         }
02301         else {
02302             MesPrint("Please increase SmallExtension setup parameter.");
02303         }
02304         MUNLOCK(ErrorMessageLock);
02305 #endif
02306         Terminate(-1);
02307     }
02308 }
02309 t = *ps1;
02310 s2 = S->sFill;
02311 m = s2;
02312 i = S->PolyWise;
02313 NCOPY(s2,t,i);
02314 i = w[1];
02315 NCOPY(s2,w,i);
02316 t = t + t[1];
02317 w = *ps1 + **ps1;
02318 while ( t < w ) *s2++ = *t++;
02319 *m = WORDDIF(s2,m);
02320 *ps1 = m;
02321 S->sFill = s2;
02322 if ( *m > AM.MaxTer/((LONG) sizeof(WORD)) ) {
02323     MLOCK(ErrorMessageLock);
02324     MesPrint("Term too complex after polynomial addition. MaxTermSize = %10l",
02325             AM.MaxTer/sizeof(WORD));
02326     MUNLOCK(ErrorMessageLock);
02327     Terminate(-1);
02328 }
02329 }
02330 return(1);
02331 }
02332
02333 /*
02334 #] AddPoly :
02335 #[ AddArgs : void AddArgs(arg1,arg2,to)
02336 */
02337
02338 #define INSLLENGTH(x) w[1] = FUNHEAD+ARGHEAD+x; w[FUNHEAD] = ARGHEAD+x;
02339
02340 void AddArgs(PHEAD WORD *s1, WORD *s2, WORD *m)
02341 {
02342     GETBIDENTITY
02343     WORD i1, i2;
02344     WORD *w = m, *mm, *t, *t1, *t2, *tstop1, *tstop2;
02345     WORD tempterm[8+FUNHEAD];
02346
02347     *m++ = AR.PolyFun; *m++ = 0; FILLFUN(m)
02348     *m++ = 0; *m++ = 0; FILLARG(m)

```

```

02356     if ( s1[FUNHEAD] < 0 || s2[FUNHEAD] < 0 ) {
02357         if ( s1[FUNHEAD] < 0 ) {
02358             if ( s2[FUNHEAD] < 0 ) { /* Both are special */
02359                 if ( s1[FUNHEAD] <= -FUNCTION ) {
02360                     if ( s2[FUNHEAD] == s1[FUNHEAD] ) {
02361                         *m++ = 4+FUNHEAD; *m++ = -s1[FUNHEAD]; *m++ = FUNHEAD;
02362                         FILLFUN(m)
02363                         *m++ = 2; *m++ = 1; *m++ = 3;
02364                         INSLNGTH(4+FUNHEAD)
02365                     }
02366                 } else if ( s2[FUNHEAD] <= -FUNCTION ) {
02367                     i1 = functions[-FUNCTION-s1[FUNHEAD]].commute != 0;
02368                     i2 = functions[-FUNCTION-s2[FUNHEAD]].commute != 0;
02369                     if ( ( !i1 && i2 ) || ( i1 == i2 && i1 > i2 ) ) {
02370                         i1 = s2[FUNHEAD];
02371                         s2[FUNHEAD] = s1[FUNHEAD];
02372                         s1[FUNHEAD] = i1;
02373                     }
02374                     *m++ = 4+FUNHEAD; *m++ = -s1[FUNHEAD]; *m++ = FUNHEAD;
02375                     FILLFUN(m)
02376                     *m++ = 1; *m++ = 1; *m++ = 3;
02377                     *m++ = 4+FUNHEAD; *m++ = -s2[FUNHEAD]; *m++ = FUNHEAD;
02378                     FILLFUN(m)
02379                     *m++ = 1; *m++ = 1; *m++ = 3;
02380                     INSLNGTH(8+2*FUNHEAD)
02381                 }
02382             } else if ( s2[FUNHEAD] == -SYMBOL ) {
02383                 *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = s2[FUNHEAD+1]; *m++ = 1;
02384                 *m++ = 1; *m++ = 1; *m++ = 3;
02385                 *m++ = 4+FUNHEAD; *m++ = -s1[FUNHEAD]; *m++ = FUNHEAD;
02386                 FILLFUN(m)
02387                 *m++ = 1; *m++ = 1; *m++ = 3;
02388                 INSLNGTH(12+FUNHEAD)
02389             }
02390         } else { /* number */
02391             *m++ = 4;
02392             *m++ = ABS(s2[FUNHEAD+1]); *m++ = 1; *m++ = s2[FUNHEAD+1] < 0 ? -3: 3;
02393             *m++ = 4+FUNHEAD; *m++ = -s1[FUNHEAD]; *m++ = FUNHEAD;
02394             FILLFUN(m)
02395             *m++ = 1; *m++ = 1; *m++ = 3;
02396             INSLNGTH(8+FUNHEAD)
02397         }
02398     }
02399     else if ( s1[FUNHEAD] == -SYMBOL ) {
02400         if ( s2[FUNHEAD] == s1[FUNHEAD] ) {
02401             if ( s1[FUNHEAD+1] == s2[FUNHEAD+1] ) {
02402                 *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = s1[FUNHEAD+1];
02403                 *m++ = 1; *m++ = 2; *m++ = 1; *m++ = 3;
02404                 INSLNGTH(8)
02405             }
02406             else {
02407                 if ( s1[FUNHEAD+1] > s2[FUNHEAD+1] )
02408                     { i1 = s2[FUNHEAD+1]; i2 = s1[FUNHEAD+1]; }
02409                 else { i1 = s1[FUNHEAD+1]; i2 = s2[FUNHEAD+1]; }
02410                 *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = i1;
02411                 *m++ = 1; *m++ = 1; *m++ = 1; *m++ = 3;
02412                 *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = i2;
02413                 *m++ = 1; *m++ = 1; *m++ = 1; *m++ = 3;
02414                 INSLNGTH(16)
02415             }
02416         }
02417         else if ( s2[FUNHEAD] <= -FUNCTION ) {
02418             *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = s1[FUNHEAD+1]; *m++ = 1;
02419             *m++ = 1; *m++ = 1; *m++ = 3;
02420             *m++ = 4+FUNHEAD; *m++ = -s2[FUNHEAD]; *m++ = FUNHEAD;
02421             FILLFUN(m)
02422             *m++ = 1; *m++ = 1; *m++ = 3;
02423             INSLNGTH(12+FUNHEAD)
02424         }
02425         else {
02426             *m++ = 4;
02427             *m++ = ABS(s2[FUNHEAD+1]); *m++ = 1; *m++ = s2[FUNHEAD+1] < 0 ? -3: 3;
02428             *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = s1[FUNHEAD+1]; *m++ = 1;
02429             *m++ = 1; *m++ = 1; *m++ = 3;
02430             INSLNGTH(12)
02431         }
02432     }
02433     else { /* Must be -SNUMBER! */
02434         if ( s2[FUNHEAD] <= -FUNCTION ) {
02435             *m++ = 4;
02436             *m++ = ABS(s1[FUNHEAD+1]); *m++ = 1; *m++ = s1[FUNHEAD+1] < 0 ? -3: 3;
02437             *m++ = 4+FUNHEAD; *m++ = -s2[FUNHEAD]; *m++ = FUNHEAD;
02438             FILLFUN(m)
02439             *m++ = 1; *m++ = 1; *m++ = 3;
02440             INSLNGTH(8+FUNHEAD)
02441         }
02442         else if ( s2[FUNHEAD] == -SYMBOL ) {

```

```

02443         *m++ = 4;
02444         *m++ = ABS(s1[FUNHEAD+1]); *m++ = 1; *m++ = s1[FUNHEAD+1] < 0 ? -3: 3;
02445         *m++ = 8; *m++ = SYMBOL; *m++ = 4; *m++ = s2[FUNHEAD+1]; *m++ = 1;
02446         *m++ = 1; *m++ = 1; *m++ = 3;
02447         INSLNGTH(12)
02448     }
02449     else { /* Both are numbers. add. */
02450         LONG x1;
02451         x1 = (LONG)s1[FUNHEAD+1] + (LONG)s2[FUNHEAD+1];
02452         if ( x1 < 0 ) { i1 = (WORD)(-x1); i2 = -3; }
02453         else { i1 = (WORD)x1; i2 = 3; }
02454         if ( x1 && AN.ncmod != 0 ) {
02455             m[0] = 4;
02456             m[1] = i1;
02457             m[2] = 1;
02458             m[3] = i2;
02459             if ( Modulus(m) ) Terminate(-1);
02460             if ( *m == 0 ) w[1] = 0;
02461             else {
02462                 if ( *m == 4 && ( m[1] & MAXPOSITIVE ) == m[1]
02463                     && m[3] == 3 ) {
02464                     i1 = m[1];
02465                     m -= ARGHEAD;
02466                     *m++ = -SNUMBER;
02467                     *m++ = i1;
02468                     INSLNGTH(4)
02469                 }
02470                 else {
02471                     INSLNGTH(*m)
02472                     m += *m;
02473                 }
02474             }
02475         }
02476     else {
02477         if ( x1 == 0 ) {
02478             w[1] = FUNHEAD;
02479         }
02480         else if ( ( i1 & MAXPOSITIVE ) == i1 ) {
02481             m -= ARGHEAD;
02482             *m++ = -SNUMBER;
02483             *m++ = (WORD)x1;
02484             w[1] = FUNHEAD+2;
02485         }
02486         else {
02487             *m++ = 4; *m++ = i1; *m++ = 1; *m++ = i2;
02488             INSLNGTH(4)
02489         }
02490     }
02491 }
02492 }
02493 }
02494 else { /* Only s1 is special */
02495 slonly:
02496 /*
02497 Compose a term in `tempterm'
02498 */
02499     t = tempterm;
02500     if ( s1[FUNHEAD] <= -FUNCTION ) {
02501         *t++ = 4+FUNHEAD; *t++ = -s1[FUNHEAD]; *t++ = FUNHEAD;
02502         FILLFUN(t)
02503         *t++ = 1; *t++ = 1; *t++ = 3;
02504     }
02505     else if ( s1[FUNHEAD] == -SYMBOL ) {
02506         *t++ = 8; *t++ = SYMBOL; *t++ = 4;
02507         *t++ = s1[FUNHEAD+1]; *t++ = 1;
02508         *t++ = 1; *t++ = 1; *t++ = 3;
02509     }
02510     else {
02511         *t++ = 4; *t++ = ABS(s1[FUNHEAD+1]);
02512         *t++ = 1; *t++ = s1[FUNHEAD+1] < 0 ? -3: 3;
02513     }
02514     tstop1 = t;
02515     s1 = tempterm;
02516     goto twogen;
02517 }
02518 }
02519 else { /* Only s2 is special */
02520     t = s1;
02521     s1 = s2;
02522     s2 = t;
02523     goto slonly;
02524 }
02525 }
02526 else {
02527     int oldPolyFlag;
02528     tstop1 = s1 + s1[1];
02529     s1 += FUNHEAD+ARGHEAD;

```



```

02530 twogen:
02531     tstop2 = s2 + s2[1];
02532     s2 += FUNHEAD+ARGHEAD;
02533 /*
02534     Now we should merge the expressions in s1 and s2 into m.
02535 */
02536     oldPolyFlag = AT.SS->PolyFlag;
02537     AT.SS->PolyFlag = 0;
02538     while ( s1 < tstop1 && s2 < tstop2 ) {
02539         i1 = CompareTerms(BHEAD s1,s2,(WORD) (-1));
02540         if ( i1 > 0 ) {
02541             i2 = *s1;
02542             NCOPY(m,s1,i2);
02543         }
02544         else if ( i1 < 0 ) {
02545             i2 = *s2;
02546             NCOPY(m,s2,i2);
02547         }
02548         else { /* Coefficients should be added. */
02549             WORD i;
02550             t = s1+*s1;
02551             i1 = t[-1];
02552             i2 = *s1 - ABS(i1);
02553             t2 = s2 + i2;
02554             s2 += *s2;
02555             mm = m;
02556             NCOPY(m,s1,i2);
02557             t1 = s1;
02558             s1 = t;
02559             i2 = s2[-1];
02560 /*
02561             t1,i1 is the first coefficient
02562             t2,i2 is the second coefficient
02563             It should be placed at m,i1
02564 */
02565             i1 = REDLENG(i1);
02566             i2 = REDLENG(i2);
02567             if ( AddRat(BHEAD (UWORD *)t1,i1,(UWORD *)t2,i2,(UWORD *)m,&i) ) {
02568                 MLOCK(ErrorMessageLock);
02569                 MesPrint("Addition of coefficients of PolyFun");
02570                 MUNLOCK(ErrorMessageLock);
02571                 Terminate(-1);
02572             }
02573             if ( i == 0 ) {
02574                 m = mm;
02575             }
02576             else {
02577                 i1 = INCLENG(i);
02578                 m += ABS(i1);
02579                 m[-1] = i1;
02580                 *mm = WORDDIF(m,mm);
02581                 if ( AN.ncmod != 0 ) {
02582                     if ( Modulus(mm) ) Terminate(-1);
02583                     if ( !*mm ) m = mm;
02584                     else m = mm + *mm;
02585                 }
02586             }
02587         }
02588     }
02589     while ( s1 < tstop1 ) *m++ = *s1++;
02590     while ( s2 < tstop2 ) *m++ = *s2++;
02591     w[1] = WORDDIF(m,w);
02592     w[FUNHEAD] = w[1] - FUNHEAD;
02593     if ( ToFast(w+FUNHEAD,w+FUNHEAD) ) {
02594         if ( w[FUNHEAD] <= -FUNCTION ) w[1] = FUNHEAD+1;
02595         else w[1] = FUNHEAD+2;
02596         if ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 ) w[1] = FUNHEAD;
02597     }
02598 /*
02599     AT.SS->PolyFlag = AR.PolyFunType;*/
02600     AT.SS->PolyFlag = oldPolyFlag;
02601 }
02602
02603 /*
02604     #] AddArgs :
02605     #[ Compare1 :                WORD Compare1(term1,term2,level)
02606 */
02640 WORD Compare1(PHEAD WORD *term1, WORD *term2, WORD level)
02641 {
02642     SORTING *S = AT.SS;
02643     WORD *stopper1, *stopper2, *t2;
02644     WORD *s1, *s2, *t1;
02645     WORD *stopex1, *stopex2;
02646     WORD c1, c2;
02647     WORD prevorder;
02648     WORD count = -1, localPoly, polyhit = -1;
02649

```

```

02650 #ifdef COUNTCOMPARES
02651     if ( AR.sLevel == 0 ) {
02652 #ifdef WITHPTHREADS
02653         numcompares[AT.identity]++;
02654 #else
02655         numcompares[0]++;
02656 #endif
02657     }
02658 #endif
02659
02660     if ( S->PolyFlag ) {
02661 /*
02662         if ( S->PolyWise != 0 ) {
02663             MLOCK(ErrorMessageLock);
02664             MesPrint("S->PolyWise is not zero!!!!");
02665             MUNLOCK(ErrorMessageLock);
02666         }
02667 */
02668         count = 0; localPoly = 1; S->PolyWise = polyhit = 0;
02669         S->PolyFlag = AR.PolyFunType;
02670         if ( AR.PolyFunType == 2 &&
02671             ( AR.PolyFunExp == 2 || AR.PolyFunExp == 3 ) ) S->PolyFlag = 1;
02672     }
02673     else { localPoly = 0; }
02674 #ifdef WITHFLOAT
02675     AT.SortFloatMode = 0;
02676 #endif
02677     prevorder = 0;
02678     GETSTOP(term1,s1);
02679     stopper1 = s1;
02680     GETSTOP(term2,stopper2);
02681     t1 = term1 + 1;
02682     t2 = term2 + 1;
02683     while ( t1 < stopper1 && t2 < stopper2 ) {
02684         if ( *t1 != *t2 ) {
02685             if ( *t1 == HAAKJE ) return(PREV(-1));
02686             if ( *t2 == HAAKJE ) return(PREV(1));
02687             if ( *t1 >= (FUNCTION-1) ) {
02688                 if ( *t2 < (FUNCTION-1) ) return(PREV(-1));
02689                 if ( *t1 < FUNCTION && *t2 < FUNCTION ) return(PREV(*t2-*t1));
02690                 if ( *t1 < FUNCTION ) return(PREV(1));
02691                 if ( *t2 < FUNCTION ) return(PREV(-1));
02692                 c1 = functions[*t1-FUNCTION].commute;
02693                 c2 = functions[*t2-FUNCTION].commute;
02694                 if ( !c1 ) {
02695                     if ( c2 ) return(PREV(1));
02696                     else return(PREV(*t2-*t1));
02697                 }
02698                 else {
02699                     if ( !c2 ) return(PREV(-1));
02700                     else return(PREV(*t2-*t1));
02701                 }
02702             }
02703             else return(PREV(*t2-*t1));
02704         }
02705         s1 = t1 + 2;
02706         s2 = t2 + 2;
02707         c1 = *t1;
02708         t1 += t1[1];
02709         t2 += t2[1];
02710         if ( localPoly && c1 < FUNCTION ) {
02711             polyhit = 1;
02712         }
02713         if ( c1 <= (FUNCTION-1)
02714             || ( c1 >= FUNCTION && functions[c1-FUNCTION].spec > 0 ) ) {
02715             if ( c1 == SYMBOL ) {
02716                 if ( *s1 == FACTORSYMBOL && *s2 == FACTORSYMBOL
02717                     && s1[-1] == 4 && s2[-1] == 4
02718                     && ( ( t1 < stopper1 && *t1 == HAAKJE )
02719                         || ( t1 == stopper1 && AT.fromindex ) ) ) {
02720 /*
02721                 We have to be very careful with the criteria here, because
02722                 Compare1 is called both in the regular sorting and by the
02723                 routine that makes the bracket index. In the last case
02724                 there is no HAAKJE subterm.
02725 */
02726                 if ( s1[1] != s2[1] ) return(s2[1]-s1[1]);
02727                 s1 += 2; s2 += 2;
02728             }
02729             else if ( AR.SortType >= SORTPOWERFIRST ) {
02730                 WORD i1 = 0, *r1;
02731                 r1 = s1;
02732                 while ( s1 < t1 ) { i1 += s1[1]; s1 += 2; }
02733                 s1 = r1; r1 = s2;
02734                 while ( s2 < t2 ) { i1 -= s2[1]; s2 += 2; }
02735                 s2 = r1;
02736                 if ( i1 ) {

```

```

02737         if ( AR.SortType >= SORTANTIPOWER ) i1 = -i1;
02738         return(PREV(i1));
02739     }
02740 }
02741 while ( s1 < t1 ) {
02742     if ( s2 >= t2 ) {
02743         /* return(PREV(1)); */
02744         if ( AR.SortType==SORTLOWFIRST ) {
02745             return(PREV((s1[1]>0?-1:1)));
02746         }
02747         else {
02748             return(PREV((s1[1]<0?-1:1)));
02749         }
02750     }
02751     if ( *s1 != *s2 ) {
02752         /* return(PREV(*s2-*s1)); */
02753         if ( AR.SortType==SORTLOWFIRST ) {
02754             if ( *s1 < *s2 ) {
02755                 return(PREV((s1[1]<0?1:-1)));
02756             }
02757             else {
02758                 return(PREV((s2[1]<0?-1:1)));
02759             }
02760         }
02761         else {
02762             if ( *s1 < *s2 ) {
02763                 return(PREV((s1[1]<0?-1:1)));
02764             }
02765             else {
02766                 return(PREV((s2[1]<0?1:-1)));
02767             }
02768         }
02769     }
02770     s1++; s2++;
02771     if ( *s1 != *s2 ) return(
02772         PREV((AR.SortType==SORTLOWFIRST?*s2-*s1:*s1-*s2)));
02773     s1++; s2++;
02774 }
02775 if ( s2 < t2 ) {
02776     /* return(PREV(-1)); */
02777     if ( AR.SortType==SORTLOWFIRST ) {
02778         return(PREV((s2[1]<0?-1:1)));
02779     }
02780     else {
02781         return(PREV((s2[1]<0?1:-1)));
02782     }
02783 }
02784 }
02785 else if ( c1 == DOTPRODUCT ) {
02786     if ( AR.SortType >= SORTPOWERFIRST ) {
02787         WORD i1 = 0, *r1;
02788         r1 = s1;
02789         while ( s1 < t1 ) { i1 += s1[2]; s1 += 3; }
02790         s1 = r1; r1 = s2;
02791         while ( s2 < t2 ) { i1 -= s2[2]; s2 += 3; }
02792         s2 = r1;
02793         if ( i1 ) {
02794             if ( AR.SortType >= SORTANTIPOWER ) i1 = -i1;
02795             return(PREV(i1));
02796         }
02797     }
02798     while ( s1 < t1 ) {
02799         if ( s2 >= t2 ) return(PREV(1));
02800         if ( *s1 != *s2 ) return(PREV(*s2-*s1));
02801         s1++; s2++;
02802         if ( *s1 != *s2 ) return(PREV(*s2-*s1));
02803         s1++; s2++;
02804         if ( *s1 != *s2 ) return(
02805             PREV((AR.SortType==SORTLOWFIRST?*s2-*s1:*s1-*s2)));
02806         s1++; s2++;
02807     }
02808     if ( s2 < t2 ) return(PREV(-1));
02809 }
02810 else {
02811     while ( s1 < t1 ) {
02812         if ( s2 >= t2 ) return(PREV(1));
02813         if ( *s1 != *s2 ) return(PREV(*s2-*s1));
02814         s1++; s2++;
02815     }
02816     if ( s2 < t2 ) return(PREV(-1));
02817 }
02818 }
02819 else {
02820     #if FUNHEAD != 2
02821     s1 += FUNHEAD-2;
02822     s2 += FUNHEAD-2;
02823     #endif

```

```

02824         if ( localPoly && c1 == AR.PolyFun ) {
02825             if ( count == 0 ) {
02826                 if ( S->PolyFlag == 1 ) {
02827                     WORD i1, i2;
02828                     if ( *s1 > 0 ) i1 = *s1;
02829                     else if ( *s1 <= -FUNCTION ) i1 = 1;
02830                     else i1 = 2;
02831                     if ( *s2 > 0 ) i2 = *s2;
02832                     else if ( *s2 <= -FUNCTION ) i2 = 1;
02833                     else i2 = 2;
02834                     if ( s1+i1 == t1 && s2+i2 == t2 ) { /* This is the stuff */
02835                         /*
02836                         Test for scalar nature
02837                         */
02838                         if ( !polyhit ) {
02839                             WORD *u1, *u2, *ustop;
02840                             if ( *s1 < 0 ) {
02841                                 if ( *s1 != -SNUMBER && *s1 != -SYMBOL && *s1 > -FUNCTION )
02842                                     goto NoPoly;
02843                             }
02844                             else {
02845                                 u1 = s1 + ARGHEAD;
02846                                 while ( u1 < t1 ) {
02847                                     u2 = u1 + *u1;
02848                                     ustop = u2 - ABS(u2[-1]);
02849                                     u1++;
02850                                     while ( u1 < ustop ) {
02851                                         if ( *u1 == INDEX ) goto NoPoly;
02852                                         u1 += u1[1];
02853                                     }
02854                                     u1 = u2;
02855                                 }
02856                             }
02857                             if ( *s2 < 0 ) {
02858                                 if ( *s2 != -SNUMBER && *s2 != -SYMBOL && *s2 > -FUNCTION )
02859                                     goto NoPoly;
02860                             }
02861                             else {
02862                                 u1 = s2 + ARGHEAD;
02863                                 while ( u1 < t2 ) {
02864                                     u2 = u1 + *u1;
02865                                     ustop = u2 - ABS(u2[-1]);
02866                                     u1++;
02867                                     while ( u1 < ustop ) {
02868                                         if ( *u1 == INDEX ) goto NoPoly;
02869                                         u1 += u1[1];
02870                                     }
02871                                     u1 = u2;
02872                                 }
02873                             }
02874                         }
02875                         S->PolyWise = WORDDIF(s1,term1);
02876                         S->PolyWise -= FUNHEAD;
02877                         count = 1;
02878                         continue;
02879                     }
02880                     else {
02881 NoPoly:
02882                         S->PolyWise = localPoly = 0;
02883                     }
02884                 }
02885             else if ( AR.PolyFunType == 2 ) {
02886                 WORD i1, i2, i1a, i2a;
02887                 if ( *s1 > 0 ) i1 = *s1;
02888                 else if ( *s1 <= -FUNCTION ) i1 = 1;
02889                 else i1 = 2;
02890                 if ( *s2 > 0 ) i2 = *s2;
02891                 else if ( *s2 <= -FUNCTION ) i2 = 1;
02892                 else i2 = 2;
02893                 if ( s1[i1] > 0 ) i1a = s1[i1];
02894                 else if ( s1[i1] <= -FUNCTION ) i1a = 1;
02895                 else i1a = 2;
02896                 if ( s2[i2] > 0 ) i2a = s2[i2];
02897                 else if ( s2[i2] <= -FUNCTION ) i2a = 1;
02898                 else i2a = 2;
02899                 if ( s1+i1+i1a == t1 && s2+i2+i2a == t2 ) { /* This is the stuff */
02900                     /*
02901                     Test for scalar nature
02902                     */
02903                     if ( !polyhit ) {
02904                         WORD *u1, *u2, *ustop;
02905                         if ( *s1 < 0 ) {
02906                             if ( *s1 != -SNUMBER && *s1 != -SYMBOL && *s1 > -FUNCTION )
02907                                 goto NoPoly;
02908                         }
02909                         else {
02910                             u1 = s1 + ARGHEAD;

```

```

02911         while ( u1 < s1+i1 ) {
02912             u2 = u1 + *u1;
02913             ustop = u2 - ABS(u2[-1]);
02914             u1++;
02915             while ( u1 < ustop ) {
02916                 if ( *u1 == INDEX ) goto NoPoly;
02917                 u1 += u1[1];
02918             }
02919             u1 = u2;
02920         }
02921     }
02922     if ( s1[i1] < 0 ) {
02923         if ( s1[i1] != -SNUMBER && s1[i1] != -SYMBOL && s1[i1] > -FUNCTION )
02924             goto NoPoly;
02925     }
02926     else {
02927         u1 = s1 + i1 + ARGHEAD;
02928         while ( u1 < t1 ) {
02929             u2 = u1 + *u1;
02930             ustop = u2 - ABS(u2[-1]);
02931             u1++;
02932             while ( u1 < ustop ) {
02933                 if ( *u1 == INDEX ) goto NoPoly;
02934                 u1 += u1[1];
02935             }
02936             u1 = u2;
02937         }
02938     }
02939     if ( *s2 < 0 ) {
02940         if ( *s2 != -SNUMBER && *s2 != -SYMBOL && *s2 > -FUNCTION )
02941             goto NoPoly;
02942     }
02943     else {
02944         u1 = s2 + ARGHEAD;
02945         while ( u1 < s2+i2 ) {
02946             u2 = u1 + *u1;
02947             ustop = u2 - ABS(u2[-1]);
02948             u1++;
02949             while ( u1 < ustop ) {
02950                 if ( *u1 == INDEX ) goto NoPoly;
02951                 u1 += u1[1];
02952             }
02953             u1 = u2;
02954         }
02955     }
02956     if ( s2[i2] < 0 ) {
02957         if ( s2[i2] != -SNUMBER && s2[i2] != -SYMBOL && s2[i2] > -FUNCTION )
02958             goto NoPoly;
02959     }
02960     else {
02961         u1 = s2 + i2 + ARGHEAD;
02962         while ( u1 < t2 ) {
02963             u2 = u1 + *u1;
02964             ustop = u2 - ABS(u2[-1]);
02965             u1++;
02966             while ( u1 < ustop ) {
02967                 if ( *u1 == INDEX ) goto NoPoly;
02968                 u1 += u1[1];
02969             }
02970             u1 = u2;
02971         }
02972     }
02973 }
02974 S->PolyWise = WORDDIF(s1,term1);
02975 S->PolyWise -= FUNHEAD;
02976 count = 1;
02977 continue;
02978 }
02979 else {
02980     S->PolyWise = localPoly = 0;
02981 }
02982 }
02983 else {
02984     S->PolyWise = localPoly = 0;
02985 }
02986 }
02987 else {
02988     t1 = term1 + S->PolyWise;
02989     t2 = term2 + S->PolyWise;
02990     S->PolyWise = 0;
02991     localPoly = 0;
02992     continue;
02993 }
02994 }
02995 #ifdef WITHFLOAT
02996     if ( level == 0 && c1 == FLOATFUN && t1 == stopper1 && t2 == stopper2 && AT.aux_ != 0 ) {
02997 /*

```

```

02998             We have two FLOATFUN's. Test whether they are 'legal'
02999 */
03000         if ( TestFloat(s1-FUNHEAD) ) {
03001             if ( TestFloat(s2-FUNHEAD) ) { AT.SortFloatMode = 3; return(0); }
03002             else { return(1); }
03003         }
03004         else if ( TestFloat(s2-FUNHEAD) ) { return(-1); }
03005     }
03006 #endif
03007     while ( s1 < t1 ) {
03008         /*
03009             The next statement was added 9-nov-2001. It repaired a bad error
03010             */
03011             if ( s2 >= t2 ) return(PREV(-1));
03012         /*
03013             There is a little problem here with fast arguments
03014             We don't want to sacrifice speed, but we like to
03015             keep a rational ordering. This last one suffers in
03016             the solution that has been chosen here.
03017             */
03018             if ( AC.properorderflag ) {
03019                 WORD oldpolyflag;
03020                 oldpolyflag = S->PolyFlag;
03021                 S->PolyFlag = 0;
03022                 if ( ( c2 = -CompArg(s1,s2) ) != 0 ) {
03023                     S->PolyFlag = oldpolyflag; return(PREV(c2));
03024                 }
03025                 S->PolyFlag = oldpolyflag;
03026                 NEXTARG(s1)
03027                 NEXTARG(s2)
03028             }
03029             else {
03030                 if ( *s1 > 0 ) {
03031                     if ( *s2 > 0 ) {
03032                         WORD oldpolyflag;
03033                         stopex1 = s1 + *s1;
03034                         if ( s2 >= t2 ) return(PREV(-1));
03035                         stopex2 = s2 + *s2;
03036                         s1 += ARGHEAD; s2 += ARGHEAD;
03037                         oldpolyflag = S->PolyFlag;
03038                         S->PolyFlag = 0;
03039                         while ( s1 < stopex1 ) {
03040                             if ( s2 >= stopex2 ) {
03041                                 S->PolyFlag = oldpolyflag; return(PREV(-1));
03042                             }
03043                             if ( ( c2 = CompareTerms(BHEAD s1,s2,(WORD)1) ) != 0 ) {
03044                                 S->PolyFlag = oldpolyflag; return(PREV(c2));
03045                             }
03046                             s1 += *s1;
03047                             s2 += *s2;
03048                         }
03049                         S->PolyFlag = oldpolyflag;
03050                         if ( s2 < stopex2 ) return(PREV(1));
03051                     }
03052                     else return(PREV(1));
03053                 }
03054                 else {
03055                     if ( *s2 > 0 ) return(PREV(-1));
03056                     if ( *s1 != *s2 ) { return(PREV(*s1-*s2)); }
03057                     if ( *s1 > -FUNCTION ) {
03058                         if ( ++s1 != ++s2 ) { return(PREV(*s2-*s1)); }
03059                     }
03060                     s1++; s2++;
03061                 }
03062             }
03063         }
03064         if ( s2 < t2 ) return(PREV(1));
03065     }
03066 }
03067 #ifdef WITHFLOAT
03068     if ( level == 0 && t1 < stopper1 && *t1 == FLOATFUN && t1+t1[1] == stopper1
03069         && TestFloat(t1) && AT.aux_ != 0 ) {
03070         AT.SortFloatMode = 1; return(0);
03071     }
03072     else if ( level == 0 && t2 < stopper2 && *t2 == FLOATFUN && t2+t2[1] == stopper2
03073         && TestFloat(t2) && AT.aux_ != 0 ) {
03074         AT.SortFloatMode = 2; return(0);
03075     }
03076 #endif
03077 {
03078     if ( AR.SortType != SORTLOWFIRST ) {
03079         if ( t1 < stopper1 ) return(PREV(1));
03080         if ( t2 < stopper2 ) return(PREV(-1));
03081     }
03082     else {
03083         if ( t1 < stopper1 ) return(PREV(-1));
03084         if ( t2 < stopper2 ) return(PREV(1));

```

```

03085     }
03086 }
03087 if ( level == 3 ) return(CompCoef(term1,term2));
03088 if ( level >= 1 )
03089     return(CompCoef(term2,term1));
03090 return(0);
03091 }
03092
03093 /*
03094     #] Compare1 :
03095     #[ CompareSymbols :          WORD CompareSymbols(term1,term2,par)
03096 */
03110 WORD CompareSymbols(PHEAD WORD *term1, WORD *term2, WORD par)
03111 {
03112     int sum1, sum2;
03113     WORD *t1, *t2, *tt1, *tt2;
03114     int low, high;
03115     DUMMYUSE(par);
03116     if ( AR.SortType == SORTLOWFIRST ) { low = 1; high = -1; }
03117     else { low = -1; high = 1; }
03118     t1 = term1 + 1; tt1 = term1+*term1; tt1 -= ABS(tt1[-1]); t1 += 2;
03119     t2 = term2 + 1; tt2 = term2+*term2; tt2 -= ABS(tt2[-1]); t2 += 2;
03120     if ( AN.polysortflag > 0 ) {
03121         sum1 = 0; sum2 = 0;
03122         while ( t1 < tt1 ) { sum1 += t1[1]; t1 += 2; }
03123         while ( t2 < tt2 ) { sum2 += t2[1]; t2 += 2; }
03124         if ( sum1 < sum2 ) return(low);
03125         if ( sum1 > sum2 ) return(high);
03126         t1 = term1+3; t2 = term2 + 3;
03127     }
03128     while ( t1 < tt1 && t2 < tt2 ) {
03129         if ( *t1 > *t2 ) return(low);
03130         if ( *t1 < *t2 ) return(high);
03131         if ( t1[1] < t2[1] ) return(low);
03132         if ( t1[1] > t2[1] ) return(high);
03133         t1 += 2; t2 += 2;
03134     }
03135     if ( t1 < tt1 ) return(high);
03136     if ( t2 < tt2 ) return(low);
03137     return(0);
03138 }
03139
03140 /*
03141     #] CompareSymbols :
03142     #[ CompareHSymbols :          WORD CompareHSymbols(term1,term2,par)
03143 */
03153 WORD CompareHSymbols(PHEAD WORD *term1, WORD *term2, WORD par)
03154 {
03155     WORD *t1, *t2, *tt1, *tt2, *ttt1, *ttt2;
03156     DUMMYUSE(par);
03157     DUMMYUSE(AT.WorkPointer);
03158     t1 = term1 + 1; tt1 = term1+*term1; tt1 -= ABS(tt1[-1]); t1 += 2;
03159     t2 = term2 + 1; tt2 = term2+*term2; tt2 -= ABS(tt2[-1]); t2 += 2;
03160     while ( t1 < tt1 && t2 < tt2 ) {
03161         if ( *t1 != *t2 ) {
03162             if ( t1[0] < t2[0] ) return(-1);
03163             return(1);
03164         }
03165         else if ( *t1 == HAAKJE ) {
03166             t1 += 3; t2 += 3; continue;
03167         }
03168         ttt1 = t1+t1[1]; ttt2 = t2+t2[1];
03169         while ( t1 < ttt1 && t2 < ttt2 ) {
03170             if ( *t1 > *t2 ) return(-1);
03171             if ( *t1 < *t2 ) return(1);
03172             if ( t1[1] < t2[1] ) return(-1);
03173             if ( t1[1] > t2[1] ) return(1);
03174             t1 += 2; t2 += 2;
03175         }
03176         if ( t1 < ttt1 ) return(1);
03177         if ( t2 < ttt2 ) return(-1);
03178     }
03179     if ( t1 < tt1 ) return(1);
03180     if ( t2 < tt2 ) return(-1);
03181     return(0);
03182 }
03183
03184 /*
03185     #] CompareHSymbols :
03186     #[ ComPress :                LONG ComPress(ss,n)
03187 */
03206 LONG ComPress(WORD **ss, LONG *n)
03207 {
03208     GETIDENTITY
03209     WORD *t, *s, j, k;
03210     LONG size = 0;
03211     int newsize, i;

```

```

03212  /*
03213      #] debug :
03214
03215      WORD **sss = ss;
03216
03217      if ( AP.DebugFlag ) {
03218          UBYTE OutBuf[140];
03219          MLOCK(ErrorMessageLock);
03220          MesPrint("ComPress:");
03221          AO.OutFill = AO.OutputLine = OutBuf;
03222          AO.OutSkip = 3;
03223          FiniLine();
03224          ss = sss;
03225          while ( *ss ) {
03226              s = *ss++;
03227              j = *s;
03228              if ( j < 0 ) {
03229                  j = s[1] + 2;
03230              }
03231              while ( --j >= 0 ) {
03232                  TalToLine((UWORD) (*s++)); TokenToLine((UBYTE *) " ");
03233              }
03234              FiniLine();
03235          }
03236          AO.OutSkip = 0;
03237          FiniLine();
03238          MUNLOCK(ErrorMessageLock);
03239          ss = sss;
03240      }
03241
03242      #] debug :
03243  */
03244  *n = 0;
03245  if ( AT.SS == AT.S0 && !AR.NoCompress ) {
03246      if ( AN.compressSize == 0 ) {
03247          if ( *ss ) { AN.compressSize = **ss + 64; }
03248          else { AN.compressSize = AM.MaxTer/sizeof(WORD) + 2; }
03249          AN.compressSpace = (WORD *) Malloc1(AN.compressSize*sizeof(WORD), "Compression");
03250      }
03251      AN.compressSpace[0] = 0;
03252      while ( *ss ) {
03253          k = 0;
03254          s = *ss;
03255          j = *s++;
03256          if ( j > AN.compressSize ) {
03257              newsize = j + 64;
03258              t = (WORD *) Malloc1(newsize*sizeof(WORD), "Compression");
03259              t[0] = 0;
03260              if ( AN.compressSpace ) {
03261                  for ( i = 0; i < *AN.compressSpace; i++ ) t[i] = AN.compressSpace[i];
03262                  M_free(AN.compressSpace, "Compression");
03263              }
03264              AN.compressSpace = t;
03265              AN.compressSize = newsize;
03266          }
03267          t = AN.compressSpace;
03268          i = *t - 1;
03269          *t++ = j; j--;
03270          if ( AR.PolyFun ) {
03271              WORD *polystop, *sa;
03272              sa = s + j;
03273              sa -= ABS(sa[-1]);
03274              polystop = s;
03275              while ( polystop < sa && *polystop != AR.PolyFun ) {
03276                  polystop += polystop[1];
03277              }
03278              while ( i > 0 && j > 0 && *s == *t && s < polystop ) {
03279                  i--; j--; s++; t++; k--;
03280              }
03281          }
03282          else {
03283              WORD *sa;
03284              sa = s + j;
03285              sa -= ABS(sa[-1]);
03286              while ( i > 0 && j > 0 && *s == *t && s < sa ) { i--; j--; s++; t++; k--; }
03287          }
03288          if ( k < -1 ) {
03289              s[-1] = j;
03290              s[-2] = k;
03291              *ss = s-2;
03292              size += j + 2;
03293          }
03294          else {
03295              size += *AN.compressSpace;
03296              if ( k == -1 ) { t--; s--; j++; }
03297          }
03298          while ( --j >= 0 ) *t++ = *s++;

```



```

03299 /*          Sabotage getting into the coefficient next time */
03300         t = AN.compressSpace + *AN.compressSpace;
03301         t[-(ABS(t[-1]))] = 0;
03302         ss++;
03303         (*n)++;
03304     }
03305 }
03306 else {
03307     while ( *ss ) {
03308         size += *(*ss++);
03309         (*n)++;
03310     }
03311 }
03312 /*
03313     #[ debug :
03314
03315     if ( AP.DebugFlag ) {
03316         UBYTE OutBuf[140];
03317         AO.OutFill = AO.OutputLine = OutBuf;
03318         AO.OutSkip = 3;
03319         FiniLine();
03320         ss = sss;
03321         while ( *ss ) {
03322             s = *ss++;
03323             j = *s;
03324             if ( j < 0 ) {
03325                 j = s[1] + 2;
03326             }
03327             while ( --j >= 0 ) {
03328                 TalToLine((UWORD) (*s++)); TokenToLine((UBYTE *) " ");
03329             }
03330             FiniLine();
03331         }
03332         AO.OutSkip = 0;
03333         FiniLine();
03334     }
03335
03336     #] debug :
03337 */
03338     return(size);
03339 }
03340
03341 /*
03342     #] ComPress :
03343     #[ SplitMerge :          void SplitMerge(Point,number)
03344 */
03370 #ifdef NEWSPLITMERGE
03371
03372 LONG SplitMerge(PHEAD WORD **Pointer, LONG number)
03373 {
03374     GETBIDENTITY
03375     SORTING *S = AT.SS;
03376     WORD **pp3, **pp1, **pp2, **pptop;
03377     LONG i, newleft, newright, split;
03378
03379 #ifdef SPLITMERGEDEBUG
03380     /* Print current array state on entry. */
03381     printf("%4ld: ", number);
03382     for (int ii = 0; ii < S->sTerms; ii++) {
03383         if ( (S->sPointer)[ii] ) {
03384             printf("%4d ", (unsigned) (S->sPointer[ii]-S->sBuffer));
03385         }
03386         else {
03387             printf(".... ");
03388         }
03389     }
03390     printf("\n");
03391     fflush(stdout);
03392 #endif
03393
03394     if ( number < 2 ) return(number);
03395     if ( number == 2 ) {
03396         pp1 = Pointer; pp2 = pp1 + 1;
03397         if ( ( i = CompareTerms(BHEAD *pp1,*pp2,(WORD)0) ) < 0 ) {
03398             pp3 = (WORD **)(*pp1); *pp1 = *pp2; *pp2 = (WORD *)pp3;
03399         }
03400         else if ( i == 0 ) {
03401             number--;
03402             if ( S->PolyWise ) { if ( AddPoly(BHEAD pp1,pp2) == 0 ) number = 0; }
03403             else { if ( AddCoef(BHEAD pp1,pp2) == 0 ) number = 0; }
03404         }
03405         return(number);
03406     }
03407     pptop = Pointer + number;
03408     split = number/2;
03409     newleft = SplitMerge(BHEAD Pointer,split);
03410     newright = SplitMerge(BHEAD Pointer+split,number-split);

```

```

03411     if ( newright == 0 ) return(newleft);
03412  /*
03413     We compare the last of the left with the first of the right
03414     If they are already in order, we will be done quickly.
03415     We may have to compactify the buffer because the recursion may
03416     have created holes. Also this compare may result in equal terms.
03417     Addition of 23-jul-1999. It makes things a bit faster.
03418  */
03419     if ( newleft > 0 && newright > 0 &&
03420         ( i = CompareTerms(BHEAD Pointer[newleft-1],Pointer[split],(WORD)0) ) >= 0 ) {
03421         pp2 = Pointer+split; pp1 = Pointer+newleft-1;
03422         if ( i == 0 ) {
03423             if ( S->PolyWise ) {
03424                 if ( AddPoly(BHEAD pp1,pp2) > 0 ) pp1++;
03425                 else newleft--;
03426             }
03427             else {
03428                 if ( AddCoef(BHEAD pp1,pp2) > 0 ) pp1++;
03429                 else newleft--;
03430             }
03431             *pp2++ = 0; newright--;
03432         }
03433         else pp1++;
03434         newleft += newright;
03435         if ( pp1 < pp2 ) {
03436             while ( --newright >= 0 ) *pp1++ = *pp2++;
03437             while ( pp1 < pptop ) *pp1++ = 0;
03438         }
03439         return(newleft);
03440     }
03441
03442     if ( split >= AN.SplitScratchSize ) {
03443         AN.SplitScratchSize = (split*3)/2+100;
03444         if ( AN.SplitScratchSize > S->Terms2InSmall/2 )
03445             AN.SplitScratchSize = S->Terms2InSmall/2;
03446         if ( AN.SplitScratch ) M_free(AN.SplitScratch,"AN.SplitScratch");
03447         AN.SplitScratch = (WORD **)Malloca(AN.SplitScratchSize*sizeof(WORD *),"AN.SplitScratch");
03448     }
03449
03450     pp3 = AN.SplitScratch; pp1 = Pointer;
03451     /* Move rather than copy, so GarbHand can't double-count. */
03452     for ( i = 0; i < newleft; i++ ) { *pp3++ = *pp1; *pp1++ = 0; }
03453     AN.InScratch = newleft;
03454     pp1 = AN.SplitScratch; pp2 = Pointer + split; pp3 = Pointer;
03455
03456 #ifndef NEWSPLITMERGETIMSORT
03457  /*
03458     An improvement in the style of Timsort
03459  */
03460     while ( newleft > 8 ) {
03461         /* Check the middle of the LHS terms */
03462         LONG nnleft = newleft/2;
03463         if ( ( i = CompareTerms(BHEAD pp1[nnleft],*pp2,(WORD)0) ) < 0 ) {
03464             /* The terms are not in order. Break out and continue as normal. */
03465             break;
03466         }
03467         /* The terms merge or are in order. Copy pointers up to this point. */
03468         /* In the copy, zero the skipped pointers so GarbHand can't double-count. */
03469         for (int iii = 0; iii < nnleft; iii++) {
03470             *pp3++ = *pp1;
03471             *pp1++ = 0;
03472         }
03473         newleft -= nnleft;
03474         if ( i == 0 ) {
03475             if ( S->PolyWise ) { i = AddPoly(BHEAD pp1,pp2); }
03476             else { i = AddCoef(BHEAD pp1,pp2); }
03477             if ( i == 0 ) {
03478                 /* The terms cancelled. The next term goes in *pp3. Don't move. */
03479             }
03480             else {
03481                 /* The terms added. Advance pp3. */
03482                 *pp3++ = *pp1;
03483             }
03484             /* We have taken a LHS (copy) and RHS term. */
03485             *pp2++ = 0;
03486             newright--;
03487             *pp1++ = 0;
03488             newleft--;
03489             break;
03490         }
03491     }
03492 #endif
03493
03494     while ( newleft > 0 && newright > 0 ) {
03495         if ( ( i = CompareTerms(BHEAD *pp1,*pp2,(WORD)0) ) < 0 ) {
03496             *pp3++ = *pp2;
03497             *pp2++ = 0;

```

```

03498         newright--;
03499     }
03500     else if ( i > 0 ) {
03501         *pp3++ = *pp1;
03502         *pp1++ = 0;
03503         newleft--;
03504     }
03505     else {
03506         if ( S->PolyWise ) { if ( AddPoly(BHEAD pp1,pp2) > 0 ) *pp3++ = *pp1; }
03507         else { if ( AddCoef(BHEAD pp1,pp2) > 0 ) *pp3++ = *pp1; }
03508         *pp1++ = 0; *pp2++ = 0; newleft--; newright--;
03509     }
03510 }
03511 for ( i = 0; i < newleft; i++ ) { *pp3++ = *pp1; *pp1++ = 0; }
03512 if ( pp3 == pp2 ) {
03513     pp3 += newright;
03514 } else {
03515     for ( i = 0; i < newright; i++ ) { *pp3++ = *pp2++; }
03516 }
03517 newleft = pp3 - Pointer;
03518 while ( pp3 < pptop ) *pp3++ = 0;
03519 AN.InScratch = 0;
03520 return(newleft);
03521 }
03522
03523 #else
03524
03525 LONG SplitMerge(PHEAD WORD **Pointer, LONG number)
03526 {
03527     GETBIDENTITY
03528     SORTING *S = AT.SS;
03529     WORD **pp3, **pp1, **pp2;
03530     LONG nleft, nright, i, newleft, newright;
03531     WORD **pptop;
03532
03533 #ifdef SPLITMERGEDEBUG
03534     /* Print current array state on entry. */
03535     printf("%4ld: ", number);
03536     for (int ii = 0; ii < S->sTerms; ii++) {
03537         if ( (S->sPointer)[ii] ) {
03538             printf("%4d ", (unsigned)(S->sPointer[ii]-S->sBuffer));
03539         }
03540         else {
03541             printf(".... ");
03542         }
03543     }
03544     printf("\n");
03545     fflush(stdout);
03546 #endif
03547
03548     if ( number < 2 ) return(number);
03549     if ( number == 2 ) {
03550         pp1 = Pointer; pp2 = pp1 + 1;
03551         if ( ( i = CompareTerms(BHEAD *pp1,*pp2,(WORD)0) ) < 0 ) {
03552             pp3 = (WORD **)(*pp1); *pp1 = *pp2; *pp2 = (WORD *)pp3;
03553         }
03554         else if ( i == 0 ) {
03555             number--;
03556             if ( S->PolyWise ) { if ( AddPoly(BHEAD pp1,pp2) == 0 ) { number = 0; } }
03557             else { if ( AddCoef(BHEAD pp1,pp2) == 0 ) { number = 0; } }
03558         }
03559         return(number);
03560     }
03561     pptop = Pointer + number;
03562     nleft = number » 1; nright = number - nleft;
03563     newleft = SplitMerge(BHEAD Pointer,nleft);
03564     newright = SplitMerge(BHEAD Pointer+nleft,nright);
03565     /*
03566     We compare the last of the left with the first of the right
03567     If they are already in order, we will be done quickly.
03568     We may have to compactify the buffer because the recursion may
03569     have created holes. Also this compare may result in equal terms.
03570     Addition of 23-jul-1999. It makes things a bit faster.
03571     */
03572     if ( newleft > 0 && newright > 0 &&
03573         ( i = CompareTerms(BHEAD Pointer[newleft-1],Pointer[nleft],(WORD)0) ) >= 0 ) {
03574         pp2 = Pointer+nleft; pp1 = Pointer+newleft-1;
03575         if ( i == 0 ) {
03576             if ( S->PolyWise ) {
03577                 if ( AddPoly(BHEAD pp1,pp2) > 0 ) pp1++;
03578                 else newleft--;
03579             }
03580             else {
03581                 if ( AddCoef(BHEAD pp1,pp2) > 0 ) pp1++;
03582                 else newleft--;
03583             }
03584             *pp2++ = 0; newright--;

```

```

03585     }
03586     else pp1++;
03587     newleft += newright;
03588     if ( pp1 < pp2 ) {
03589         while ( --newright >= 0 ) *pp1++ = *pp2++;
03590         while ( pp1 < pptop ) *pp1++ = 0;
03591     }
03592     return(newleft);
03593 }
03594 if ( nleft > AN.SplitScratchSize ) {
03595     AN.SplitScratchSize = (nleft*3)/2+100;
03596     if ( AN.SplitScratchSize > S->Terms2InSmall/2 )
03597         AN.SplitScratchSize = S->Terms2InSmall/2;
03598     if ( AN.SplitScratch ) M_free(AN.SplitScratch,"AN.SplitScratch");
03599     AN.SplitScratch = (WORD **)Malloca(AN.SplitScratchSize*sizeof(WORD *),"AN.SplitScratch");
03600 }
03601 pp3 = AN.SplitScratch; pp1 = Pointer; i = nleft;
03602 do { *pp3++ = *pp1; *pp1++ = 0; } while ( *pp1 && --i > 0 );
03603 if ( i > 0 ) { *pp3 = 0; i--; }
03604 AN.InScratch = nleft - i;
03605 pp1 = AN.SplitScratch; pp2 = Pointer + nleft; pp3 = Pointer;
03606 while ( nleft > 0 && nright > 0 && *pp1 && *pp2 ) {
03607     if ( ( i = CompareTerms(BHEAD *pp1,*pp2,(WORD)0) ) < 0 ) {
03608         *pp3++ = *pp2;
03609         *pp2++ = 0;
03610         nright--;
03611     }
03612     else if ( i > 0 ) {
03613         *pp3++ = *pp1;
03614         *pp1++ = 0;
03615         nleft--;
03616     }
03617     else {
03618         if ( S->PolyWise ) { if ( AddPoly(BHEAD pp1,pp2) > 0 ) *pp3++ = *pp1; }
03619         else { if ( AddCoef(BHEAD pp1,pp2) > 0 ) *pp3++ = *pp1; }
03620         *pp1++ = 0; *pp2++ = 0; nleft--; nright--;
03621     }
03622 }
03623 while ( --nleft >= 0 && *pp1 ) { *pp3++ = *pp1; *pp1++ = 0; }
03624 while ( --nright >= 0 && *pp2 ) { *pp3++ = *pp2++; }
03625 nleft = pp3 - Pointer;
03626 while ( pp3 < pptop ) *pp3++ = 0;
03627 AN.InScratch = 0;
03628 return(nleft);
03629 }
03630
03631 #endif
03632
03633 /*
03634     #] SplitMerge :
03635     #[ GarbHand :          void GarbHand()
03636 */
03652 void GarbHand(void)
03653 {
03654     GETIDENTITY
03655     SORTING *S = AT.SS;
03656     WORD **Point, *s2, *t, *garbuf, i;
03657     LONG k, total = 0;
03658     int tobereturned = 0;
03659     /*
03660     Compute the size needed. Put it in total.
03661     */
03662     #ifdef TESTGARB
03663         MLOCK(ErrorMessageLock);
03664         MesPrint("in: S->sFill = %l, S->sTop2 = %l",S->sFill,S->sTop2-S->sBuffer,S->sTop2-S->sBuffer);
03665     #endif
03666     Point = S->sPointer;
03667     k = S->sTerms;
03668     while ( --k >= 0 ) {
03669         if ( ( s2 = *Point++ ) != 0 ) { total += *s2; }
03670     }
03671     Point = AN.SplitScratch;
03672     k = AN.InScratch;
03673     while ( --k >= 0 ) {
03674         if ( ( s2 = *Point++ ) != 0 ) { total += *s2; }
03675     }
03676     #ifdef TESTGARB
03677         MesPrint("total = %l, nterms = %l",total,AN.InScratch);
03678         MUNLOCK(ErrorMessageLock);
03679     #endif
03680     /*
03681     Test now whether it fits. If so deal with the problem inside
03682     the memory at the tail of the large buffer.
03683     */
03684     if ( S->lBuffer != 0 && S->lFill + total <= S->lTop ) {
03685         garbuf = S->lFill;
03686     }

```

```

03687     else {
03688         garbuf = (WORD *)Malloca1(total*sizeof(WORD),"Garbage buffer");
03689         tobereturned = 1;
03690     }
03691     t = garbuf;
03692     Point = S->sPointer;
03693     k = S->sTerms;
03694     while ( --k >= 0 ) {
03695         if ( *Point ) {
03696             s2 = *Point++;
03697             i = *s2;
03698             NCOPY(t,s2,i);
03699         }
03700         else { Point++; }
03701     }
03702     Point = AN.SplitScratch;
03703     k = AN.InScratch;
03704     while ( --k >= 0 ) {
03705         if ( *Point ) {
03706             s2 = *Point++;
03707             i = *s2;
03708             NCOPY(t,s2,i);
03709         }
03710         else Point++;
03711     }
03712     s2 = S->sBuffer;
03713     t = garbuf;
03714     Point = S->sPointer;
03715     k = S->sTerms;
03716     while ( --k >= 0 ) {
03717         if ( *Point ) {
03718             *Point++ = s2;
03719             i = *t;
03720             NCOPY(s2,t,i);
03721         }
03722         else { Point++; }
03723     }
03724     Point = AN.SplitScratch;
03725     k = AN.InScratch;
03726     while ( --k >= 0 ) {
03727         if ( *Point ) {
03728             *Point++ = s2;
03729             i = *t;
03730             NCOPY(s2,t,i);
03731         }
03732         else Point++;
03733     }
03734     S->sFill = s2;
03735 #ifdef TESTGARB
03736     MLOCK(ErrorMessageLock);
03737     MesPrint("out: S->sFill = %1, S->sTop2 = %1", S->sFill-S->sBuffer, S->sTop2-S->sBuffer);
03738     if ( S->sFill >= S->sTop2 ) {
03739         MesPrint("We are in deep trouble");
03740     }
03741     MUNLOCK(ErrorMessageLock);
03742 #endif
03743     if ( tobereturned ) M_free(garbuf,"Garbage buffer");
03744     return;
03745 }
03746
03747 /*
03748     #] GarbHand :
03749     #[ MergePatches :          WORD MergePatches(par)
03750 */
03751 int MergePatches(WORD par)
03752 {
03753     GETIDENTITY
03754     SORTING *S = AT.SS;
03755     WORD **poin, **poin2, ul, k, i, im, *m1;
03756     WORD *p, lpat, mpat, level, l1, l2, r1, r2, r3, c;
03757     WORD *m2, *m3, r31, r33, ki, *rr;
03758     UWORD *coef;
03759     POSITION position;
03760     FILEHANDLE *fin, *fout;
03761     int fhandle;
03762
03763     /*
03764     UBYTE *s;
03765     */
03766 #ifdef WITHZLIB
03767     POSITION position2;
03768     int oldgzipCompress = AR.gzipCompress;
03769     if ( par == 2 ) {
03770         AR.gzipCompress = 0;
03771     }
03772 #endif
03773     fin = &S->file;
03774     fout = &(AR.FoStage4[0]);

```

```

03790 NewMerge:
03791     coef = AN.SoScratC;
03792     poin = S->poina; poin2 = S->poin2a;
03793     rr = AR.CompressPointer;
03794     *rr = 0;
03795     /*
03796         #[] Setup :
03797     */
03798     if ( par == 1 ) {
03799         fout = &(S->file);
03800         if ( fout->handle < 0 ) {
03801 FileMake:
03802             PUTZERO(AN.OldPosOut);
03803             if ( ( fhandle = CreateFile(fout->name) ) < 0 ) {
03804                 MLOCK(ErrorMessageLock);
03805                 MesPrint("Cannot create file %s",fout->name);
03806                 MUNLOCK(ErrorMessageLock);
03807                 goto ReturnError;
03808             }
03809 #ifdef GZIPDEBUG
03810             MLOCK(ErrorMessageLock);
03811             MesPrint("%w MergePatches created output file %s",fout->name);
03812             MUNLOCK(ErrorMessageLock);
03813 #endif
03814             fout->handle = fhandle;
03815             PUTZERO(fout->filesize);
03816             PUTZERO(fout->POposition);
03817         /*
03818             Should not be here?
03819         #ifdef WITHZLIB
03820             fout->ziobuffer = 0;
03821         #endif
03822     */
03823 #ifdef ALLLOCK
03824             LOCK(fout->pthreadlock);
03825 #endif
03826             SeekFile(fout->handle,&(fout->filesize),SEEK_SET);
03827 #ifdef ALLLOCK
03828             UNLOCK(fout->pthreadlock);
03829 #endif
03830             S->fPatchN = 0;
03831             PUTZERO(S->fPatches[0]);
03832             fout->POfill = fout->PObuffer;
03833             PUTZERO(fout->POposition);
03834         }
03835 ConMer:
03836         StageSort(fout);
03837 #ifdef WITHZLIB
03838         if ( S == AT.S0 && AR.NoCompress == 0 && AR.zipCompress > 0 )
03839             S->fpcompressed[S->fPatchN] = 1;
03840         else
03841             S->fpcompressed[S->fPatchN] = 0;
03842         SetupOutputGZIP(fout);
03843 #endif
03844     }
03845     else if ( par == 0 && S->stage4 > 0 ) {
03846     /*
03847         We will have to do our job more than once.
03848         Input is from S->file and output will go to AR.FoStage4.
03849         The file corresponding to this last one must be made now.
03850     */
03851         AR.Stage4Name ^= 1;
03852     /*
03853         s = (UBYTE *) (fout->name); while ( *s ) s++;
03854         if ( AR.Stage4Name ) s[-1] += 1;
03855         else
03856             s[-1] -= 1;
03857     */
03857         S->iPatches = S->fPatches;
03858         S->fPatches = S->inPatches;
03859         S->inPatches = S->iPatches;
03860         (S->inNum) = S->fPatchN;
03861         AN.OldPosIn = AN.OldPosOut;
03862 #ifdef WITHZLIB
03863         m1 = S->fpincompressed;
03864         S->fpincompressed = S->fpcompressed;
03865         S->fpcompressed = m1;
03866         for ( i = 0; i < S->inNum; i++ ) {
03867             S->fPatchesStop[i] = S->iPatches[i+1];
03868 #ifdef GZIPDEBUG
03869             MLOCK(ErrorMessageLock);
03870             MesPrint("%w fPatchesStop[%d] = %l0p",i,&(S->fPatchesStop[i]));
03871             MUNLOCK(ErrorMessageLock);
03872 #endif
03873         }
03874 #endif
03875         S->stage4 = 0;
03876         goto FileMake;

```

```

03877     }
03878     else {
03879 #ifdef WITHZLIB
03880 /*
03881     The next statement is just for now
03882 */
03883     AR.gzipCompress = 0;
03884 #endif
03885     if ( par == 0 ) {
03886         S->iPatches = S->fPatches;
03887         S->inNum = S->fPatchN;
03888 #ifdef WITHZLIB
03889         m1 = S->fpincompressed;
03890         S->fpincompressed = S->fpcompressed;
03891         S->fpcompressed = m1;
03892         for ( i = 0; i < S->inNum; i++ ) {
03893             S->fPatchesStop[i] = S->fPatches[i+1];
03894 #ifdef GZIPDEBUG
03895             MLOCK(ErrorMessageLock);
03896             MesPrint("%w fPatchesStop[%d] = %10p", i, &(S->fPatchesStop[i]));
03897             MUNLOCK(ErrorMessageLock);
03898 #endif
03899         }
03900 #endif
03901     }
03902     fout = AR.outfile;
03903 }
03904 if ( par ) { /* Mark end of patches */
03905     S->Patches[S->lPatch] = S->lFill;
03906     for ( i = 0; i < S->lPatch; i++ ) {
03907         S->pStop[i] = S->Patches[i+1]-1;
03908         S->Patches[i] = (WORD *)(((UBYTE *) (S->Patches[i])) + AM.MaxTer);
03909     }
03910 }
03911 else { /* Load the patches */
03912     S->lPatch = (S->inNum);
03913 #ifdef WITHMPI
03914     if ( S->lPatch > 1 || ( (PF.exprtodo < 0) && (fout == AR.outfile || fout == AR.hidefile ) ) ) {
03915 #else
03916     if ( S->lPatch > 1 ) {
03917 #endif
03918 #ifdef WITHZLIB
03919         SetupAllInputGZIP(S);
03920 #endif
03921         p = S->lBuffer;
03922         for ( i = 0; i < S->lPatch; i++ ) {
03923             p = (WORD *)(((UBYTE *)p)+2*AM.MaxTer+COMPINC*sizeof(WORD));
03924             S->Patches[i] = p;
03925             p = (WORD *)(((UBYTE *)p) + fin->POsize);
03926             S->pStop[i] = m2 = p;
03927 #ifdef WITHZLIB
03928             PutIn(fin, &(S->iPatches[i]), S->Patches[i], &m2, i);
03929 #else
03930             ADDPOS(S->iPatches[i], PutIn(fin, &(S->iPatches[i]), S->Patches[i], &m2, i));
03931 #endif
03932         }
03933     }
03934 }
03935 if ( fout->handle >= 0 ) {
03936     PUTZERO(position);
03937 #ifdef ALLLOCK
03938     LOCK(fout->pthreadlock);
03939 #endif
03940     SeekFile(fout->handle, &position, SEEK_END);
03941     ADDPOS(position, ((fout->POfill-fout->PObuffer)*sizeof(WORD)));
03942 #ifdef ALLLOCK
03943     UNLOCK(fout->pthreadlock);
03944 #endif
03945 }
03946 else {
03947     SETBASEPOSITION(position, (fout->POfill-fout->PObuffer)*sizeof(WORD));
03948 }
03949 /*
03950     #] Setup :
03951
03952     The old code had to be replaced because all output needs to go
03953     through PutOut. For this we have to go term by term and keep
03954     track of the compression.
03955 */
03956 if ( S->lPatch == 1 ) { /* Single patch --> direct copy. Very rare. */
03957     LONG length;
03958
03959     if ( fout->handle < 0 ) if ( Sflush(fout) ) goto PatCall;
03960     if ( par ) { /* Memory to file */
03961 #ifdef WITHZLIB
03962 /*
03963         We fix here the problem that the thing needs to go through PutOut

```

```

03964 */
03965     m2 = m1 = *S->Patches; /* The m2 is to keep the compiler from complaining */
03966     while ( *m1 ) {
03967         if ( *m1 < 0 ) { /* Need to uncompress */
03968             i = -( *m1++ ); m2 += i; im = *m1+i+1;
03969             while ( i > 0 ) { *m1-- = *m2--; i--; }
03970             *m1 = im;
03971         }
03972 #ifndef WITHPTHREADS
03973         /* Check here (and in the following) that we are at ground level
03974         (so S == AT.S0) to use PutToMaster. Control can reach here,
03975         with AS.MasterSort, but with S != AT.S0, when the SortBots are
03976         adding PolyRatFun and the sorting of their arguments requires
03977         large buffer patches to be merged. In this case, terms escape
03978         the PolyRatFun argument and end up at ground level, if we use
03979         PutToMaster. */
03980         if ( AS.MasterSort && ( fout == AR.outfile ) && S == AT.S0 ) {
03981             im = PutToMaster(BHEAD m1);
03982         }
03983         else
03984 #endif
03985             if ( ( im = PutOut(BHEAD m1,&position,fout,1) ) < 0 ) goto ReturnError;
03986             ADDPOS(S->SizeInFile[par],im);
03987             m2 = m1;
03988             m1 += *m1;
03989         }
03990 #ifndef WITHPTHREADS
03991         if ( AS.MasterSort && ( fout == AR.outfile ) && S == AT.S0 ) {
03992             PutToMaster(BHEAD 0);
03993         }
03994         else
03995 #endif
03996             if ( FlushOut(&position,fout,1) ) goto ReturnError;
03997             ADDPOS(S->SizeInFile[par],1);
03998     #else
03999     /* old code */
04000     length = (LONG) (*S->pStop) - (LONG) (*S->Patches) + sizeof(WORD);
04001     if ( WriteFile(fout->handle, (UBYTE *) (*S->Patches), length) != length )
04002         goto PatwCall;
04003     ADDPOS(position, length);
04004     ADDPOS(fout->PPosition, length);
04005     ADDPOS(fout->filesize, length);
04006     ADDPOS(S->SizeInFile[par], length/sizeof(WORD));
04007 #endif
04008     }
04009     else { /* File to file */
04010 #ifndef WITHZLIB
04011     /*
04012     Note: if we change FRONTSIZE we need to make the minimum value
04013     of SmallSize in AllocSort correspondingly larger or smaller.
04014     Theoretically we could get close to 2*AM.MaxTer!
04015     */
04016     #define FRONTSIZE (2*AM.MaxTer)
04017     WORD *copybuf = (WORD *) (((UBYTE *) (S->sBuffer)) + FRONTSIZE);
04018     WORD *copytop;
04019     SetupAllInputGZIP(S);
04020     m1 = m2 = copybuf;
04021     position2 = S->iPatches[0];
04022     while ( ( length = FillInputGZIP(fin,&position2,
04023         (UBYTE *) copybuf,
04024         (S->SmallSize*sizeof(WORD)-FRONTSIZE), 0) ) > 0 ) {
04025         copytop = (WORD *) (((UBYTE *) copybuf) + length);
04026         while ( *m1 && ( ( *m1 > 0 && m1+m1 < copytop ) ||
04027             ( *m1 < 0 && ( m1+1 < copytop ) && ( m1+m1[1]+1 < copytop ) ) ) )
04028             /*
04029             22-jun-2013 JV Extremely nasty bug that has been around for a while.
04030             What if the end is in the remaining part? We will loose terms!
04031             while ( *m1 && ( (WORD *) (((UBYTE *) (m1)) + AM.MaxTer ) < S->sTop2 ) )
04032             */
04033             {
04034                 if ( *m1 < 0 ) { /* Need to uncompress */
04035                     i = -( *m1++ ); m2 += i; im = *m1+i+1;
04036                     while ( i > 0 ) { *m1-- = *m2--; i--; }
04037                     *m1 = im;
04038                 }
04039 #ifndef WITHPTHREADS
04040                 if ( AS.MasterSort && ( fout == AR.outfile ) && S == AT.S0 ) {
04041                     im = PutToMaster(BHEAD m1);
04042                 }
04043                 else
04044 #endif
04045                     if ( ( im = PutOut(BHEAD m1,&position,fout,1) ) < 0 ) goto ReturnError;
04046                     ADDPOS(S->SizeInFile[par],im);
04047                     m2 = m1;
04048                     m1 += *m1;
04049                 }
04050             if ( m1 < copytop && *m1 == 0 ) break;

```



```

04051 /*
04052         Now move the remaining part 'back'
04053 */
04054         m3 = copybuf;
04055         m1 = copytop;
04056         while ( m1 > m2 ) *--m3 = *--m1;
04057         m2 = m3;
04058         m1 = m2 + *m2;
04059     }
04060     if ( length < 0 ) {
04061         MLOCK(ErrorMessageLock);
04062         MesPrint("Readerror");
04063         goto PatCall2;
04064     }
04065 #ifdef WITHPTHREADS
04066     if ( AS.MasterSort && ( fout == AR.outfile ) && S == AT.S0 ) {
04067         PutToMaster(BHEAD 0);
04068     }
04069     else
04070 #endif
04071     if ( FlushOut(&position,fout,1) ) goto ReturnError;
04072     ADDPOS(S->SizeInFile[par],1);
04073 #else
04074 /* old code */
04075     SeekFile(fin->handle,&(S->iPatches[0]),SEEK_SET); /* needed for stage4 */
04076     while ( ( length = ReadFile(fin->handle,
04077         (UBYTE *) (S->sBuffer),S->SmallEsize*sizeof(WORD)) ) > 0 ) {
04078         if ( WriteFile(fout->handle,(UBYTE *) (S->sBuffer),length) != length )
04079             goto PatwCall;
04080         ADDPOS(position,length);
04081         ADDPOS(fout->PPosition,length);
04082         ADDPOS(fout->filesize,length);
04083         ADDPOS(S->SizeInFile[par],length/sizeof(WORD));
04084     }
04085     if ( length < 0 ) {
04086         MLOCK(ErrorMessageLock);
04087         MesPrint("Readerror");
04088         goto PatCall2;
04089     }
04090 #endif
04091     }
04092     goto EndOfAll;
04093 }
04094 else if ( S->lPatch > 0 ) {
04095     /* More than one patch. Construct the tree. */
04096     lpat = 1;
04097     do { lpat *= 2; } while ( lpat < S->lPatch );
04098     mpat = ( lpat » 1 ) - 1;
04099     k = lpat - S->lPatch;
04100     /* k is the number of empty places in the tree. they will
04101        be at the even positions from 2 to 2*k */
04102     for ( i = 1; i < lpat; i++ ) {
04103         S->tree[i] = -1;
04104     }
04105     for ( i = 1; i <= k; i++ ) {
04106         im = ( i * 2 ) - 1;
04107         poin[im] = S->Patches[i-1];
04108         poin2[im] = poin[im] + *(poin[im]);
04109         S->used[i] = im;
04110         S->ktoi[im] = i-1;
04111         S->tree[mpat+i] = 0;
04112         poin[im-1] = poin2[im-1] = 0;
04113     }
04114     for ( i = (k*2)+1; i <= lpat; i++ ) {
04115         S->used[i-k] = i;
04116         S->ktoi[i] = i-k-1;
04117         poin[i] = S->Patches[i-k-1];
04118         poin2[i] = poin[i] + *(poin[i]);
04119     }
04120 }
04121 /*
04122 the array poin tells the position of the i-th element of the S->tree
04123 'S->used' is a stack with the S->tree elements that need to be entered
04124 into the S->tree. at the beginning this is S->lPatch. during the
04125 sort there will be only very few elements.
04126 poin2 is the next value of poin. it has to be determined
04127 before the comparisons as the position or the size of the
04128 term indicated by poin may change.
04129 S->ktoi translates a S->tree element back to its stream number.
04130
04131 start the sort
04132 */
04133 level = S->lPatch;
04134
04135
04136
04137

```

```

04138      /* introduce one term */
04139 OneTerm:
04140     k = S->used[level];
04141     i = k + lpat - 1;
04142     if ( !*(poin[k]) ) {
04143         do { if ( !( i >= 1 ) ) goto EndOfMerge; } while ( !S->tree[i] );
04144         if ( S->tree[i] == -1 ) {
04145             S->tree[i] = 0;
04146             level--;
04147             goto OneTerm;
04148         }
04149         k = S->tree[i];
04150         S->used[level] = k;
04151         S->tree[i] = 0;
04152     }
04153 /*
04154     move terms down the tree
04155 */
04156     while ( i >= 1 ) {
04157         if ( S->tree[i] > 0 ) {
04158             if ( ( c = CompareTerms(BHEAD poin[S->tree[i]],poin[k],(WORD)0) ) > 0 ) {
04159 /*
04160                 S->tree[i] is the smaller. Exchange and go on.
04161 */
04162                 S->used[level] = S->tree[i];
04163                 S->tree[i] = k;
04164                 k = S->used[level];
04165             }
04166             else if ( !c ) { /* Terms are equal */
04167                 S->TermsLeft--;
04168 /*
04169                 Here the terms are equal and their coefficients
04170                 have to be added.
04171 */
04172                 l1 = *( m1 = poin[S->tree[i]] );
04173                 l2 = *( m2 = poin[k] );
04174                 if ( S->PolyWise ) { /* Here we work with PolyFun */
04175                     WORD *ttl, *w;
04176                     ttl = m1;
04177                     m1 += S->PolyWise;
04178                     m2 += S->PolyWise;
04179                     if ( S->PolyFlag == 2 ) {
04180                         w = poly_ratfun_add(BHEAD m1,m2);
04181                         if ( *ttl + w[l1] - m1[l1] > AM.MaxTer/((LONG)sizeof(WORD)) ) {
04182                             MLOCK(ErrorMessageLock);
04183                             MesPrint("Term too complex in PolyRatFun addition. MaxTermSize of %10l
04184 is too small",AM.MaxTer);
04185                             MUNLOCK(ErrorMessageLock);
04186                             Terminate(-1);
04187                         }
04188                         AT.WorkPointer = w;
04189                     }
04190                     else {
04191                         w = AT.WorkPointer;
04192                         if ( w + m1[l1] + m2[l1] > AT.WorkTop ) {
04193                             MLOCK(ErrorMessageLock);
04194                             MesPrint("A Workspace of %10l is too small",AM.WorkSize);
04195                             MUNLOCK(ErrorMessageLock);
04196                             Terminate(-1);
04197                         }
04198                         AddArgs(BHEAD m1,m2,w);
04199                     }
04200                     r1 = w[l1];
04201                     if ( r1 <= FUNHEAD
04202                         || ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 ) ) {
04203                         { goto cancelled; }
04204                     if ( r1 == m1[l1] ) {
04205                         NCOPY(m1,w,r1);
04206                     }
04207                     else if ( r1 < m1[l1] ) {
04208                         r2 = m1[l1] - r1;
04209                         m2 = w + r1;
04210                         m1 += m1[l1];
04211                         while ( --r1 >= 0 ) *--m1 = *--m2;
04212                         m2 = m1 - r2;
04213                         r1 = S->PolyWise;
04214                         while ( --r1 >= 0 ) *--m1 = *--m2;
04215                         *m1 -= r2;
04216                         poin[S->tree[i]] = m1;
04217                     }
04218                     else {
04219                         r2 = r1 - m1[l1];
04220                         m2 = ttl - r2;
04221                         r1 = S->PolyWise;
04222                         m1 = ttl;
04223                         *m1 += r2;
04224                         poin[S->tree[i]] = m2;

```

```

04224             NCOPY(m2,m1,r1);
04225             r1 = w[1];
04226             NCOPY(m2,w,r1);
04227         }
04228     }
04229 #ifdef WITHFLOAT
04230     else if ( AT.SortFloatMode ) {
04231         WORD *term1, *term2;
04232         term1 = poin[S->tree[i]];
04233         term2 = poin[k];
04234         if ( MergeWithFloat(BHEAD &term1,&term2) == 0 )
04235             goto cancelled;
04236         poin[S->tree[i]] = term1;
04237     }
04238 #endif
04239     else {
04240         r1 = *( m1 += l1 - 1 );
04241         m1 -= ABS(r1) - 1;
04242         r1 = ( ( r1 > 0 ) ? (r1-1) : (r1+1) ) >> 1;
04243         r2 = *( m2 += l2 - 1 );
04244         m2 -= ABS(r2) - 1;
04245         r2 = ( ( r2 > 0 ) ? (r2-1) : (r2+1) ) >> 1;
04246
04247         if ( AddRat(BHEAD (UWORD *)m1,r1,(UWORD *)m2,r2,coef,&r3) ) {
04248             MLOCK(ErrorMessageLock);
04249             MesCall("MergePatches");
04250             MUNLOCK(ErrorMessageLock);
04251             SETERROR(-1)
04252         }
04253
04254         if ( AN.ncmod != 0 ) {
04255             if ( ( AC.modmode & POSNEG ) != 0 ) {
04256                 NormalModulus(coef,&r3);
04257             }
04258             else if ( BigLong(coef,r3,(UWORD *)AC.cmod,ABS(AN.ncmod)) >= 0 ) {
04259                 WORD ii;
04260                 SubPLon(coef,r3,(UWORD *)AC.cmod,ABS(AN.ncmod),coef,&r3);
04261                 coef[r3] = 1;
04262                 for ( ii = 1; ii < r3; ii++ ) coef[r3+ii] = 0;
04263             }
04264         }
04265         r3 *= 2;
04266         r33 = ( r3 > 0 ) ? ( r3 + 1 ) : ( r3 - 1 );
04267         if ( r3 < 0 ) r3 = -r3;
04268         if ( r1 < 0 ) r1 = -r1;
04269         r1 *= 2;
04270         r31 = r3 - r1;
04271         if ( !r3 ) { /* Terms cancel */
04272 cancelled:
04273             ul = S->used[level] = S->tree[i];
04274             S->tree[i] = -1;
04275             /*
04276              We skip to the next term in stream ul
04277             */
04278             im = *poin2[ul];
04279             if ( im < 0 ) {
04280                 r1 = poin2[ul][1] - im + 1;
04281                 m1 = poin2[ul] + 2;
04282                 m2 = poin[ul] - im + 1;
04283                 while ( ++im <= 0 ) *--m1 = *--m2;
04284                 *--m1 = r1;
04285                 poin2[ul] = m1;
04286                 im = r1;
04287             }
04288             poin[ul] = poin2[ul];
04289             ki = S->ktoi[ul];
04290             if ( !par && (poin[ul] + im + COMPINC) >= S->pStop[ki]
04291                 && im > 0 ) {
04292                 PutIn(fin,&(S->iPatches[ki]),S->Patches[ki],&(poin[ul]),ki);
04293             }
04294             else
04295                 ADDPOS(S->iPatches[ki],PutIn(fin,&(S->iPatches[ki]),
04296                     S->Patches[ki],&(poin[ul]),ki));
04297             #endif
04298             poin2[ul] = poin[ul] + im;
04299         }
04300         else {
04301             poin2[ul] += im;
04302         }
04303         S->used[++level] = k;
04304         S->TermsLeft--;
04305     }
04306     else if ( !r31 ) { /* copy coef into term1 */
04307         goto CopCof2;
04308     }
04309     else if ( r31 < 0 ) { /* copy coef into term1
04310                          and adjust the length of term1 */

```

```

04311         goto CopCoef;
04312     }
04313     else {
04314         /*
04315             this is the dreaded calamity.
04316             is there enough space?
04317         */
04318         if( (poin[S->tree[i]]+l1+r31) >= poin2[S->tree[i]] ) {
04319             /*
04320                 no space! now the special trick for which
04321                 we left 2*maxlng spaces open at the beginning
04322                 of each patch.
04323             */
04324             if ( (l1 + r31) > AM.MaxTer/((LONG)sizeof(WORD)) ) {
04325                 MLOCK(ErrorMessageLock);
04326                 MesPrint("Coefficient overflow during sort");
04327                 MUNLOCK(ErrorMessageLock);
04328                 goto ReturnError;
04329             }
04330             m2 = poin[S->tree[i]];
04331             m3 = ( poin[S->tree[i]] - r31 );
04332             do { *m3++ = *m2++; } while ( m2 < m1 );
04333             m1 = m3;
04334         }
04335         CopCoef:
04336             *(poin[S->tree[i]]) += r31;
04337         CopCof2:
04338             m2 = (WORD *)coef; im = r3;
04339             NCOPY(m1,m2,im);
04340             *m1 = r33;
04341         }
04342     }
04343     /*
04344         Now skip to the next term in stream k.
04345     */
04346     NextTerm:
04347         im = poin2[k][0];
04348         if ( im < 0 ) {
04349             r1 = poin2[k][1] - im + 1;
04350             m1 = poin2[k] + 2;
04351             m2 = poin[k] - im + 1;
04352             while ( ++im <= 0 ) *--m1 = *--m2;
04353             *--m1 = r1;
04354             poin2[k] = m1;
04355             im = r1;
04356         }
04357         poin[k] = poin2[k];
04358         ki = S->ktoi[k];
04359         if ( !par && ( (poin[k] + im + COMPINC) >= S->pStop[ki] )
04360             && im > 0 ) {
04361             #ifdef WITHZLIB
04362                 PutIn(fin,&(S->iPatches[ki]),S->Patches[ki],&(poin[k]),ki);
04363             #else
04364                 ADDPOS(S->iPatches[ki],PutIn(fin,&(S->iPatches[ki]),
04365                     S->Patches[ki],&(poin[k]),ki));
04366             #endif
04367             poin2[k] = poin[k] + im;
04368         }
04369         else {
04370             poin2[k] += im;
04371         }
04372         goto OneTerm;
04373     }
04374 }
04375 else if ( S->tree[i] < 0 ) {
04376     S->tree[i] = k;
04377     level--;
04378     goto OneTerm;
04379 }
04380 }
04381 /*
04382     found the smallest in the set. indicated by k.
04383     write to its destination.
04384 */
04385 #ifdef WITHPTHREADS
04386     if ( AS.MasterSort && ( fout == AR.outfile ) && S == AT.S0 ) {
04387         im = PutToMaster(BHEAD poin[k]);
04388     }
04389     else
04390 #endif
04391     if ( ( im = PutOut(BHEAD poin[k],&position,fout,1) ) < 0 ) {
04392         MLOCK(ErrorMessageLock);
04393         MesPrint("Called from MergePatches with k = %d (stream %d)",k,S->ktoi[k]);
04394         MUNLOCK(ErrorMessageLock);
04395         goto ReturnError;
04396     }
04397     ADDPOS(S->SizeInFile[par],im);

```

```

04398         goto NextTerm;
04399     }
04400     else {
04401         goto NormalReturn;
04402     }
04403 EndOfMerge:
04404 #ifdef WITHPTHREADS
04405     if ( AS.MasterSort && ( fout == AR.outfile ) && S == AT.S0 ) {
04406         PutToMaster(BHEAD 0);
04407     }
04408     else
04409 #endif
04410     if ( FlushOut(&position,fout,1) ) goto ReturnError;
04411     ADDPOS(S->SizeInFile[par],1);
04412 EndOfAll:
04413     if ( par == 1 ) { /* Set the fpatch pointers */
04414 #ifdef WITHZLIB
04415         SeekFile(fout->handle,&position,SEEK_CUR);
04416 #endif
04417         (S->fPatchN)++;
04418         S->fPatches[S->fPatchN] = position;
04419     }
04420     if ( par == 0 && fout != AR.outfile ) {
04421 /*
04422         Output went to sortfile. We have two possibilities:
04423         1: We are not finished with the current in-out cycle
04424            In that case we should pop to the next set of patches
04425         2: We finished a cycle and should clean up the in file
04426            Then we restart the sort.
04427 */
04428         (S->fPatchN)++;
04429         S->fPatches[S->fPatchN] = position;
04430         if ( ISNOTZEROPOS(AN.OldPosIn) ) { /* We are not done */
04431
04432             SeekFile(fin->handle,&(AN.OldPosIn),SEEK_SET);
04433 /*
04434             We don't need extra provisions for the zlib compression here.
04435             If part of an expression has been sorted, the whole has been so.
04436             This means that S->fpincompressed[] will remain the same
04437 */
04438             if ( (ULONG)ReadFile(fin->handle,(UBYTE *)(&(S->inNum)),(LONG)sizeof(WORD)) !=
04439                 sizeof(WORD)
04440             || (ULONG)ReadFile(fin->handle,(UBYTE *)(&AN.OldPosIn),(LONG)sizeof(POSITION)) !=
04441                 sizeof(POSITION)
04442             || (ULONG)ReadFile(fin->handle,(UBYTE *)S->iPatches,(LONG)((S->inNum)+1)
04443                 *sizeof(POSITION)) != ((S->inNum)+1)*sizeof(POSITION) ) {
04444                 MLOCK(ErrorMessageLock);
04445                 MesPrint("Read error fourth stage sorting");
04446                 MUNLOCK(ErrorMessageLock);
04447                 goto ReturnError;
04448             }
04449             *rr = 0;
04450 #ifdef WITHZLIB
04451             for ( i = 0; i < S->inNum; i++ ) {
04452                 S->fPatchesStop[i] = S->iPatches[i+1];
04453 #ifdef GZIPDEBUG
04454                 MLOCK(ErrorMessageLock);
04455                 MesPrint("%w fPatchesStop[%d] = %10p",i,&(S->fPatchesStop[i]));
04456                 MUNLOCK(ErrorMessageLock);
04457 #endif
04458             }
04459 #endif
04460             goto ConMer;
04461         }
04462     }
04463 /*
04464     if ( fin == &(AR.FoStage4[0]) ) {
04465         s = (UBYTE *) (fin->name); while ( *s ) s++;
04466         if ( AR.Stage4Name == 1 ) s[-1] -= 1;
04467         else
04468             s[-1] += 1;
04469     }
04470 */
04471 /* TruncateFile(fin->handle); */
04472     UpdateMaxSize();
04473 #ifdef WITHZLIB
04474     ClearSortGZIP(fin);
04475 #endif
04476     CloseFile(fin->handle);
04477     remove(fin->name); /* Gives disk space free again. */
04478 #ifdef GZIPDEBUG
04479     MLOCK(ErrorMessageLock);
04480     MesPrint("%w MergePatches removed in file %s",fin->name);
04481     MUNLOCK(ErrorMessageLock);
04482 #endif
04483 /*
04484     if ( fin == &(AR.FoStage4[0]) ) {
04485         s = (UBYTE *) (fin->name); while ( *s ) s++;

```

```

04485             if ( AR.Stage4Name == 1 ) s[-1] += 1;
04486             else s[-1] -= 1;
04487         }
04488     /*
04489         fin->handle = -1;
04490         { FILEHANDLE *ff = fin; fin = fout; fout = ff; }
04491         PUTZERO(S->SizeInFile[0]);
04492         goto NewMerge;
04493     }
04494 }
04495 if ( par == 0 ) {
04496     /* TruncateFile(fin->handle); */
04497     UpdateMaxSize();
04498     #ifdef WITHZLIB
04499     ClearSortGZIP(fin);
04500 #endif
04501     CloseFile(fin->handle);
04502     remove(fin->name);
04503     fin->handle = -1;
04504     #ifdef GZIPDEBUG
04505     MLOCK(ErrorMessageLock);
04506     MesPrint("%w MergePatches removed in file %s", fin->name);
04507     MUNLOCK(ErrorMessageLock);
04508 #endif
04509 }
04510 NormalReturn:
04511 #ifdef WITHZLIB
04512     AR.gzipCompress = oldgzipCompress;
04513 #endif
04514     return(0);
04515 ReturnError:
04516 #ifdef WITHZLIB
04517     AR.gzipCompress = oldgzipCompress;
04518 #endif
04519     return(-1);
04520 #ifndef WITHZLIB
04521 PatwCall:
04522     MLOCK(ErrorMessageLock);
04523     MesPrint("Error while writing to file.");
04524     goto PatCall2;
04525 #endif
04526 PatCall:;
04527     MLOCK(ErrorMessageLock);
04528 PatCall2:;
04529     MesCall("MergePatches");
04530     MUNLOCK(ErrorMessageLock);
04531     #ifdef WITHZLIB
04532     AR.gzipCompress = oldgzipCompress;
04533 #endif
04534     SETERROR(-1)
04535 }
04536
04537 /*
04538     #] MergePatches :
04539     #[ StoreTerm : WORD StoreTerm(term)
04540 */
04550 int StoreTerm(PHEAD WORD *term)
04551 {
04552     GETBIDENTITY
04553     SORTING *S = AT.SS;
04554     WORD **ss, *lfill, j, *t;
04555     POSITION pp;
04556     LONG lSpace, sSpace, RetCode, over, tover;
04557
04558     if ( ( ( AP.PreDebug & DUMPTOSORT ) == DUMPTOSORT ) && AR.sLevel == 0 ) {
04559     #ifdef WITHPTHREADS
04560         snprintf((char *) (THRbuf), 100, "StoreTerm(%d)", AT.identity);
04561         PrintTerm(term, (char *) (THRbuf));
04562     #else
04563         PrintTerm(term, "StoreTerm");
04564     #endif
04565     }
04566     if ( AM.exitflag && AR.sLevel == 0 ) return(0);
04567     S->sFill = *(S->PoinFill);
04568     if ( S->sTerms >= S->TermsInSmall || ( S->sFill + *term ) >= S->sTop ) {
04569     /*
04570         The small buffer is full. It has to be sorted and written.
04571     */
04572         tover = over = S->sTerms;
04573         ss = S->sPointer;
04574         ss[over] = 0;
04575     #ifdef SPLITTIME
04576         PrintTime((UBYTE *) "Before SplitMerge");
04577     #endif
04578         ss[SplitMerge(BHEAD ss, over)] = 0;
04579     #ifdef SPLITTIME
04580         PrintTime((UBYTE *) "After SplitMerge");

```

```

04581 #endif
04582     sSpace = 0;
04583     if ( over > 0 ) {
04584         sSpace = ComPress(ss,&RetCode);
04585         S->TermsLeft -= over - RetCode;
04586     }
04587     sSpace++;
04588
04589     lSpace = sSpace + (S->lFill - S->lBuffer)
04590         - (AM.MaxTer/sizeof(WORD))*((LONG)S->lPatch);
04591     SETBASEPOSITION(pp,lSpace);
04592     MULPOS(pp,sizeof(WORD));
04593     if ( S->file.handle >= 0 ) {
04594         ADD2POS(pp,S->fPatches[S->fPatchN]);
04595     }
04596     if ( S == AT.S0 ) { /* Only statistics at ground level */
04597         WriteStats(&pp,STATSSPLITMERGE,CHECKLOGTYPE);
04598     }
04599     if ( ( S->lPatch >= S->MaxPatches ) ||
04600         ( ( (WORD *)(((UBYTE *) (S->lFill + sSpace)) + 2*AM.MaxTer ) ) >= S->lTop ) ) {
04601 /*
04602     The large buffer is too full. Merge and write it
04603 */
04604     if ( MergePatches(1) ) goto StoreCall;
04605 /*
04606     pp = S->SizeInFile[1];
04607     ADDPOS(pp,sSpace);
04608     MULPOS(pp,sizeof(WORD));
04609 */
04610     SETBASEPOSITION(pp,sSpace);
04611     MULPOS(pp,sizeof(WORD));
04612     ADD2POS(pp,S->fPatches[S->fPatchN]);
04613
04614     if ( S == AT.S0 ) { /* Only statistics at ground level */
04615         WriteStats(&pp,STATSMERGETOFILE,CHECKLOGTYPE);
04616     }
04617     S->lPatch = 0;
04618     S->lFill = S->lBuffer;
04619 }
04620 S->Patches[S->lPatch++] = S->lFill;
04621 lfill = (WORD *)(((UBYTE *) (S->lFill)) + AM.MaxTer);
04622 if ( tover > 0 ) {
04623     ss = S->sPointer;
04624     while ( ( t = *ss++ ) != 0 ) {
04625         j = *t;
04626         if ( j < 0 ) j = t[1] + 2;
04627         while ( --j >= 0 ){
04628             *lfill++ = *t++;
04629         }
04630     }
04631 }
04632 *lfill++ = 0;
04633 S->lFill = lfill;
04634 S->sTerms = 0;
04635 S->PoinFill = S->sPointer;
04636 *(S->PoinFill) = S->sFill = S->sBuffer;
04637 }
04638 j = *term;
04639 while ( --j >= 0 ) *S->sFill++ = *term++;
04640 S->sTerms++;
04641 S->GenTerms++;
04642 S->TermsLeft++;
04643 **S->PoinFill = S->sFill;
04644
04645     return(0);
04646
04647 StoreCall:
04648     MLOCK(ErrorMessageLock);
04649     MesCall("StoreTerm");
04650     MUNLOCK(ErrorMessageLock);
04651     SETERROR(-1)
04652 }
04653
04654 /*
04655     #] StoreTerm :
04656     #[ StageSort :                void StageSort(FILEHANDLE *fout)
04657 */
04664 void StageSort(FILEHANDLE *fout)
04665 {
04666     GETIDENTITY
04667     SORTING *S = AT.SS;
04668     if ( S->fPatchN >= S->MaxFpatches ) {
04669         POSITION position;
04670         if ( S != AT.S0 ) {
04671 /*
04672     There are no proper provisions for stage 4 or higher sorts
04673     for function arguments and $ variables. The reason:

```

```

04674         The current code maps out the patches, based on the size of
04675         the buffers in the FoStage4 structs, while they are used
04676         inside the S->file struct that may have far smaller buffers.
04677         By itself that might still be repairable, but it goes completely
04678         wrong when during the sort polyRatFuns have to be added and they
04679         would go into stage4 (very rare but possible).
04680         The only really correct solution would be to put FoStage4 structs
04681         in all sort levels. Messy. (JV 8-oct-2018).
04682     */
04683     MLOCK(ErrorMessageLock);
04684     MesPrint("Currently Stage 4 sorts are not allowed for function arguments or $
variables.");
04685     MesPrint("Please increase correspondingsorting parameters (sub-) in the setup.");
04686     MUNLOCK(ErrorMessageLock);
04687     Terminate(-1);
04688 }
04689 PUTZERO(position);
04690 MLOCK(ErrorMessageLock);
04691 #ifdef WITHPTHREADS
04692     MesPrint("StageSort in thread %d",identity);
04693 #elif defined(WITHMPI)
04694     MesPrint("StageSort in process %d",PF.me);
04695 #else
04696     MesPrint("StageSort");
04697 #endif
04698     MUNLOCK(ErrorMessageLock);
04699     SeekFile(fout->handle,&position,SEEK_END);
04700 /*
04701     No extra compression data has to be written.
04702     S->fpincompressed should remain valid.
04703 */
04704     if ( (ULONG)WriteFile(fout->handle, (UBYTE *) (&(S->fPatchN)), (LONG) sizeof(WORD)) !=
sizeof(WORD)
|| (ULONG)WriteFile(fout->handle, (UBYTE *) (&(AN.OldPosOut)), (LONG) sizeof(POSITION)) !=
sizeof(POSITION)
|| (ULONG)WriteFile(fout->handle, (UBYTE *) (S->fPatches), (LONG) (S->fPatchN+1)
*sizeof(POSITION)) != (S->fPatchN+1)*sizeof(POSITION) ) {
04705         MLOCK(ErrorMessageLock);
04706         MesPrint("Write error while staging sort. Disk full?");
04707         MUNLOCK(ErrorMessageLock);
04708         Terminate(-1);
04709     }
04710     AN.OldPosOut = position;
04711     fout->filesize = position;
04712     ADDPOS(fout->filesize, (S->fPatchN+2)*sizeof(POSITION) + sizeof(WORD));
04713     fout->PPosition = fout->filesize;
04714     S->fPatches[0] = fout->filesize;
04715     S->fPatchN = 0;
04716
04717     if ( AR.FoStage4[0].PObuffer == 0 ) {
04718         AR.FoStage4[0].PObuffer = (WORD *)Malloc1(AR.FoStage4[0].POsize*sizeof(WORD)
,"Stage 4 buffer");
04719         AR.FoStage4[0].POfill = AR.FoStage4[0].PObuffer;
04720         AR.FoStage4[0].POstop = AR.FoStage4[0].PObuffer
+ AR.FoStage4[0].POsize/sizeof(WORD);
04721 #ifdef WITHPTHREADS
04722         AR.FoStage4[0].pthreadslock = dummylock;
04723 #endif
04724     }
04725     if ( AR.FoStage4[1].PObuffer == 0 ) {
04726         AR.FoStage4[1].PObuffer = (WORD *)Malloc1(AR.FoStage4[1].POsize*sizeof(WORD)
,"Stage 4 buffer");
04727         AR.FoStage4[1].POfill = AR.FoStage4[1].PObuffer;
04728         AR.FoStage4[1].POstop = AR.FoStage4[1].PObuffer
+ AR.FoStage4[1].POsize/sizeof(WORD);
04729 #ifdef WITHPTHREADS
04730         AR.FoStage4[1].pthreadslock = dummylock;
04731 #endif
04732     }
04733     S->stage4 = 1;
04734 }
04735
04736 /*
04737     #] StageSort :
04738     #[ SortWild :                WORD SortWild(w,nw)
04739 */
04740 int SortWild(WORD *w, WORD nw)
04741 {
04742     GETIDENTITY
04743     WORD *v, *s, *m, k, i;
04744     WORD *pScrat, *stop, *sv;
04745     int error = 0;
04746     pScrat = AT.WorkPointer;
04747     if ( ( AT.WorkPointer + 8 * AM.MaxWildcards ) >= AT.WorkTop ) {
04748         MLOCK(ErrorMessageLock);
04749         MesWork();

```



```

04773     MUNLOCK(ErrorMessageLock);
04774     return(-1);
04775 }
04776 stop = w + nw;
04777 i = 0;
04778 while ( i < nw ) {
04779     m = w + i;
04780     v = m + m[1];
04781     while ( v < stop && (
04782         *v == FROMSET || *v == SETTONUM || *v == LOADDOLLAR ) ) v += v[1];
04783     while ( v < stop ) {
04784         if ( *v >= 0 ) {
04785             if ( AM.Ordering[*v] < AM.Ordering[*m] ) {
04786                 m = v;
04787             }
04788             else if ( *v == *m ) {
04789                 if ( v[2] < m[2] ) {
04790                     m = v;
04791                 }
04792             }
04793             else if ( v[2] == m[2] ) {
04794                 s = m + m[1];
04795                 sv = v + v[1];
04796                 if ( s < stop && ( *s == FROMSET
04797                     || *s == SETTONUM || *s == LOADDOLLAR ) ) {
04798                     if ( sv < stop && ( *sv == FROMSET
04799                         || *sv == SETTONUM || *sv == LOADDOLLAR ) ) {
04800                         if ( s[2] != sv[2] ) {
04801                             error = -1;
04802                             MLOCK(ErrorMessageLock);
04803                             MesPrint("&Wildcard set conflict");
04804                             MUNLOCK(ErrorMessageLock);
04805                         }
04806                     }
04807                     *v = -1;
04808                 }
04809                 else {
04810                     if ( sv < stop && ( *sv == FROMSET
04811                         || *sv == SETTONUM || *sv == LOADDOLLAR ) ) {
04812                         *m = -1;
04813                         m = v;
04814                     }
04815                     else {
04816                         *v = -1;
04817                     }
04818                 }
04819             }
04820         }
04821         v += v[1];
04822         while ( v < stop && ( *v == FROMSET
04823             || *v == SETTONUM || *v == LOADDOLLAR ) ) v += v[1];
04824     }
04825     s = pScrat;
04826     v = m;
04827     k = m[1];
04828     NCOPY(s,m,k);
04829     while ( m < stop && ( *m == FROMSET
04830         || *m == SETTONUM || *m == LOADDOLLAR ) ) {
04831         k = m[1];
04832         NCOPY(s,m,k);
04833     }
04834     *v = -1;
04835     pScrat = s;
04836     i = 0;
04837     while ( i < nw && ( w[i] < 0 || w[i] == FROMSET
04838         || w[i] == SETTONUM || w[i] == LOADDOLLAR ) ) i += w[i+1];
04839 }
04840 AC.NwildC = k = WORDDIF(pScrat,AT.WorkPointer);
04841 s = AT.WorkPointer;
04842 m = w;
04843 NCOPY(m,s,k);
04844 AC.WildC = m;
04845 return(error);
04846 }
04847
04848 /*
04849     #] SortWild :
04850     #[ CleanUpSort :          void CleanUpSort(num)
04851 */
04852 void CleanUpSort(int num)
04853 {
04854     GETIDENTITY
04855     SORTING *S;
04856     int minnum = num, i;
04857     if ( AN.FunSorts ) {
04858         if ( num == -1 ) {
04859             if ( AN.MaxFunSorts > 3 ) {

```

```

04864         minnum = (AN.MaxFunSorts+4)/2;
04865     }
04866     else minnum = 4;
04867 }
04868 else if ( minnum == 0 ) minnum = 1;
04869 for ( i = minnum; i < AN.NumFunSorts; i++ ) {
04870     S = AN.FunSorts[i];
04871     if ( S ) {
04872         if ( S->file.handle >= 0 ) {
04873             /* TruncateFile(S->file.handle); */
04874             UpdateMaxSize();
04875 #ifdef WITHZLIB
04876             ClearSortGZIP (&(S->file));
04877 #endif
04878             CloseFile(S->file.handle);
04879             S->file.handle = -1;
04880             remove(S->file.name);
04881 #ifdef GZIPDEBUG
04882             MLOCK(ErrorMessageLock);
04883             MesPrint("%w CleanUpSort removed file %s",S->file.name);
04884             MUNLOCK(ErrorMessageLock);
04885 #endif
04886         }
04887         M_free(S->sPointer, "CleanUpSort: sPointer");
04888         M_free(S->Patches, "CleanUpSort: Patches");
04889         M_free(S->pStop, "CleanUpSort: pStop");
04890         M_free(S->poina, "CleanUpSort: poina");
04891         M_free(S->poin2a, "CleanUpSort: poin2a");
04892         M_free(S->fPatches, "CleanUpSort: fPatches");
04893         M_free(S->fPatchesStop, "CleanUpSort: fPatchesStop");
04894         M_free(S->inPatches, "CleanUpSort: inPatches");
04895         M_free(S->tree, "CleanUpSort: tree");
04896         M_free(S->used, "CleanUpSort: used");
04897 #ifdef WITHZLIB
04898         M_free(S->fpcompressed, "CleanUpSort: fpcompressed");
04899         M_free(S->fpincompressed, "CleanUpSort: fpincompressed");
04900 #endif
04901         M_free(S->ktoi, "CleanUpSort: ktoi");
04902         M_free(S->lBuffer, "CleanUpSort: lBuffer+sBuffer");
04903         M_free(S->file.PObuffer, "CleanUpSort: PObuffer");
04904         M_free(S, "CleanUpSort: sorting struct");
04905     }
04906     AN.FunSorts[i] = 0;
04907 }
04908 AN.MaxFunSorts = minnum;
04909 if ( num == 0 ) {
04910     S = AN.FunSorts[0];
04911     if ( S ) {
04912         if ( S->file.handle >= 0 ) {
04913             /* TruncateFile(S->file.handle); */
04914             UpdateMaxSize();
04915 #ifdef WITHZLIB
04916             ClearSortGZIP (&(S->file));
04917 #endif
04918             CloseFile(S->file.handle);
04919             S->file.handle = -1;
04920             remove(S->file.name);
04921 #ifdef GZIPDEBUG
04922             MLOCK(ErrorMessageLock);
04923             MesPrint("%w CleanUpSort removed file %s",S->file.name);
04924             MUNLOCK(ErrorMessageLock);
04925 #endif
04926         }
04927     }
04928 }
04929 }
04930 for ( i = 0; i < 2; i++ ) {
04931     if ( AR.FoStage4[i].handle >= 0 ) {
04932         UpdateMaxSize();
04933 #ifdef WITHZLIB
04934         ClearSortGZIP (&(AR.FoStage4[i]));
04935 #endif
04936         CloseFile(AR.FoStage4[i].handle);
04937         remove(AR.FoStage4[i].name);
04938         AR.FoStage4[i].handle = -1;
04939 #ifdef GZIPDEBUG
04940         MLOCK(ErrorMessageLock);
04941         MesPrint("%w CleanUpSort removed stage4 file %s",AR.FoStage4[i].name);
04942         MUNLOCK(ErrorMessageLock);
04943 #endif
04944     }
04945 }
04946 }
04947
04948 /*
04949 #] CleanUpSort :
04950 #[ LowerSortLevel :          void LowerSortLevel()

```

```

04951 */
04956 void LowerSortLevel(void)
04957 {
04958     GETIDENTITY
04959     if ( AR.sLevel >= 0 ) {
04960         AR.sLevel--;
04961         if ( AR.sLevel >= 0 ) AT.SS = AN.FunSorts[AR.sLevel];
04962     }
04963 }
04964
04965 /*
04966     #] LowerSortLevel :
04967     #[ PolyRatFunSpecial :
04968
04969     Keeps only the most divergent term in AR.PolyFunVar
04970     We assume that the terms are already in that notation.
04971 */
04972
04973 WORD *PolyRatFunSpecial(PHEAD WORD *t1, WORD *t2)
04974 {
04975     WORD *oldworkpointer = AT.WorkPointer, *t, *r;
04976     WORD expl, exp2;
04977     int i;
04978     t = t1+FUNHEAD;
04979     if ( *t == -SYMBOL ) {
04980         if ( t[1] != AR.PolyFunVar ) goto Illegal;
04981         expl = 1;
04982         if ( t[2] != -SNUMBER ) goto Illegal;
04983         t[3] = 1;
04984     }
04985     else if ( *t == -SNUMBER ) {
04986         t[1] = 1;
04987         t += 2;
04988         if ( *t == -SYMBOL ) {
04989             if ( t[1] != AR.PolyFunVar ) goto Illegal;
04990             expl = -1;
04991         }
04992         else if ( *t == -SNUMBER ) {
04993             t[1] = 1;
04994             expl = 0;
04995         }
04996         else if ( *t == ARGHEAD+8 && t[ARGHEAD] == 8 && t[ARGHEAD+1] == SYMBOL
04997             && t[ARGHEAD+3] == AR.PolyFunVar ) {
04998             t[ARGHEAD+5] = 1;
04999             t[ARGHEAD+6] = 1;
05000             t[ARGHEAD+7] = 3;
05001             expl = -t[ARGHEAD+4];
05002         }
05003         else goto Illegal;
05004     }
05005     else if ( *t == ARGHEAD+8 && t[ARGHEAD] == 8 && t[ARGHEAD+1] == SYMBOL
05006         && t[ARGHEAD+3] == AR.PolyFunVar ) {
05007         t[ARGHEAD+5] = 1;
05008         t[ARGHEAD+6] = 1;
05009         t[ARGHEAD+7] = 3;
05010         expl = t[ARGHEAD+4];
05011         t += *t;
05012         if ( *t != -SNUMBER ) goto Illegal;
05013         t[1] = 1;
05014     }
05015     else goto Illegal;
05016
05017     t = t2+FUNHEAD;
05018     if ( *t == -SYMBOL ) {
05019         if ( t[1] != AR.PolyFunVar ) goto Illegal;
05020         exp2 = 1;
05021         if ( t[2] != -SNUMBER ) goto Illegal;
05022         t[3] = 1;
05023     }
05024     else if ( *t == -SNUMBER ) {
05025         t[1] = 1;
05026         t += 2;
05027         if ( *t == -SYMBOL ) {
05028             if ( t[1] != AR.PolyFunVar ) goto Illegal;
05029             exp2 = -1;
05030         }
05031         else if ( *t == -SNUMBER ) {
05032             t[1] = 1;
05033             exp2 = 0;
05034         }
05035         else if ( *t == ARGHEAD+8 && t[ARGHEAD] == 8 && t[ARGHEAD+1] == SYMBOL
05036             && t[ARGHEAD+3] == AR.PolyFunVar ) {
05037             t[ARGHEAD+5] = 1;
05038             t[ARGHEAD+6] = 1;
05039             t[ARGHEAD+7] = 3;
05040             exp2 = -t[ARGHEAD+4];
05041         }

```

```

05042         else goto Illegal;
05043     }
05044     else if ( *t == ARGHEAD+8 && t[ARGHEAD] == 8 && t[ARGHEAD+1] == SYMBOL
05045         && t[ARGHEAD+3] == AR.PolyFunVar ) {
05046         t[ARGHEAD+5] = 1;
05047         t[ARGHEAD+6] = 1;
05048         t[ARGHEAD+7] = 3;
05049         exp2 = t[ARGHEAD+4];
05050         t += *t;
05051         if ( *t != -SNUMBER ) goto Illegal;
05052         t[1] = 1;
05053     }
05054     else goto Illegal;
05055
05056     if ( exp1 <= exp2 ) { i = t1[1]; r = t1; }
05057     else { i = t2[1]; r = t2; }
05058     t = oldworkpointer;
05059     NCOPY(t,r,i)
05060
05061     return(oldworkpointer);
05062 Illegal:
05063     MesPrint("Illegal occurrence of PolyRatFun with divergent option");
05064     Terminate(-1);
05065     return(0);
05066 }
05067
05068 /*
05069     #] PolyRatFunSpecial :
05070     #[ SimpleSplitMerge :
05071
05072     Sorts an array of WORDs. No adding of equal objects.
05073 */
05074
05075 void SimpleSplitMergeRec(WORD *array,WORD num,WORD *auxarray)
05076 {
05077     WORD n1,n2,i,j,k,*t1,*t2;
05078     if ( num < 2 ) return;
05079     if ( num == 2 ) {
05080         if ( array[0] > array[1] ) {
05081             EXCH(array[0],array[1])
05082         }
05083         return;
05084     }
05085     n1 = num/2;
05086     n2 = num - n1;
05087     SimpleSplitMergeRec(array,n1,auxarray);
05088     SimpleSplitMergeRec(array+n1,n2,auxarray);
05089     if ( array[n1-1] <= array[n1] ) return;
05090
05091     t1 = array; t2 = auxarray; i = n1; NCOPY(t2,t1,i);
05092     i = 0; j = n1; k = 0;
05093     while ( i < n1 && j < num ) {
05094         if ( auxarray[i] <= array[j] ) { array[k++] = auxarray[i++]; }
05095         else { array[k++] = array[j++]; }
05096     }
05097     while ( i < n1 ) array[k++] = auxarray[i++];
05098 /*
05099     Remember: remnants of j are still in place!
05100 */
05101 }
05102
05103 void SimpleSplitMerge(WORD *array,WORD num)
05104 {
05105     WORD *auxarray = Malloc1(sizeof(WORD)*num/2,"SimpleSplitMerge");
05106     SimpleSplitMergeRec(array,num,auxarray);
05107     M_free(auxarray,"SimpleSplitMerge");
05108 }
05109
05110 /*
05111     #] SimpleSplitMerge :
05112     #[ BinarySearch :
05113
05114     Searches in the sorted array with length num for the object x.
05115     If x is in the list, it returns the number of the array element
05116     that matched. If it is not in the list, it returns -1.
05117     If there are identical objects in the list, which one will
05118     match is quasi random.
05119 */
05120
05121 WORD BinarySearch(WORD *array,WORD num,WORD x)
05122 {
05123     WORD i, bot, top, med;
05124     if ( num < 8 ) {
05125         for ( i = 0; i < num; i++ ) if ( array[i] == x ) return(i);
05126         return(-1);
05127     }
05128     if ( array[0] > x || array[num-1] < x ) return(-1);

```

```

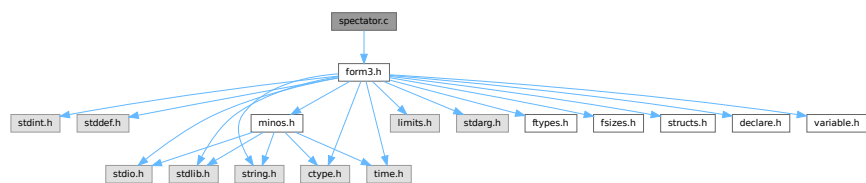
05129     bot = 0; top = num-1; med = (top+bot)/2;
05130     do {
05131         if ( array[med] == x ) return(med);
05132         if ( array[med] < x ) { bot = med+1; }
05133         else { top = med-1; }
05134         med = (top+bot)/2;
05135     } while ( med >= bot && med <= top );
05136     return(-1);
05137 }
05138
05139 /*
05140     #] BinarySearch :
05141     #] SortUtilities :
05142 */

```

4.131 spectator.c File Reference

```
#include "form3.h"
```

Include dependency graph for spectator.c:



Functions

- int [CoCreateSpectator](#) (UBYTE *inp)
- int [CoToSpectator](#) (UBYTE *inp)
- int [CoRemoveSpectator](#) (UBYTE *inp)
- int [CoEmptySpectator](#) (UBYTE *inp)
- int [PutInSpectator](#) (WORD *term, WORD specnum)
- void [FlushSpectators](#) (void)
- int [CoCopySpectator](#) (UBYTE *inp)
- WORD [GetFromSpectator](#) (WORD *term, WORD specnum)
- void [ClearSpectators](#) (WORD par)

4.131.1 Detailed Description

File contains the code for the spectator files and their control.

Definition in file [spectator.c](#).

4.131.2 Function Documentation

4.131.2.1 CoCreateSpectator()

```

int CoCreateSpectator (
    UBYTE * inp )

```

Definition at line 89 of file [spectator.c](#).

4.131.2.2 CoToSpectator()

```
int CoToSpectator (
    UBYTE * inp )
```

Definition at line 181 of file [spectator.c](#).

4.131.2.3 CoRemoveSpectator()

```
int CoRemoveSpectator (
    UBYTE * inp )
```

Definition at line 233 of file [spectator.c](#).

4.131.2.4 CoEmptySpectator()

```
int CoEmptySpectator (
    UBYTE * inp )
```

Definition at line 293 of file [spectator.c](#).

4.131.2.5 PutInSpectator()

```
int PutInSpectator (
    WORD * term,
    WORD specnum )
```

Definition at line 345 of file [spectator.c](#).

4.131.2.6 FlushSpectators()

```
void FlushSpectators (
    void )
```

Definition at line 402 of file [spectator.c](#).

4.131.2.7 CoCopySpectator()

```
int CoCopySpectator (
    UBYTE * inp )
```

Definition at line 459 of file [spectator.c](#).

4.131.2.8 GetFromSpectator()

```
WORD GetFromSpectator (
    WORD * term,
    WORD specnum )
```

Definition at line 601 of file [spectator.c](#).

4.131.2.9 ClearSpectators()

```
void ClearSpectators (
    WORD par )
```

Definition at line 675 of file [spectator.c](#).

4.132 spectator.c

[Go to the documentation of this file.](#)

```
00001
00006 /* #[ License : */
00007 /*
00008 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009 *   When using this file you are requested to refer to the publication
00010 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011 *   This is considered a matter of courtesy as the development was paid
00012 *   for by FOM the Dutch physics granting agency and we would like to
00013 *   be able to track its scientific use to convince FOM of its value
00014 *   for the community.
00015 *
00016 *   This file is part of FORM.
00017 *
00018 *   FORM is free software: you can redistribute it and/or modify it under the
00019 *   terms of the GNU General Public License as published by the Free Software
00020 *   Foundation, either version 3 of the License, or (at your option) any later
00021 *   version.
00022 *
00023 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00026 *   details.
00027 *
00028 *   You should have received a copy of the GNU General Public License along
00029 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00030 */
00031 /* #] License : */
00032 /*
00033 *   #[ includes :
00034 */
00035
00036 #include "form3.h"
00037
00038 /*
00039 *   #] includes :
00040 *   #[ Commentary :
00041
00042 *   We use an array of SPECTATOR structs in AM.SpectatorFiles.
00043 *   When a spectator is removed this leaves a hole. This means that
00044 *   we cannot use AM.NumSpectatorFiles but always have to scan up to
00045 *   AM.SizeForSpectatorFiles which is the size of the array.
00046 *   An element is in use when it has a name. This is the name of the
00047 *   expression that is associated with it. There is also the number of
00048 *   the expression, but because the expressions are frequently renumbered
00049 *   at the end of a module, we always search for the spectators by name.
00050 *   The expression number is only valid in the current module.
00051 *   During execution we use the number of the spectator.
00052
00053 *   The FILEHANDLE struct is borrowed from the structs for the scratch
00054 *   files, but we do not keep copies for all workers as with the scratch
00055 *   files. This brings some limitations (but saves much space). Basically
00056 *   the reading can only be done by one master or worker. And because
00057 *   we use the buffer both for writing and for reading we cannot read and
00058 *   write in the same module.
00059
00060 *   Processor can see that an expression is to be filled with a spectator
00061 *   because we replace the compiler buffer number in the prototype by
00062 *   -specnum-1. Of course, after this filling has taken place we should
00063 *   make sure that in the next module there is a nonnegative number there.
00064 *   The input is then obtained from GetFromSpectator instead from GetTerm.
00065 *   This needed an extra argument in ThreadsProcessor. InParallelProcessor
00066 *   can figure it out by itself. ParFORM still needs to be treated for this.
00067
00068 *   The writing is to a single buffer. Hence it needs a lock. It is possible
00069 *   to give all workers their own buffers (at great memory cost) and merge
00070 *   the results when needed. That would be friendlier on ParFORM. We ALWAYS
00071 *   assume that the order of the terms in the spectator file is random.
00072
```

```

00073     In the first version there is no compression in the file. This could
00074     change in later versions because both the writing and the reading are
00075     purely sequential. Brackets are not possible.
00076
00077     Currently, ParFORM allows use of spectators only in the sequential
00078     mode. The parallelization is switched off in modules containing
00079     ToSpectator or CopySpectator. Workers never create or access to
00080     spectator files. Their file handles are always -1. We leave the
00081     parallelization of modules with spectators for future work.
00082
00083     #] Commentary :
00084     #[ CoCreateSpectator :
00085
00086     Syntax: CreateSpectator name_of_expr "filename";
00087  */
00088
00089 int CoCreateSpectator(UBYTE *inp)
00090 {
00091     UBYTE *p, *q, *filename, c, cc;
00092     WORD c1, c2, numexpr = 0, specnum, HadOne = 0;
00093     FILEHANDLE *fh;
00094     while ( *inp == ',' ) inp++;
00095     if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
00096         MesPrint("&Illegal name for expression");
00097         return(1);
00098     }
00099     c = *q; *q = 0;
00100     if ( GetVar(inp,&c1,&c2,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
00101         if ( c2 == CEXPRESSION &&
00102             Expressions[c1].status == DROPSPECTATOREXPRESSION ) {
00103             numexpr = c1;
00104             Expressions[numexpr].status = SPECTATOREXPRESSION;
00105             HadOne = 1;
00106         }
00107         else {
00108             MesPrint("&The name %s has been used already.",inp);
00109             *q = c;
00110             return(1);
00111         }
00112     }
00113     p = q+1;
00114     while ( *p == ',' ) p++;
00115     if ( *p != '"' ) goto Syntax;
00116     p++; filename = p;
00117     while ( *p && *p != '"' ) {
00118         if ( *p == '\\\\' ) p++;
00119         p++;
00120     }
00121     if ( *p != '"' ) goto Syntax;
00122     q = p+1;
00123     while ( *q && ( *q == ',' || *q == ' ' || *q == '\\t' ) ) q++;
00124     if ( *q ) goto Syntax;
00125     cc = *p; *p = 0;
00126  /*
00127     Now we need to: create a struct for the spectator file.
00128  */
00129     if ( HadOne == 0 )
00130         numexpr = EntVar(CEXPRESSION,inp,SPECTATOREXPRESSION,0,0,0);
00131     fh = AllocFileHandle(1,(char *)filename);
00132  /*
00133     Make sure there is space in the AM.spectatorfiles array
00134  */
00135     if ( AM.NumSpectatorFiles >= AM.SizeForSpectatorFiles || AM.SpectatorFiles == 0 ) {
00136         int newsize, i;
00137         SPECTATOR *newspectators;
00138         if ( AM.SizeForSpectatorFiles == 0 ) {
00139             newsize = 10;
00140             AM.NumSpectatorFiles = AM.SizeForSpectatorFiles = 0;
00141         }
00142         else newsize = AM.SizeForSpectatorFiles*2;
00143         newspectators = (SPECTATOR *)Malloc1(newsize*sizeof(SPECTATOR),"Spectators");
00144         for ( i = 0; i < AM.NumSpectatorFiles; i++ )
00145             newspectators[i] = AM.SpectatorFiles[i];
00146         for ( ; i < newsize; i++ ) {
00147             newspectators[i].fh = 0;
00148             newspectators[i].name = 0;
00149             newspectators[i].exprnumber = -1;
00150             newspectators[i].flags = 0;
00151             PUTZERO(newspectators[i].position);
00152             PUTZERO(newspectators[i].readpos);
00153         }
00154         AM.SizeForSpectatorFiles = newsize;
00155         if ( AM.SpectatorFiles != 0 ) M_free(AM.SpectatorFiles,"Spectators");
00156         AM.SpectatorFiles = newspectators;
00157         specnum = AM.NumSpectatorFiles++;
00158     }
00159     else {

```



```

00160         for ( specnum = 0; specnum < AM.SizeForSpectatorFiles; specnum++ ) {
00161             if ( AM.SpectatorFiles[specnum].name == 0 ) break;
00162         }
00163         AM.NumSpectatorFiles++;
00164     }
00165     PUTZERO(AM.SpectatorFiles[specnum].position);
00166     AM.SpectatorFiles[specnum].name = (char *) (strDup1(inp, "Spectator expression name"));
00167     AM.SpectatorFiles[specnum].fh = fh;
00168     AM.SpectatorFiles[specnum].exprnumber = numexpr;
00169     *p = cc;
00170     return(0);
00171 Syntax:
00172     MesPrint("&Proper syntax is: CreateSpectator,exprname,\"filename\";");
00173     return(-1);
00174 }
00175
00176 /*
00177  #] CoCreateSpectator :
00178  #[ CoToSpectator :
00179  */
00180
00181 int CoToSpectator(UBYTE *inp)
00182 {
00183     UBYTE *q;
00184     WORD c1, numexpr;
00185     int i;
00186     while ( *inp == ',' ) inp++;
00187     if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
00188         MesPrint("&Illegal name for expression");
00189         return(1);
00190     }
00191     if ( *q != 0 ) goto Syntax;
00192     if ( GetVar(inp, &c1, &numexpr, ALLVARIABLES, NOAUTO) == NAMENOTFOUND ||
00193         c1 != CEXPRESSION ) {
00194         MesPrint("&%s is not a valid expression.", inp);
00195         return(1);
00196     }
00197     if ( Expressions[numexpr].status != SPECTATOREXPRESSION ) {
00198         MesPrint("&%s is not an active spectator.", inp);
00199         return(1);
00200     }
00201     for ( i = 0; i < AM.SizeForSpectatorFiles; i++ ) {
00202         if ( AM.SpectatorFiles[i].name != 0 ) {
00203             if ( StrCmp((UBYTE *) (AM.SpectatorFiles[i].name), (UBYTE *) (inp)) == 0 ) break;
00204         }
00205     }
00206     if ( i >= AM.SizeForSpectatorFiles ) {
00207         MesPrint("&Spectator %s not found.", inp);
00208         return(1);
00209     }
00210     if ( ( AM.SpectatorFiles[i].flags & READSPECTATORFLAG ) != 0 ) {
00211         MesPrint("&Spectator %s: It is not permitted to read from and write to the same spectator in
one module.", inp);
00212         return(1);
00213     }
00214     AM.SpectatorFiles[i].exprnumber = numexpr;
00215     Add3Com(TYPETOSPECTATOR, i);
00216 #ifdef WITHMPI
00217     /*
00218      * In ParFORM, ToSpectator has to be executed on the master.
00219      */
00220     AC.mparallelflag |= NOPARALLEL_SPECTATOR;
00221 #endif
00222     return(0);
00223 Syntax:
00224     MesPrint("&Proper syntax is: ToSpectator,exprname;");
00225     return(-1);
00226 }
00227
00228 /*
00229  #] CoToSpectator :
00230  #[ CoRemoveSpectator :
00231  */
00232
00233 int CoRemoveSpectator(UBYTE *inp)
00234 {
00235     UBYTE *q;
00236     WORD c1, numexpr;
00237     int i;
00238     SPECTATOR *sp;
00239     while ( *inp == ',' ) inp++;
00240     if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
00241         MesPrint("&Illegal name for expression");
00242         return(1);
00243     }
00244     if ( *q != 0 ) goto Syntax;
00245     if ( GetVar(inp, &c1, &numexpr, ALLVARIABLES, NOAUTO) == NAMENOTFOUND ||

```

```

00246         c1 != CEXPRESSION ) {
00247     MesPrint("&%s is not a valid expression.",inp);
00248     return(1);
00249 }
00250 if ( Expressions[numexpr].status != SPECTATOREXPRESSION ) {
00251     MesPrint("&%s is not a spectator.",inp);
00252     return(1);
00253 }
00254 for ( i = 0; i < AM.SizeForSpectatorFiles; i++ ) {
00255     if ( AM.SpectatorFiles[i].name != 0 ) {
00256         if ( StrCmp((UBYTE *) (AM.SpectatorFiles[i].name), (UBYTE *) (inp)) == 0 ) {
00257             break;
00258         }
00259     }
00260 }
00261 if ( i >= AM.SizeForSpectatorFiles ) {
00262     MesPrint("&Spectator %s not found.",inp);
00263     return(1);
00264 }
00265 sp = AM.SpectatorFiles+i;
00266 Expressions[numexpr].status = DROPSPECTATOREXPRESSION;
00267 if ( sp->fh->handle != -1 ) {
00268     CloseFile(sp->fh->handle);
00269     sp->fh->handle = -1;
00270     remove(sp->fh->name);
00271 }
00272 M_free(sp->fh,"Temporary FileHandle");
00273 M_free(sp->name,"Spectator expression name");
00274
00275 PUTZERO(sp->position);
00276 PUTZERO(sp->readpos);
00277 sp->fh = 0;
00278 sp->name = 0;
00279 sp->exprnumber = -1;
00280 sp->flags = 0;
00281 AM.NumSpectatorFiles--;
00282 return(0);
00283 Syntax:
00284 MesPrint("&Proper syntax is: RemoveSpectator,exprname;");
00285 return(-1);
00286 }
00287
00288 /*
00289 #] CoRemoveSpectator :
00290 #[ CoEmptySpectator :
00291 */
00292
00293 int CoEmptySpectator(UBYTE *inp)
00294 {
00295     UBYTE *q;
00296     WORD c1, numexpr;
00297     int i;
00298     SPECTATOR *sp;
00299     while ( *inp == ',' ) inp++;
00300     if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
00301         MesPrint("&Illegal name for expression");
00302         return(1);
00303     }
00304     if ( *q != 0 ) goto Syntax;
00305     if ( GetVar(inp,&c1,&numexpr,ALLVARIABLES,NOAUTO) == NAMENOTFOUND ||
00306         c1 != CEXPRESSION ) {
00307         MesPrint("&%s is not a valid expression.",inp);
00308         return(1);
00309     }
00310     if ( Expressions[numexpr].status != SPECTATOREXPRESSION ) {
00311         MesPrint("&%s is not a spectator.",inp);
00312         return(1);
00313     }
00314     for ( i = 0; i < AM.SizeForSpectatorFiles; i++ ) {
00315         if ( StrCmp((UBYTE *) (AM.SpectatorFiles[i].name), (UBYTE *) (inp)) == 0 ) break;
00316     }
00317     if ( i >= AM.SizeForSpectatorFiles ) {
00318         MesPrint("&Spectator %s not found.",inp);
00319         return(1);
00320     }
00321     sp = AM.SpectatorFiles+i;
00322     if ( sp->fh->handle != -1 ) {
00323         CloseFile(sp->fh->handle);
00324         sp->fh->handle = -1;
00325         remove(sp->fh->name);
00326     }
00327     sp->fh->Pofill = sp->fh->POfull = sp->fh->PObuffer;
00328     PUTZERO(sp->position);
00329     PUTZERO(sp->readpos);
00330     return(0);
00331 Syntax:
00332 MesPrint("&Proper syntax is: EmptySpectator,exprname;");

```

```

00333     return(-1);
00334 }
00335
00336 /*
00337  #] CoEmptySpectator :
00338  #[ PutInSpectator :
00339
00340  We need to use locks! There is only one file.
00341  The code was copied (and modified) from PutOut.
00342  Here we use no compression.
00343 */
00344
00345 int PutInSpectator(WORD *term,WORD specnum)
00346 {
00347     GETBIDENTITY
00348     WORD i, *p, ret;
00349     LONG RetCode;
00350     SPECTATOR *sp = &(AM.SpectatorFiles[specnum]);
00351     FILEHANDLE *fi = sp->fh;
00352
00353     if ( ( i = *term ) <= 0 ) return(0);
00354     LOCK(fi->pthreadlock);
00355     ret = i;
00356     p = fi->POfill;
00357     do {
00358         if ( p >= fi->PStop ) {
00359             if ( fi->handle < 0 ) {
00360                 if ( ( RetCode = CreateFile(fi->name) ) >= 0 ) {
00361                     fi->handle = (WORD)RetCode;
00362                     PUTZERO(fi->filesize);
00363                     PUTZERO(fi->POposition);
00364                 }
00365                 else {
00366                     MLOCK(ErrorMessageLock);
00367                     MesPrint("Cannot create spectator file %s",fi->name);
00368                     MUNLOCK(ErrorMessageLock);
00369                     UNLOCK(fi->pthreadlock);
00370                     return(-1);
00371                 }
00372             }
00373             SeekFile(fi->handle,&(sp->position),SEEK_SET);
00374             if ( ( RetCode = WriteFile(fi->handle,(UBYTE *) (fi->PObuffer),fi->POsize) ) != fi->POsize
00375         ) {
00376             MLOCK(ErrorMessageLock);
00377             MesPrint("Error during spectator write. Disk full?");
00378             MesPrint("Attempt to write %l bytes on file %d at position %l5p",
00379                     fi->POsize,fi->handle,&(fi->POposition));
00380             MesPrint("RetCode = %l, Buffer address = %l",RetCode,(LONG) (fi->PObuffer));
00381             MUNLOCK(ErrorMessageLock);
00382             UNLOCK(fi->pthreadlock);
00383             return(-1);
00384         }
00385         ADDPOS(fi->filesize,fi->POsize);
00386         p = fi->PObuffer;
00387         ADDPOS(sp->position,fi->POsize);
00388         fi->POposition = sp->position;
00389         *p++ = *term++;
00390     } while ( --i > 0 );
00391     fi->POfull = fi->POfill = p;
00392     Expressions[AM.SpectatorFiles[specnum].exprnumber].counter++;
00393     UNLOCK(fi->pthreadlock);
00394     return(ret);
00395 }
00396
00397 /*
00398  #] PutInSpectator :
00399  #[ FlushSpectators :
00400 */
00401
00402 void FlushSpectators(void)
00403 {
00404     SPECTATOR *sp = AM.SpectatorFiles;
00405     FILEHANDLE *fh;
00406     LONG RetCode;
00407     int i;
00408     LONG size;
00409     if ( AM.NumSpectatorFiles <= 0 ) return;
00410     for ( i = 0; i < AM.SizeForSpectatorFiles; i++, sp++ ) {
00411         if ( sp->name == 0 ) continue;
00412         fh = sp->fh;
00413         if ( ( sp->flags & READSPECTATORFLAG ) != 0 ) { /* reset for writing */
00414             sp->flags &= ~READSPECTATORFLAG;
00415             fh->POfill = fh->PObuffer;
00416             if ( fh->handle >= 0 ) {
00417                 SeekFile(fh->handle,&(sp->position),SEEK_SET);
00418                 fh->POposition = sp->position;

```

```

00419         }
00420         continue;
00421     }
00422     if ( fh->POfill <= fh->PObuffer ) continue; /* is clean */
00423     if ( fh->handle < 0 ) { /* File needs to be created */
00424         if ( ( RetCode = CreateFile(fh->name) ) >= 0 ) {
00425             PUTZERO(fh->filesize);
00426             PUTZERO(fh->POposition);
00427             fh->handle = (WORD)RetCode;
00428         }
00429         else {
00430             MLOCK(ErrorMessageLock);
00431             MesPrint("Cannot create spectator file %s",fh->name);
00432             MUNLOCK(ErrorMessageLock);
00433             Terminate(-1);
00434         }
00435         PUTZERO(sp->position);
00436     }
00437     SeekFile(fh->handle,&(sp->position),SEEK_SET);
00438     size = (fh->POfill - fh->PObuffer)*sizeof(WORD);
00439     if ( ( RetCode = WriteFile(fh->handle,(UBYTE *) (fh->PObuffer),size) ) != size ) {
00440         MLOCK(ErrorMessageLock);
00441         MesPrint("Write error synching spectator file. Disk full?");
00442         MesPrint("Attempt to write %l bytes on file %s at position %l5p",
00443             size,fh->name,&(sp->position));
00444         MUNLOCK(ErrorMessageLock);
00445         Terminate(-1);
00446     }
00447     fh->POfill = fh->PObuffer;
00448     SeekFile(fh->handle,&(sp->position),SEEK_END);
00449     fh->POposition = sp->position;
00450 }
00451 return;
00452 }
00453
00454 /*
00455 #] FlushSpectators :
00456 #[ CoCopySpectator :
00457 */
00458
00459 int CoCopySpectator(UBYTE *inp)
00460 {
00461     GETIDENTITY
00462     UBYTE *q, c, *exprname, *p;
00463     WORD c1, c2, numexpr;
00464     int specnum, error = 0;
00465     SPECTATOR *sp;
00466     while ( *inp == ',' ) inp++;
00467     if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
00468         MesPrint("%Illegal name for expression");
00469         return(1);
00470     }
00471     if ( *q == 0 ) goto Syntax;
00472     c = *q; *q = 0;
00473     if ( GetVar(inp,&c1,&c2,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
00474         MesPrint("%s is the name of an existing variable.",inp);
00475         return(1);
00476     }
00477     numexpr = EntVar(CEXPRESSION,inp,LOCALEXPRESSION,0,0,0);
00478     p = q;
00479     exprname = inp;
00480     *q = c;
00481     while ( *q == ' ' || *q == ',' || *q == '\t' ) q++;
00482     if ( *q != '=' ) goto Syntax;
00483     q++;
00484     while ( *q == ' ' || *q == ',' || *q == '\t' ) q++;
00485     inp = q;
00486     if ( ( q = SkipAName(inp) ) == 0 || q[-1] == '_' ) {
00487         MesPrint("%Illegal name for spectator expression");
00488         return(1);
00489     }
00490     if ( *q != 0 ) goto Syntax;
00491     if ( AM.NumSpectatorFiles <= 0 ) {
00492         MesPrint("%CopySpectator: There are no spectator expressions!");
00493         return(1);
00494     }
00495     sp = AM.SpectatorFiles;
00496     for ( specnum = 0; specnum < AM.SizeForSpectatorFiles; specnum++, sp++ ) {
00497         if ( sp->name != 0 ) {
00498             if ( StrCmp((UBYTE *) (sp->name),(UBYTE *) (inp)) == 0 ) break;
00499         }
00500     }
00501     if ( specnum >= AM.SizeForSpectatorFiles ) {
00502         MesPrint("%Spectator %s not found.",inp);
00503         return(1);
00504     }
00505     sp->flags |= READSPECTATORFLAG;

```

```

00506     PUTZERO(sp->fh->POposition);
00507     PUTZERO(sp->readpos);
00508     sp->fh->POfill = sp->fh->PObuffer;
00509     if ( sp->fh->handle >= 0 ) {
00510         SeekFile(sp->fh->handle, &(sp->fh->POposition), SEEK_SET);
00511     }
00512     /*
00513     Now we have:
00514     1: The name of the target expression: numexpr
00515     2: The spectator: sp (or specnum).
00516     Time for some action. We need:
00517     a: Write a prototype to create the expression
00518     b: Signal to Processor that this is a spectator.
00519     We do this by giving a negative compiler buffer number.
00520     */
00521     {
00522         WORD *OldWork, *w;
00523         POSITION pos;
00524         OldWork = w = AT.WorkPointer;
00525         *w++ = TYPEEXPRESSION;
00526         *w++ = 3+SUBEXPSIZE;
00527         *w++ = numexpr;
00528         AC.ProtoType = w;
00529         AR.CurExpr = numexpr;          /* Block expression numexpr */
00530         *w++ = SUBEXPRESSION;
00531         *w++ = SUBEXPSIZE;
00532         *w++ = numexpr;
00533         *w++ = 1;
00534         *w++ = -specnum-1;          /* Indicates "spectator" to Processor */
00535         FILLSUB(w)
00536         *w++ = 1;
00537         *w++ = 1;
00538         *w++ = 3;
00539         *w++ = 0;
00540         SeekScratch(AR.outfile, &pos);
00541         Expressions[numexpr].counter = 1;
00542         Expressions[numexpr].onfile = pos;
00543         Expressions[numexpr].whichbuffer = 0;
00544         #ifdef PARALLELCODE
00545         Expressions[numexpr].partodo = AC.inparallelflag;
00546         #endif
00547         OldWork[2] = w - OldWork - 3;
00548         AT.WorkPointer = w;
00549
00550         if ( PutOut(BHEAD OldWork+2, &pos, AR.outfile, 0) < 0 ) {
00551             c = *p; *p = 0;
00552             MesPrint("&Cannot create expression %s", exprname);
00553             *p = c;
00554             error = -1;
00555         }
00556         else {
00557             OldWork[2] = 4+SUBEXPSIZE;
00558             OldWork[4] = SUBEXPSIZE;
00559             OldWork[5] = numexpr;
00560             OldWork[SUBEXPSIZE+3] = 1;
00561             OldWork[SUBEXPSIZE+4] = 1;
00562             OldWork[SUBEXPSIZE+5] = 3;
00563             OldWork[SUBEXPSIZE+6] = 0;
00564             if ( PutOut(BHEAD OldWork+2, &pos, AR.outfile, 0) < 0
00565                 || FlushOut(&pos, AR.outfile, 0) ) {
00566                 c = *p; *p = 0;
00567                 MesPrint("&Cannot create expression %s", exprname);
00568                 *p = c;
00569                 error = -1;
00570             }
00571             AR.outfile->POfull = AR.outfile->POfill;
00572         }
00573         OldWork[2] = numexpr;
00574     /*
00575     Seems unnecessary (13-feb-2018)
00576
00577     AddNtoL(OldWork[1], OldWork);
00578     */
00579     AT.WorkPointer = OldWork;
00580     if ( AC.dumnumflag ) Add2Com(TYPEDETCURDUM)
00581     }
00582     #ifdef WITHMPI
00583     /*
00584     * In ParFORM, substitutions of spectators has to be done on the master.
00585     */
00586     AC.mparallelflag |= NOPARALLEL_SPECTATOR;
00587     #endif
00588     return(error);
00589 Syntax:
00590     MesPrint("&Proper syntax is: CopySpectator, exprname=spectatorname;");
00591     return(-1);
00592 }

```

```

00593
00594 /*
00595 #] CoCopySpectator :
00596 #[ GetFromSpectator :
00597
00598     Note that if we did things right, we do not need a lock for the reading.
00599 */
00600
00601 WORD GetFromSpectator(WORD *term,WORD specnum)
00602 {
00603     SPECTATOR *sp = &(AM.SpectatorFiles[specnum]);
00604     FILEHANDLE *fh = sp->fh;
00605     WORD i, size, *t = term;
00606     LONG InIn;
00607     if ( fh->handle < 0 ) {
00608         *term = 0;
00609         return(0);
00610     }
00611 /*
00612     sp->position marks the 'end' of the file: the point where writing should
00613     take place. sp->readpos marks from where to read.
00614     fh->POposition marks where the file is currently positioned.
00615     Note that when we read, we need to
00616 */
00617     if ( ISZEROPOS(sp->readpos) ) { /* we start reading. Fill buffer. */
00618         FillBuffer:
00619         SeekFile(fh->handle,&(sp->readpos),SEEK_SET);
00620         InIn = ReadFile(fh->handle,(UBYTE *) (fh->PObuffer),fh->POsize);
00621         if ( InIn < 0 || ( InIn & 1 ) ) {
00622             MLOCK(ErrorMessageLock);
00623             MesPrint("Error reading information for %s spectator",sp->name);
00624             MUNLOCK(ErrorMessageLock);
00625             Terminate(-1);
00626         }
00627         InIn /= sizeof(WORD);
00628         if ( InIn == 0 ) { *term = 0; return(0); }
00629         SeekFile(fh->handle,&(sp->readpos),SEEK_CUR);
00630         fh->POposition = sp->readpos;
00631         fh->POfull = fh->PObuffer+InIn;
00632         fh->POfill = fh->PObuffer;
00633     }
00634     if ( fh->POfill == fh->POfull ) { /* not even the size of the term! */
00635         if ( ISLESSPOS(sp->readpos,sp->position) ) goto FillBuffer;
00636         *term = 0;
00637         return(0);
00638     }
00639     size = *fh->POfill++; *t++ = size;
00640     for ( i = 1; i < size; i++ ) {
00641         if ( fh->POfill >= fh->POfull ) {
00642             SeekFile(fh->handle,&(sp->readpos),SEEK_SET);
00643             InIn = ReadFile(fh->handle,(UBYTE *) (fh->PObuffer),fh->POsize);
00644             if ( InIn < 0 || ( InIn & 1 ) ) {
00645                 MLOCK(ErrorMessageLock);
00646                 MesPrint("Error reading information for %s spectator",sp->name);
00647                 MUNLOCK(ErrorMessageLock);
00648                 Terminate(-1);
00649             }
00650             InIn /= sizeof(WORD);
00651             if ( InIn == 0 ) {
00652                 MLOCK(ErrorMessageLock);
00653                 MesPrint("Reading incomplete information for %s spectator",sp->name);
00654                 MUNLOCK(ErrorMessageLock);
00655                 Terminate(-1);
00656             }
00657             SeekFile(fh->handle,&(sp->readpos),SEEK_CUR);
00658             fh->POposition = sp->readpos;
00659             fh->POfull = fh->PObuffer+InIn;
00660             fh->POfill = fh->PObuffer;
00661         }
00662         *t++ = *fh->POfill++;
00663     }
00664     return(size);
00665 }
00666
00667 /*
00668 #] GetFromSpectator :
00669 #[ ClearSpectators :
00670
00671     Removes all spectators.
00672     In case of .store, the ones that are protected by .global stay.
00673 */
00674
00675 void ClearSpectators(WORD par)
00676 {
00677     SPECTATOR *sp = AM.SpectatorFiles;
00678     WORD numexpr, c1;
00679     int i;

```

```

00680     if ( AM.NumSpectatorFiles > 0 ) {
00681         for ( i = 0; i < AM.SizeForSpectatorFiles; i++, sp++ ) {
00682             if ( sp->name == 0 ) continue;
00683             if ( ( sp->flags & GLOBALSPECTATORFLAG ) == 1 && par == STOREMODULE ) continue;
00684
00685             if ( GetVar((UBYTE *) (sp->name), &c1, &numexpr, ALLVARIABLES, NOAUTO) == NAMENOTFOUND ||
00686                 c1 != CEXPRESSION ) {
00687                 MesPrint("&#x26;s is not a valid expression.", sp->name);
00688                 continue;
00689             }
00690             Expressions[numexpr].status = DROPPEDEXPRESSION;
00691             if ( sp->fh->handle != -1 ) {
00692                 CloseFile(sp->fh->handle);
00693                 sp->fh->handle = -1;
00694                 remove(sp->fh->name);
00695             }
00696             M_free(sp->fh, "Temporary FileHandle");
00697             M_free(sp->name, "Spectator expression name");
00698             PUTZERO(sp->position);
00699             sp->fh = 0;
00700             sp->name = 0;
00701             sp->exprnumber = -1;
00702             sp->flags = 0;
00703             AM.NumSpectatorFiles--;
00704         }
00705     }
00706 }
00707
00708 /*
00709 #] ClearSpectators :
00710 */

```

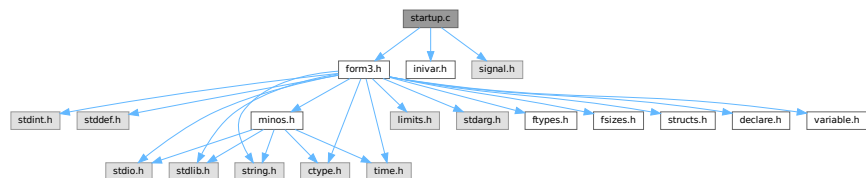
4.133 startup.c File Reference

```

#include "form3.h"
#include "inivar.h"
#include <signal.h>

```

Include dependency graph for startup.c:



Macros

- #define **STRINGIFY**(x) STRINGIFY__(x)
- #define **STRINGIFY__**(x) #x
- #define **FORMNAME** "FORM"
- #define **VERSIONSTR__** STRINGIFY(MAJORVERSION) "." STRINGIFY(MINORVERSION) "Beta"
- #define **VERSIONSTR** FORMNAME " " VERSIONSTR__ " (" PRODUCTIONDATE ")"
- #define **TAKEPATH**(x) if(s[1]== '='){x=s+2;} else{x=*argv++;argc--;}
- #define **DEFAULTFNAMELENGTH** 16
- #define **STR2**(x) #x
- #define **STR**(x) STR2(x)

Functions

- int [DoTail](#) (int argc, UBYTE **argv)
- int [OpenInput](#) (void)
- void [ReserveTempFiles](#) (int par)
- void [StartVariables](#) (void)
- void [StartMore](#) (void)
- void [IniVars](#) (void)
- int [main](#) (int argc, char **argv)
- void [CleanUp](#) (WORD par)
- void [TerminateImpl](#) (int errorcode, const char *file, int line, const char *function)
- void [PrintDeprecation](#) (const char *feature, const char *issue)
- void [PrintRunningTime](#) (void)
- LONG [GetRunningTime](#) (void)

Variables

- UBYTE * [emptystring](#) = (UBYTE *)"."
- UBYTE * [defaulttempfilename](#) = (UBYTE *)"xformxxx.str"

4.133.1 Detailed Description

This file contains the main program. It also deals with the very early stages of the startup of FORM and the final stages when the program attempts some cleanup. Here is the routine that analyses the command tail.

Definition in file [startup.c](#).

4.133.2 Macro Definition Documentation

4.133.2.1 STRINGIFY

```
#define STRINGIFY(  
    x ) STRINGIFY__(x)
```

Definition at line 57 of file [startup.c](#).

4.133.2.2 STRINGIFY__

```
#define STRINGIFY__(  
    x ) #x
```

Definition at line 58 of file [startup.c](#).

4.133.2.3 FORMNAME

```
#define FORMNAME "FORM"
```

Definition at line 68 of file [startup.c](#).

4.133.2.4 VERSIONSTR__

```
#define VERSIONSTR__ STRINGIFY(MAJORVERSION) "." STRINGIFY(MINORVERSION) "Beta"
```

Definition at line 97 of file [startup.c](#).

4.133.2.5 VERSIONSTR

```
#define VERSIONSTR FORMNAME " " VERSIONSTR__ " (" PRODUCTIONDATE ")"
```

Definition at line 101 of file [startup.c](#).

4.133.2.6 TAKEPATH

```
#define TAKEPATH(  
    x ) if(s[1]== '=' ) {x=s+2;} else {x=*argv++;argc--;}  
    x )
```

Definition at line 235 of file [startup.c](#).

4.133.2.7 DEFAULTFNAMELENGTH

```
#define DEFAULTFNAMELENGTH 16
```

Definition at line 664 of file [startup.c](#).

4.133.3 Function Documentation

4.133.3.1 DoTail()

```
int DoTail (  
    int argc,  
    UBYTE ** argv )
```

Definition at line 237 of file [startup.c](#).

4.133.3.2 OpenInput()

```
int OpenInput (  
    void )
```

Definition at line 542 of file [startup.c](#).

4.133.3.3 ReserveTempFiles()

```
void ReserveTempFiles (  
    int par )
```

Definition at line 666 of file [startup.c](#).

4.133.3.4 StartVariables()

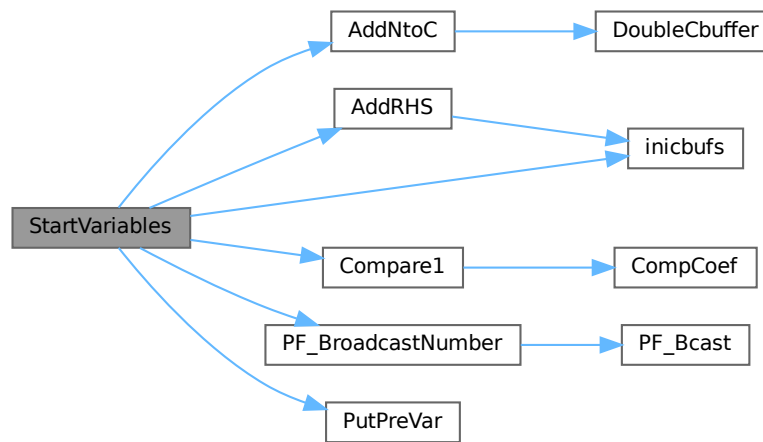
```
void StartVariables (
    void )
```

All functions (well, nearly all) are declared here.

Definition at line 905 of file [startup.c](#).

References [AddNtoC\(\)](#), [AddRHS\(\)](#), [Compare1\(\)](#), [inicbufs\(\)](#), [FuNcTiOn::name](#), [PF_BroadcastNumber\(\)](#), [PutPreVar\(\)](#), and [FuNcTiOn::symmetric](#).

Here is the call graph for this function:



4.133.3.5 StartMore()

```
void StartMore (
    void )
```

Definition at line 1333 of file [startup.c](#).

4.133.3.6 IniVars()

```
void IniVars (
    void )
```

Definition at line 1371 of file [startup.c](#).

4.133.3.7 main()

```
int main (
    int argc,
    char ** argv )
```

Definition at line 1678 of file [startup.c](#).

4.133.3.8 CleanUp()

```
void CleanUp (
    WORD par )
```

Definition at line 1780 of file [startup.c](#).

4.133.3.9 TerminateImpl()

```
void TerminateImpl (
    int errorcode,
    const char * file,
    int line,
    const char * function )
```

Definition at line 1869 of file [startup.c](#).

4.133.3.10 PrintDeprecation()

```
void PrintDeprecation (
    const char * feature,
    const char * issue )
```

Prints a deprecation warning for a given feature.

Parameters

<i>feature</i>	The name of the deprecated feature.
<i>issue</i>	The associated issue, e.g., "issues/700".

Definition at line 2082 of file [startup.c](#).

4.133.3.11 PrintRunningTime()

```
void PrintRunningTime (
    void )
```

Definition at line 2106 of file [startup.c](#).

4.133.3.12 GetRunningTime()

```
LONG GetRunningTime (
    void )
```

Definition at line 2147 of file [startup.c](#).

4.133.4 Variable Documentation

4.133.4.1 emptystring

```
UBYTE* emptystring = (UBYTE *)"."
```

Definition at line 660 of file [startup.c](#).

4.133.4.2 defaulttempfilename

```
UBYTE* defaulttempfilename = (UBYTE *)"xformxxx.str"
```

Definition at line 661 of file [startup.c](#).

4.134 startup.c

[Go to the documentation of this file.](#)

```
00001
00008 /* #[ License : */
00009 /*
00010 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00011 *   When using this file you are requested to refer to the publication
00012 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013 *   This is considered a matter of courtesy as the development was paid
00014 *   for by FOM the Dutch physics granting agency and we would like to
00015 *   be able to track its scientific use to convince FOM of its value
00016 *   for the community.
00017 *
00018 *   This file is part of FORM.
00019 *
00020 *   FORM is free software: you can redistribute it and/or modify it under the
00021 *   terms of the GNU General Public License as published by the Free Software
00022 *   Foundation, either version 3 of the License, or (at your option) any later
00023 *   version.
00024 *
00025 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00028 *   details.
00029 *
00030 *   You should have received a copy of the GNU General Public License along
00031 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00032 */
00033 /* #[ License : */
00034 /*
00035     #[ includes :
00036 */
00037
00038 #include "form3.h"
00039 #include "inivar.h"
00040
00041 #ifdef TRAP_SIGNALS
00042 #include "portsignals.h"
00043 #else
00044 #include <signal.h>
00045 #endif
00046 #ifdef ENABLE_BACKTRACE
00047     #include <execinfo.h>
00048 #endif
00049 #ifdef LINUX
00049     #include <stdint.h>
00050     #include <inttypes.h>
00051 #endif
00052 #endif
00053
00054 /*
00055 * A macro for translating the contents of `x' into a string after expanding.
00056 */
00057 #define STRINGIFY(x) STRINGIFY__(x)
00058 #define STRINGIFY__(x) #x
00059
00060 /*
```

```

00061  * FORMNAME = "FORM" or "TFORM" or "ParFORM".
00062  */
00063  #if defined(WITHPTHREADS)
00064      #define FORMNAME "TFORM"
00065  #elif defined(WITHMPI)
00066      #define FORMNAME "ParFORM"
00067  #else
00068      #define FORMNAME "FORM"
00069  #endif
00070
00071  /*
00072  * VERSIONSTR is the version information printed in the header line.
00073  */
00074  #ifdef HAVE_CONFIG_H
00075      /* We have also version.h. */
00076      #include "version.h"
00077      #ifndef REPO_VERSION
00078          #define REPO_VERSION STRINGIFY(REPO_MAJOR_VERSION) "." STRINGIFY(REPO_MINOR_VERSION)
00079      #endif
00080      #ifndef REPO_DATE
00081          /* The build date, instead of the repo date. */
00082          #define REPO_DATE __DATE__
00083      #endif
00084      #ifndef REPO_REVISION
00085          #define VERSIONSTR FORMNAME " " REPO_VERSION " (" REPO_DATE ", " REPO_REVISION ")"
00086      #else
00087          #define VERSIONSTR FORMNAME " " REPO_VERSION " (" REPO_DATE ")"
00088      #endif
00089      #define MAJORVERSION REPO_MAJOR_VERSION
00090      #define MINORVERSION REPO_MINOR_VERSION
00091  #else
00092      /*
00093       * Otherwise, form3.h defines MAJORVERSION, MINORVERSION and PRODUCTIONDATE,
00094       * possibly BETAVERSION.
00095       */
00096      #ifdef BETAVERSION
00097          #define VERSIONSTR__ STRINGIFY(MAJORVERSION) "." STRINGIFY(MINORVERSION) "Beta"
00098      #else
00099          #define VERSIONSTR__ STRINGIFY(MAJORVERSION) "." STRINGIFY(MINORVERSION)
00100      #endif
00101      #define VERSIONSTR FORMNAME " " VERSIONSTR__ " (" PRODUCTIONDATE ")"
00102  #endif
00103
00104  /*
00105      #] includes :
00106      #[ PrintHeader :
00107  */
00108
00114  static void PrintHeader(int with_full_info)
00115  {
00116      #ifdef WITHMPI
00117          if ( PF.me == MASTER && !AM.silent ) {
00118      #else
00119          if ( !AM.silent ) {
00120      #endif
00121          char buffer1[250], buffer2[80], *s = buffer1, *t = buffer2;
00122          WORD length, n;
00123          for ( n = 0; n < 250; n++ ) buffer1[n] = ' ';
00124          /*
00125           * NOTE: we expect that the compiler optimizes strlen("string literal")
00126           * to just a number.
00127           */
00128          if ( strlen(VERSIONSTR) <= 100 ) {
00129              strcpy(s, VERSIONSTR);
00130              s += strlen(VERSIONSTR);
00131              *s = 0;
00132          }
00133          else {
00134              /*
00135               * Truncate when it is too long.
00136               */
00137              strncpy(s, VERSIONSTR, 97);
00138              s[97] = '.';
00139              s[98] = '.';
00140              s[99] = ')';
00141              s[100] = '\0';
00142              s += 100;
00143          }
00144      /*
00145          By now we omit the message about 32/64 bits. It should all be 64.
00146
00147          s += sprintf(s, 250 - (s - buffer1), " %d-bits", (WORD) (sizeof(WORD) * 16));
00148          *s = 0;
00149      */
00150          if ( with_full_info ) {
00151      #if defined(WITHPTHREADS) || defined(WITHMPI)
00152      #if defined(WITHPTHREADS)

```

```

00153         int nworkers = AM.totalnumberofthreads-1;
00154 #elif defined(WITHMPI)
00155         int nworkers = PF.numtasks-1;
00156 #endif
00157         s += sprintf(s,250-(s-buffer1)," %d worker",nworkers);
00158         *s = 0;
00159 /*         while ( *s ) s++; */
00160         if ( nworkers != 1 ) {
00161             *s++ = 's';
00162             *s = '\\0';
00163         }
00164 #endif
00165
00166         sprintf(t,80-(t-buffer2),"Run: %s",MakeDate());
00167         while ( *t ) t++;
00168
00169         /*
00170          * Align the date to the right, if it fits in a line.
00171          */
00172         length = (s-buffer1) + (t-buffer2);
00173         if ( length+2 <= AC.LineLength ) {
00174             for ( n = AC.LineLength-length; n > 0; n-- ) *s++ = ' ';
00175             *s = 0;
00176             strcat(s,buffer2);
00177             while ( *s ) s++;
00178         }
00179         else {
00180             *s = 0;
00181             strcat(s," ");
00182             while ( *s ) s++;
00183             *s = 0;
00184             strcat(s,buffer2);
00185             while ( *s ) s++;
00186         }
00187     }
00188
00189     /*
00190     * If the header information doesn't fit in a line, we need to extend
00191     * the line length temporarily.
00192     */
00193     length = s-buffer1;
00194     if ( length <= AC.LineLength ) {
00195         MesPrint("%s",buffer1);
00196     }
00197     else {
00198         WORD oldLineLength = AC.LineLength;
00199         AC.LineLength = length;
00200         MesPrint("%s",buffer1);
00201         AC.LineLength = oldLineLength;
00202     }
00203 }
00204 #ifdef WINDOWS
00205     PrintDeprecation("the native Windows version", "issues/623");
00206 #endif
00207 #if BITSINWORD == 16
00208     PrintDeprecation("the 32-bit version", "issues/624");
00209 #endif
00210 #ifdef WITHMPI
00211     PrintDeprecation("the MPI version (ParFORM)", "issues/625");
00212 #endif
00213     if ( AC.CheckpointFlag ) {
00214         PrintDeprecation("the checkpoint mechanism", "issues/626");
00215     }
00216 }
00217
00218 /*
00219     #] PrintHeader :
00220     #[ DoTail :
00221
00222     Routine reads the command tail and handles the commandline options.
00223     It sets the flags for later actions and stored pathnames for
00224     the setup file, include/prc/sub directories etc.
00225     Finally the name of the program is passed on.
00226     Note that we do not support interactive use yet. This will come
00227     to pass in the distant future when we can couple STedi to FORM.
00228     Routine made 23-feb-1993 by J.Vermaseren
00229 */
00230
00231 #ifdef WITHINTERACTION
00232     static UBYTE deflogname[] = "formsession.log";
00233 #endif
00234
00235 #define TAKEPATH(x) if(s[1]== '=' ){x=s+2;} else{x=*argv++;argc--;}
00236
00237 int DoTail(int argc, UBYTE **argv)
00238 {
00239     int errorflag = 0, onlyversion = 1;

```

```

00240     UBYTE *s, *t, *copy;
00241     int threadnum = 0;
00242     argc--; argv++;
00243     AM.ClearStore = 0;
00244     AM.TimeLimit = 0;
00245     AM.LogType = -1;
00246     AM.HoldFlag = AM.qError = AM.Interact = AM.FileOnlyFlag = 0;
00247     AM.InputFileName = AM.LogFileName = AM.IncDir = AM.TempDir = AM.TempSortDir =
00248     AM.SetupDir = AM.SetupFile = AM.Path = 0;
00249     AM.FromStdin = 0;
00250     /* Always use MultiRun, "-M" option is now ignored. */
00251     AM.MultiRun = 1;
00252     if ( argc < 1 ) {
00253         onlyversion = 0;
00254         goto printversion;
00255     }
00256     while ( argc >= 1 ) {
00257         s = *argv++; argc--;
00258         if ( *s == '-' || ( *s == '/' && ( argc > 0 || AM.Interact ) ) ) {
00259             s++;
00260             switch (*s) {
00261                 case 'c': /* Error checking only */
00262                     AM.qError = 1; break;
00263                 case 'C': /* Next arg is filename of log file */
00264                     TAKEPATH(AM.LogFileName) break;
00265                 case 'D':
00266                 case 'd': /* Next arg is define preprocessor var. */
00267                     t = copy = strDupl(*argv,"Dotail");
00268                     while ( *t && *t != '=' ) t++;
00269                     if ( *t == 0 ) {
00270                         if ( PutPreVar(copy, (UBYTE *) "1", 0, 0) < 0 ) return(-1);
00271                     }
00272                     else {
00273                         *t++ = 0;
00274                         if ( PutPreVar(copy, t, 0, 0) < 0 ) return(-1);
00275                         t[-1] = '=';
00276                     }
00277                     M_free(copy, "-d prevar");
00278                     argv++; argc--; break;
00279                 case 'f': /* Output only to regular log file */
00280                     AM.FileOnlyFlag = 1; AM.LogType = 0; break;
00281                 case 'F': /* Output only to log file. Further like L. */
00282                     AM.FileOnlyFlag = 1; AM.LogType = 1; break;
00283                 case 'h': /* For old systems: wait for key before exit */
00284                     AM.HoldFlag = 1; break;
00285                 case 'i':
00286                     if ( StrCmp(s, (UBYTE *) "ignore-deprecation") == 0 ) {
00287                         AM.IgnoreDeprecation = 1;
00288                         break;
00289                     }
00290 #ifndef WITHINTERACTION
00291                     /* Interactive session (not used yet) */
00292                     AM.Interact = 1;
00293                     break;
00294 #else
00295                     goto IllegalOption;
00296 #endif
00297                 case 'I': /* Next arg is dir for inc/prc/sub files */
00298                     TAKEPATH(AM.IncDir) break;
00299                 case 'l': /* Make regular log file */
00300                     if ( s[1] == 'l' ) AM.LogType = 1; /*compatibility! */
00301                     else AM.LogType = 0;
00302                     break;
00303                 case 'L': /* Make log file with only final statistics */
00304                     AM.LogType = 1; break;
00305                 case 'M': /* Multirun. Name of tempfiles will contain PID */
00306                     /* This option is now ignored. We always use MultiRun. */
00307                     /* AM.MultiRun = 1; */
00308                     break;
00309                 case 'm': /* Read number of threads */
00310                 case 'w': /* Read number of workers */
00311                     t = s++;
00312                     threadnum = 0;
00313                     while ( *s >= '0' && *s <= '9' )
00314                         threadnum = 10*threadnum + *s++ - '0';
00315                     if ( *s ) {
00316 #ifndef WITHMPI
00317                         if ( PF.me == MASTER )
00318 #endif
00319                         printf("Illegal value for option m or w: %s\n", t);
00320                         errorflag++;
00321                     }
00322                     /* if ( threadnum == 1 ) threadnum = 0; */
00323                     threadnum++;
00324                     break;
00325                 case 'W': /* Print the wall-clock time on the master. */
00326                     AM.ggWTimeStatsFlag = 1;

```

```

00327                                     break;
00328 /*
00329         case 'n':
00330             Reserved for number of slaves without MPI
00331 */
00332         case 'p':
00333 #ifdef WITHEXTERNALCHANNEL
00334     /*There are two possibilities: -p|-pipe*/
00335     if(s[1]=='i'){
00336         if( (s[2]=='p') && (s[3]=='e') && (s[4]=='\0') ){
00337             argc--;
00338             /*Initialize pre-set external channels, see
00339             the file extcmd.c:*/
00340             if(initPresetExternalChannels(*argv++,AX.timeout)<1){
00341 #ifdef WITHMPI
00342                 if ( PF.me == MASTER )
00343 #endif
00344                 printf("Error initializing preset external channels\n");
00345                 errorflag++;
00346             }
00347             AX.timeout=-1; /*This indicates that preset channels
00348             are initialized from cmdline*/
00349         }else{
00350 #ifdef WITHMPI
00351             if ( PF.me == MASTER )
00352 #endif
00353             printf("Illegal option in call of FORM: %s\n",s);
00354             errorflag++;
00355         }
00356     }else
00357 #else
00358     if ( s[1] ) {
00359         if ( ( s[1]=='i' ) && ( s[2] == 'p' ) && (s[3] == 'e' )
00360             && ( s[4] == '\0' ) ){
00361 #ifdef WITHMPI
00362             if ( PF.me == MASTER )
00363 #endif
00364             printf("Illegal option: Pipes not supported on this system.\n");
00365         }
00366         else {
00367 #ifdef WITHMPI
00368             if ( PF.me == MASTER )
00369 #endif
00370             printf("Illegal option: %s\n",s);
00371         }
00372         errorflag++;
00373     }
00374     else
00375 #endif
00376     {
00377         /* Next arg is a path variable like in environment */
00378         TAKEPATH(AM.Path)
00379     }
00380     break;
00381 case 'q': /* Quiet option. Only output. Same as -si */
00382     AM.silent = 1; break;
00383 case 'R': /* recover from saved snapshot */
00384     AC.CheckpointFlag = -1;
00385     break;
00386 case 's': /* Next arg is dir with form.set to be used */
00387     if ( ( s[1] == 'o' ) && ( s[2] == 'r' ) && ( s[3] == 't' ) ) {
00388         if(s[4]=='=' ) {
00389             AM.TempSortDir = s+5;
00390         }
00391         else {
00392             AM.TempSortDir = *argv++;
00393             argc--;
00394         }
00395     }
00396     else if ( s[1] == 'i' ) { /* compatibility: silent/quiet */
00397         AM.silent = 1;
00398     }
00399     else {
00400         TAKEPATH(AM.SetupDir)
00401     }
00402     break;
00403 case 'S': /* Next arg is setup file */
00404     TAKEPATH(AM.SetupFile) break;
00405 case 't': /* Next arg is directory for temp files */
00406     if ( s[1] == 's' ) {
00407         s++;
00408         AM.havesortdir = 1;
00409         TAKEPATH(AM.TempSortDir)
00410     }
00411     else {
00412         TAKEPATH(AM.TempDir)
00413     }

```



```

00414             break;
00415         case 'T': /* Print the total size used at end of job */
00416             AM.PrintTotalSize = 1; break;
00417         case 'v':
00418             printversion;;
00419 #ifdef WITHMPI
00420             if ( PF.me == MASTER )
00421 #endif
00422                 PrintHeader(0);
00423             if ( onlyversion ) return(1);
00424             goto NoFile;
00425         case 'y': /* Preprocessor dumps output. No compilation. */
00426             AP.PreDebug = PREPROONLY; break;
00427         case 'z': /* The number following is a time limit in sec. */
00428             t = s++;
00429             AM.TimeLimit = 0;
00430             while ( *s >= '0' && *s <= '9' )
00431                 AM.TimeLimit = 10*AM.TimeLimit + *s++ - '0';
00432             break;
00433         case 'Z': /* Removes the .str file on crash, no matter its contents */
00434             AM.ClearStore = 1; break;
00435         case '\0': /* "-" to use STDIN for the input. */
00436 #ifdef WITHMPI
00437             /* At the moment, ParFORM doesn't implement STDIN broadcasts. */
00438             if ( PF.me == MASTER )
00439                 printf("Sorry, reading STDIN as input is currently not supported by
ParFORM\n");
00440             errorflag++;
00441 #endif
00442             AM.FromStdin = 1;
00443             AC.NoShowInput = 1; // disable input echoing by default
00444             break;
00445         default:
00446             if ( FG.cTable[*s] == 1 ) {
00447                 AM.SkipClears = 0; t = s;
00448                 while ( FG.cTable[*t] == 1 )
00449                     AM.SkipClears = 10*AM.SkipClears + *t++ - '0';
00450                 if ( *t != 0 ) {
00451 #ifdef WITHMPI
00452                     if ( PF.me == MASTER )
00453 #endif
00454                         printf("Illegal numerical option in call of FORM: %s\n",s);
00455                         errorflag++;
00456                     }
00457                 }
00458                 else {
00459                     IllegalOption:
00460 #ifdef WITHMPI
00461                     if ( PF.me == MASTER )
00462 #endif
00463                         printf("Illegal option in call of FORM: %s\n",s);
00464                         errorflag++;
00465                     }
00466                     break;
00467             }
00468         }
00469         else if ( argc == 0 && !AM.Interact ) AM.InputFileName = argv[-1];
00470         else {
00471 #ifdef WITHMPI
00472             if ( PF.me == MASTER )
00473 #endif
00474             printf("Illegal option in call of FORM: %s\n",s);
00475             errorflag++;
00476         }
00477     }
00478     AM.totalnumberofthreads = threadnum;
00479     if ( AM.InputFileName ) {
00480         if ( AM.FromStdin ) {
00481             printf("STDIN and the input filename cannot be specified simultaneously\n");
00482             errorflag++;
00483         }
00484         s = AM.InputFileName;
00485         while ( *s ) s++;
00486         if ( s < AM.InputFileName+4 ||
00487             s[-4] != '.' || s[-3] != 'f' || s[-2] != 'r' || s[-1] != 'm' ) {
00488             t = (UBYTE *)Malloc1((s-AM.InputFileName)+5,"adding .frm");
00489             s = AM.InputFileName;
00490             AM.InputFileName = t;
00491             while ( *s ) *t++ = *s++;
00492             *t++ = '.'; *t++ = 'f'; *t++ = 'r'; *t++ = 'm'; *t = 0;
00493         }
00494         if ( AM.LogType >= 0 && AM.LogFileName == 0 ) {
00495             AM.LogFileName = strDup1(AM.InputFileName,"name of logfile");
00496             s = AM.LogFileName;
00497             while ( *s ) s++;
00498             s[-3] = 'l'; s[-2] = 'o'; s[-1] = 'g';
00499         }

```

```

00500     }
00501 #ifdef WITHINTERACTION
00502     else if ( AM.Interact ) {
00503         if ( AM.LogType >= 0 ) {
00504             /*
00505                 We may have to do better than just taking a name.
00506                 It is not unique! This will be left for later.
00507             */
00508             AM.LogFileName = deflogname;
00509         }
00510     }
00511 #endif
00512     else if ( AM.FromStdin ) {
00513         /* Do nothing. */
00514     }
00515     else {
00516 NoFile:
00517 #ifdef WITHMPI
00518         if ( PF.me == MASTER )
00519 #endif
00520         printf("No filename specified in call of FORM\n");
00521         errorflag++;
00522     }
00523     if ( AM.Path == 0 ) AM.Path = (UBYTE *)getenv("FORMPATH");
00524     if ( AM.Path ) {
00525         /*
00526          * AM.Path is taken from argv or getenv. Reallocate it to avoid invalid
00527          * frees when AM.Path has to be changed.
00528          */
00529         AM.Path = strdup(AM.Path, "DoTail Path");
00530     }
00531     return(errorflag);
00532 }
00533
00534 /*
00535     #] DoTail :
00536     #[ OpenInput :
00537
00538     Major task here after opening is to skip the proper number of
00539     .clear instructions if so desired without using interpretation
00540 */
00541
00542 int OpenInput(void)
00543 {
00544     int oldNoShowInput = AC.NoShowInput;
00545     UBYTE c;
00546     if ( !AM.Interact ) {
00547         if ( AM.FromStdin ) {
00548             if ( OpenStream(0, INPUTSTREAM, 0, PRENOACTION) == 0 ) {
00549                 Error0("Cannot open STDIN");
00550                 return(-1);
00551             }
00552         }
00553         else {
00554             if ( OpenStream(AM.InputFileName, FILESTREAM, 0, PRENOACTION) == 0 ) {
00555                 Error1("Cannot open file", AM.InputFileName);
00556                 return(-1);
00557             }
00558             if ( AC.CurrentStream->inbuffer <= 0 ) {
00559                 Error1("No input in file", AM.InputFileName);
00560                 return(-1);
00561             }
00562         }
00563         AC.NoShowInput = 1;
00564         while ( AM.SkipClears > 0 ) {
00565             c = GetInput();
00566             if ( c == ENDOFINPUT ) {
00567                 Error0("Not enough .clear instructions in input file");
00568             }
00569             if ( c == '\\\\' ) {
00570                 c = GetInput();
00571                 if ( c == ENDOFINPUT )
00572                     Error0("Not enough .clear instructions in input file");
00573                 continue;
00574             }
00575             if ( c == ' ' || c == '\t' ) continue;
00576             if ( c == '.' ) {
00577                 c = GetInput();
00578                 if ( tolower(c) == 'c' ) {
00579                     c = GetInput();
00580                     if ( tolower(c) == 'l' ) {
00581                         c = GetInput();
00582                         if ( tolower(c) == 'e' ) {
00583                             c = GetInput();
00584                             if ( tolower(c) == 'a' ) {
00585                                 c = GetInput();
00586                                 if ( tolower(c) == 'r' ) {

```

```

00587             c = GetInput();
00588             if ( FG.cTable[c] > 2 ) {
00589                 AM.SkipClears--;
00590             }
00591         }
00592     }
00593 }
00594 }
00595 }
00596 while ( c != '\n' && c != '\r' && c != ENDOFINPUT ) {
00597     c = GetInput();
00598     if ( c == '\\ ' ) c = GetInput();
00599 }
00600 }
00601 else if ( c == '\n' || c == '\r' ) continue;
00602 else {
00603     while ( ( c = GetInput() ) != '\n' && c != '\r' ) {
00604         if ( c == ENDOFINPUT ) {
00605             Error0("Not enough .clear instructions in input file");
00606         }
00607     }
00608 }
00609 }
00610 AC.NoShowInput = oldNoShowInput;
00611 }
00612 if ( AM.LogFileName ) {
00613 #ifndef WITHMPI
00614     if ( PF.me != MASTER ) {
00615         /*
00616          * Only the master writes to the log file. On slaves, we need
00617          * a dummy handle, without opening the file.
00618          */
00619         extern FILES **filelist; /* in tools.c */
00620         int i = CreateHandle();
00621         RWLOCKW(AM.handlelock);
00622         filelist[i] = (FILES *)123; /* Must be nonzero to prevent a reuse in CreateHandle. */
00623         UNRWLOCK(AM.handlelock);
00624         AC.LogHandle = i;
00625     }
00626     else
00627 #endif
00628     if ( AC.CheckpointFlag != -1 ) {
00629         if ( ( AC.LogHandle = CreateLogFile((char *) (AM.LogFileName)) ) < 0 ) {
00630             Error1("Cannot create logfile",AM.LogFileName);
00631             return(-1);
00632         }
00633     }
00634     else {
00635         if ( ( AC.LogHandle = OpenAddFile((char *) (AM.LogFileName)) ) < 0 ) {
00636             Error1("Cannot re-open logfile",AM.LogFileName);
00637             return(-1);
00638         }
00639     }
00640 }
00641 return(0);
00642 }
00643
00644 /*
00645     #] OpenInput :
00646     #[ ReserveTempFiles :
00647
00648     Order of preference:
00649     a: if there is a path in the commandtail, take that.
00650     b: if none, try in the form.set file.
00651     c: if still none, try in the environment for the variable FORMTMP
00652     d: if still none, try the current directory.
00653
00654     The parameter indicates action in the case of multithreaded running.
00655     par = 0 : We just run on a single processor. Keep everything normal.
00656     par = 1 : Multithreaded running startup phase 1.
00657     par = 2 : Multithreaded running startup phase 2.
00658 */
00659
00660 UBYTE *emptystring = (UBYTE *)".";
00661 UBYTE *defaulttempfilename = (UBYTE *)"xformxxx.str";
00662 /* This is the length of the above default, but with 7 spaces for PID digits
00663    instead of the "xxx". (Previously FORM used 5 digits and this value was 14) */
00664 #define DEFAULTFNAMLENGTH 16
00665
00666 void ReserveTempFiles(int par)
00667 {
00668     GETIDENTITY
00669     SETUPPARAMETERS *sp;
00670     UBYTE *s, *t, *tendir, *tendir2, c;
00671     int i = 0;
00672     WORD j;
00673     if ( par == 0 || par == 1 ) {

```

```

00674     if ( AM.TempDir == 0 ) {
00675         sp = GetSetupPar((UBYTE *)"tempdir");
00676         if ( ( sp->flags & USEDFLAG ) != USEDFLAG ) {
00677             AM.TempDir = (UBYTE *)getenv("FORMTMP");
00678             if ( AM.TempDir == 0 ) AM.TempDir = emptystring;
00679         }
00680         else AM.TempDir = (UBYTE *) (sp->value);
00681     }
00682     if ( AM.TempSortDir == 0 ) {
00683         if ( AM.havesortdir ) {
00684             sp = GetSetupPar((UBYTE *)"tempSortdir");
00685             AM.TempSortDir = (UBYTE *) (sp->value);
00686         }
00687         else {
00688             AM.TempSortDir = (UBYTE *)getenv("FORMTMPSORT");
00689             if ( AM.TempSortDir == 0 ) AM.TempSortDir = AM.TempDir;
00690         }
00691     }
00692     /*
00693     We have now in principle a path but we will use its first element only.
00694     Later that should become more complicated. Then we will use a path and
00695     when one device is full we can continue on the next one.
00696     */
00697     s = AM.TempDir; i = 200; /* Some extra for VMS */
00698     while ( *s && *s != PATHSEPARATOR ) { if ( *s == '\\\' ) s++; s++; i++; }
00699     FG.fnamesize = sizeof(UBYTE)*(i+DEFAULTFNAMELENGTH);
00700     FG.fname = (char *)Malloc1(FG.fnamesize,"name for temporary files");
00701     s = AM.TempDir; t = (UBYTE *)FG.fname;
00702     while ( *s && *s != PATHSEPARATOR ) { if ( *s == '\\\' ) s++; *t++ = *s++; }
00703     if ( (char *)t > FG.fname && t[-1] != SEPARATOR && t[-1] != ALTSEPARATOR )
00704         *t++ = SEPARATOR;
00705     *t = 0;
00706     tenddir = t;
00707     FG.fnamebase = t-(UBYTE *) (FG.fname);
00708
00709     s = AM.TempSortDir; i = 200; /* Some extra for VMS */
00710     while ( *s && *s != PATHSEPARATOR ) { if ( *s == '\\\' ) s++; s++; i++; }
00711
00712     FG.fname2size = sizeof(UBYTE)*(i+DEFAULTFNAMELENGTH);
00713     FG.fname2 = (char *)Malloc1(FG.fname2size,"name for sort files");
00714     s = AM.TempSortDir; t = (UBYTE *)FG.fname2;
00715     while ( *s && *s != PATHSEPARATOR ) { if ( *s == '\\\' ) s++; *t++ = *s++; }
00716     if ( (char *)t > FG.fname2 && t[-1] != SEPARATOR && t[-1] != ALTSEPARATOR )
00717         *t++ = SEPARATOR;
00718     *t = 0;
00719     tenddir2 = t;
00720     FG.fname2base = t-(UBYTE *) (FG.fname2);
00721
00722     t = tenddir;
00723     s = defaulttmpfilename;
00724     #ifdef WITHMPI
00725     {
00726         int iii;
00727         iii = sprintf((char*)t,FG.fnamesize-((char*)t)-FG.fname),"%d",PF.me);
00728         t+= iii;
00729         s+= iii; /* in case defaulttmpfilename is too short */
00730     }
00731     #endif
00732     while ( *s ) *t++ = *s++;
00733     *t = 0;
00734     /*
00735     There are problems when running many FORM jobs at the same time
00736     from make or minos. If they start up simultaneously, occasionally
00737     they can make the same .str file. We prevent this with first trying
00738     a file that contains the digits of the pid. If this file
00739     has already been taken we fall back on the old scheme.
00740     The whole is controlled with the -M (MultiRun) parameter in the
00741     command tail.
00742     */
00743     if ( AM.MultiRun ) {
00744         int num = ((int)GetPID())%10000000;
00745         t += 4;
00746         *t = 0;
00747         t[-1] = 'r';
00748         t[-2] = 't';
00749         t[-3] = 's';
00750         t[-4] = '.';
00751         t[-5] = (UBYTE) ('0' + num%10);
00752         t[-6] = (UBYTE) ('0' + (num/10)%10);
00753         t[-7] = (UBYTE) ('0' + (num/100)%10);
00754         t[-8] = (UBYTE) ('0' + (num/1000)%10);
00755         t[-9] = (UBYTE) ('0' + (num/10000)%10);
00756         t[-10] = (UBYTE) ('0' + (num/100000)%10);
00757         t[-11] = (UBYTE) ('0' + num/1000000);
00758         if ( ( AC.StoreHandle = CreateFile((char *)FG.fname) ) < 0 ) {
00759             t[-5] = 'x'; t[-6] = 'x'; t[-7] = 'x'; t[-8] = 'x'; t[-9] = 'x'; t[-10] = 'x'; t[-11] =

```

```

00760         goto classic;
00761     }
00762 }
00763 else
00764 {
00765 classic;;
00766     for(;;) {
00767         if ( ( AC.StoreHandle = OpenFile((char *)FG.fname) ) < 0 ) {
00768             if ( ( AC.StoreHandle = CreateFile((char *)FG.fname) ) >= 0 ) break;
00769         }
00770         else CloseFile(AC.StoreHandle);
00771         c = t[-5];
00772         if ( c == 'x' ) t[-5] = '0';
00773         else if ( c == '9' ) {
00774             t[-5] = '0';
00775             c = t[-6];
00776             if ( c == 'x' ) t[-6] = '0';
00777             else if ( c == '9' ) {
00778                 t[-6] = '0';
00779                 c = t[-7];
00780                 if ( c == 'x' ) t[-7] = '0';
00781                 else if ( c == '9' ) {
00782 /*
00783             Note that we tried 1111 names!
00784 */
00785             MesPrint("Name space for temp files exhausted");
00786             t[-7] = 0;
00787             MesPrint("Please remove files of the type %s or try a different directory"
00788                 ,FG.fname);
00789             Terminate(-1);
00790         }
00791         else t[-7] = (UBYTE)(c+1);
00792     }
00793     else t[-6] = (UBYTE)(c+1);
00794 }
00795     else t[-5] = (UBYTE)(c+1);
00796 }
00797 }
00798 /*
00799 Now we should make sure that the tempsortdir cq tempsortfilename makes it
00800 into a similar construction.
00801 */
00802 s = tenddir; t = tenddir2; while ( *s ) *t++ = *s++;
00803 *t = 0;
00804
00805 /*
00806 Now we should assign a name to the main sort file and the two stage 4 files.
00807 */
00808 AM.S0->file.name = (char *)Malloc1(sizeof(char)*(i+DEFAULTFNAMELENGTH),"name for temporary
files");
00809 s = (UBYTE *)AM.S0->file.name;
00810 t = (UBYTE *)FG.fname2;
00811 i = 1;
00812 while ( *t ) { *s++ = *t++; i++; }
00813 s[-2] = 'o'; *s = 0;
00814 }
00815 /*
00816 Try to create the sort file already, so we can Terminate earlier if this fails.
00817 */
00818 #ifdef WITHPTHREADS
00819 if ( par <= 1 ) {
00820 #endif
00821     if ( ( AM.S0->file.handle = CreateFile((char *)AM.S0->file.name) ) < 0 ) {
00822         MesPrint("Could not create sort file: %s", AM.S0->file.name);
00823         Terminate(-1);
00824     };
00825     /* Close and clean up the test file */
00826     CloseFile(AM.S0->file.handle);
00827     AM.S0->file.handle = -1;
00828     remove(AM.S0->file.name);
00829 #ifdef WITHPTHREADS
00830 }
00831 #endif
00832 /*
00833 With the stage4 and scratch file names we have to be a bit more careful.
00834 They are to be allocated after the threads are initialized when there
00835 are threads of course.
00836 */
00837 if ( par == 0 ) {
00838     s = (UBYTE *)((void *) (FG.fname2)); i = 0;
00839     while ( *s ) { s++; i++; }
00840     /* +1 for null terminator */
00841     s = (UBYTE *)Malloc1(sizeof(char)*(i+1),"name for stage4 file a");
00842     AR.FoStage4[1].name = (char *)s;
00843     t = (UBYTE *)FG.fname2;
00844     while ( *t ) *s++ = *t++;
00845     s[-2] = '4'; s[-1] = 'a'; *s = 0;

```

```

00846     s = (UBYTE *) ((void *) (FG.fname)); i = 0;
00847     while ( *s ) { s++; i++; }
00848     /* +1 for null terminator */
00849     s = (UBYTE *) Malloc1(sizeof(char)*(i+1), "name for stage4 file b");
00850     AR.FoStage4[0].name = (char *)s;
00851     t = (UBYTE *)FG.fname;
00852     while ( *t ) *s++ = *t++;
00853     s[-2] = '4'; s[-1] = 'b'; *s = 0;
00854     for ( j = 0; j < 3; j++ ) {
00855         /* +1 for null terminator */
00856         s = (UBYTE *) Malloc1(sizeof(char)*(i+1), "name for scratch file");
00857         AR.Fscr[j].name = (char *)s;
00858         t = (UBYTE *)FG.fname;
00859         while ( *t ) *s++ = *t++;
00860         s[-2] = 'c'; s[-1] = (UBYTE) ('0'+j); *s = 0;
00861     }
00862 }
00863 #ifdef WITHPTHREADS
00864     else if ( par == 2 ) {
00865         size_t tname;
00866         s = (UBYTE *) ((void *) (FG.fname2)); i = 0;
00867         while ( *s ) { s++; i++; }
00868         /* +1 for null terminator, +10 for 32bit int, +1 for "." */
00869         tname = sizeof(char)*(i+12);
00870         s = (UBYTE *) Malloc1(tname, "name for stage4 file a");
00871         snprintf((char *)s, tname, "%s.%d", FG.fname2, AT.identity);
00872         s[i-2] = '4'; s[i-1] = 'a';
00873         AR.FoStage4[1].name = (char *)s;
00874         s = (UBYTE *) ((void *) (FG.fname)); i = 0;
00875         while ( *s ) { s++; i++; }
00876         /* +1 for null terminator, +10 for 32bit int, +1 for "." */
00877         tname = sizeof(char)*(i+12);
00878         s = (UBYTE *) Malloc1(tname, "name for stage4 file b");
00879         snprintf((char *)s, tname, "%s.%d", FG.fname, AT.identity);
00880         s[i-2] = '4'; s[i-1] = 'b';
00881         AR.FoStage4[0].name = (char *)s;
00882         if ( AT.identity == 0 ) {
00883             for ( j = 0; j < 3; j++ ) {
00884                 /* +1 for null terminator */
00885                 s = (UBYTE *) Malloc1(sizeof(char)*(i+1), "name for scratch file");
00886                 AR.Fscr[j].name = (char *)s;
00887                 t = (UBYTE *)FG.fname;
00888                 while ( *t ) *s++ = *t++;
00889                 s[-2] = 'c'; s[-1] = (UBYTE) ('0'+j); *s = 0;
00890             }
00891         }
00892     }
00893 #endif
00894 }
00895
00896 /*
00897     #] ReserveTempFiles :
00898     #[ StartVariables :
00899 */
00900
00901 #ifdef WITHPTHREADS
00902 ALLPRIVATES *DummyPointer = 0;
00903 #endif
00904
00905 void StartVariables(void)
00906 {
00907     int i, ii;
00908     PUTZERO(AM.zeropos);
00909     StartPrepro();
00910     /*
00911         The module counter:
00912     */
00913     AC.CModule=0;
00914 #ifdef WITHPTHREADS
00915     /*
00916         We need a value in AB because in the startup some routines may call AB[0].
00917     */
00918     AB = (ALLPRIVATES **) &DummyPointer;
00919 #endif
00920     /*
00921         separators used to delimit arguments in #call and #do, by default ', ' and '|':
00922         Be sure, it is en empty set:
00923     */
00924     set_sub(AC.separators, AC.separators, AC.separators);
00925     set_set(' ', AC.separators);
00926     set_set('|', AC.separators);
00927
00928     AM.BracketFactors[0] = 8;
00929     AM.BracketFactors[1] = SYMBOL;
00930     AM.BracketFactors[2] = 4;
00931     AM.BracketFactors[3] = FACTORSYMBOL;
00932     AM.BracketFactors[4] = 1;

```

```

00933     AM.BraceFactors[5] = 1;
00934     AM.BraceFactors[6] = 1;
00935     AM.BraceFactors[7] = 3;
00936
00937     AM.SkipClears = 0;
00938     AC.Cnumpows = 0;
00939     AC.OutputMode = 72;
00940     AC.OutputSpaces = NORMALFORMAT;
00941     AC.LineLength = 79;
00942     AM.gIsFortran90 = AC.IsFortran90 = ISNOTFORTRAN90;
00943     AM.gFortran90Kind = AC.Fortran90Kind = 0;
00944     AM.gCnumpows = 0;
00945     AC.exprfillwarning = 0;
00946     AM.gLineLength = 79;
00947     AM.OutBufSize = 80;
00948     AM.MaxStreamSize = MAXFILESTREAMSIZE;
00949     AP.MaxPreAssignLevel = 4;
00950     AC.iBufferSize = 512;
00951     AP.pSize = 128;
00952     AP.MaxPreIfLevel = 10;
00953     AP.cComChar = AP.ComChar = '*';
00954     AP.firstnamespace = 0;
00955     AP.lastnamespace = 0;
00956     AP.fullnamesize = 127;
00957     AP.fullname = (UBYTE *)Malloc1((AP.fullnamesize+1)*sizeof(UBYTE), "AP.fullname");
00958     AM.OffsetVector = -2*WILDOFFSET+MINSPEC;
00959     AC.cbufList.num = 0;
00960     AM.hparallelflag = AM.gparallelflag =
00961     AC.parallelflag = AC.mparallelflag = PARALLELFLAG;
00962 #ifdef WITHMPI
00963     if ( PF.numtasks < 2 ) AM.hparallelflag |= NOPARALLEL_NPROC;
00964 #endif
00965     AC.tablefilling = 0;
00966     AC.models = 0;
00967     AC.nummodels = 0;
00968     AC.ModelLevel = 0;
00969     AC.modelspace = 0;
00970     AM.resetTimeOnClear = 1;
00971     AM.gnumextrasym = AM.ggnumextrasym = 0;
00972     AM.havesortdir = 0;
00973     AM.SpectatorFiles = 0;
00974     AM.NumSpectatorFiles = 0;
00975     AM.SizeForSpectatorFiles = 0;
00976 /*
00977     Information for the lists of variables. Part of error message and size:
00978 */
00979     AP.ProcList.message = "procedure";
00980     AP.ProcList.size = sizeof(PROCEDURE);
00981     AP.LoopList.message = "doloop";
00982     AP.LoopList.size = sizeof(DOLOOP);
00983     AP.PreVarList.message = "PreVariable";
00984     AP.PreVarList.size = sizeof(PREVAR);
00985     AC.SymbolList.message = "symbol";
00986     AC.SymbolList.size = sizeof(struct SyMbOl);
00987     AC.IndexList.message = "index";
00988     AC.IndexList.size = sizeof(struct InDeX);
00989     AC.VectorList.message = "vector";
00990     AC.VectorList.size = sizeof(struct VeCtOr);
00991     AC.FunctionList.message = "function";
00992     AC.FunctionList.size = sizeof(struct FuNcTiOn);
00993     AC.SetList.message = "set";
00994     AC.SetList.size = sizeof(struct SeTs);
00995     AC.SetElementList.message = "set element";
00996     AC.SetElementList.size = sizeof(WORD);
00997     AC.ExpressionList.message = "expression";
00998     AC.ExpressionList.size = sizeof(struct ExPrEsSiOn);
00999     AC.cbufList.message = "compiler buffer";
01000     AC.cbufList.size = sizeof(CBUF);
01001     AC.Channellist.message = "channel buffer";
01002     AC.Channellist.size = sizeof(CHANNEL);
01003     AP.DollarList.message = "$-variable";
01004     AP.DollarList.size = sizeof(struct DoLLaRs);
01005     AC.DubiousList.message = "ambiguous variable";
01006     AC.DubiousList.size = sizeof(struct DuBiOuS);
01007     AC.TableBaseList.message = "list of tablebases";
01008     AC.TableBaseList.size = sizeof(DBASE);
01009     AC.TestValue = 0;
01010     AC.InnerTest = 0;
01011
01012     AC.AutoSymbolList.message = "autosymbol";
01013     AC.AutoSymbolList.size = sizeof(struct SyMbOl);
01014     AC.AutoIndexList.message = "autoindex";
01015     AC.AutoIndexList.size = sizeof(struct InDeX);
01016     AC.AutoVectorList.message = "autovector";
01017     AC.AutoVectorList.size = sizeof(struct VeCtOr);
01018     AC.AutoFunctionList.message = "autofunction";
01019     AC.AutoFunctionList.size = sizeof(struct FuNcTiOn);

```

```

01020     AC.PotModDolList.message = "potentially modified dollar";
01021     AC.PotModDolList.size = sizeof(WORD);
01022     AC.ModOptDolList.message = "moduleoptiondollar";
01023     AC.ModOptDolList.size = sizeof(MODOPTDOLLAR);
01024
01025     AO.FortDotChar = '-';
01026     AO.ErrorBlock = 0;
01027     AC.firstconstindex = 1;
01028     AO.Optimize.mctsconstant.fval = 1.0;
01029     AO.Optimize.horner = O_MCTS;
01030     AO.Optimize.hornerdirection = O_FORWARDORBACKWARD;
01031     AO.Optimize.method = O_GREEDY;
01032     AO.Optimize.mctstimelimit = 0;
01033     AO.Optimize.mctsnumexpand = 1000;
01034     AO.Optimize.mctsnumkeep = 10;
01035     AO.Optimize.mctsnumrepeat = 1;
01036     AO.Optimize.greedytimelimit = 0;
01037     AO.Optimize.greedyminnum = 10;
01038     AO.Optimize.greedymaxperc = 5;
01039     AO.Optimize.printstats = 0;
01040     AO.Optimize.debugflags = 0;
01041     AO.OptimizeResult.code = NULL;
01042     AO.inscheme = 0;
01043     AO.schemenum = 0;
01044     AO.wpos = 0;
01045     AO.wpoin = 0;
01046     AO.wlen = 0;
01047     AM.dollarzero = 0;
01048     AC.doloopstack = 0;
01049     AC.doloopstacksize = 0;
01050     AC.dolooplevel = 0;
01051 /*
01052     Set up the main name trees:
01053 */
01054     AC.varnames = MakeNameTree();
01055     AC.exprnames = MakeNameTree();
01056     AC.dollarnames = MakeNameTree();
01057     AC.autonames = MakeNameTree();
01058     AC.activenames = &(AC.varnames);
01059     AP.preError = 0;
01060 /*
01061     Initialize the compiler:
01062 */
01063     inictable();
01064     AM.rbufnum = inicbufs(); /* Regular compiler buffer */
01065 #ifndef WITHPTHREADS
01066     AT.ebufnum = inicbufs(); /* Buffer for extras during execution */
01067     AT.fbufnum = inicbufs(); /* Buffer for caching in factorization */
01068     AT.allbufnum = inicbufs(); /* Buffer for id,all */
01069     AT.aebufnum = inicbufs(); /* Buffer for id,all */
01070     AN.tryterm = 0;
01071 #else
01072     AS.MasterSort = 0;
01073 #endif
01074     AM.dbufnum = inicbufs(); /* Buffer for dollar variables */
01075     AM.sbufnum = inicbufs(); /* Subterm buffer for polynomials and optimization */
01076     AC.ffbufnum = inicbufs(); /* Buffer number for user defined factorizations */
01077     AM.zbufnum = inicbufs(); /* For very special values */
01078     {
01079         CBUF *C = cbuf+AM.zbufnum;
01080         WORD one[5] = {4,1,1,3,0};
01081         WORD zero = 0;
01082         AddRHS(AM.zbufnum,1);
01083         AM.zerorhs = C->numrhs;
01084         AddNtoC(AM.zbufnum,1,&zero,17);
01085         AddRHS(AM.zbufnum,1);
01086         AM.onerhs = C->numrhs;
01087         AddNtoC(AM.zbufnum,5,one,17);
01088     }
01089     AP.inside.inscbuf = inicbufs(); /* For the #inside instruction */
01090 /*
01091     Enter the built in objects
01092 */
01093     AC.Symbols = &(AC.SymbolList);
01094     AC.Indices = &(AC.IndexList);
01095     AC.Vectors = &(AC.VectorList);
01096     AC.Functions = &(AC.FunctionList);
01097     AC.vetofilling = 0;
01098
01099     AddDollar((UBYTE *) "$", DOLUNDEFINED, 0, 0);
01100
01101     cbuf[AM.dbufnum].mnumlhs = cbuf[AM.dbufnum].numlhs;
01102     cbuf[AM.dbufnum].mnumrhs = cbuf[AM.dbufnum].numrhs;
01103
01104     AddSymbol((UBYTE *) "i_", -MAXPOWER, MAXPOWER, VARTYPEIMAGINARY, 0);
01105     AddSymbol((UBYTE *) "pi_", -MAXPOWER, MAXPOWER, VARTYPENONE, 0);
01106 /*

```



```

01107     coeff_ should have the number COEFFSYMBOL and den_ the number DENOMINATOR
01108     and the three should be in this order!
01109 */
01110     AddSymbol((UBYTE *)"coeff_", -MAXPOWER, MAXPOWER, VARTYPENONE, 0);
01111     AddSymbol((UBYTE *)"num_", -MAXPOWER, MAXPOWER, VARTYPENONE, 0);
01112     AddSymbol((UBYTE *)"den_", -MAXPOWER, MAXPOWER, VARTYPENONE, 0);
01113     AddSymbol((UBYTE *)"xarg_", -MAXPOWER, MAXPOWER, VARTYPENONE, 0);
01114     AddSymbol((UBYTE *)"dimension_", -MAXPOWER, MAXPOWER, VARTYPENONE, 0);
01115     AddSymbol((UBYTE *)"factor_", -MAXPOWER, MAXPOWER, VARTYPENONE, 0);
01116     AddSymbol((UBYTE *)"sep_", -MAXPOWER, MAXPOWER, VARTYPENONE, 0);
01117     AddSymbol((UBYTE *)"ee_", -MAXPOWER, MAXPOWER, VARTYPENONE, 0);
01118     AddSymbol((UBYTE *)"em_", -MAXPOWER, MAXPOWER, VARTYPENONE, 0);
01119     i = BUILTINSYMBOLS; /* update this in ftypes.h when we add new symbols */
01120 */
01121     Next we add a number of dummy symbols for ensuring that the user defined
01122     symbols start at a fixed given number FIRSTUSERSYMBOL
01123     We do want to give them unique names though that the user cannot access.
01124 */
01125     {
01126         char dumstr[20];
01127         for ( ; i < FIRSTUSERSYMBOL; i++ ) {
01128             snprintf(dumstr, 20, ":%d:", i);
01129             AddSymbol((UBYTE *)dumstr, -MAXPOWER, MAXPOWER, VARTYPENONE, 0);
01130         }
01131     }
01132
01133     AddIndex((UBYTE *)"iarg_", 4, 0);
01134     AddVector((UBYTE *)"parg_", VARTYPENONE, 0);
01135
01136     AM.NumFixedFunctions = sizeof(fixedfunctions)/sizeof(struct fixedfun);
01137     for ( i = 0; i < AM.NumFixedFunctions; i++ ) {
01138         ii = AddFunction((UBYTE *)fixedfunctions[i].name
01139             , fixedfunctions[i].commu
01140             , fixedfunctions[i].tensor
01141             , fixedfunctions[i].complx
01142             , fixedfunctions[i].symmetric
01143             , 0, -1, -1);
01144         if ( fixedfunctions[i].tensor == GAMMAFUNCTION )
01145             functions[ii].flags |= COULDCOMMUTE;
01146     }
01147 */
01148     Next we add a number of dummy functions for ensuring that the user defined
01149     functions start at a fixed given number FIRSTUSERFUNCTION.
01150     We do want to give them unique names though that the user cannot access.
01151 */
01152     {
01153         char dumstr[20];
01154         for ( ; i < FIRSTUSERFUNCTION-FUNCTION; i++ ) {
01155             snprintf(dumstr, 20, ":%d::%d:", i);
01156             AddFunction((UBYTE *)dumstr, 0, 0, 0, 0, 0, -1, -1);
01157         }
01158     }
01159     AM.NumFixedSets = sizeof(fixedsets)/sizeof(struct fixedset);
01160     for ( i = 0; i < AM.NumFixedSets; i++ ) {
01161         ii = AddSet((UBYTE *)fixedsets[i].name, fixedsets[i].dimension);
01162         Sets[ii].type = fixedsets[i].type;
01163     }
01164     AM.RepMax = MAXREPEAT;
01165 #ifndef WITHPTHREADS
01166     AT.RepCount = (int *)Malloc1((LONG)((AM.RepMax+3)*sizeof(int)), "repeat buffers");
01167     AN.RepPoint = AT.RepCount;
01168     AT.RepTop = AT.RepCount + AM.RepMax;
01169     AN.polysortflag = 0;
01170     AN.subsubveto = 0;
01171 #endif
01172     AC.NumWildcardNames = 0;
01173     AC.WildcardBufferSize = 50;
01174     AC.WildcardNames = (UBYTE *)Malloc1((LONG)AC.WildcardBufferSize, "argument list names");
01175 #ifndef WITHPTHREADS
01176     AT.WildArgTaken = (WORD *)Malloc1((LONG)AC.WildcardBufferSize*sizeof(WORD)/2
01177         , "argument list names");
01178     AT.WildcardBufferSize = AC.WildcardBufferSize;
01179     AR.CompareRoutine = (COMPAREDUMMY)(&Compare1);
01180     AT.nfac = AT.nBer = 0;
01181     AT.factorials = 0;
01182     AT.bernoullis = 0;
01183     AR.wranfia = 0;
01184     AR.wranfcall = 0;
01185     AR.wranfnpair1 = NPAIR1;
01186     AR.wranfnpair2 = NPAIR2;
01187     AR.wranfseed = 0;
01188 #endif
01189     AM.atstartup = 1;
01190     AM.oldnumextrasyms = strDup1((UBYTE *)"OLDNUMEXTRASYNBOLS_", "oldnumextrasyms");
01191     PutPreVar((UBYTE *)"VERSION_", (UBYTE *)STRINGIFY(MAJORVERSION), 0, 0);
01192     PutPreVar((UBYTE *)"SUBVERSION_", (UBYTE *)STRINGIFY(MINORVERSION), 0, 0);
01193     PutPreVar((UBYTE *)"DATE_", (UBYTE *)MakeDate(), 0, 0);

```

```

01194     PutPreVar((UBYTE *)"random_", (UBYTE *)"_____", 0,0);
01195     PutPreVar((UBYTE *)"optiminvar_", (UBYTE *)("0"), 0,0);
01196     PutPreVar((UBYTE *)"optimmaxvar_", (UBYTE *)("0"), 0,0);
01197     PutPreVar(AM.oldnumextrasympols, (UBYTE *)("0"), 0,0);
01198     PutPreVar((UBYTE *)"optimvalue_", (UBYTE *)("0"), 0,0);
01199     PutPreVar((UBYTE *)"optimscheme_", (UBYTE *)("0"), 0,0);
01200     PutPreVar((UBYTE *)"tolower_", (UBYTE *)("0"), 0,0);
01201     PutPreVar((UBYTE *)"toupper_", (UBYTE *)("0"), 0,0);
01202     PutPreVar((UBYTE *)"takeleft_", (UBYTE *)("0"), 0,0);
01203     PutPreVar((UBYTE *)"takeright_", (UBYTE *)("0"), 0,0);
01204     PutPreVar((UBYTE *)"keepleft_", (UBYTE *)("0"), 0,0);
01205     PutPreVar((UBYTE *)"keepright_", (UBYTE *)("0"), 0,0);
01206     PutPreVar((UBYTE *)"SYSTEMERROR_", (UBYTE *)("0"), 0,0);
01207 /*
01208     Next are the flags to control diagram generation filters
01209 */
01210     #define STR2(x) #x
01211     #define STR(x) STR2(x)
01212     PutPreVar((UBYTE *)"TOPOLOGIESONLY_" , (UBYTE*)(STR(TOPOLOGIESONLY)) , 0,0);
01213     PutPreVar((UBYTE *)"WITHOUTNODES_" , (UBYTE*)(STR(WITHOUTNODES)) , 0,0);
01214     PutPreVar((UBYTE *)"WITHEDGES_" , (UBYTE*)(STR(WITHEDGES)) , 0,0);
01215     PutPreVar((UBYTE *)"WITHBLOCKS_" , (UBYTE*)(STR(WITHBLOCKS)) , 0,0);
01216     PutPreVar((UBYTE *)"WITHONEPISETS_" , (UBYTE*)(STR(WITHONEPISETS)) , 0,0);
01217     PutPreVar((UBYTE *)"WITHSYMMETRIZEI_" , (UBYTE*)(STR(WITHSYMMETRIZEI)) , 0,0);
01218     PutPreVar((UBYTE *)"WITHSYMMETRIZEF_" , (UBYTE*)(STR(WITHSYMMETRIZEF)) , 0,0);
01219     // This is not an "option" preprocessor var but is set by "external" particle definitions
01220     // PutPreVar((UBYTE *)"CHECKEXTERN_" , (UBYTE*)(STR(CHECKEXTERN)) , 0,0);
01221     PutPreVar((UBYTE *)"ONEPI_" , (UBYTE*)(STR(ONEPARTI)) , 0,0);
01222     PutPreVar((UBYTE *)"ONEPR_" , (UBYTE*)(STR(ONEPARTR)) , 0,0);
01223     PutPreVar((UBYTE *)"ONSHELL_" , (UBYTE*)(STR(ONSHELL)) , 0,0);
01224     PutPreVar((UBYTE *)"OFFSHELL_" , (UBYTE*)(STR(OFFSHELL)) , 0,0);
01225     PutPreVar((UBYTE *)"NOSIGMA_" , (UBYTE*)(STR(NOSIGMA)) , 0,0);
01226     PutPreVar((UBYTE *)"SIGMA_" , (UBYTE*)(STR(SIGMA)) , 0,0);
01227     PutPreVar((UBYTE *)"NOSNAIL_" , (UBYTE*)(STR(NOSNAIL)) , 0,0);
01228     PutPreVar((UBYTE *)"SNAIL_" , (UBYTE*)(STR(SNAIL)) , 0,0);
01229     PutPreVar((UBYTE *)"NOTADPOLE_" , (UBYTE*)(STR(NOTADPOLE)) , 0,0);
01230     PutPreVar((UBYTE *)"TADPOLE_" , (UBYTE*)(STR(TADPOLE)) , 0,0);
01231     PutPreVar((UBYTE *)"SIMPLE_" , (UBYTE*)(STR(SIMPLE)) , 0,0);
01232     PutPreVar((UBYTE *)"NOTSIMPLE_" , (UBYTE*)(STR(NOTSIMPLE)) , 0,0);
01233     PutPreVar((UBYTE *)"BIPART_" , (UBYTE*)(STR(BIPART)) , 0,0);
01234     PutPreVar((UBYTE *)"NONBIPART_" , (UBYTE*)(STR(NONBIPART)) , 0,0);
01235     PutPreVar((UBYTE *)"CYCLI_" , (UBYTE*)(STR(CYCLI)) , 0,0);
01236     PutPreVar((UBYTE *)"CYCLR_" , (UBYTE*)(STR(CYCLR)) , 0,0);
01237     PutPreVar((UBYTE *)"FLOOP_" , (UBYTE*)(STR(FLOOP)) , 0,0);
01238     PutPreVar((UBYTE *)"NOTFLOOP_" , (UBYTE*)(STR(NOTFLOOP)) , 0,0);
01239
01240     {
01241         char buf[41]; /* up to 128-bit */
01242         LONG pid;
01243         #ifndef WITHMPI
01244             pid = GetPID();
01245         #else
01246             pid = ( PF.me == MASTER ) ? GetPID() : (LONG)0;
01247             pid = PF_BroadcastNumber(pid);
01248         #endif
01249         LongCopy(pid,buf);
01250         PutPreVar((UBYTE *)"PID_" , (UBYTE *)buf, 0,0);
01251     }
01252     AM.atstartup = 0;
01253     AP.MaxPreTypes = 10;
01254     AP.NumPreTypes = 0;
01255     AP.PreTypes = (int *)Malloc1(sizeof(int)*(AP.MaxPreTypes+1),"preprocessor types");
01256     AP.inside.buffer = 0;
01257     AP.inside.size = 0;
01258
01259     AC.SortType = AC.lSortType = AM.gSortType = SORTLOWFIRST;
01260     #ifndef WITHPTHREADS
01261     #else
01262         AR.SortType = AC.SortType;
01263     #endif
01264     AC.LogHandle = -1;
01265     AC.SetList.numtemp = AC.SetList.num;
01266     AC.SetElementList.numtemp = AC.SetElementList.num;
01267
01268     GetName(AC.varnames, (UBYTE *)"exp_" , &AM.expnum, NOAUTO);
01269     GetName(AC.varnames, (UBYTE *)"denom_" , &AM.denomnum, NOAUTO);
01270     GetName(AC.varnames, (UBYTE *)"fac_" , &AM.facnum, NOAUTO);
01271     GetName(AC.varnames, (UBYTE *)"invfac_" , &AM.invfacnum, NOAUTO);
01272     GetName(AC.varnames, (UBYTE *)"sum_" , &AM.sumnum, NOAUTO);
01273     GetName(AC.varnames, (UBYTE *)"sump_" , &AM.sumpnum, NOAUTO);
01274     GetName(AC.varnames, (UBYTE *)"term_" , &AM.termfunnum, NOAUTO);
01275     GetName(AC.varnames, (UBYTE *)"match_" , &AM.matchfunnum, NOAUTO);
01276     GetName(AC.varnames, (UBYTE *)"count_" , &AM.countfunnum, NOAUTO);
01277     AM.termfunnum += FUNCTION;
01278     AM.matchfunnum += FUNCTION;
01279     AM.countfunnum += FUNCTION;
01280

```

```

01281     AC.ThreadStats = AM.gThreadStats = AM.ggThreadStats = 1;
01282     AC.FinalStats = AM.gFinalStats = AM.ggFinalStats = 1;
01283     AC.StatsFlag = AM.gStatsFlag = AM.ggStatsFlag = 1;
01284     AC.ThreadsFlag = AM.gThreadsFlag = AM.ggThreadsFlag = 1;
01285     AC.ThreadBalancing = AM.gThreadBalancing = AM.ggThreadBalancing = 1;
01286     AC.ThreadSortFileSynch = AM.gThreadSortFileSynch = AM.ggThreadSortFileSynch = 0;
01287     AC.ProcessStats = AM.gProcessStats = AM.ggProcessStats = 1;
01288     AC.OldParallelStats = AM.gOldParallelStats = AM.ggOldParallelStats = 0;
01289     AC.OldFactArgFlag = AM.gOldFactArgFlag = AM.ggOldFactArgFlag = NEWFACTARG;
01290     AC.OldGCDflag = AM.gOldGCDflag = AM.ggOldGCDflag = 1;
01291     AC.WTimeStatsFlag = AM.gWTimeStatsFlag = AM.ggWTimeStatsFlag = 0;
01292     AC.gNumDollars = AP.DollarList.num;
01293     AC.SizeCommuteInSet = AM.gSizeCommuteInSet = 0;
01294     #ifdef WITHFLOAT
01295     AC.MaxWeight = AM.gMaxWeight = AM.ggMaxWeight = MAXWEIGHT;
01296     AC.DefaultPrecision = AM.gDefaultPrecision = AM.ggDefaultPrecision = DEFAULTPRECISION;
01297     #endif
01298     AC.CommuteInSet = 0;
01299
01300     AM.PrintTotalSize = 0;
01301
01302     AO.NoSpacesInNumbers = AM.gNoSpacesInNumbers = AM.ggNoSpacesInNumbers = 0;
01303     AO.IndentSpace = AM.gIndentSpace = AM.ggIndentSpace = INDENTSPACE;
01304     AO.BlockSpaces = 0;
01305     AO.OptimizationLevel = 0;
01306     PUTZERO(AS.MaxExprSize);
01307     PUTZERO(AC.StoreFileSize);
01308
01309     #ifdef WITHPTHREADS
01310     AC.inputnumbers = 0;
01311     AC.pfirstnum = 0;
01312     AC.numppfirstnum = AC.sizeppfirstnum = 0;
01313     #endif
01314     AC.MemDebugFlag = 1;
01315
01316     #ifdef WITHEXTERNALCHANNEL
01317     AX.currentExternalChannel=0;
01318     AX.killSignal=SIGKILL;
01319     AX.killWholeGroup=1;
01320     AX.daemonize=1;
01321     AX.currentPrompt=0;
01322     AX.timeout=1000; /*One second to initialize preset channels*/
01323     AX.shellname=strDup1((UBYTE *)"/bin/sh -c","external channel shellname");
01324     AX.stdername=strDup1((UBYTE *)"/dev/null","external channel stderrname");
01325     #endif
01326 }
01327
01328 /*
01329     #] StartVariables :
01330     #[ StartMore :
01331 */
01332
01333 void StartMore(void)
01334 {
01335     #ifdef WITHEXTERNALCHANNEL
01336     /*If env.variable "FORM_PIPES" is defined, we have to initialize
01337     corresponding pre-set external channels, see file extcmd.c.*/
01338     /*This line must be after all setup settings: in future, timeout
01339     could be changed at setup.*/
01340     if(AX.timeout>=0)/*if AX.timeout<0, this was done by cmdline option -pipe*/
01341         initPresetExternalChannels((UBYTE*)getenv("FORM_PIPES"),AX.timeout);
01342     #endif
01343
01344     #ifdef WITHMPI
01345     /*
01346     Define preprocessor variable PARALLELTASK_ as a process number, 0 is the master
01347     Define preprocessor variable NPARALLELTASKS_ as a total number of processes
01348     */
01349     {
01350         UBYTE buf[32];
01351         snprintf((char*)buf,32,"%d",PF.me);
01352         PutPreVar((UBYTE *) "PARALLELTASK_",buf,0,0);
01353         snprintf((char*)buf,32,"%d",PF.numtasks);
01354         PutPreVar((UBYTE *) "NPARALLELTASKS_",buf,0,0);
01355     }
01356     #else
01357     PutPreVar((UBYTE *) "PARALLELTASK_",(UBYTE *) "0",0,0);
01358     PutPreVar((UBYTE *) "NPARALLELTASKS_",(UBYTE *) "1",0,0);
01359     #endif
01360
01361     PutPreVar((UBYTE *) "NAME_",AM.InputFileName ? AM.InputFileName : (UBYTE *) "STDIN",0,0);
01362 }
01363
01364 /*
01365     #] StartMore :
01366     #[ IniVars :
01367

```

```

01368         This routine initializes the parameters that may change during the run.
01369     */
01370
01371 void IniVars(void)
01372 {
01373     #ifdef WITHPTHREADS
01374         GETIDENTITY
01375     #else
01376         WORD *t;
01377     #endif
01378     WORD *fi, i, one = 1;
01379     CBUF *C = cbuf+AC.cbunum;
01380
01381     #ifdef WITHPTHREADS
01382         UBYTE buf[32];
01383         snprintf((char*)buf, 32, "%d", AM.totalnumberofthreads);
01384         PutPreVar((UBYTE *) "NTHREADS_", buf, 0, 1);
01385     #else
01386         PutPreVar((UBYTE *) "NTHREADS_", (UBYTE *) "1", 0, 1);
01387     #endif
01388
01389     AC.ShortStats = 0;
01390     AC.WarnFlag = 1;
01391     AR.SortType = AC.SortType = AC.lSortType = AM.gSortType;
01392     AC.OutputMode = 72;
01393     AC.OutputSpaces = NORMALFORMAT;
01394     AR.Eside = 0;
01395     AC.DumNum = 0;
01396     AC.ncmod = AM.gncmod = 0;
01397     AC.modmode = AM.gmodmode = 0;
01398     AC.npowmod = AM.gnpowmod = 0;
01399     AC.halfmod = 0; AC.nhalfmod = 0;
01400     AC.modinverses = 0;
01401     AC.lPolyFun = AM.gPolyFun = 0;
01402     AC.lPolyFunInv = AM.gPolyFunInv = 0;
01403     AC.lPolyFunType = AM.gPolyFunType = 0;
01404     AC.lPolyFunExp = AM.gPolyFunExp = 0;
01405     AC.lPolyFunVar = AM.gPolyFunVar = 0;
01406     AC.lPolyFunPow = AM.gPolyFunPow = 0;
01407     #ifdef WITHFLINT
01408         AC.FlintPolyFlag = 1;
01409     #else
01410         AC.FlintPolyFlag = 0;
01411     #endif
01412     AC.DirtPow = 0;
01413     AC.lDefDim = AM.gDefDim = 4;
01414     AC.lDefDim4 = AM.gDefDim4 = 0;
01415     AC.lUnitTrace = AM.gUnitTrace = 4;
01416     AC.NamesFlag = AM.gNamesFlag = 0;
01417     AC.CodesFlag = AM.gCodesFlag = 0;
01418     /* Printing a backtrace on crash is on by default for both normal and debug
01419        modes if FORM has been compiled with backtrace support. */
01420     #ifdef ENABLE_BACKTRACE
01421         AC.PrintBacktraceFlag = 1;
01422     #else
01423         AC.PrintBacktraceFlag = 0;
01424     #endif
01425     /* Human-readable statistics are off by default */
01426     AC.HumanStatsFlag = 0;
01427     AC.extrasymbols = AM.gextrasymbols = AM.ggextrasymbols = 0;
01428     AC.extrasym = (UBYTE *) Malloc1(2*sizeof(UBYTE), "extrasym");
01429     AM.gextrasym = (UBYTE *) Malloc1(2*sizeof(UBYTE), "extrasym");
01430     AM.ggextrasym = (UBYTE *) Malloc1(2*sizeof(UBYTE), "extrasym");
01431     AC.extrasym[0] = AM.gextrasym[0] = AM.ggextrasym[0] = 'Z';
01432     AC.extrasym[1] = AM.gextrasym[1] = AM.ggextrasym[1] = 0;
01433     AC.TokensWriteFlag = AM.gTokensWriteFlag = 0;
01434     AC.SetupFlag = 0;
01435     AC.LineLength = AM.gLineLength = 79;
01436     AC.NwildC = 0;
01437     AC.OutputMode = 0;
01438     AM.gOutputMode = 0;
01439     AC.OutputSpaces = NORMALFORMAT;
01440     AM.gOutputSpaces = NORMALFORMAT;
01441     AC.OutNumberType = RATIONALMODE;
01442     AM.gOutNumberType = RATIONALMODE;
01443     #ifdef WITHZLIB
01444         AR.zipCompress = GZIPDEFAULT;
01445         AR.FoStage4[0].ziobuffer = 0;
01446         AR.FoStage4[1].ziobuffer = 0;
01447     #ifdef WITHZSTD
01448         /* Zstd compression is on by default, if we have compiled with it */
01449         ZWRAP_useZSTDcompression(1);
01450     #endif
01451     #endif
01452     AR.BracketOn = 0;
01453     AC.bracketindexflag = 0;
01454     AT.bracketindexflag = 0;

```

```

01455     AT.bracketinfo = 0;
01456     AO.IsBracket = 0;
01457     AM.gfunpowers = AC.funpowers = COMFUNPOWERS;
01458     AC.parallelfldflag = AM.gparallelfldflag;
01459     AC.properorderflag = AM.gproperorderflag = PROPERORDERFLAG;
01460     AC.ProcessBucketSize = AC.mProcessBucketSize = AM.gProcessBucketSize;
01461     AC.ThreadBucketSize = AM.gThreadBucketSize;
01462     AC.ShortStatsMax = 0;
01463     AM.gShortStatsMax = 0;
01464     AM.ggShortStatsMax = 0;
01465
01466     GlobalSymbols      = NumSymbols;
01467     GlobalIndices      = NumIndices;
01468     GlobalVectors      = NumVectors;
01469     GlobalFunctions    = NumFunctions;
01470     GlobalSets         = NumSets;
01471     GlobalSetElements  = NumSetElements;
01472     AC.modpowers = (UWORD *)0;
01473
01474     i = AM.OffsetIndex;
01475     fi = AC.FixIndices;
01476     if ( i > 0 ) do { *fi++ = one; } while ( --i >= 0 );
01477     AR.sLevel = -1;
01478     AM.Ordering[0] = 5;
01479     AM.Ordering[1] = 6;
01480     AM.Ordering[2] = 7;
01481     AM.Ordering[3] = 0;
01482     AM.Ordering[4] = 1;
01483     AM.Ordering[5] = 2;
01484     AM.Ordering[6] = 3;
01485     AM.Ordering[7] = 4;
01486     for ( i = 8; i < 15; i++ ) AM.Ordering[i] = i;
01487     AM.gUniTrace[0] =
01488     AC.lUniTrace[0] = SNUMBER;
01489     AM.gUniTrace[1] =
01490     AC.lUniTrace[1] =
01491     AM.gUniTrace[2] =
01492     AC.lUniTrace[2] = 4;
01493     AM.gUniTrace[3] =
01494     AC.lUniTrace[3] = 1;
01495 #ifdef WITHPTHREADS
01496     AS.Balancing = 0;
01497 #else
01498     AT.MinVecArg[0] = 7+ARGHEAD;
01499     AT.MinVecArg[ARGHEAD] = 7;
01500     AT.MinVecArg[1+ARGHEAD] = INDEX;
01501     AT.MinVecArg[2+ARGHEAD] = 3;
01502     AT.MinVecArg[3+ARGHEAD] = 0;
01503     AT.MinVecArg[4+ARGHEAD] = 1;
01504     AT.MinVecArg[5+ARGHEAD] = 1;
01505     AT.MinVecArg[6+ARGHEAD] = -3;
01506     t = AT.FunArg;
01507     *t++ = 4+ARGHEAD+FUNHEAD;
01508     for ( i = 1; i < ARGHEAD; i++ ) *t++ = 0;
01509     *t++ = 4+FUNHEAD;
01510     *t++ = 0;
01511     *t++ = FUNHEAD;
01512     for ( i = 2; i < FUNHEAD; i++ ) *t++ = 0;
01513     *t++ = 1; *t++ = 1; *t++ = 3;
01514
01515 #ifdef WITHMPI
01516     AS.printflag = 0;
01517 #endif
01518
01519     AT.comsym[0] = 8;
01520     AT.comsym[1] = SYMBOL;
01521     AT.comsym[2] = 4;
01522     AT.comsym[3] = 0;
01523     AT.comsym[4] = 1;
01524     AT.comsym[5] = 1;
01525     AT.comsym[6] = 1;
01526     AT.comsym[7] = 3;
01527     AT.comnum[0] = 4;
01528     AT.comnum[1] = 1;
01529     AT.comnum[2] = 1;
01530     AT.comnum[3] = 3;
01531     AT.comfun[0] = FUNHEAD+4;
01532     AT.comfun[1] = FUNCTION;
01533     AT.comfun[2] = FUNHEAD;
01534     AT.comfun[3] = 0;
01535 #if FUNHEAD == 4
01536     AT.comfun[4] = 0;
01537 #endif
01538     AT.comfun[FUNHEAD+1] = 1;
01539     AT.comfun[FUNHEAD+2] = 1;
01540     AT.comfun[FUNHEAD+3] = 3;
01541     AT.comind[0] = 7;

```

```

01542     AT.comind[1] = INDEX;
01543     AT.comind[2] = 3;
01544     AT.comind[3] = 0;
01545     AT.comind[4] = 1;
01546     AT.comind[5] = 1;
01547     AT.comind[6] = 3;
01548     AT.locwildvalue[0] = SUBEXPRESSION;
01549     AT.locwildvalue[1] = SUBEXPSIZE;
01550     for ( i = 2; i < SUBEXPSIZE; i++ ) AT.locwildvalue[i] = 0;
01551     AT.mulpat[0] = TYPEMULT;
01552     AT.mulpat[1] = SUBEXPSIZE+3;
01553     AT.mulpat[2] = 0;
01554     AT.mulpat[3] = SUBEXPRESSION;
01555     AT.mulpat[4] = SUBEXPSIZE;
01556     AT.mulpat[5] = 0;
01557     AT.mulpat[6] = 1;
01558     for ( i = 7; i < SUBEXPSIZE+5; i++ ) AT.mulpat[i] = 0;
01559     AT.proexp[0] = SUBEXPSIZE+4;
01560     AT.proexp[1] = EXPRESSION;
01561     AT.proexp[2] = SUBEXPSIZE;
01562     AT.proexp[3] = -1;
01563     AT.proexp[4] = 1;
01564     for ( i = 5; i < SUBEXPSIZE+1; i++ ) AT.proexp[i] = 0;
01565     AT.proexp[SUBEXPSIZE+1] = 1;
01566     AT.proexp[SUBEXPSIZE+2] = 1;
01567     AT.proexp[SUBEXPSIZE+3] = 3;
01568     AT.proexp[SUBEXPSIZE+4] = 0;
01569     AT.dummysubexp[0] = SUBEXPRESSION;
01570     AT.dummysubexp[1] = SUBEXPSIZE+4;
01571     for ( i = 2; i < SUBEXPSIZE; i++ ) AT.dummysubexp[i] = 0;
01572     AT.dummysubexp[SUBEXPSIZE] = WILDDUMMY;
01573     AT.dummysubexp[SUBEXPSIZE+1] = 4;
01574     AT.dummysubexp[SUBEXPSIZE+2] = 0;
01575     AT.dummysubexp[SUBEXPSIZE+3] = 0;
01576
01577     AT.inprimelist = -1;
01578     AT.sizeprimelist = 0;
01579     AT.primelist = 0;
01580     AT.LeaveNegative = 0;
01581     AT.TrimPower = 0;
01582     AN.SplitScratch = 0;
01583     AN.SplitScratchSize = AN.InScratch = 0;
01584     AN.SplitScratch1 = 0;
01585     AN.SplitScratchSize1 = AN.InScratch1 = 0;
01586     AN.idfunctionflag = 0;
01587 #endif
01588     AO.OutputLine = AO.OutFill = BufferForOutput;
01589     AO.FactorMode = 0;
01590     C->Pointer = C->Buffer;
01591
01592     AP.PreOut = 0;
01593     AP.ComChar = AP.cComChar;
01594     AC.cbufnum = AM.rbufnum; /* Select the default compiler buffer */
01595     AC.HideLevel = 0;
01596     AP.PreAssignFlag = 0;
01597 }
01598
01599 /*
01600     #] IniVars :
01601     #[ Signal handlers :
01602 */
01603 /*[28apr2004 mt]:*/
01604 #ifdef TRAP_SIGNALS
01605
01606 static int exitInProgress = 0;
01607 static int trappedTerminate = 0;
01608
01609 /*INTSIGHANDLER : some systems require a signal handler to return an integer,
01610 so define the macro INTSIGHANDLER if compiler fails:*/
01611 #ifndef INTSIGHANDLER
01612 static int onErrSig(int i)
01613 #else
01614 static void onErrSig(int i)
01615 #endif
01616 {
01617     if (exitInProgress){
01618         signal(i, SIG_DFL); /* Use default behaviour*/
01619         raise (i); /*reproduce trapped signal*/
01620 #ifdef INTSIGHANDLER
01621         return(i);
01622 #else
01623         return;
01624 #endif
01625     }
01626     trappedTerminate = 1;
01627     /*[13jul2005 mt]:/*TerminateImpl(-1) on signal is here:*/
01628     Terminate(-1);

```

```

01629 }
01630
01631 #ifdef INTSIGHANDLER
01632 static void setNewSig(int i, int (*handler)(int))
01633 #else
01634 static void setNewSig(int i, void (*handler)(int))
01635 #endif
01636 {
01637     if(! (i<NSIG) )/* Invalid signal -- see comments in the file */
01638         return;
01639     if ( signal(i,SIG_IGN) !=SIG_IGN)
01640         /* if compiler fails here, try to define INTSIGHANDLER:*/
01641         signal(i,handler);
01642 }
01643
01644 void setSignalHandlers(void)
01645 {
01646     /* Reset various unrecoverable error signals:*/
01647     setNewSig(SIGSEGV,onErrSig);
01648     setNewSig(SIGFPE,onErrSig);
01649     setNewSig(SIGILL,onErrSig);
01650     setNewSig(SIGEMT,onErrSig);
01651     setNewSig(SIGSYS,onErrSig);
01652     setNewSig(SIGPIPE,onErrSig);
01653     setNewSig(SIGLOST,onErrSig);
01654     setNewSig(SIGXCPU,onErrSig);
01655     setNewSig(SIGXFSZ,onErrSig);
01656
01657     /* Reset interrupt signals:*/
01658     setNewSig(SIGTERM,onErrSig);
01659     setNewSig(SIGINT,onErrSig);
01660     setNewSig(SIGQUIT,onErrSig);
01661     setNewSig(SIGHUP,onErrSig);
01662     setNewSig(SIGALRM,onErrSig);
01663     setNewSig(SIGVTALRM,onErrSig);
01664     /* setNewSig(SIGPROF,onErrSig); */ /* Why did Tentukov forbid profilers?? */
01665 }
01666
01667 #endif
01668 /*:[28apr2004 mt]*/
01669 /*
01670     #] Signal handlers :
01671     #[ main :
01672 */
01673
01674 #ifdef WITHPTHREADS
01675 ALLPRIVATES *ABdummy[10];
01676 #endif
01677
01678 int main(int argc, char **argv)
01679 {
01680     int retval;
01681     bzero((void *)(&A),sizeof(A)); /* make sure A is initialized at zero */
01682     iniTools();
01683 #ifdef TRAP_SIGNALS
01684     setSignalHandlers();
01685 #endif
01686 #ifdef WINDOWS
01687     _setmode(_fileno(stdout),O_BINARY);
01688 #endif
01689
01690 #ifdef WITHPTHREADS
01691     AB = ABdummy;
01692     StartHandleLock();
01693     BeginIdentities();
01694 #else
01695     AM.SumTime = TimeCPU(0);
01696     TimeWallClock(0);
01697 #endif
01698
01699 #ifdef WITHMPI
01700     if ( PF_Init(&argc,&argv) ) exit(-1);
01701 #endif
01702
01703     StartFiles();
01704     StartVariables();
01705 #ifdef WITHMPI
01706     /*
01707     * Here MesPrint() is ready. We turn on AS.printflag to print possible
01708     * errors occurring on slaves in the initialization. With AS.printflag = -1
01709     * MesPrint() does not use the synchronized output. This may lead broken
01710     * texts in the output somewhat, but it is safer to implement in this way
01711     * for the situation in which some of MesPrint() calls use MLOCK()-MUNLOCK()
01712     * and some do not. In future if we set AS.printflag = 1 and modify the
01713     * source code such that all MesPrint() calls are sandwiched by MLOCK()-
01714     * MUNLOCK(), we need also to modify the code for the master to catch
01715     * messages corresponding to MUNLOCK() calls at some point.

```

```

01716      *
01717      * AS.printflag will be set to 0 in IniVars() to prevent slaves from
01718      * printing redundant errors in the preprocessor and compiler (e.g., syntax
01719      * errors).
01720      */
01721      AS.printflag = -1;
01722 #endif
01723
01724      if ( ( retval = DoTail(argc, (UBYTE **)argv) ) != 0 ) {
01725          if ( retval > 0 ) Terminate(0);
01726          else Terminate(-1);
01727      }
01728      if ( DoSetups() ) Terminate(-2);
01729 #ifdef WITHMPI
01730      /* It is messy if all errors in OpenInput() on slaves are printed. */
01731      AS.printflag = 0;
01732 #endif
01733      if ( OpenInput() ) Terminate(-3);
01734 #ifdef WITHMPI
01735      AS.printflag = -1;
01736 #endif
01737      if ( TryEnvironment() ) Terminate(-2);
01738      if ( TryFileSetups() ) Terminate(-2);
01739      if ( MakeSetupAllocs() ) Terminate(-2);
01740      StartMore();
01741      InitRecovery();
01742      CheckRecoveryFile();
01743      if ( AM.totalnumberofthreads == 0 ) AM.totalnumberofthreads = 1;
01744      AS.MultiThreaded = 0;
01745 #ifdef WITHPTHREADS
01746      if ( AM.totalnumberofthreads > 1 ) AS.MultiThreaded = 1;
01747      ReserveTempFiles(1);
01748      StartAllThreads(AM.totalnumberofthreads);
01749      IniFbufs();
01750 #else
01751      ReserveTempFiles(0);
01752      IniFbuffer(AT.fbufnum);
01753 #endif
01754      if ( !AM.FromStdin ) PrintHeader(1);
01755      IniVars();
01756      Globalize(1);
01757 #ifdef WITH_ALARM
01758      if ( AM.TimeLimit > 0 ) alarm(AM.TimeLimit);
01759 #endif
01760 #ifdef WITHFLINT
01761      flint_check_version();
01762 #endif
01763      TimeCPU(0);
01764      TimeChildren(0);
01765      TimeWallClock(0);
01766      PreProcessor();
01767      Terminate(0);
01768      return(0);
01769 }
01770 /*
01771      #] main :
01772      #[ CleanUp :
01773
01774      if par < 0 we have to keep the storage file.
01775      when par > 0 we ran into a .clear statement.
01776      In that case we keep the zero level input and the log file.
01777 */
01778 void CleanUp(WORD par)
01779 {
01780     GETIDENTITY
01781     int i;
01782
01783     if ( FG.fname ) {
01784 #ifdef WITHPTHREADS
01785         if ( B ) {
01786 #endif
01787             CleanUpSort(0);
01788             for ( i = 0; i < 3; i++ ) {
01789                 if ( AR.Fscr[i].handle >= 0 ) {
01790                     if ( AR.Fscr[i].name ) {
01791 /*
01792                         If there are more threads referring to the same file
01793                         only the one with the name is the owner of the file.
01794 */
01795                         CloseFile(AR.Fscr[i].handle);
01796                         remove(AR.Fscr[i].name);
01797                     }
01798                     AR.Fscr[i].handle = - 1;
01799                     AR.Fscr[i].POfill = 0;
01800                 }
01801             }
01802

```



```

01803     }
01804 #ifdef WITHPTHREADS
01805     }
01806 #endif
01807     if ( par > 0 ) {
01808 /*
01809     Close all input levels above the lowest?
01810 */
01811     }
01812     if ( AC.StoreHandle >= 0 && par <= 0 ) {
01813 #ifdef TRAP_SIGNALS
01814         if ( trappedTerminate ) { /* We don't throw .str if it has contents */
01815             POSITION pos;
01816             PUTZERO(pos);
01817             SeekFile(AC.StoreHandle,&pos,SEEK_END);
01818             if ( ISNOTZEROPOS(pos) ) {
01819                 CloseFile(AC.StoreHandle);
01820                 goto dontremove;
01821             }
01822         }
01823         CloseFile(AC.StoreHandle);
01824 #ifdef WITHPTHREADS
01825         if ( B )
01826 #endif
01827         if ( par >= 0 || AR.StoreData.Handle < 0 || AM.ClearStore ) {
01828             remove(FG.fname);
01829         }
01830 dontremove:;
01831 #else
01832         CloseFile(AC.StoreHandle);
01833 #ifdef WITHPTHREADS
01834         if ( B )
01835 #endif
01836         if ( par >= 0 || AR.StoreData.Handle < 0 || AM.ClearStore > 0 ) {
01837             remove(FG.fname);
01838         }
01839 #endif
01840     }
01841 }
01842 ClearSpectators(CLEARMODULE);
01843 /*
01844 Remove recovery file on exit if everything went well
01845 */
01846 if ( par == 0 ) {
01847     DeleteRecoveryFile();
01848 }
01849 /*
01850 Now the final message concerning the total time
01851 */
01852 if ( AC.LogHandle >= 0 && par <= 0 ) {
01853     WORD lh = AC.LogHandle;
01854     AC.LogHandle = -1;
01855 #ifdef WITHMPI
01856     if ( PF.me == MASTER ) /* Only the master opened the real file. */
01857 #endif
01858     CloseFile(lh);
01859 }
01860 }
01861 /*
01862 #] CleanUp :
01863 #[ TerminateImpl :
01864 */
01865 static int firstterminate = 1;
01866 void TerminateImpl(int errorcode, const char* file, int line, const char* function)
01867 {
01868     if ( errorcode && firstterminate ) {
01869         firstterminate = 0;
01870         MLOCK(ErrorMessageLock);
01871 #ifdef WITHPTHREADS
01872         MesPrint("Program terminating in thread %w at %");
01873 #elif defined(WITHMPI)
01874         MesPrint("Program terminating in process %w at %");
01875 #else
01876         MesPrint("Program terminating at %");
01877 #endif
01878         MesPrint("Terminate called from %s:%d (%s)", file, line, function);
01879         if ( AC.PrintBacktraceFlag ) {
01880             #ifdef ENABLE_BACKTRACE
01881                 void *stack[64];
01882                 int stacksize, stop = 0;
01883                 stacksize = backtrace(stack, sizeof(stack)/sizeof(stack[0]));
01884             }
01885         }
01886     }

```

```

01890         /* First check whether eu-addr2line is available */
01891         if ( !system("command -v eu-addr2line > /dev/null 2>&1") ) {
01892             MesPrint("Backtrace:");
01893             for (int i = 0; i < stacksize && !stop; i++) {
01894                 FILE *fp;
01895                 char cmd[512];
01896                 // Leave an initial space
01897                 cmd[0] = ' ';
01898                 MesPrint("%#2d:", i);
01899                 snprintf(cmd+1, sizeof(cmd)-1, "eu-addr2line -s --pretty-print -f -i '%p'
--pid=%d\n", stack[i], getpid());
01900                 fp = popen(cmd+1, "r");
01901                 while ( fgets(cmd+1, sizeof(cmd)-1, fp) != NULL ) {
01902                     MesPrint("%s", cmd);
01903                     /* Don't show functions lower than "main" (or thread equivalent) */
01904                     if ( strstr(cmd, " main ") || strstr(cmd, " RunThread ") || strstr(cmd, "
RunSortBot ") ) {
01905                         stop = 1;
01906                     }
01907                 }
01908                 pclose(fp);
01909             }
01910         }
01911 #ifdef LINUX
01912         else if ( !system("command -v addr2line > /dev/null 2>&1") ) {
01913             /* Get the executable path. */
01914             char exe_path[PATH_MAX];
01915             {
01916                 ssize_t len = readlink("/proc/self/exe", exe_path, sizeof(exe_path) - 1);
01917                 if ( len != -1 ) {
01918                     exe_path[len] = '\0';
01919                 }
01920                 else {
01921                     goto backtrace_fallback;
01922                 }
01923             }
01924             /* Assume PIE binary and get the base address. */
01925             uintptr_t base_address = 0;
01926             {
01927                 char line[256];
01928                 FILE *maps = fopen("/proc/self/maps", "r");
01929                 if ( !maps ) {
01930                     goto backtrace_fallback;
01931                 }
01932                 /* See the format used by nommu_region_show() in fs/proc/nommu.c of the Linux
source. */
01933                 if ( fgets(line, sizeof(line), maps) ) {
01934                     sscanf(line, "%* SCNxPTR -", &base_address);
01935                 }
01936                 else {
01937                     fclose(maps);
01938                     goto backtrace_fallback;
01939                 }
01940                 fclose(maps);
01941             }
01942             char **strings;
01943             strings = backtrace_symbols(stack, stacksize);
01944             MesPrint("Backtrace:");
01945             for ( int i = 0; i < stacksize && !stop; i++ ) {
01946                 FILE *fp;
01947                 char cmd[PATH_MAX + 512];
01948                 // Leave an initial space
01949                 cmd[0] = ' ';
01950                 uintptr_t addr = (uintptr_t)stack[i] - base_address;
01951                 MesPrint("%#2d:", i);
01952                 snprintf(cmd+1, sizeof(cmd)-1, "addr2line -e \"%s\" -i -p -s -f -C 0x% PRIxPTR,
exe_path, addr);
01953                 fp = popen(cmd+1, "r");
01954                 while ( fgets(cmd+1, sizeof(cmd)-1, fp) != NULL ) {
01955                     MesPrint("%s", cmd);
01956                     /* Don't show functions lower than "main" */
01957                     if ( strstr(cmd, " main ") || strstr(cmd, " RunThread ") || strstr(cmd, "
RunSortBot ") ) {
01958                         stop = 1;
01959                     }
01960                 }
01961                 pclose(fp);
01962             }
01963             free(strings);
01964         }
01965 #endif
01966         else {
01967             /* eu-addr2line not found */
01968 #ifdef LINUX
01969             backtrace_fallback: ;
01970 #endif
01971             char **strings;

```

```

01972         strings = backtrace_symbols(stack, stacksize);
01973         MesPrint("Backtrace:");
01974         for ( int i = 0; i < stacksize && !stop; i++ ) {
01975             char *p = strings[i];
01976             while ( *p && *p != ' (' ) p++;
01977             MesPrint("###%2d: %s\n", i, p);
01978             /* Don't show functions lower than "main" (or thread equivalent) */
01979             if ( strstr(p, "(main)") || strstr(p, "(RunThread)") || strstr(p, "(RunSortBot)")
01980         ) {
01981             stop = 1;
01982         }
01983 #ifdef LINUX
01984         MesPrint("Please install addr2line or eu-addr2line for readable stack information.");
01985 #else
01986         MesPrint("Please install eu-addr2line for readable stack information.");
01987 #endif
01988         free(strings);
01989     }
01990 #else
01991         MesPrint("FORM compiled without backtrace support.");
01992 #endif
01993     } /* if ( AC.PrintBacktraceFlag ) { */
01994
01995     MUNLOCK(ErrorMessageLock);
01996
01997     Crash();
01998 }
01999 #ifdef TRAP_SIGNALS
02000     exitInProgress=1;
02001 #endif
02002 #ifdef WITHEXTERNALCHANNEL
02003 /*
02004     This function can be called from the error handler, so it is better to
02005     clean up all started processes before any activity:
02006 */
02007     closeAllExternalChannels();
02008     AX.currentExternalChannel=0;
02009     /*[08may2006 mt]:*/
02010     AX.killSignal=SIGKILL;
02011     AX.killWholeGroup=1;
02012     AX.daemonize=1;
02013     /*:[08may2006 mt]*/
02014     if(AX.currentPrompt){
02015         M_free(AX.currentPrompt,"external channel prompt");
02016         AX.currentPrompt=0;
02017     }
02018     /*[08may2006 mt]:*/
02019     if(AX.shellname){
02020         M_free(AX.shellname,"external channel shellname");
02021         AX.shellname=0;
02022     }
02023     if(AX.stdername){
02024         M_free(AX.stdername,"external channel stderrname");
02025         AX.stdername=0;
02026     }
02027     /*:[08may2006 mt]*/
02028 #endif
02029 #ifdef WITHPTHREADS
02030     if ( !WhoAmI() && !errorcode ) {
02031         TerminateAllThreads();
02032     }
02033 #endif
02034     if ( AC.FinalStats ) {
02035         if ( AM.PrintTotalSize ) {
02036             MesPrint("Max. space for expressions: %19p bytes",&(AS.MaxExprSize));
02037         }
02038         PrintRunningTime();
02039     }
02040 #ifdef WITHMPI
02041     if ( AM.HoldFlag && PF.me == MASTER ) {
02042         WriteFile(AM.StdOut,(UBYTE *)("Hit any key "),12);
02043         PF_FlushStdOutBuffer();
02044         getchar();
02045     }
02046 #else
02047     if ( AM.HoldFlag ) {
02048         WriteFile(AM.StdOut,(UBYTE *)("Hit any key "),12);
02049         getchar();
02050     }
02051 #endif
02052 #ifdef WITHMPI
02053     PF_Terminate(errorcode);
02054 #endif
02055 /*
02056     We are about to terminate the program. If we are using flint, call the cleanup function.
02057     This keeps valgrind happy.

```

```

02058 */
02059 #ifdef WITHFLINT
02060     flint_final_cleanup_master();
02061 #endif
02062     CleanUp(errorcode);
02063     M_print();
02064 #ifdef VMS
02065     P_term(errorcode? 0: 1);
02066 #else
02067     P_term(errorcode);
02068 #endif
02069 }
02070
02071 /*
02072     #] TerminateImpl :
02073     #[ PrintDeprecation :
02074 */
02075
02082 void PrintDeprecation(const char *feature, const char *issue) {
02083 #ifdef WITHMPI
02084     if ( PF.me != MASTER ) return;
02085 #endif
02086     if ( AM.IgnoreDeprecation ) return;
02087
02088     UBYTE *e = (UBYTE *)getenv("FORM_IGNORE_DEPRECATION");
02089     if ( e && e[0] && StrCmp(e, (UBYTE *)"0") && StrICmp(e, (UBYTE *)"false") && StrICmp(e, (UBYTE
*)"no") ) return;
02090
02091     MesPrint("DeprecationWarning: We are considering deprecating %s.", feature);
02092     MesPrint("If you want this support to continue, leave a comment at:");
02093     MesPrint("");
02094     MesPrint("    https://github.com/vermaseren/form/%s", issue);
02095     MesPrint("");
02096     MesPrint("Otherwise, it will be discontinued in the future.");
02097     MesPrint("To suppress this warning, use the -ignore-deprecation command line option or");
02098     MesPrint("set the environment variable FORM_IGNORE_DEPRECATION=1.");
02099 }
02100
02101 /*
02102     #] PrintDeprecation :
02103     #[ PrintRunningTime :
02104 */
02105
02106 void PrintRunningTime(void)
02107 {
02108 #if (defined(WITHPTHREADS) && (defined(WITHPOSIXCLOCK) || defined(WINDOWS))) || defined(WITHMPI)
02109     LONG mastertime;
02110     LONG workertime;
02111     LONG wallclocktime;
02112     LONG totaltime;
02113 #if defined(WITHPTHREADS)
02114     if ( AB[0] != 0 ) {
02115         workertime = GetWorkerTimes();
02116 #else
02117         workertime = PF_GetSlaveTimes(); /* must be called on all processors */
02118         if ( PF.me == MASTER ) {
02119 #endif
02120             mastertime = AM.SumTime + TimeCPU(1);
02121             wallclocktime = TimeWallClock(1);
02122             totaltime = mastertime+workertime;
02123             if ( !AM.silent ) {
02124                 MesPrint(" %1.2i sec + %1.2i sec: %1.2i sec out of %1.2i sec",
02125                     mastertime/1000, (WORD)((mastertime%1000)/10),
02126                     workertime/1000, (WORD)((workertime%1000)/10),
02127                     totaltime/1000, (WORD)((totaltime%1000)/10),
02128                     wallclocktime/100, (WORD)(wallclocktime%100));
02129             }
02130         }
02131 #else
02132         LONG mastertime = AM.SumTime + TimeCPU(1);
02133         LONG wallclocktime = TimeWallClock(1);
02134         if ( !AM.silent ) {
02135             MesPrint(" %1.2i sec out of %1.2i sec",
02136                 mastertime/1000, (WORD)((mastertime%1000)/10),
02137                 wallclocktime/100, (WORD)(wallclocktime%100));
02138         }
02139 #endif
02140 }
02141
02142 /*
02143     #] PrintRunningTime :
02144     #[ GetRunningTime :
02145 */
02146
02147 LONG GetRunningTime(void)
02148 {
02149 #if defined(WITHPTHREADS) && (defined(WITHPOSIXCLOCK) || defined(WINDOWS))

```

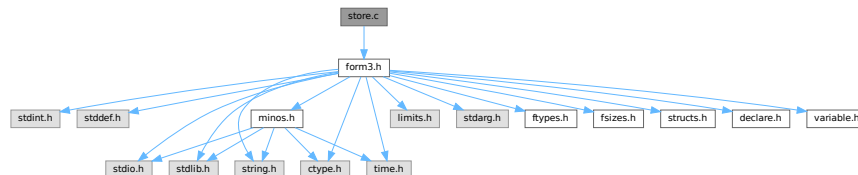
```

02150     LONG mastertime;
02151     if ( AB[0] != 0 ) {
02152     /*
02153     #if ( defined(APPLE64) || defined(APPLE32) )
02154         mastertime = AM.SumTime + TimeCPU(1);
02155         return(mastertime);
02156     #else
02157     */
02158         LONG workertime = GetWorkerTimes();
02159         mastertime = AM.SumTime + TimeCPU(1);
02160         return(mastertime+workertime);
02161     /*
02162     #endif
02163     */
02164     }
02165     else {
02166         return(AM.SumTime + TimeCPU(1));
02167     }
02168     #elif defined(WITHMPI)
02169     LONG mastertime, t = 0;
02170     LONG workertime = PF_GetSlaveTimes(); /* must be called on all processors */
02171     if ( PF.me == MASTER ) {
02172         mastertime = AM.SumTime + TimeCPU(1);
02173         t = mastertime + workertime;
02174     }
02175     return PF_BroadcastNumber(t); /* must be called on all processors */
02176     #else
02177     return(AM.SumTime + TimeCPU(1));
02178     #endif
02179 }
02180
02181 /*
02182     #] GetRunningTime :
02183 */

```

4.135 store.c File Reference

#include "form3.h"
 Include dependency graph for store.c:



Macros

- #define [SAVEREVISION](#) 0x02

Functions

- void [OpenTemp](#) (void)
- void [SeekScratch](#) (FILEHANDLE *fi, POSITION *pos)
- void [SetEndScratch](#) (FILEHANDLE *f, POSITION *position)
- void [SetEndHScratch](#) (FILEHANDLE *f, POSITION *position)
- void [SetScratch](#) (FILEHANDLE *f, POSITION *position)
- int [RevertScratch](#) (void)
- int [ResetScratch](#) (void)
- int [ReadFromScratch](#) (FILEHANDLE *fi, POSITION *pos, UBYTE *buffer, POSITION *length)

- int [AddToScratch](#) ([FILEHANDLE](#) *fi, [POSITION](#) *pos, [UBYTE](#) *buffer, [POSITION](#) *length, int withflush)
- int [CoSave](#) ([UBYTE](#) *inp)
- int [CoLoad](#) ([UBYTE](#) *inp)
- int [DeleteStore](#) ([WORD](#) par)
- int [PutInStore](#) ([INDEXENTRY](#) *ind, [WORD](#) num)
- [WORD](#) [GetTerm](#) ([PHEAD](#) [WORD](#) *term)
- [WORD](#) [GetOneTerm](#) ([PHEAD](#) [WORD](#) *term, [FILEHANDLE](#) *fi, [POSITION](#) *pos, int par)
- [WORD](#) [GetMoreTerms](#) ([WORD](#) *term)
- int [GetMoreFromMem](#) ([WORD](#) *term, [WORD](#) **tpoin)
- [WORD](#) [GetFromStore](#) ([WORD](#) *to, [POSITION](#) *position, [RENUMBER](#) renumber, [WORD](#) *InCompState, [WORD](#) nexpr)
- void [DetVars](#) ([WORD](#) *term, [WORD](#) par)
- int [ToStorage](#) ([EXPRESSIONS](#) e, [POSITION](#) *length)
- [INDEXENTRY](#) * [NextFileIndex](#) ([POSITION](#) *indexpos)
- int [SetFileIndex](#) (void)
- int [VarStore](#) ([UBYTE](#) *s, [WORD](#) n, [WORD](#) name, [WORD](#) namesize)
- int [TermRenumber](#) ([WORD](#) *term, [RENUMBER](#) renumber, [WORD](#) nexpr)
- [WORD](#) [FindrNumber](#) ([WORD](#) n, [VARRENUM](#) *v)
- [INDEXENTRY](#) * [FindInIndex](#) ([WORD](#) expr, [FILEDATA](#) *f, [WORD](#) par, [WORD](#) mode)
- [RENUMBER](#) [GetTable](#) ([WORD](#) expr, [POSITION](#) *position, [WORD](#) mode)
- int [CopyExpression](#) ([FILEHANDLE](#) *from, [FILEHANDLE](#) *to)
- [LONG](#) [WriteStoreHeader](#) ([WORD](#) handle)
- int [ReadSaveHeader](#) (void)
- [LONG](#) [ReadSaveIndex](#) ([FILEINDEX](#) *fileind)
- [LONG](#) [ReadSaveVariables](#) ([UBYTE](#) *buffer, [UBYTE](#) *top, [LONG](#) *size, [LONG](#) *outsize, [INDEXENTRY](#) *ind, [LONG](#) *stage)
- [UBYTE](#) * [ReadSaveTerm32](#) ([UBYTE](#) *bin, [UBYTE](#) *binend, [UBYTE](#) **bout, [UBYTE](#) *boutend, [UBYTE](#) *top, int terminbuf)
- [LONG](#) [ReadSaveExpression](#) ([UBYTE](#) *buffer, [UBYTE](#) *top, [LONG](#) *size, [LONG](#) *outsize)

4.135.1 Detailed Description

Contains all functions that deal with store-files and the system independent save-files.

Definition in file [store.c](#).

4.135.2 Macro Definition Documentation

4.135.2.1 SAVEREVISION

```
#define SAVEREVISION 0x02
```

Definition at line [3953](#) of file [store.c](#).

4.135.3 Function Documentation

4.135.3.1 OpenTemp()

```
void OpenTemp (
    void )
```

Definition at line [48](#) of file [store.c](#).

4.135.3.2 SeekScratch()

```
void SeekScratch (
    FILEHANDLE * fi,
    POSITION * pos )
```

Definition at line 63 of file [store.c](#).

4.135.3.3 SetEndScratch()

```
void SetEndScratch (
    FILEHANDLE * f,
    POSITION * position )
```

Definition at line 74 of file [store.c](#).

4.135.3.4 SetEndHScratch()

```
void SetEndHScratch (
    FILEHANDLE * f,
    POSITION * position )
```

Definition at line 88 of file [store.c](#).

4.135.3.5 SetScratch()

```
void SetScratch (
    FILEHANDLE * f,
    POSITION * position )
```

Definition at line 114 of file [store.c](#).

4.135.3.6 RevertScratch()

```
int RevertScratch (
    void )
```

Definition at line 189 of file [store.c](#).

4.135.3.7 ResetScratch()

```
int ResetScratch (
    void )
```

Definition at line 234 of file [store.c](#).

4.135.3.8 ReadFromScratch()

```
int ReadFromScratch (
    FILEHANDLE * fi,
    POSITION * pos,
    UBYTE * buffer,
    POSITION * length )
```

Definition at line 272 of file [store.c](#).

4.135.3.9 AddToScratch()

```
int AddToScratch (
    FILEHANDLE * fi,
    POSITION * pos,
    UBYTE * buffer,
    POSITION * length,
    int withflush )
```

Definition at line 299 of file [store.c](#).

4.135.3.10 CoSave()

```
int CoSave (
    UBYTE * inp )
```

Definition at line 362 of file [store.c](#).

4.135.3.11 CoLoad()

```
int CoLoad (
    UBYTE * inp )
```

Definition at line 572 of file [store.c](#).

4.135.3.12 DeleteStore()

```
int DeleteStore (
    WORD par )
```

Definition at line 772 of file [store.c](#).

4.135.3.13 PutInStore()

```
int PutInStore (
    INDEXENTRY * ind,
    WORD num )
```

Definition at line 873 of file [store.c](#).

4.135.3.14 GetTerm()

```
WORD GetTerm (
    PHEAD WORD * term )
```

Definition at line 992 of file [store.c](#).

4.135.3.15 GetOneTerm()

```
WORD GetOneTerm (
    PHEAD WORD * term,
    FILEHANDLE * fi,
    POSITION * pos,
    int par )
```

Definition at line 1267 of file [store.c](#).

4.135.3.16 GetMoreTerms()

```
WORD GetMoreTerms (
    WORD * term )
```

Definition at line 1425 of file [store.c](#).

4.135.3.17 GetMoreFromMem()

```
int GetMoreFromMem (
    WORD * term,
    WORD ** tpoin )
```

Definition at line 1521 of file [store.c](#).

4.135.3.18 GetFromStore()

```
WORD GetFromStore (
    WORD * to,
    POSITION * position,
    RENUMBER renumber,
    WORD * InCompState,
    WORD nexpr )
```

Definition at line 1628 of file [store.c](#).

4.135.3.19 DetVars()

```
void DetVars (
    WORD * term,
    WORD par )
```

Definition at line 1829 of file [store.c](#).

4.135.3.20 ToStorage()

```
int ToStorage (
    EXPRESSIONS e,
    POSITION * length )
```

Definition at line 2016 of file [store.c](#).

4.135.3.21 NextFileIndex()

```
INDEXENTRY * NextFileIndex (
    POSITION * indexpos )
```

Definition at line 2257 of file [store.c](#).

4.135.3.22 SetFileIndex()

```
int SetFileIndex (
    void )
```

Reads the next file index and puts it into AR.StoreData.Index. TODO

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 2320 of file [store.c](#).

References [WriteStoreHeader\(\)](#).

Here is the call graph for this function:



4.135.3.23 VarStore()

```
int VarStore (
    UBYTE * s,
    WORD n,
    WORD name,
    WORD namesize )
```

Definition at line 2369 of file [store.c](#).

4.135.3.24 TermRenumber()

```
int TermRenumber (
    WORD * term,
    RENUMBER renumber,
    WORD nexpr )
```

!! WORD *memterm=term; static LONG ctrap=0; !!!

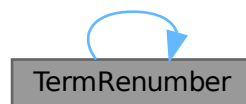
!! ctrap++; !!!

Definition at line 2427 of file [store.c](#).

References [ReNuMbEr::func](#), [ReNuMbEr::funnum](#), [ReNuMbEr::indi](#), [ReNuMbEr::indnum](#), [ReNuMbEr::symb](#), [ReNuMbEr::symnum](#), [TermRenumber\(\)](#), [ReNuMbEr::vecnum](#), and [ReNuMbEr::vect](#).

Referenced by [TermRenumber\(\)](#).

Here is the call graph for this function:



4.135.3.25 FindrNumber()

```
WORD FindrNumber (
    WORD n,
    VARRENUM * v )
```

Definition at line 2587 of file [store.c](#).

4.135.3.26 FindInIndex()

```
INDEXENTRY * FindInIndex (
    WORD expr,
    FILEDATA * f,
    WORD par,
    WORD mode )
```

Definition at line 2664 of file [store.c](#).

4.135.3.27 GetTable()

```
RENUMBER GetTable (
    WORD expr,
    POSITION * position,
    WORD mode )
```

Definition at line [2825](#) of file [store.c](#).

4.135.3.28 CopyExpression()

```
int CopyExpression (
    FILEHANDLE * from,
    FILEHANDLE * to )
```

Definition at line [3307](#) of file [store.c](#).

4.135.3.29 WriteStoreHeader()

```
LONG WriteStoreHeader (
    WORD handle )
```

Writes header with information about system architecture and FORM revision to an open store file.

Called by [SetFileIndex\(\)](#).

Parameters

<i>handle</i>	specifies open file to which header will be written

Returns

= 0 everything okay, != 0 an error occurred

Definition at line [3964](#) of file [store.c](#).

References [STOREHEADER::endianness](#), [STOREHEADER::lenLONG](#), [STOREHEADER::lenPOINTER](#), [STOREHEADER::lenPOS](#), [STOREHEADER::lenWORD](#), [STOREHEADER::maxpower](#), [STOREHEADER::sFun](#), [STOREHEADER::slnd](#), [STOREHEADER::sSym](#), [STOREHEADER::sVec](#), and [STOREHEADER::wildoffset](#).

Referenced by [SetFileIndex\(\)](#).

4.135.3.30 ReadSaveHeader()

```
int ReadSaveHeader (
    void )
```

Reads the header in the save file and sets function pointers and flags according to the information found there. Must be called before any other ReadSave... function.

Currently works only for the exchange between 32bit and 64bit machines (WORD size must be 2 or 4 bytes)!

It is called by [CoLoad\(\)](#).

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 4057 of file [store.c](#).

4.135.3.31 ReadSaveIndex()

```
LONG ReadSaveIndex (
    FILEINDEX * fileind )
```

Reads a FILEINDEX from the open save file specified by AO.SaveData.Handle. Translations for adjusting endianness and data sizes are done if necessary.

Depends on the assumption that sizeof(FILEINDEX) is the same everywhere. If FILEINDEX or INDEXENTRY change, then this functions has to be adjusted.

Called by CoLoad() and FindInIndex().

Parameters

<i>fileind</i>	contains the read FILEINDEX after successful return. must point to allocated, big enough memory.

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 4175 of file [store.c](#).

References [INFILEINDEX](#), [FiLeInDeX::next](#), [FiLeInDeX::number](#), and [FuNcTiOn::number](#).

4.135.3.32 ReadSaveVariables()

```
LONG ReadSaveVariables (
    UBYTE * buffer,
    UBYTE * top,
    LONG * size,
    LONG * outsize,
    INDEXENTRY * ind,
    LONG * stage )
```

Reads the variables from the open file specified by AO.SaveData.Handle. It reads the *size bytes and writes them to the *buffer. It is called by PutInStore().

If translation is necessary, the data might shrink or grow in size, then *size is adjusted so that the reading and writing fits into the memory from the buffer to the top. The actual number of read bytes is returned in *size, the number of written bytes is returned in *outsize.

If the *size is smaller than the actual size of the variables, this function will be called several times and needs to remember the current position in the variable structure. The parameter *stage* does this job. When [ReadSaveVariables\(\)](#) is called for the first time, this parameter should have the value -1.

The parameter *ind* is used to get the number of variables.

Parameters

<i>buffer</i>	read variables are written into this allocated memory
<i>top</i>	upper end of allocated memory
<i>size</i>	number of bytes to read. might return a smaller number of read bytes if translation was necessary
<i>outsize</i>	if translation has be done, outsize contains the number of written bytes
<i>ind</i>	pointer of INDEXENTRY for the current expression. read-only
<i>stage</i>	should be -1 for the first call, will be increased by ReadSaveVariables to memorize the position in the variable structure

Returns

= 0 everything okay, != 0 an error occurred

Definition at line 4361 of file [store.c](#).

References [InDeXeNtRy::nfunctions](#), [InDeXeNtRy::nindices](#), [InDeXeNtRy::nsymbols](#), [InDeXeNtRy::nvectors](#), and [FuNcTiOn::spec](#).

4.135.3.33 ReadSaveTerm32()

```

UBYTE * ReadSaveTerm32 (
    UBYTE * bin,
    UBYTE * binend,
    UBYTE ** bout,
    UBYTE * boutend,
    UBYTE * top,
    int terminbuf )

```

Reads a single term from the given buffer at *bin* and write the translated term back to this buffer at *bout*.

[ReadSaveTerm32\(\)](#) is currently the only instantiation of a ReadSaveTerm-function. It only deals with data that already has the correct endianness and that is resized to 32bit words but without being renumbered or translated in any other way. It uses the compress buffer [AR.CompressBuffer](#).

The function is reentrant in order to cope with nested function arguments. It is called by [ReadSaveExpression\(\)](#) and itself.

The *return value* indicates the position in the input buffer up to which the data has already been successfully processed. The parameter *bout* returns the corresponding position in the output buffer.

Parameters

<i>bin</i>	start of the input buffer
<i>binend</i>	end of the input buffer
<i>bout</i>	as input points to the beginning of the output buffer, as output points behind the already translated data in the output buffer
<i>boutend</i>	end of already decompressed data in output buffer
<i>top</i>	end of output buffer
<i>terminbuf</i>	flag whether decompressed data is already in the output buffer. used in recursive calls

Returns

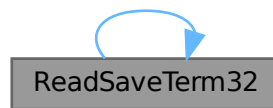
pointer to the next unprocessed data in the input buffer

Definition at line 4728 of file [store.c](#).

References [ReadSaveTerm32\(\)](#).

Referenced by [ReadSaveExpression\(\)](#), and [ReadSaveTerm32\(\)](#).

Here is the call graph for this function:

**4.135.3.34 ReadSaveExpression()**

```
LONG ReadSaveExpression (
    UBYTE * buffer,
    UBYTE * top,
    LONG * size,
    LONG * outsize )
```

Reads an expression from the open file specified by `AO.SaveData.Handle`. The endianness flip and a resizing without renumbering is done in this function. Thereafter the buffer consists of chunks with a uniform maximal word size (32bit at the moment). The actual renumbering is then done by calling the function [ReadSaveTerm32\(\)](#). The result is returned in *buffer*.

If the translation at some point doesn't fit into the buffer anymore, the function returns and must be called again. In any case *size* returns the number of successfully read bytes, *outsize* returns the number of successfully written bytes, and the file will be positioned at the next byte after the successfully read data.

It is called by `PutInStore()`.

Parameters

<i>buffer</i>	output buffer, holds the (translated) expression
<i>top</i>	end of buffer
<i>size</i>	number of read bytes
<i>outsize</i>	number of written bytes

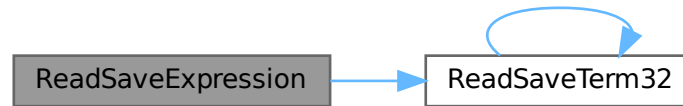
Returns

= 0 everything okay, != 0 an error occurred

Definition at line 5138 of file [store.c](#).

References [ReadSaveTerm32\(\)](#).

Here is the call graph for this function:



4.136 store.c

[Go to the documentation of this file.](#)

```

00001
00006 /* #[] License : */
00007 /*
00008 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00009 * When using this file you are requested to refer to the publication
00010 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011 * This is considered a matter of courtesy as the development was paid
00012 * for by FOM the Dutch physics granting agency and we would like to
00013 * be able to track its scientific use to convince FOM of its value
00014 * for the community.
00015 *
00016 * This file is part of FORM.
00017 *
00018 * FORM is free software: you can redistribute it and/or modify it under the
00019 * terms of the GNU General Public License as published by the Free Software
00020 * Foundation, either version 3 of the License, or (at your option) any later
00021 * version.
00022 *
00023 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00026 * details.
00027 *
00028 * You should have received a copy of the GNU General Public License along
00029 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00030 */
00031 /* #[] License : */
00032 /*
00033 #define HIDEDEBUG
00034 #[] Includes : store.c
00035 */
00036
00037 #include "form3.h"
00038
00039 /*
00040 #[] Includes :
00041 #[] StoreExpressions :
00042 #[] OpenTemp :
00043
00044 Opens the scratch files for the input -> output operations.
00045
00046 */
00047
00048 void OpenTemp(void)
00049 {
00050     GETIDENTITY
00051     if ( AR.outfile->handle >= 0 ) {
00052         SeekFile(AR.outfile->handle, &(AR.outfile->filesize), SEEK_SET);
00053         AR.outfile->PPosition = AR.outfile->filesize;
00054         AR.outfile->POfill = AR.outfile->PObuffer;
00055     }
00056 }
00057

```



```

00058 /*
00059     #] OpenTemp :
00060     #[ SeekScratch :
00061 */
00062
00063 void SeekScratch(FILEHANDLE *fi, POSITION *pos)
00064 {
00065     *pos = fi->POposition;
00066     ADDPOS(*pos, (TOLONG(fi->POfill)-TOLONG(fi->PObuffer)));
00067 }
00068
00069 /*
00070     #] SeekScratch :
00071     #[ SetEndScratch :
00072 */
00073
00074 void SetEndScratch(FILEHANDLE *f, POSITION *position)
00075 {
00076     if ( f->handle < 0 ) {
00077         SETBASEPOSITION(*position, (f->POfull-f->PObuffer)*sizeof(WORD));
00078     }
00079     else *position = f->filesize;
00080     SetScratch(f, position);
00081 }
00082
00083 /*
00084     #] SetEndScratch :
00085     #[ SetEndHScratch :
00086 */
00087
00088 void SetEndHScratch(FILEHANDLE *f, POSITION *position)
00089 {
00090     if ( f->handle < 0 ) {
00091         SETBASEPOSITION(*position, (f->POfull-f->PObuffer)*sizeof(WORD));
00092         f->POfill = f->POfull;
00093     }
00094     else {
00095 #ifdef HIDEDEBUG
00096         POSITION possize;
00097         PUTZERO(possiz);
00098         SeekFile(f->handle, &possiz, SEEK_END);
00099         MesPrint("SetEndHScratch: filesize(th) = %12p, filesize(ex) = %12p", &(f->filesize),
00100             &(possiz));
00101 #endif
00102         *position = f->filesize;
00103         f->POposition = f->filesize;
00104         f->POfill = f->POfull = f->PObuffer;
00105     }
00106     /* SetScratch(f, position); */
00107 }
00108
00109 /*
00110     #] SetEndHScratch :
00111     #[ SetScratch :
00112 */
00113
00114 void SetScratch(FILEHANDLE *f, POSITION *position)
00115 {
00116     GETIDENTITY
00117     POSITION possiz;
00118     LONG size, *whichInInBuf;
00119     if ( f == AR.hidefile ) whichInInBuf = &(AR.InHiBuf);
00120     else whichInInBuf = &(AR.InInBuf);
00121 #ifdef HIDEDEBUG
00122     if ( f == AR.hidefile ) MesPrint("In the hide file");
00123     else MesPrint("In the input file");
00124     MesPrint("SetScratch to position %15p", position);
00125     MesPrint("POposition = %15p, full = %1, fill = %1"
00126         , &(f->POposition), (f->POfull-f->PObuffer)*sizeof(WORD)
00127         , (f->POfill-f->PObuffer)*sizeof(WORD));
00128 #endif
00129     if ( ISLESSPOS(*position, f->POposition) ||
00130         ISGEPOSINC(*position, f->POposition, (f->POfull-f->PObuffer)*sizeof(WORD)) ) {
00131         if ( f->handle < 0 ) {
00132             if ( ISEQUALPOSINC(*position, f->POposition,
00133                 (f->POfull-f->PObuffer)*sizeof(WORD)) ) goto endpos;
00134             MesPrint("Illegal position in SetScratch");
00135             Terminate(-1);
00136         }
00137         possiz = *position;
00138         LOCK(AS.inputslock);
00139         SeekFile(f->handle, &possiz, SEEK_SET);
00140         if ( ISNOTEQUALPOS(possiz, *position) ) {
00141             UNLOCK(AS.inputslock);
00142             MesPrint("Cannot position file in SetScratch");
00143             Terminate(-1);
00144         }
00145     }

```

```

00145 #ifdef HIDEDEBUG
00146     MesPrint("SetScratch1(%w): position = %l2p, size = %l, address =
00147     %x",position,f->POsize,f->PObuffer);
00148 #endif
00149     if ( ( size = ReadFile(f->handle, (UBYTE *) (f->PObuffer),f->POsize) ) < 0
00150         || ( size & 1 ) != 0 ) {
00151         UNLOCK(AS.inputslock);
00152         MesPrint("Read error in SetScratch");
00153         Terminate(-1);
00154     }
00155     UNLOCK(AS.inputslock);
00156     if ( size == 0 ) {
00157         f->PObuffer[0] = 0;
00158     }
00159     f->POfill = f->PObuffer;
00160     f->POposition = *position;
00161 #ifdef WORD2
00162     *whichInInBuf = size » 1;
00163 #else
00164     *whichInInBuf = size / TABLESIZE(WORD,UBYTE);
00165 #endif
00166     f->POfull = f->PObuffer + *whichInInBuf;
00167 #ifdef HIDEDEBUG
00168     MesPrint("SetScratch2: size = %l, InInBuf = %l, fill = %l, full = %l"
00169             ,size,*whichInInBuf,(f->POfill-f->PObuffer)*sizeof(WORD)
00170             ,(f->POfull-f->PObuffer)*sizeof(WORD));
00171 #endif
00172     }
00173     else {
00174     endpos:
00175         DIFPOS(possiz,*position,f->POposition);
00176         f->POfill = (WORD *) (BASEPOSITION(possiz)+(UBYTE *) (f->PObuffer));
00177         *whichInInBuf = f->POfull-f->POfill;
00178     }
00179 }
00180 /*
00181     #] SetScratch :
00182     #[ RevertScratch :
00183
00184     Reverts the input/output directions. This way input comes
00185     always from AR.infile
00186 */
00187 */
00188
00189 int RevertScratch(void)
00190 {
00191     GETIDENTITY
00192     FILEHANDLE *f;
00193     if ( AR.infile->handle >= 0 && AR.infile->handle != AR.outfile->handle ) {
00194         CloseFile(AR.infile->handle);
00195         AR.infile->handle = -1;
00196         remove(AR.infile->name);
00197     }
00198     f = AR.infile; AR.infile = AR.outfile; AR.outfile = f;
00199     AR.infile->POfull = AR.infile->POfill;
00200     AR.infile->POfill = AR.infile->PObuffer;
00201     if ( AR.infile->handle >= 0 ) {
00202         POSITION scrpos;
00203         PUTZERO(scrpos);
00204         SeekFile(AR.infile->handle,&scrpos,SEEK_SET);
00205         if ( ISNOTZEROPOS(scrpos) ) {
00206             return(MesPrint("Error with scratch output."));
00207         }
00208         if ( ( AR.InInBuf = ReadFile(AR.infile->handle, (UBYTE *) (AR.infile->PObuffer)
00209             ,AR.infile->POsize) ) < 0 || AR.InInBuf & 1 ) {
00210             return(MesPrint("Error while reading from scratch file"));
00211         }
00212         else {
00213             AR.InInBuf /= TABLESIZE(WORD,UBYTE);
00214         }
00215         AR.infile->POfull = AR.infile->PObuffer + AR.InInBuf;
00216     }
00217     PUTZERO(AR.infile->POposition);
00218     AR.outfile->POfill = AR.outfile->POfull = AR.outfile->PObuffer;
00219     PUTZERO(AR.outfile->POposition);
00220     PUTZERO(AR.outfile->filesize);
00221     return(0);
00222 }
00223
00224 /*
00225     #] RevertScratch :
00226     #[ ResetScratch :
00227
00228     Resets the output scratch file to its beginning in such a way
00229     that the write routines can read it. The output buffers are
00230     left untouched as they may still be needed for extra declarations.

```

```

00231
00232 */
00233
00234 int ResetScratch(void)
00235 {
00236     GETIDENTITY
00237     FILEHANDLE *f;
00238     if ( AR.infile->handle >= 0 ) {
00239         CloseFile(AR.infile->handle); AR.infile->handle = -1;
00240         remove(AR.infile->name);
00241         PUTZERO(AR.infile->POposition);
00242         AR.infile->POfill = AR.infile->POfull = AR.infile->PObuffer;
00243     }
00244     if ( AR.outfile->handle >= 0 ) {
00245         POSITION scrpos;
00246         PUTZERO(scrpos);
00247         SeekFile(AR.outfile->handle,&scrpos,SEEK_SET);
00248         if ( ISNOTZEROPOS(scrpos) ) {
00249             return(MesPrint("Error with scratch output."));
00250         }
00251         if ( ( AR.InInBuf = ReadFile(AR.outfile->handle,(UBYTE *) (AR.outfile->PObuffer)
00252 ,AR.outfile->POsize) ) < 0 || AR.InInBuf & 1 ) {
00253             return(MesPrint("Error while reading from scratch file"));
00254         }
00255         else AR.InInBuf /= TABLESIZE(WORD,UBYTE);
00256         AR.outfile->POfull = AR.outfile->PObuffer + AR.InInBuf;
00257     }
00258     else AR.outfile->POfull = AR.outfile->POfill;
00259     AR.outfile->POfill = AR.outfile->PObuffer;
00260     PUTZERO(AR.outfile->POposition);
00261     f = AR.outfile; AR.outfile = AR.infile; AR.infile = f;
00262     return(0);
00263 }
00264
00265 /*
00266     #] ResetScratch :
00267     #[ ReadFromScratch :
00268
00269     Routine is used to copy files from scratch to hide.
00270 */
00271
00272 int ReadFromScratch(FILEHANDLE *fi, POSITION *pos, UBYTE *buffer, POSITION *length)
00273 {
00274     GETIDENTITY
00275     LONG l = BASEPOSITION(*length);
00276     if ( fi->handle < 0 ) {
00277         memcpy(buffer,fi->POfill,l);
00278     }
00279     else {
00280         SeekFile(fi->handle,pos,SEEK_SET);
00281         if ( ReadFile(fi->handle,buffer,l) != l ) {
00282             if ( fi == AR.hidefile )
00283                 MesPrint("Error reading from hide file.");
00284             else
00285                 MesPrint("Error reading from scratch file.");
00286             return(-1);
00287         }
00288     }
00289     return(0);
00290 }
00291
00292 /*
00293     #] ReadFromScratch :
00294     #[ AddToScratch :
00295
00296     Routine is used to copy files from scratch to hide.
00297 */
00298
00299 int AddToScratch(FILEHANDLE *fi, POSITION *pos, UBYTE *buffer, POSITION *length,
00300 int withflush)
00301 {
00302     GETIDENTITY
00303     LONG l = BASEPOSITION(*length), avail;
00304     DUMMYUSE(pos)
00305     fi->POfill = fi->POfull;
00306     while ( fi->POfill+l/sizeof(WORD) > fi->POstop ) {
00307         avail = (fi->POstop-fi->POfill)*sizeof(WORD);
00308         if ( avail > 0 ) {
00309             memcpy(fi->POfill,buffer,avail);
00310             l -= avail; buffer += avail;
00311         }
00312         if ( fi->handle < 0 ) {
00313             if ( ( fi->handle = (WORD)CreateFile(fi->name) ) < 0 ) {
00314                 if ( fi == AR.hidefile )
00315                     MesPrint("Cannot create hide file %s",fi->name);
00316                 else
00317                     MesPrint("Cannot create scratch file %s",fi->name);

```

```

00318         return(-1);
00319     }
00320     PUTZERO(fi->POposition);
00321 }
00322 SeekFile(fi->handle, &(fi->POposition), SEEK_SET);
00323 if ( WriteFile(fi->handle, (UBYTE *)fi->PObuffer, fi->POsize) != fi->POsize )
00324     goto writeerror;
00325 ADDPOS(fi->POposition, fi->POsize);
00326 fi->POfill = fi->POfull = fi->PObuffer;
00327 }
00328 if ( l > 0 ) {
00329     memcpy(fi->POfill, buffer, l);
00330     fi->POfill += l/sizeof(WORD);
00331     fi->POfull = fi->POfill;
00332 }
00333 if ( withflush && fi->handle >= 0 && fi->POfill > fi->PObuffer ) { /* flush */
00334     l = (LONG)fi->POfill - (LONG)fi->PObuffer;
00335     SeekFile(fi->handle, &(fi->POposition), SEEK_SET);
00336     if ( WriteFile(fi->handle, (UBYTE *)fi->PObuffer, l) != l ) goto writeerror;
00337     ADDPOS(fi->POposition, fi->POsize);
00338     fi->POfill = fi->POfull = fi->PObuffer;
00339 }
00340 if ( withflush && fi->handle >= 0 )
00341     SETBASEPOSITION(fi->filesize, TellFile(fi->handle));
00342 return(0);
00343 writeerror:
00344 if ( fi == AR.hidefile )
00345     MesPrint("Error writing to hide file. Disk full?");
00346 else
00347     MesPrint("Error writing to scratch file. Disk full?");
00348 return(-1);
00349 }
00350
00351 /*
00352     #] AddToScratch :
00353     #[ CoSave :
00354
00355     The syntax of the save statement is:
00356
00357     save filename
00358     save filename expr1 expr2
00359 */
00360
00361 int CoSave(UBYTE *inp)
00362 {
00363     GETIDENTITY
00364     UBYTE *p, c;
00365     WORD n = 0, i;
00366     WORD error = 0, type, number;
00367     WORD exprInStorageFlag = 0;
00368     LONG RetCode = 0, wSize;
00369     EXPRESSIONS e;
00370     INDEXENTRY *ind;
00371     INDEXENTRY *indold;
00372     WORD TMproto[SUBEXPSIZE];
00373     POSITION scrpos, scrpos1, filesize;
00374     int ii, j = sizeof(FILEINDEX)/(sizeof(LONG));
00375     LONG *lo;
00376     while ( *inp == ',' ) inp++;
00377     p = inp;
00378
00379 #ifdef WITHMPI
00380     if ( PF.me != MASTER ) return(0);
00381 #endif
00382
00383     if ( !*p ) return(MesPrint("No filename in save statement"));
00384     if ( FG.cTable[*p] > 1 && ( *p != '.' ) && ( *p != SEPARATOR ) && ( *p != ALTSEPARATOR ) )
00385         return(MesPrint("Illegal filename"));
00386     while ( ++p && *p != ',' ) {}
00387     c = *p;
00388     *p = 0;
00389     if ( !AP.preError ) {
00390         if ( ( RetCode = CreateFile((char *)inp) ) < 0 ) {
00391             return(MesPrint("Cannot open file %s", inp));
00392         }
00393     }
00394     AO.SaveData.Handle = (WORD)RetCode;
00395     PUTZERO(filesize);
00396
00397     e = Expressions;
00398     n = NumExpressions;
00399     if ( c ) { /* There follows a list of expressions */
00400         *p++ = c;
00401         inp = p;
00402         i = (WORD)(INFILEINDEX);
00403         if ( WriteStoreHeader(AO.SaveData.Handle) ) return(MesPrint("Error writing storage file

```

```

header"));
00405 /* PUTZERO(AO.SaveData.Index.number); */
00406 /* PUTZERO(AO.SaveData.Index.next); */
00407 lo = (LONG *)(&AO.SaveData.Index);
00408 for ( ii = 0; ii < j; ii++ ) *lo++ = 0;
00409 SETBASEPOSITION(AO.SaveData.Position, (LONG)sizeof(STOREHEADER));
00410 ind = AO.SaveData.Index.expression;
00411 if ( !AP.preError && WriteFile(AO.SaveData.Handle, (UBYTE *)(&(AO.SaveData.Index))
00412 , (LONG)sizeof(struct FileInDeX))!= (LONG)sizeof(struct FileInDeX) ) goto SavWrt;
00413 SeekFile(AO.SaveData.Handle, &(filesize), SEEK_END);
00414 /* ADDPOS(filesize, sizeof(struct FileInDeX)); */
00415
00416 do { /* Scan the list */
00417     if ( !FG.cTable[*p] || *p == '[' ) {
00418         p = SkipAName(p);
00419         if ( p == 0 ) return(-1);
00420     }
00421     c = *p; *p = 0;
00422     if ( GetVar(inp, &type, &number, CEXPRESSION, NOAUTO) != NAMENOTFOUND ) {
00423         if ( e[number].status == STOREDEXPRESSION ) {
00424             if ( StrLen(AC.exprnames->namebuffer+e[number].name) > MAXENAME ) {
00425                 char msg[100];
00426                 sprintf(msg, "saved expr name over %d char: %s", MAXENAME,
AC.exprnames->namebuffer+e[number].name);
00427                 Warning(msg);
00428             }
00429 /*
00430 Here we have to locate the stored expression, copy its index entry
00431 possibly after making a new fileindex and then copy the whole
00432 expression.
00433 */
00434         if ( AP.preError ) goto NextExpr;
00435         TMproto[0] = EXPRESSION;
00436         TMproto[1] = SUBEXPSIZE;
00437         TMproto[2] = number;
00438         TMproto[3] = 1;
00439         { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
00440         AT.TMaddr = TMproto;
00441         if ( ( indold = FindInIndex(number, &AR.StoreData, 0, 0) ) != 0 ) {
00442             if ( i <= 0 ) {
00443 /*
00444             AO.SaveData.Index.next = filesize;
00445 */
00446             SeekFile(AO.SaveData.Handle, &(AO.SaveData.Index.next), SEEK_END);
00447             scrpos = AO.SaveData.Position;
00448             SeekFile(AO.SaveData.Handle, &scrpos, SEEK_SET);
00449             if ( ISNOTEQUALPOS(scrpos, AO.SaveData.Position) ) goto SavWrt;
00450             if ( WriteFile(AO.SaveData.Handle, (UBYTE *)(&(AO.SaveData.Index))
, (LONG)sizeof(struct FileInDeX)) != (LONG)sizeof(struct FileInDeX)
00451 )
00452                 goto SavWrt;
00453             i = (WORD)(INFILEINDEX);
00454             AO.SaveData.Position = AO.SaveData.Index.next;
00455             lo = (LONG *)(&AO.SaveData.Index);
00456             for ( ii = 0; ii < j; ii++ ) *lo++ = 0;
00457             ind = AO.SaveData.Index.expression;
00458             scrpos = AO.SaveData.Position;
00459             SeekFile(AO.SaveData.Handle, &scrpos, SEEK_SET);
00460             if ( ISNOTEQUALPOS(scrpos, AO.SaveData.Position) ) goto SavWrt;
00461             if ( WriteFile(AO.SaveData.Handle, (UBYTE *)(&(AO.SaveData.Index))
, (LONG)sizeof(struct FileInDeX)) != (LONG)sizeof(struct FileInDeX) )
00462                 goto SavWrt;
00463             ADDPOS(filesize, sizeof(struct FileInDeX));
00464         }
00465         *ind = *indold;
00466 /*
00467 ind->variables = SeekFile(AO.SaveData.Handle, &(AM.zeropos), SEEK_END);
00468 */
00469 /*
00470 ind->variables = filesize;
00471 ind->position = ind->variables;
00472 ADDPOS(ind->position, DIFBASE(indold->position, indold->variables));
00473 SeekFile(AR.StoreData.Handle, &(indold->variables), SEEK_SET);
00474 wSize = TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer);
00475 scrpos = ind->length;
00476 ADDPOS(scrpos, DIFBASE(ind->position, ind->variables));
00477 ADD2POS(filesize, scrpos);
00478 SETBASEPOSITION(scrpos1, wSize);
00479 do {
00480     if ( ISLESSPOS(scrpos, scrpos1) ) wSize = BASEPOSITION(scrpos);
00481     if ( ReadFile(AR.StoreData.Handle, (UBYTE *)AT.WorkPointer, wSize)
!= wSize ) {
00482         MesPrint("ReadError");
00483         error = -1;
00484         goto EndSave;
00485     }
00486     if ( WriteFile(AO.SaveData.Handle, (UBYTE *)AT.WorkPointer, wSize)
!= wSize ) goto SavWrt;

```

```

00489             ADDPOS(scrpos,-wSize);
00490         } while ( ISPOSPOS(scrpos) );
00491         ADDPOS(AO.SaveData.Index.number,1);
00492         ind++;
00493     }
00494     else error = -1;
00495     i--;
00496 }
00497 else {
00498     MesPrint("%s is not a stored expression",inp);
00499     error = -1;
00500 }
00501 NextExpr;;
00502 }
00503 else {
00504     MesPrint("%s is not an expression",inp);
00505     error = -1;
00506 }
00507 *p = c;
00508 if ( c != ',' && c ) {
00509     MesComp("Illegal character",inp,p);
00510     error = -1;
00511     goto EndSave;
00512 }
00513 if ( c ) c = *++p;
00514 inp = p;
00515 } while ( c );
00516 if ( !AP.preError ) {
00517     scrpos = AO.SaveData.Position;
00518     SeekFile(AO.SaveData.Handle,&scrpos,SEEK_SET);
00519     if ( ISNOTEQUALPOS(scrpos,AO.SaveData.Position) ) goto SavWrt;
00520 }
00521 if ( !AP.preError &&
00522 WriteFile(AO.SaveData.Handle,(UBYTE *)(&(AO.SaveData.Index))
00523 , (LONG)sizeof(struct FileInDeX) ) != (LONG)sizeof(struct FileInDeX) ) goto SavWrt;
00524 }
00525 else if ( !AP.preError ) { /* All stored expressions should be saved. Easy */
00526     /* Make sure there is at least one stored expression: */
00527     if ( n > 0 ) { do {
00528         if ( StrLen(AC.exprnames->namebuffer+e->name) > MAXENAME ) {
00529             char msg[100];
00530             snprintf(msg, sizeof(msg), "saved expr name over %d char: %s", MAXENAME,
00531 AC.exprnames->namebuffer+e->name);
00532             Warning(msg);
00533         }
00534         if ( e->status == STOREDEXPRESSION ) exprInStorageFlag = 1;
00535         e++;
00536     } while ( --n > 0 ); }
00537     if ( exprInStorageFlag ) {
00538         wSize = TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer);
00539         PUTZERO(scrpos);
00540         SeekFile(AR.StoreData.Handle,&scrpos,SEEK_SET); /* Start at the beginning */
00541         scrpos = AR.StoreData.Fill; /* Number of bytes to be copied */
00542         SETBASEPOSITION(scrpos1,wSize);
00543         do {
00544             if ( ISLESSPOS(scrpos,scrpos1) ) wSize = BASEPOSITION(scrpos);
00545             if ( ReadFile(AR.StoreData.Handle,(UBYTE *)AT.WorkPointer,wSize) != wSize ) {
00546                 MesPrint("ReadError");
00547                 error = -1;
00548                 goto EndSave;
00549             }
00550             if ( WriteFile(AO.SaveData.Handle,(UBYTE *)AT.WorkPointer,wSize) != wSize )
00551                 goto SavWrt;
00552             ADDPOS(scrpos,-wSize);
00553         } while ( ISPOSPOS(scrpos) );
00554     }
00555 EndSave:
00556     if ( !AP.preError ) {
00557         CloseFile(AO.SaveData.Handle);
00558         AO.SaveData.Handle = -1;
00559     }
00560     return(error);
00561 SavWrt:
00562     MesPrint("WriteError");
00563     error = -1;
00564     goto EndSave;
00565 }
00566
00567 /*
00568     #] CoSave :
00569     #[ CoLoad :
00570 */
00571
00572 int CoLoad(UBYTE *inp)
00573 {
00574     GETIDENTITY

```

```

00575     INDEXENTRY *ind;
00576     LONG RetCode;
00577     UBYTE *p, c;
00578     WORD num, i, error = 0;
00579     WORD type, number, silentload = 0;
00580     WORD TMproto[SUBEXPSIZE];
00581     POSITION scrpos, firstposition;
00582     while ( *inp == ',' ) inp++;
00583     p = inp;
00584     if ( ( *p == ',' && p[1] == '-' ) || *p == '-' ) {
00585         if ( *p == ',' ) p++;
00586         p++;
00587         if ( *p == 's' || *p == 'S' ) {
00588             silentload = 1;
00589             while ( *p && ( *p != ',' && *p != '-' && *p != '+'
00590                 && *p != SEPARATOR && *p != ALTSEPARATOR && *p != '.' ) ) p++;
00591         }
00592         else if ( *p != ',' ) {
00593             return(MesPrint("Illegal option in Load statement"));
00594         }
00595         while ( *p == ',' ) p++;
00596     }
00597     inp = p;
00598     if ( !*p ) return(MesPrint("No filename in load statement"));
00599     if ( FG.cTable[*p] > 1 && ( *p != '.' ) && ( *p != SEPARATOR ) && ( *p != ALTSEPARATOR ) )
00600         return(MesPrint("Illegal filename"));
00601     while ( ++p && *p != ',' ) {}
00602     c = *p;
00603     *p = 0;
00604     if ( ( RetCode = OpenFile((char *)inp) ) < 0 ) {
00605         return(MesPrint("Cannot open file %s",inp));
00606     }
00607
00608     if ( SetFileIndex() ) {
00609         MesCall("CoLoad");
00610         SETERROR(-1)
00611     }
00612
00613     AO.SaveData.Handle = (WORD) (RetCode);
00614
00615     #ifdef SYSDEPENDENTSAVE
00616     if ( ReadFile(AO.SaveData.Handle, (UBYTE *) (&(AO.SaveData.Index)),
00617         (LONG)sizeof(struct FileInDeX) != (LONG)sizeof(struct FileInDeX) ) goto LoadRead;
00618     #else
00619     if ( ReadSaveHeader() ) goto LoadRead;
00620     TELLFILE(AO.SaveData.Handle,&firstposition);
00621     if ( ReadSaveIndex(&AO.SaveData.Index) ) goto LoadRead;
00622     #endif
00623     if ( c ) { /* There follows a list of expressions */
00624         *p++ = c;
00625         inp = p;
00626
00627         do { /* Scan the list */
00628             if ( !FG.cTable[*p] || *p == '[' ) {
00629                 p = SkipAName(p);
00630                 if ( p == 0 ) return(-1);
00631             }
00632             c = *p; *p = 0;
00633             if ( GetVar(inp,&type,&number,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
00634                 MesPrint("Conflicting name: %s",inp);
00635                 error = -1;
00636             }
00637             else {
00638                 if ( ( num = EntVar(CEXPRESSION,inp,STOREDEXPRESSION,0,0,0) ) >= 0 ) {
00639                     TMproto[0] = EXPRESSION;
00640                     TMproto[1] = SUBEXPSIZE;
00641                     TMproto[2] = num;
00642                     TMproto[3] = 1;
00643                     { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) TMproto[ie] = 0; }
00644                     AT.TMaddr = TMproto;
00645                     SeekFile(AO.SaveData.Handle,&firstposition,SEEK_SET);
00646                     AO.SaveData.Position = firstposition;
00647                     if ( ReadSaveIndex(&AO.SaveData.Index) ) goto LoadRead;
00648                     if ( ( ind = FindInIndex(num,&AO.SaveData,1,0) ) != 0 ) {
00649                         if ( !error ) {
00650                             if ( PutInStore(ind,num) ) error = -1;
00651                             else if ( !AM.silent && silentload == 0 )
00652                                 MesPrint(" %s loaded",ind->name);
00653                         }
00654                     }
00655                     /*
00656                     !!! Added 1-feb-1998
00657                     */
00657                     Expressions[num].counter = -1;
00658                 }
00659             else {
00660                 MesPrint(" %s not found",inp);
00661                 error = -1;

```

```

00662         }
00663     }
00664     else error = -1;
00665 }
00666 *p = c;
00667 if ( c != ',' && c ) {
00668     MesComp("Illegal character",inp,p);
00669     error = -1;
00670     goto EndLoad;
00671 }
00672 if ( c ) c = *++p;
00673 inp = p;
00674 } while ( c );
00675 scrpos = AR.StoreData.Position;
00676 SeekFile(AR.StoreData.Handle,&scrpos,SEEK_SET);
00677 if ( ISNOTEQUALPOS(scrpos,AR.StoreData.Position) ) goto LoadWrt;
00678 if ( WriteFile(AR.StoreData.Handle,(UBYTE *) (&(AR.StoreData.Index))
00679     ,(LONG)sizeof(struct FileInDeX)) != (LONG)sizeof(struct FileInDeX) ) goto LoadWrt;
00680 }
00681 else { /* All saved expressions should be stored. Easy */
00682     i = (WORD)BASEPOSITION(AO.SaveData.Index.number);
00683     ind = AO.SaveData.Index.expression;
00684 #ifndef SYSDEPENDENTSAVE
00685     if ( i > 0 ) { do {
00686         if ( GetVar((UBYTE *) (ind->name),&type,&number,ALLVARIABLES,NOAUTO) != NAMENOTFOUND ) {
00687             MesPrint("Conflicting name: %s",ind->name);
00688             error = -1;
00689         }
00690     } else {
00691         if ( ( num = EntVar(CEXPRESSION,(UBYTE *) (ind->name),STOREDEXPRESSION,0,0,0) ) >= 0 )
00692         {
00693             if ( !error ) {
00694                 if ( PutInStore(ind,num) ) error = -1;
00695                 else if ( !AM.silent && silentload == 0 )
00696                     MesPrint(" %s loaded",ind->name);
00697             }
00698             else error = -1;
00699         }
00700         i--;
00701         if ( i == 0 && ISNOTZEROPOS(AO.SaveData.Index.next) ) {
00702             SeekFile(AO.SaveData.Handle,&(AO.SaveData.Index.next),SEEK_SET);
00703             if ( ReadFile(AO.SaveData.Handle,(UBYTE *) (&(AO.SaveData.Index)),
00704                 (LONG)sizeof(struct FileInDeX)) != (LONG)sizeof(struct FileInDeX) ) goto LoadRead;
00705             i = (WORD)BASEPOSITION(AO.SaveData.Index.number);
00706             ind = AO.SaveData.Index.expression;
00707         }
00708         else ind++;
00709     } while ( i > 0 ); }
00710 #else
00711     if ( i > 0 ) {
00712         do {
00713             if ( GetVar((UBYTE *) (ind->name),&type,&number,ALLVARIABLES,NOAUTO) != NAMENOTFOUND )
00714             {
00715                 MesPrint("Conflicting name: %s",ind->name);
00716                 error = -1;
00717             }
00718             else {
00719                 if ( ( num = EntVar(CEXPRESSION,(UBYTE *) (ind->name),STOREDEXPRESSION,0,0,0) ) >=
00720                     0 ) {
00721                     if ( !error ) {
00722                         if ( PutInStore(ind,num) ) error = -1;
00723                         else if ( !AM.silent && silentload == 0 )
00724                             MesPrint(" %s loaded",ind->name);
00725                     }
00726                     else error = -1;
00727                 }
00728                 i--;
00729                 if ( i == 0 && (ISNOTZEROPOS(AO.SaveData.Index.next) || AO.bufferedInd) ) {
00730                     SeekFile(AO.SaveData.Handle,&(AO.SaveData.Index.next),SEEK_SET);
00731                     if ( ReadSaveIndex(&AO.SaveData.Index) ) goto LoadRead;
00732                     i = (WORD)BASEPOSITION(AO.SaveData.Index.number);
00733                     ind = AO.SaveData.Index.expression;
00734                 }
00735                 else ind++;
00736             } while ( i > 0 ); }
00737 #endif
00738 }
00739 EndLoad:
00740 #ifndef SYSDEPENDENTSAVE
00741     if ( AO.powerFlag ) {
00742         MesPrint("WARNING: min-/maxpower had to be adjusted!");
00743     }
00744     if ( AO.resizeFlag ) {
00745         MesPrint("ERROR: could not downsize data!");

```



```

00746         return ( -2 );
00747     }
00748 #endif
00749     CloseFile(AO.SaveData.Handle);
00750     AO.SaveData.Handle = -1;
00751     SeekFile(AR.StoreData.Handle,&(AC.StoreFileSize),SEEK_END);
00752     return(error);
00753 LoadWrt:
00754     MesPrint("WriteError");
00755     error = -1;
00756     goto EndLoad;
00757 LoadRead:
00758     MesPrint("ReadError");
00759     error = -1;
00760     goto EndLoad;
00761 }
00762
00763 /*
00764     #] CoLoad :
00765     #[ DeleteStore :
00766
00767     Routine deletes the contents of the entire storage file.
00768     We close the file and recreate it.
00769     If par > 0 we have to remove the expressions from the namelists.
00770 */
00771
00772 int DeleteStore(WORD par)
00773 {
00774     GETIDENTITY
00775     char *s;
00776     WORD j, n = 0;
00777     EXPRESSIONS e_in, e_out;
00778     WORD DidClean = 0;
00779     if ( AR.StoreData.Handle >= 0 ) {
00780         if ( par > 0 ) {
00781             n = NumExpressions;
00782             j = 0;
00783             e_in = e_out = Expressions;
00784             if ( n > 0 ) { do {
00785                 if ( e_in->status == STOREDEXPRESSION ) {
00786                     NAMENODE *node = GetNode(AC.exprnames,
00787                                             AC.exprnames->namebuffer+e_in->name);
00788                     node->type = CDELETE;
00789                     DidClean = 1;
00790                 }
00791                 else {
00792                     if ( e_out != e_in ) {
00793                         NAMENODE *node;
00794                         node = GetNode(AC.exprnames,
00795                                         AC.exprnames->namebuffer+e_in->name);
00796                         node->number = (WORD)(e_out - Expressions);
00797                         e_out->onfile = e_in->onfile;
00798                         e_out->prototype = e_in->prototype;
00799                         e_out->printfflag = 0;
00800                         e_out->status = e_in->status;
00801                         e_out->name = e_in->name;
00802                         e_out->inmem = e_in->inmem;
00803                         e_out->counter = e_in->counter;
00804                         e_out->numfactors = e_in->numfactors;
00805                         e_out->numdummies = e_in->numdummies;
00806                         e_out->compression = e_in->compression;
00807                         e_out->namesize = e_in->namesize;
00808                         e_out->whichbuffer = e_in->whichbuffer;
00809                         e_out->hidelevel = e_in->hidelevel;
00810                         e_out->node = e_in->node;
00811                         e_out->replace = e_in->replace;
00812                         e_out->vflags = e_in->vflags;
00813                         e_out->uflags = e_in->uflags;
00814 #ifdef PARALLELCODE
00815                         e_out->partodo = e_in->partodo;
00816 #endif
00817                     }
00818                     e_out++;
00819                     j++;
00820                 }
00821                 e_in++;
00822             } while ( --n > 0 ); }
00823             NumExpressions = j;
00824             if ( DidClean ) CompactifyTree(AC.exprnames,EXPRNAMES);
00825         }
00826         AR.StoreData.Handle = -1;
00827         CloseFile(AC.StoreHandle);
00828         AC.StoreHandle = -1;
00829     }
00830     /*
00831         Knock out the storage caches (25-apr-1990!)
00832     */

```

```

00833         STORECACHE st;
00834         st = (STORECACHE) (AT.StoreCache);
00835         while ( st ) {
00836             SETBASEPOSITION(st->position,-1);
00837             SETBASEPOSITION(st->toppos,-1);
00838             st = st->next;
00839         }
00840 #ifdef WITHPTHREADS
00841         for ( j = 1; j < AM.totalnumberofthreads; j++ ) {
00842             st = (STORECACHE) (AB[j]->T.StoreCache);
00843             while ( st ) {
00844                 SETBASEPOSITION(st->position,-1);
00845                 SETBASEPOSITION(st->toppos,-1);
00846                 st = st->next;
00847             }
00848         }
00849 #endif
00850     }
00851     PUTZERO(AC.StoreFileSize);
00852     s = FG.fname; while ( *s ) s++;
00853 #ifdef VMS
00854     *s = '/'; s[1] = '*'; s[2] = 0;
00855     remove(FG.fname);
00856     *s = 0;
00857 #endif
00858     return(AC.StoreHandle = CreateFile(FG.fname));
00859 }
00860 else return(0);
00861 }
00862
00863 /*
00864     #] DeleteStore :
00865     #[ PutInStore :
00866
00867     Copies the expression indicated by ind from a load file to the
00868     internal storage file. A return value of zero indicates that
00869     everything is OK.
00870 */
00871 */
00872
00873 int PutInStore(INDEXENTRY *ind, WORD num)
00874 {
00875     GETIDENTITY
00876     INDEXENTRY *newind;
00877     LONG wSize;
00878 #ifndef SYSDEPENDENTSAVE
00879     LONG wSizeOut;
00880     LONG stage;
00881 #endif
00882     POSITION scrpos,scrpos1;
00883     newind = NextFileIndex(&(Expressions[num].onfile));
00884     *newind = *ind;
00885 #ifndef SYSDEPENDENTSAVE
00886     SETBASEPOSITION(newind->length, 0);
00887 #endif
00888     newind->variables = AR.StoreData.Fill;
00889     SeekFile(AR.StoreData.Handle,&(newind->variables),SEEK_SET);
00890     if ( ISNOTEQUALPOS(newind->variables,AR.StoreData.Fill) ) goto PutErrS;
00891     newind->position = newind->variables;
00892 #ifdef SYSDEPENDENTSAVE
00893     ADDPOS(newind->position,DIFBASE(ind->position,ind->variables));
00894 #endif
00895     /* set read position to ind->variables */
00896     scrpos = ind->variables;
00897     SeekFile(AO.SaveData.Handle,&scrpos,SEEK_SET);
00898     if ( ISNOTEQUALPOS(scrpos,ind->variables) ) goto PutErrS;
00899     /* set max size for read-in */
00900     wSize = TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer);
00901 #ifdef SYSDEPENDENTSAVE
00902     scrpos = ind->length;
00903     ADDPOS(scrpos,DIFBASE(ind->position,ind->variables));
00904     ADD2POS(AR.StoreData.Fill,scrpos);
00905 #endif
00906     SETBASEPOSITION(scrpos1,wSize);
00907 #ifndef SYSDEPENDENTSAVE
00908     /* prepare look-up table for tensor functions */
00909     if ( ind->nfunctions ) {
00910         AO.tensorList = (UBYTE *)Malloc1(MAXSAVEFUNCTION,"PutInStore");
00911     }
00912     SETBASEPOSITION(scrpos, DIFBASE(ind->position,ind->variables));
00913     /* copy variables first */
00914     stage = -1;
00915     do {
00916         wSize = TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer);
00917         if ( ISLESSPOS(scrpos,scrpos1) ) wSize = BASEPOSITION(scrpos);
00918         wSizeOut = wSize;
00919         if ( ReadSaveVariables(

```

```

00920         (UBYTE *)AT.WorkPointer, (UBYTE *)AT.WorkTop, &wSize, &wSizeOut, ind, &stage) ) {
00921         goto PutErrS;
00922     }
00923     if ( WriteFile(AR.StoreData.Handle, (UBYTE *)AT.WorkPointer, wSizeOut)
00924         != wSizeOut ) goto PutErrS;
00925     ADDPOS(scrpos,-wSize);
00926     ADDPOS(newind->position, wSizeOut);
00927     ADDPOS(AR.StoreData.Fill, wSizeOut);
00928     } while ( ISPOSPOS(scrpos) );
00929     /* then copy the expression itself */
00930     wSize = TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer);
00931     scrpos = ind->length;
00932 #endif
00933     do {
00934         wSize = TOLONG(AT.WorkTop) - TOLONG(AT.WorkPointer);
00935         if ( ISLESSPOS(scrpos,scrpos1) ) wSize = BASEPOSITION(scrpos);
00936 #ifndef SYSDEPENDENTSAVE
00937         if ( ReadFile(AO.SaveData.Handle, (UBYTE *)AT.WorkPointer,wSize)
00938             != wSize ) goto PutErrS;
00939         if ( WriteFile(AR.StoreData.Handle, (UBYTE *)AT.WorkPointer,wSize)
00940             != wSize ) goto PutErrS;
00941         ADDPOS(scrpos,-wSize);
00942 #else
00943         wSizeOut = wSize;
00944         if ( ReadSaveExpression((UBYTE *)AT.WorkPointer, (UBYTE *)AT.WorkTop, &wSize, &wSizeOut) ) {
00945             goto PutErrS;
00946         }
00947         if ( WriteFile(AR.StoreData.Handle, (UBYTE *)AT.WorkPointer, wSizeOut)
00948             != wSizeOut ) goto PutErrS;
00949         ADDPOS(scrpos,-wSize);
00950         ADDPOS(AR.StoreData.Fill, wSizeOut);
00951         ADDPOS(newind->length, wSizeOut);
00952 #endif
00953     } while ( ISPOSPOS(scrpos) );
00954     /* free look-up table for tensor functions */
00955     if ( ind->nfunctions ) {
00956         M_free(AO.tensorList,"PutInStore");
00957     }
00958     scrpos = AR.StoreData.Position;
00959     SeekFile(AR.StoreData.Handle,&scrpos,SEEK_SET);
00960     if ( ISNOTEQUALPOS(scrpos,AR.StoreData.Position) ) goto PutErrS;
00961     if ( WriteFile(AR.StoreData.Handle, (UBYTE *)(&AR.StoreData.Index), (LONG)sizeof(FILEINDEX))
00962         == (LONG)sizeof(FILEINDEX) ) return(0);
00963 PutErrS:
00964     return(MesPrint("File error"));
00965 }
00966
00967 /*
00968     #] PutInStore :
00969     #[ GetTerm :
00970
00971     Gets one term from input scratch stream.
00972     Puts it in 'term'.
00973     Returns the length of the term.
00974
00975     Used by Processor      (proces.c)
00976     WriteAll              (sch.c)
00977     WriteOne              (sch.c)
00978     GetMoreTerms          (store.c)
00979     ToStorage              (store.c)
00980     CoFillExpression      (comexpr.c)
00981     FactorInExpr          (factor.c)
00982     LoadOpti              (optim.c)
00983     PF_Processor          (parallel.c)
00984     ThreadsProcessor      (threads.c)
00985
00986     In multi thread/processor mode all calls are done by the master.
00987     Note however that other routines, used by the threads, can use
00988     the same file. Hence we need to be careful about SeekFile and locks.
00989 */
00990
00991 WORD GetTerm(PHEAD WORD *term)
00992 {
00993     GETBIDENTITY
00994     WORD *inp, i, j = 0, len;
00995     LONG InIn, *whichInInBuf;
00996     WORD *r, *m, *mstop = 0, minsiz = 0, *bra = 0, *from;
00997     WORD first, *start = 0, testing = 0;
00998     FILEHANDLE *fi;
00999     AN.deferskipped = 0;
01000     if ( AR.GetFile == 2 ) {
01001         fi = AR.hidefile;
01002         whichInInBuf = &(AR.InHiBuf);
01003     }
01004     else {
01005         fi = AR.infile;

```

```

01007         whichInInBuf = &(AR.InInBuf);
01008     }
01009     InIn = *whichInInBuf;
01010     from = term;
01011     if ( AR.KeptInHold ) {
01012         r = AR.CompressBuffer;
01013         i = *r;
01014         AR.KeptInHold = 0;
01015         if ( i <= 0 ) { *term = 0; goto RegRet; }
01016         m = term;
01017         NCOPY(m,r,i);
01018         goto RegRet;
01019     }
01020     if ( AR.DeferFlag ) {
01021         m = AR.CompressBuffer;
01022         if ( *m > 0 ) {
01023             mstop = m + *m;
01024             mstop -= ABS(mstop[-1]);
01025             m++;
01026             while ( m < mstop ) {
01027                 if ( *m == HAAKJE ) {
01028                     testing = 1;
01029                     mstop = m + m[1];
01030                     bra = (WORD *)(((UBYTE *) (term)) + 2*AM.MaxTer);
01031                     m = AR.CompressBuffer+1;
01032                     r = bra;
01033                     while ( m < mstop ) *r++ = *m++;
01034                     mstop = r;
01035                     minsiz = WORDDIF(mstop,bra);
01036                     goto ReStart;
01037                 /*
01038                     We have the bracket to be tested in bra till mstop
01039                 */
01040             }
01041             m += m[1];
01042         }
01043     }
01044     bra = (WORD *)(((UBYTE *) (term)) + 2*AM.MaxTer);
01045     mstop = bra+1;
01046     *bra = 0;
01047     minsiz = 1;
01048     testing = 1;
01049 }
01050 ReStart:
01051     first = 0;
01052     r = AR.CompressBuffer;
01053     if ( fi->handle >= 0 ) {
01054         if ( InIn <= 0 ) {
01055             ADDPOS(fi->POposition, (fi->POfull-fi->PObuffer)*sizeof(WORD));
01056             LOCK(AS.inputslock);
01057             SeekFile(fi->handle, &(fi->POposition), SEEK_SET);
01058             InIn = ReadFile(fi->handle, (UBYTE *) (fi->PObuffer), fi->POsize);
01059             UNLOCK(AS.inputslock);
01060             if ( ( InIn < 0 ) || ( InIn & 1 ) ) {
01061                 goto GTerr;
01062             }
01063 #ifdef WORD2
01064             InIn >>= 1;
01065 #else
01066             InIn /= TABLESIZE(WORD,UBYTE);
01067 #endif
01068             *whichInInBuf = InIn;
01069             if ( !InIn ) { *r = 0; *from = 0; goto RegRet; }
01070             fi->POfill = fi->PObuffer;
01071             fi->POfull = fi->PObuffer + InIn;
01072         }
01073         inp = fi->POfill;
01074         if ( ( len = i = *inp ) == 0 ) {
01075             (*whichInInBuf)--;
01076             (fi->POfill)++;
01077             *r = 0;
01078             *from = 0;
01079             goto RegRet;
01080         }
01081         if ( i < 0 ) {
01082             InIn--;
01083             inp++;
01084             r++;
01085             start = term;
01086             *term++ = -i + 1;
01087             while ( ++i <= 0 ) *term++ = *r++;
01088             if ( InIn > 0 ) {
01089                 i = *inp++;
01090                 InIn--;
01091                 *start += i;
01092                 *(AR.CompressBuffer) = len = *start;
01093             }

```

```

01094         else {
01095             first = 1;
01096             goto NewIn;
01097         }
01098     }
01099     InIn -= i;
01100     if ( InIn < 0 ) {
01101         j = (WORD) (- InIn);
01102         i -= j;
01103     }
01104     else j = 0;
01105     while ( --i >= 0 ) {
01106         *r++ = *term++ = *inp++;
01107     }
01108     if ( j ) {
01109 NewIn:
01110         ADDPOS(fi->PPosition, (fi->POfull-fi->PObuffer)*sizeof(WORD));
01111         LOCK(AS.inputslock);
01112         SeekFile(fi->handle, &(fi->PPosition), SEEK_SET);
01113         InIn = ReadFile(fi->handle, (UBYTE *) (fi->PObuffer), fi->POsize);
01114         UNLOCK(AS.inputslock);
01115         if ( ( InIn <= 0 ) || ( InIn & 1 ) ) {
01116             goto GTerr;
01117         }
01118 #ifdef WORD2
01119         InIn >= 1;
01120 #else
01121         InIn /= TABLESIZE(WORD, UBYTE);
01122 #endif
01123         inp = fi->PObuffer;
01124         fi->POfull = inp + InIn;
01125
01126         if ( first ) {
01127             j = *inp++;
01128             InIn--;
01129             *start += j;
01130             *(AR.CompressBuffer) = len = *start;
01131         }
01132         InIn -= j;
01133         while ( --j >= 0 ) { *r++ = *term++ = *inp++; }
01134     }
01135     fi->POfill = inp;
01136     *whichInInBuf = InIn;
01137     AR.DefPosition = fi->PPosition;
01138     ADDPOS(AR.DefPosition, ((UBYTE *) (fi->POfill) - (UBYTE *) (fi->PObuffer)));
01139 }
01140 else {
01141     inp = fi->POfill;
01142     if ( inp >= fi->POfull ) { *from = 0; goto RegRet; }
01143     len = j = *inp;
01144     if ( j < 0 ) {
01145         inp++;
01146         *term++ = *r++ = len = - j + 1 + *inp;
01147         while ( ++j <= 0 ) *term++ = *r++;
01148         j = *inp++;
01149     }
01150     else if ( !j ) j = 1;
01151     while ( --j >= 0 ) { *r++ = *term++ = *inp++; }
01152     fi->POfill = inp;
01153 /*%%%%ADDED 7-apr-2006 for Keep Brackets in bucket */
01154     SETBASEPOSITION(AR.DefPosition, ((UBYTE *) (fi->POfill) - (UBYTE *) (fi->PObuffer)));
01155     if ( inp > fi->POfull ) {
01156         goto GTerr;
01157     }
01158 }
01159 if ( r >= AR.ComprTop ) {
01160     MesPrint("CompressSize of %10l is insufficient", AM.CompressSize);
01161     Terminate(-1);
01162 }
01163 AR.CompressPointer = r; *r = 0;
01164 /*
01165     The next *from is a bug fix that made the program read in forbidden
01166     territory.
01167 */
01168 if ( testing && *from != 0 ) {
01169     WORD jj;
01170     r = from;
01171     jj = *r - 1 - ABS(*(r+r-1));
01172     if ( jj < minsiz ) goto strip;
01173     r++;
01174     m = bra;
01175     while ( m < mstop ) {
01176         if ( *m != *r ) {
01177 strip:
01178             r = from;
01179             m = r + *r;
01180             mstop = m - ABS(m[-1]);
01181             r++;

```

```

01181         while ( r < mstop ) {
01182             if ( *r == HAAKJE ) {
01183                 *r++ = 1;
01184                 *r++ = 1;
01185                 *r++ = 3;
01186                 len = WORDDIF(r,from);
01187                 *from = len;
01188                 goto RegRet;
01189             }
01190             r += r[1];
01191         }
01192         goto RegRet;
01193     }
01194     m++;
01195     r++;
01196 }
01197 term = from;
01198 AN.deferskipped++;
01199 goto ReStart;
01200 }
01201 RegRet;;
01202 /*
01203     #] debug :
01204 */
01205 {
01206     UBYTE OutBuf[140];
01207     /* if ( AP.DebugFlag ) { */
01208     if ( ( AP.PreDebug & DUMPINTERMS ) == DUMPINTERMS ) {
01209         MLOCK(ErrorMessageLock);
01210         AO.OutFill = AO.OutputLine = OutBuf;
01211         AO.OutSkip = 3;
01212         FiniLine();
01213         r = from;
01214         i = *r;
01215         TokenToLine((UBYTE *) ("Input: "));
01216         if ( i == 0 ) {
01217             TokenToLine((UBYTE *) "zero");
01218         }
01219         else if ( i < 0 ) {
01220             TokenToLine((UBYTE *) "negative!!");
01221         }
01222         else {
01223             while ( --i >= 0 ) {
01224                 TalToLine((UWORD) (*r++)); TokenToLine((UBYTE *) " ");
01225             }
01226         }
01227         FiniLine();
01228         MUNLOCK(ErrorMessageLock);
01229     }
01230 }
01231 /*
01232     #] debug :
01233 */
01234     return(*from);
01235 GTerr:
01236     MesPrint("Error while reading scratch file in GetTerm");
01237     Terminate(-1);
01238     return(-1);
01239 }
01240
01241 /*
01242     #] GetTerm :
01243     #] GetOneTerm :
01244
01245     Gets one term from stream AR.infile->handle.
01246     Puts it in 'term'.
01247     Returns the length of the term.
01248     Input is unbuffered.
01249     Compression via AR.CompressPointer
01250     par is actually in all calls a file handle
01251
01252     Routine is called from
01253         DoOnePow          Get one power of an expression
01254         Deferred          Get the contents of a bracket
01255         GetFirstBracket
01256         FindBracket
01257
01258     We should do something about the lack of buffering.
01259     Maybe a buffer of a few times AM.MaxTer (MaxTermSize*sizeof(WORD)).
01260     Each thread will need its own buffer!
01261
01262     If par == 0 we use ReadPosFile which can fill the whole buffer.
01263     If par == 1 we use ReadFile and do actual read operations.
01264
01265     Note: we cannot use ReadPosFile when running in the master thread.
01266 */
01267 WORD GetOneTerm(PHEAD WORD *term, FILEHANDLE *fi, POSITION *pos, int par)

```

```

01268 {
01269     GETBIDENTITY
01270     WORD i, *p;
01271     LONG j, siz;
01272     WORD *r, *rr = AR.CompressPointer;
01273     int error = 0;
01274     r = rr;
01275     if ( fi->handle >= 0 ) {
01276 #ifdef READONEBYONE
01277 #ifdef WITHPTHREADS
01278 /*
01279     This code needs some investigation.
01280     It may be that we should do this always.
01281     It may be that even for workers it is no good.
01282     We may have to make a variable like AM.ReadDirect with
01283     if ( AM.ReadDirect ) par = 1;
01284     and a user command like
01285     On ReadDirect;
01286 */
01287     if ( AT.identity > 0 ) par = 1;
01288 #endif
01289 #endif
01290 /*
01291     To be changed:
01292     1: check first whether the term lies completely inside the buffer
01293     2: if not a: use old strategy for AT.identity == 0 (master)
01294         b: for workers, position file and read buffer
01295 */
01296     if ( par == 0 ) {
01297         siz = ReadPosFile(BHEAD fi, (UBYTE *)term, 1L, pos);
01298     }
01299     else {
01300         LOCK(AS.inputslock);
01301         SeekFile(fi->handle, pos, SEEK_SET);
01302         siz = ReadFile(fi->handle, (UBYTE *)term, sizeof(WORD));
01303         UNLOCK(AS.inputslock);
01304         ADDPOS(*pos, siz);
01305     }
01306     if ( siz == sizeof(WORD) ) {
01307         p = term;
01308         j = i = *term++;
01309         if ( ( i > AM.MaxTer/((WORD)sizeof(WORD)) ) || ( -i >= AM.MaxTer/((WORD)sizeof(WORD)) ) )
01310         {
01311             error = 1;
01312             goto ErrGet;
01313         }
01314         r++;
01315         if ( i < 0 ) { /* Loading a compressed term */
01316             *p = -i + 1;
01317             while ( ++i <= 0 ) *term++ = *r++;
01318             if ( par == 0 ) {
01319                 siz = ReadPosFile(BHEAD fi, (UBYTE *)term, 1L, pos);
01320             }
01321             else {
01322                 LOCK(AS.inputslock);
01323                 SeekFile(fi->handle, pos, SEEK_SET);
01324                 siz = ReadFile(fi->handle, (UBYTE *)term, sizeof(WORD));
01325                 UNLOCK(AS.inputslock);
01326                 ADDPOS(*pos, sizeof(WORD));
01327             }
01328             if ( siz != sizeof(WORD) ) {
01329                 error = 2;
01330                 goto ErrGet;
01331             }
01332             *p += *term;
01333             j = *term;
01334             if ( ( j > AM.MaxTer/((WORD)sizeof(WORD)) ) || ( j <= 0 ) )
01335             {
01336                 error = 3;
01337                 goto ErrGet;
01338             }
01339             *rr = *p; /* Write the proper term size to *AR.CompressPointer */
01340         }
01341         else { /* Loading a regular term */
01342             if ( !j ) return(0);
01343             *rr = i; /* Write the proper term size to *AR.CompressPointer */
01344             j--;
01345         }
01346         i = (WORD)j;
01347         if ( par == 0 ) {
01348             siz = ReadPosFile(BHEAD fi, (UBYTE *)term, j, pos);
01349             j *= TABLESIZE(WORD, UBYTE);
01350         }
01351         else {
01352             j *= TABLESIZE(WORD, UBYTE);
01353             LOCK(AS.inputslock);
01354             SeekFile(fi->handle, pos, SEEK_SET);

```

```

01355         siz = ReadFile(fi->handle, (UBYTE *)term, j);
01356         UNLOCK(AS.inputslock);
01357         ADDPOS(*pos, j);
01358     }
01359     if ( siz != j ) {
01360         error = 4;
01361         goto ErrGet;
01362     }
01363     while ( --i >= 0 ) *r++ = *term++;
01364     if ( r >= AR.ComprTop ) {
01365         MLOCK(ErrorMessageLock);
01366         MesPrint("CompressSize of %10l is insufficient", AM.CompressSize);
01367         MUNLOCK(ErrorMessageLock);
01368         Terminate(-1);
01369     }
01370     AR.CompressPointer = r; *r = 0;
01371     return(*p);
01372 }
01373 error = 5;
01374 }
01375 else {
01376 /*
01377     Here the whole expression is in the buffer.
01378 */
01379     fi->POfill = (WORD *)((UBYTE *) (fi->PObuffer) + BASEPOSITION(*pos));
01380     p = fi->POfill;
01381     if ( p >= fi->POfull ) { *term = 0; return(0); }
01382     j = i = *p;
01383     if ( i < 0 ) {
01384         p++;
01385         j = *r++ = *term++ = -i + 1 + *p;
01386         while ( ++i <= 0 ) *term++ = *r++;
01387         i = *p++;
01388     }
01389     if ( i == 0 ) { i = 1; *r++ = 0; *term++ = 0; }
01390     else { while ( --i >= 0 ) { *r++ = *term++ = *p++; } }
01391     fi->POfill = p;
01392     SETBASEPOSITION(*pos, (UBYTE *) (fi->POfill) - (UBYTE *) (fi->PObuffer));
01393     if ( p <= fi->POfull ) {
01394         if ( r >= AR.ComprTop ) {
01395             MLOCK(ErrorMessageLock);
01396             MesPrint("CompressSize of %10l is insufficient", AM.CompressSize);
01397             MUNLOCK(ErrorMessageLock);
01398             Terminate(-1);
01399         }
01400         AR.CompressPointer = r; *r = 0;
01401         return((WORD)j);
01402     }
01403     error = 6;
01404 }
01405 ErrGet:
01406     MLOCK(ErrorMessageLock);
01407     MesPrint("Error while reading scratch file in GetOneTerm (%d)", error);
01408     MUNLOCK(ErrorMessageLock);
01409     Terminate(-1);
01410     return(-1);
01411 }
01412
01413 /*
01414     #] GetOneTerm :
01415     #[ GetMoreTerms :
01416     Routine collects more contents of brackets inside a function,
01417     indicated by the number in AC.CollectFun.
01418     The first term is in term already.
01419     We can keep calling GetTerm either till a bracket is finished
01420     or till it would make the term too long (> AM.MaxTer/2)
01421     In all cases this function makes that the routine GetTerm
01422     has a term in 'hold', so the AR.KeptInHold flag must be turned on.
01423 */
01424
01425 WORD GetMoreTerms(WORD *term)
01426 {
01427     GETIDENTITY
01428     WORD *t, *r, *m, *h, *tstop, i, inc, same;
01429     WORD extra;
01430     WORD retval = 0;
01431 /*
01432     We use 23% as a quasi-random default value.
01433 */
01434     extra = ((AM.MaxTer/sizeof(WORD)) * ((LONG)100-AC.CollectPercentage))/100;
01435     if ( extra < 23 ) extra = 23;
01436 /*
01437     First find the bracket pointer
01438 */
01439     t = term + *term;
01440     tstop = t - ABS(t[-1]);
01441     h = term+1;

```



```

01442     while ( *h != HAAKJE && h < tstop ) h += h[1];
01443     if ( h >= tstop ) return(retval);
01444     inc = FUNHEAD+ARGHEAD+1-h[1];
01445     same = WORDDIFF(h,term) + h[1] - 1;
01446     r = m + t + inc;
01447     tstop = h + h[1];
01448     while ( t > tstop ) *--r = *--t;
01449     r--;
01450     *r = WORDDIFF(m,r);
01451     while ( GetTerm(BHEAD m) > 0 ) {
01452         r = m + 1;
01453         t = m + *m - 1;
01454         if ( same > ( i = ( *m - ABS(*t) - 1 ) ) ) { /* Must fail */
01455             if ( AC.AltCollectFun && AS.CollectOverFlag == 2 ) AS.CollectOverFlag = 3;
01456             break;
01457         }
01458         t = term+1;
01459         i = same;
01460         while ( --i >= 0 ) {
01461             if ( *r != *t ) {
01462                 if ( AC.AltCollectFun && AS.CollectOverFlag == 2 ) AS.CollectOverFlag = 3;
01463                 goto FullTerm;
01464             }
01465             r++; t++;
01466         }
01467         if ( ( WORDDIFF(m,term) + i + extra ) > (WORD)(AM.MaxTer/sizeof(WORD)) ) {
01468             /* 23 = 3 +20. The 20 is to have some extra for substitutions or whatever */
01469             if ( AS.CollectOverFlag == 0 && AC.AltCollectFun == 0 ) {
01470                 Warning("Bracket contents too long in Collect statement");
01471                 Warning("Contents spread over more than one term");
01472                 Warning("If possible: increase MaxTermSize in setfile");
01473                 AS.CollectOverFlag = 1;
01474             }
01475             else if ( AC.AltCollectFun ) {
01476                 AS.CollectOverFlag = 2;
01477             }
01478             break;
01479         }
01480         tstop = m + *m;
01481         *m -= same;
01482         m++;
01483         while ( r < tstop ) *m++ = *r++;
01484         retval++;
01485         if ( extra == 23 ) extra = ((AM.MaxTer/sizeof(WORD))/6);
01486     }
01487 FullTerm:
01488     h[1] = WORDDIFF(m,h);
01489     if ( AS.CollectOverFlag > 1 ) {
01490         *h = AC.AltCollectFun;
01491         if ( AS.CollectOverFlag == 3 ) AS.CollectOverFlag = 1;
01492     }
01493     else *h = AC.CollectFun;
01494     h[2] |= DIRTYFLAG;
01495     h[FUNHEAD] = h[1] - FUNHEAD;
01496     h[FUNHEAD+1] = 0;
01497     if ( ToFast(h+FUNHEAD,h+FUNHEAD) ) {
01498         if ( h[FUNHEAD] <= -FUNCTION ) {
01499             h[1] = FUNHEAD+1;
01500             m = h + FUNHEAD+1;
01501         }
01502         else {
01503             h[1] = FUNHEAD+2;
01504             m = h + FUNHEAD+2;
01505         }
01506     }
01507     *m++ = 1;
01508     *m++ = 1;
01509     *m++ = 3;
01510     *term = WORDDIFF(m,term);
01511     AR.KeptInHold = 1;
01512     return(retval);
01513 }
01514
01515 /*
01516     #] GetMoreTerms :
01517     #[ GetMoreFromMem :
01518
01519 */
01520
01521 int GetMoreFromMem(WORD *term, WORD **tpoin)
01522 {
01523     GETIDENTITY
01524     WORD *t, *r, *m, *h, *tstop, i, j, inc, same;
01525     LONG extra = 23;
01526 /*
01527     First find the bracket pointer
01528 */

```

```

01529     t = term + *term;
01530     tstop = t - ABS(t[-1]);
01531     h = term+1;
01532     while ( *h != HAAKJE && h < tstop ) h += h[1];
01533     if ( h >= tstop ) return(0);
01534     inc = FUNHEAD+ARGHEAD+1-h[1];
01535     same = WORDDIF(h,term) + h[1] - 1;
01536     r = m = t + inc;
01537     tstop = h + h[1];
01538     while ( t > tstop ) *--r = *--t;
01539     r--;
01540     *r = WORDDIF(m,r);
01541     while ( **tpoin ) {
01542         r = *tpoin; j = *r;
01543         for ( i = 0; i < j; i++ ) m[i] = *r++;
01544         *tpoin = r;
01545         r = m + 1;
01546         t = m + *m - 1;
01547         if ( same > ( i = ( *m - ABS(*t) - 1 ) ) ) { /* Must fail */
01548             if ( AC.AltCollectFun && AS.CollectOverFlag == 2 ) AS.CollectOverFlag = 3;
01549             break;
01550         }
01551         t = term+1;
01552         i = same;
01553         while ( --i >= 0 ) {
01554             if ( *r != *t ) {
01555                 if ( AC.AltCollectFun && AS.CollectOverFlag == 2 ) AS.CollectOverFlag = 3;
01556                 goto FullTerm;
01557             }
01558             r++; t++;
01559         }
01560         if ( ( WORDDIF(m,term) + i + extra ) > (LONG)(AM.MaxTer/(2*sizeof(WORD))) ) {
01561             /* 23 = 3 +20. The 20 is to have some extra for substitutions or whatever */
01562             if ( AS.CollectOverFlag == 0 && AC.AltCollectFun == 0 ) {
01563                 Warning("Bracket contents too long in Collect statement");
01564                 Warning("Contents spread over more than one term");
01565                 Warning("If possible: increase MaxTermSize in setfile");
01566                 AS.CollectOverFlag = 1;
01567             }
01568             else if ( AC.AltCollectFun ) {
01569                 AS.CollectOverFlag = 2;
01570             }
01571             break;
01572         }
01573         tstop = m + *m;
01574         *m -= same;
01575         m++;
01576         while ( r < tstop ) *m++ = *r++;
01577         if ( extra == 23 ) extra = ((AM.MaxTer/sizeof(WORD))/6);
01578     }
01579 FullTerm:
01580     h[1] = WORDDIF(m,h);
01581     if ( AS.CollectOverFlag > 1 ) {
01582         *h = AC.AltCollectFun;
01583         if ( AS.CollectOverFlag == 3 ) AS.CollectOverFlag = 1;
01584     }
01585     else *h = AC.CollectFun;
01586     h[2] |= DIRTYFLAG;
01587     h[FUNHEAD] = h[1] - FUNHEAD;
01588     h[FUNHEAD+1] = 0;
01589     if ( ToFast(h+FUNHEAD,h+FUNHEAD) ) {
01590         if ( h[FUNHEAD] <= -FUNCTION ) {
01591             h[1] = FUNHEAD+1;
01592             m = h + FUNHEAD+1;
01593         }
01594         else {
01595             h[1] = FUNHEAD+2;
01596             m = h + FUNHEAD+2;
01597         }
01598     }
01599     *m++ = 1;
01600     *m++ = 1;
01601     *m++ = 3;
01602     *term = WORDDIF(m,term);
01603     AR.KeptInHold = 1;
01604     return(0);
01605 }
01606
01607 /*
01608     #] GetMoreFromMem :
01609     #[ GetFromStore :
01610
01611     Gets a single term from the storage file at position and puts
01612     it at 'to'.
01613     The value to be returned is the number of words read.
01614     Renumbering is done also.
01615     This is controlled by the renumber table, given in 'renumber'

```

```

01616
01617     This routine should work with a number of cache buffers. The
01618     exact number should be definable in form.set.
01619     The parameters are:
01620     AM.SizeStoreCache    (4096)
01621     The numbers are the proposed default values.
01622
01623     The cache is a pure read cache.
01624 */
01625
01626 static int gfs = 0;
01627
01628 WORD GetFromStore(WORD *to, POSITION *position, RENUMBER renumber, WORD *InCompState, WORD nexpr)
01629 {
01630     GETIDENTITY
01631     LONG RetCode, num, first = 0;
01632     WORD *from, *m;
01633     struct StOrEcAcHe sc;
01634     STORECACHE s;
01635     STORECACHE snext, sold;
01636     WORD *r, *rr = AR.CompressPointer;
01637     r = rr;
01638     gfs++;
01639     sc.next = AT.StoreCache;
01640     sold = s = &sc;
01641     snext = s->next;
01642     while ( snext ) {
01643         sold = s;
01644         s = snext;
01645         snext = s->next;
01646         if ( BASEPOSITION(s->position) == -1 ) break;
01647         if ( ISLESSPOS(*position,s->toppos) &&
01648             ISGEPOS(*position,s->position) ) { /* Hit */
01649             if ( AT.StoreCache != s ) {
01650                 sold->next = s->next;
01651                 s->next = AT.StoreCache->next;
01652                 AT.StoreCache = s;
01653             }
01654             from = (WORD *) ((UBYTE *) (s->buffer)) + DIFBASE(*position,s->position));
01655             num = *from;
01656             if ( !num ) { return(*to = 0); }
01657             *InCompState = (WORD) num;
01658             m = to;
01659             if ( num < 0 ) {
01660                 from++;
01661                 ADDPOS(*position,sizeof(WORD));
01662                 *m++ = (WORD) (-num+1);
01663                 r++;
01664                 while ( ++num <= 0 ) *m++ = *r++;
01665                 if ( ISLESSPOS(*position,s->toppos) ) {
01666                     num = *from++;
01667                     *to += (WORD) num;
01668                     ADDPOS(*position,sizeof(WORD));
01669                     *InCompState = (WORD) (num + 2);
01670                 }
01671                 else {
01672                     first = 1;
01673                     goto InNew;
01674                 }
01675             }
01676             PastCon:;
01677             while ( num > 0 && ISLESSPOS(*position,s->toppos) ) {
01678                 *r++ = *m++ = *from++; ADDPOS(*position,sizeof(WORD)); num--;
01679             }
01680             if ( num > 0 ) {
01681 InNew:
01682                 SETBASEPOSITION(s->position,-1);
01683                 SETBASEPOSITION(s->toppos,-1);
01684                 LOCK(AM.storefilelock);
01685                 SeekFile(AR.StoreData.Handle,position,SEEK_SET);
01686                 RetCode = ReadFile(AR.StoreData.Handle,(UBYTE *) (s->buffer),AM.SizeStoreCache);
01687                 UNLOCK(AM.storefilelock);
01688                 if ( RetCode < 0 ) goto PastErr;
01689                 if ( !RetCode ) return(*to = 0 );
01690                 s->position = *position;
01691                 s->toppos = *position;
01692                 ADDPOS(s->toppos,RetCode);
01693                 from = s->buffer;
01694                 if ( first ) {
01695                     num = *from++;
01696                     ADDPOS(*position,sizeof(WORD));
01697                     *to += (WORD) num;
01698                     /* This next line has always been commented, but uncommenting it
01699                     fixes a rare bug when loading certain save files. */
01700                     first = 0;
01701                     *InCompState = (WORD) (num + 2);
01702                 }

```

```

01703         goto PastCon;
01704     }
01705     goto PastEnd;
01706 }
01707 }
01708 if ( AT.StoreCache ) { /* Fill the last buffer */
01709     s->position = *position;
01710     LOCK(AM.storefilelock);
01711     SeekFile(AR.StoreData.Handle,position,SEEK_SET);
01712     RetCode = ReadFile(AR.StoreData.Handle,(UBYTE *) (s->buffer),AM.SizeStoreCache);
01713     UNLOCK(AM.storefilelock);
01714     if ( RetCode < 0 ) goto PastErr;
01715     if ( !RetCode ) return( *to = 0 );
01716     s->toppos = *position;
01717     ADDPOS(s->toppos,RetCode);
01718     if ( AT.StoreCache != s ) {
01719         sold->next = s->next;
01720         s->next = AT.StoreCache->next;
01721         AT.StoreCache = s;
01722     }
01723     m = to;
01724     from = s->buffer;
01725     num = *from;
01726     if ( !num ) { return( *to = 0 ); }
01727     *InCompState = (WORD)num;
01728     if ( num < 0 ) {
01729         *m++ = (WORD) (-num+1);
01730         r++;
01731         from++;
01732         ADDPOS(*position,sizeof(WORD));
01733         while ( ++num <= 0 ) *m++ = *r++;
01734         num = *from++;
01735         *to += (WORD)num;
01736         ADDPOS(*position,sizeof(WORD));
01737         *InCompState = (WORD) (num+2);
01738     }
01739     goto PastCon;
01740 }
01741 /* No caching available */
01742 LOCK(AM.storefilelock);
01743 SeekFile(AR.StoreData.Handle,position,SEEK_SET);
01744 RetCode = ReadFile(AR.StoreData.Handle,(UBYTE *)to,(LONG)sizeof(WORD));
01745 SeekFile(AR.StoreData.Handle,position,SEEK_CUR);
01746 UNLOCK(AM.storefilelock);
01747 if ( RetCode != sizeof(WORD) ) {
01748     *to = 0;
01749     return( (WORD)RetCode);
01750 }
01751 if ( !*to ) return(0);
01752 m = to;
01753 if ( *to < 0 ) {
01754     num = *m++;
01755     *to = *r++ = (WORD) (-num + 1);
01756     while ( ++num <= 0 ) *m++ = *r++;
01757     LOCK(AM.storefilelock);
01758     SeekFile(AR.StoreData.Handle,position,SEEK_SET);
01759     RetCode = ReadFile(AR.StoreData.Handle,(UBYTE *)m,(LONG)sizeof(WORD));
01760     SeekFile(AR.StoreData.Handle,position,SEEK_CUR);
01761     UNLOCK(AM.storefilelock);
01762     if ( RetCode != sizeof(WORD) ) {
01763         MLOCK(ErrorMessageLock);
01764         MesPrint("@Error in compression of store file");
01765         MUNLOCK(ErrorMessageLock);
01766         return(-1);
01767     }
01768     num = *m;
01769     *to += (WORD)num;
01770     *InCompState = (WORD) (num + 2);
01771 }
01772 else {
01773     *InCompState = *to;
01774     num = *to - 1; m = to + 1; r = rr + 1;
01775 }
01776 first = num;
01777 num *= wsizeof(WORD);
01778 if ( num < 0 ) {
01779     MLOCK(ErrorMessageLock);
01780     MesPrint("@Error in stored expressions file at position %9p",position);
01781     MUNLOCK(ErrorMessageLock);
01782     return(-1);
01783 }
01784 LOCK(AM.storefilelock);
01785 SeekFile(AR.StoreData.Handle,position,SEEK_SET);
01786 RetCode = ReadFile(AR.StoreData.Handle,(UBYTE *)m,num);
01787 SeekFile(AR.StoreData.Handle,position,SEEK_CUR);
01788 UNLOCK(AM.storefilelock);
01789 if ( RetCode != num ) {

```

```

01790         MLOCK(ErrorMessageLock);
01791         MesPrint("@Error in stored expressions file at position %9p",position);
01792         MUNLOCK(ErrorMessageLock);
01793         return(-1);
01794     }
01795     NCOPY(r,m,first);
01796 PastEnd:
01797     *rr = *to;
01798     if ( r >= AR.ComprTop ) {
01799         MLOCK(ErrorMessageLock);
01800         MesPrint("CompressSize of %10l is insufficient",AM.CompressSize);
01801         MUNLOCK(ErrorMessageLock);
01802         Terminate(-1);
01803     }
01804     AR.CompressPointer = r; *r = 0;
01805     if ( !TermRenumber(to,renumber,nexpr) ) {
01806         MarkDirty(to,DIRTYSYMFLAG);
01807         if ( AR.CurDum > AM.IndDum && Expressions[nexpr].numdummies > 0 )
01808             MoveDummies(BHEAD to,AR.CurDum - AM.IndDum);
01809         return((WORD)*to);
01810     }
01811 PastErr:
01812     MLOCK(ErrorMessageLock);
01813     MesCall("GetFromStore");
01814     MUNLOCK(ErrorMessageLock);
01815     SETERROR(-1)
01816 }
01817
01818 /*
01819     #] GetFromStore :
01820     #[ DetVars :          void DetVars(term)
01821
01822     Determines which variables are used in term.
01823
01824     When par = 1 we are scanning a prototype expression which involves
01825     completely different rules.
01826
01827 */
01828
01829 void DetVars(WORD *term, WORD par)
01830 {
01831     GETIDENTITY
01832     WORD *stopper;
01833     WORD *t, sym;
01834     WORD *sarg;
01835     stopper = term + *term - 1;
01836     stopper = stopper - ABS(*stopper) + 1;
01837     term++;
01838     if ( par ) {          /* Prototype expression */
01839         WORD n;
01840         if ( ( n = NumSymbols ) > 0 ) {
01841             SYMBOLS tt;
01842             tt = symbols;
01843             do {
01844                 (tt++)->flags &= ~INUSE;
01845             } while ( --n > 0 );
01846         }
01847         if ( ( n = NumIndices ) > 0 ) {
01848             INDICES tt;
01849             tt = indices;
01850             do {
01851                 (tt++)->flags &= ~INUSE;
01852             } while ( --n > 0 );
01853         }
01854         if ( ( n = NumVectors ) > 0 ) {
01855             VECTORS tt;
01856             tt = vectors;
01857             do {
01858                 (tt++)->flags &= ~INUSE;
01859             } while ( --n > 0 );
01860         }
01861         if ( ( n = NumFunctions ) > 0 ) {
01862             FUNCTIONS tt;
01863             tt = functions;
01864             do {
01865                 (tt++)->flags &= ~INUSE;
01866             } while ( --n > 0 );
01867         }
01868         term += SUBEXPSIZE;
01869         while ( term < stopper ) {
01870             if ( *term == SYMTOSYM || *term == SYMTONUM ) {
01871                 term += 2;
01872                 AN.UsedSymbol[*term] = 1;
01873                 symbols[*term].flags |= INUSE;
01874             }
01875             else if ( *term == VECTOVEC ) {
01876                 term += 2;

```

```

01877         AN.UsedVector[*term-AM.OffsetVector] = 1;
01878         vectors[*term-AM.OffsetVector].flags |= INUSE;
01879     }
01880     else if ( *term == INDTOIND ) {
01881         term += 2;
01882         sym = indices[*term - AM.OffsetIndex].dimension;
01883         if ( sym < 0 ) AN.UsedSymbol[-sym] = 1;
01884         AN.UsedIndex[*term - AM.OffsetIndex] = 1;
01885         sym = indices[*term-AM.OffsetIndex].nmin4;
01886         if ( sym < -NMIN4SHIFT ) AN.UsedSymbol[-sym-NMIN4SHIFT] = 1;
01887         indices[*term-AM.OffsetIndex].flags |= INUSE;
01888     }
01889     else if ( *term == FUNTOFUN ) {
01890         term += 2;
01891         AN.UsedFunction[*term-FUNCTION] = 1;
01892         functions[*term-FUNCTION].flags |= INUSE;
01893     }
01894     term += 2;
01895 }
01896 }
01897 else {
01898     while ( term < stopper ) {
01899         t = term + term[1];
01900         if ( *term == SYMBOL ) {
01901             term += 2;
01902             do {
01903                 AN.UsedSymbol[*term] = 1;
01904                 term += 2;
01905             } while ( term < t );
01906         }
01907         else if ( *term == DOTPRODUCT ) {
01908             term += 2;
01909             do {
01910                 AN.UsedVector[(+term++) - AM.OffsetVector] = 1;
01911                 AN.UsedVector[(+term) - AM.OffsetVector] = 1;
01912                 term += 2;
01913             } while ( term < t );
01914         }
01915         else if ( *term == VECTOR ) {
01916             term += 2;
01917             do {
01918                 AN.UsedVector[(+term++) - AM.OffsetVector] = 1;
01919                 if ( *term >= AM.OffsetIndex && *term < AM.DumInd ) {
01920                     sym = indices[*term - AM.OffsetIndex].dimension;
01921                     if ( sym < 0 ) AN.UsedSymbol[-sym] = 1;
01922                     AN.UsedIndex[*term - AM.OffsetIndex] = 1;
01923                     sym = indices[(+term++)-AM.OffsetIndex].nmin4;
01924                     if ( sym < -NMIN4SHIFT ) AN.UsedSymbol[-sym-NMIN4SHIFT] = 1;
01925                 }
01926                 else term++;
01927             } while ( term < t );
01928         }
01929         else if ( *term == INDEX || *term == LEVICIVITA || *term == GAMMA
01930             || *term == DELTA ) {
01931             /*
01932             Tensors:
01933             */
01934             term += 2;
01935             if ( *term == INDEX || *term == DELTA ) term += 2;
01936             else {
01937                 Tensors:
01938                 term += FUNHEAD;
01939             }
01940             while ( term < t ) {
01941                 if ( *term >= AM.OffsetIndex && *term < AM.DumInd ) {
01942                     sym = indices[*term - AM.OffsetIndex].dimension;
01943                     if ( sym < 0 ) AN.UsedSymbol[-sym] = 1;
01944                     AN.UsedIndex[*term] - AM.OffsetIndex] = 1;
01945                     sym = indices[*term-AM.OffsetIndex].nmin4;
01946                     if ( sym < -NMIN4SHIFT ) AN.UsedSymbol[-sym-NMIN4SHIFT] = 1;
01947                 }
01948                 else if ( *term < (WILDOFFSET+AM.OffsetVector) )
01949                     AN.UsedVector[(+term) - AM.OffsetVector] = 1;
01950                 term++;
01951             }
01952         }
01953         else if ( *term == HAAKJE ) term = t;
01954         else {
01955             if ( *term > MAXBUILTINFUNCTION )
01956                 AN.UsedFunction[(+term)-FUNCTION] = 1;
01957             if ( *term >= FUNCTION && functions[*term-FUNCTION].spec
01958                 >= TENSORFUNCTION && term[1] > FUNHEAD ) goto Tensors;
01959             term += FUNHEAD;
01960             /* First argument */
01961             while ( term < t ) {
01962                 sarg = term;
01963                 NEXTARG(sarg)
01964                 if ( *term > 0 ) {

```

```

01964         sarg = term + *term;      /* End of argument */
01965         term += ARGHEAD;           /* First term in argument */
01966         if ( term < sarg ) { do {
01967             DetVars(term,par);
01968             term += *term;
01969         } while ( term < sarg ); }
01970     }
01971     else {
01972         if ( *term < -MAXBUILTINFUNCTION ) {
01973             AN.UsedFunction[-*term-FUNCTION] = 1;
01974         }
01975         else if ( *term == -SYMBOL ) {
01976             AN.UsedSymbol[term[1]] = 1;
01977         }
01978         else if ( *term == -INDEX ) {
01979             if ( term[1] < (WILDOffset+AM.OffsetVector) ) {
01980                 AN.UsedVector[term[1]-AM.OffsetVector] = 1;
01981             }
01982             else if ( term[1] >= AM.OffsetIndex && term[1] < AM.DumInd ) {
01983                 sym = indices[term[1] - AM.OffsetIndex].dimension;
01984                 if ( sym < 0 ) AN.UsedSymbol[-sym] = 1;
01985                 AN.UsedIndex[term[1] - AM.OffsetIndex] = 1;
01986                 sym = indices[term[1]-AM.OffsetIndex].nmin4;
01987                 if ( sym < -NMIN4SHIFT ) AN.UsedSymbol[-sym-NMIN4SHIFT] = 1;
01988             }
01989         }
01990         else if ( *term == -VECTOR || *term == -MINVECTOR ) {
01991             AN.UsedVector[term[1]-AM.OffsetVector] = 1;
01992         }
01993     }
01994     term = sarg;          /* Next argument */
01995     term = t;
01996 }
01997 }
01998 }
01999 }
02000 }
02001
02002 /*
02003     #] DetVars :
02004     #[ ToStorage :
02005
02006     This routine takes an expression in the scratch buffer (indicated by e)
02007     and puts it in the storage file. The necessary actions are:
02008
02009     1: determine the list of the used variables.
02010     2: make an index entry.
02011     3: write the namelists.
02012     4: copy the 'length' bytes of the expression.
02013
02014 */
02015
02016 int ToStorage(EXPRESSIONS e, POSITION *length)
02017 {
02018     GETIDENTITY
02019     WORD *w, i, j;
02020     WORD *term;
02021     INDEXENTRY *indexent;
02022     LONG size;
02023     POSITION indexpos, scrpos;
02024     FILEHANDLE *f;
02025     if ( ( indexent = NextFileIndex(&indexpos) ) == 0 ) {
02026         MesCall("ToStorage");
02027         SETERROR(-1)
02028     }
02029     indexent->CompressSize = 0;      /* thus far no compression */
02030     f = AR.infile; AR.infile = AR.outfile; AR.outfile = f;
02031     if ( e->status == HIDDENEXPRESSION ) {
02032         AR.InHiBuf = 0; f = AR.hidefile; AR.GetFile = 2;
02033     }
02034     else {
02035         AR.InInBuf = 0; f = AR.infile; AR.GetFile = 0;
02036     }
02037     if ( f->handle >= 0 ) {
02038         scrpos = e->onfile;
02039         SeekFile(f->handle,&scrpos,SEEK_SET);
02040         if ( !ISNOTEQUALPOS(scrpos,e->onfile) ) {
02041             MesPrint(":::Error in Scratch file");
02042             goto ErrReturn;
02043         }
02044         f->POposition = e->onfile;
02045         f->POfull = f->PObuffer;
02046         if ( e->status == HIDDENEXPRESSION ) AR.InHiBuf = 0;
02047         else AR.InInBuf = 0;
02048     }
02049     else {
02050         f->POfill = (WORD *) ((UBYTE *) (f->PObuffer)+BASEPOSITION(e->onfile));

```

```

02051     }
02052     w = AT.WorkPointer;
02053     AN.UsedSymbol = w;    w += NumSymbols;
02054     AN.UsedVector = w;    w += NumVectors;
02055     AN.UsedIndex = w;    w += NumIndices;
02056     AN.UsedFunction = w; w += NumFunctions;
02057     term = w;
02058     w = (WORD *)(((UBYTE *) (w)) + AM.MaxTer);
02059     if ( w > AT.WorkTop ) {
02060         MesWork();
02061         goto ErrReturn;
02062     }
02063     w = AN.UsedSymbol;
02064     i = NumSymbols + NumVectors + NumIndices + NumFunctions;
02065     do { *w++ = 0; } while ( --i > 0 );
02066     if ( GetTerm(BHEAD term) > 0 ) {
02067         DetVars(term,1);
02068         if ( GetTerm(BHEAD term) ) {
02069             do { DetVars(term,0); } while ( GetTerm(BHEAD term) > 0 );
02070         }
02071     }
02072     j = 0;
02073     w = AN.UsedSymbol;
02074     i = NumSymbols;
02075     while ( --i >= 0 ) { if ( *w++ ) j++; }
02076     indexent->nsymbols = j;
02077     /* size = j * sizeof(struct SyMbOl); */
02078     j = 0;
02079     w = AN.UsedIndex;
02080     i = NumIndices;
02081     while ( --i >= 0 ) { if ( *w++ ) j++; }
02082     indexent->nindices = j;
02083     /* size += j * sizeof(struct InDeX); */
02084     j = 0;
02085     w = AN.UsedVector;
02086     i = NumVectors;
02087     while ( --i >= 0 ) { if ( *w++ ) j++; }
02088     indexent->nvectors = j;
02089     /* size += j * sizeof(struct VeCtOr); */
02090     j = 0;
02091     w = AN.UsedFunction;
02092     i = NumFunctions;
02093     while ( --i >= 0 ) { if ( *w++ ) j++; }
02094     indexent->nfunctions = j;
02095     /* size += j * sizeof(struct FuNcTiOn); */
02096     indexent->length = *length;
02097     indexent->variables = AR.StoreData.Fill;
02098     /* indexent->position = AR.StoreData.Fill + size; */
02099     StrCopy(AC.exprnames->namebuffer+e->name, (UBYTE *) (indexent->name));
02100     SeekFile(AR.StoreData.Handle, &(AR.StoreData.Fill), SEEK_SET);
02101     AO.wlen = 100000;
02102     AO.wpos = (UBYTE *) Malloc1(AO.wlen, "AO.wpos buffer");
02103     AO.wpoin = AO.wpos;
02104     {
02105         SYMBOLS a;
02106         w = AN.UsedSymbol;
02107         a = symbols;
02108         j = 0;
02109         i = indexent->nsymbols;
02110         while ( --i >= 0 ) {
02111             while ( !*w ) { w++; a++; j++; }
02112             a->number = j;
02113             if ( VarStore((UBYTE *) a, (WORD) (sizeof(struct SyMbOl)), a->name,
02114                 a->namesize) ) goto ErrToSto;
02115             w++; j++; a++;
02116         }
02117     }
02118     {
02119         INDICES a;
02120         w = AN.UsedIndex;
02121         a = indices;
02122         j = 0;
02123         i = indexent->nindices;
02124         while ( --i >= 0 ) {
02125             while ( !*w ) { w++; a++; j++; }
02126             a->number = j;
02127             if ( VarStore((UBYTE *) a, (WORD) (sizeof(struct InDeX)), a->name,
02128                 a->namesize) ) goto ErrToSto;
02129             w++; j++; a++;
02130         }
02131     }
02132     {
02133         VECTORS a;
02134         w = AN.UsedVector;
02135         a = vectors;
02136         j = 0;
02137         i = indexent->nvectors;

```



```

02138     while ( --i >= 0 ) {
02139         while ( !*w ) { w++; a++; j++; }
02140         a->number = j;
02141         if ( VarStore((UBYTE *)a, (WORD) (sizeof(struct VeCtOr)), a->name,
02142             a->namesize) ) goto ErrToSto;
02143         w++; j++; a++;
02144     }
02145 }
02146 {
02147     FUNCTIONS a;
02148     w = AN.UsedFunction;
02149     a = functions;
02150     j = 0;
02151     i = indexent->nfunctions;
02152     while ( --i >= 0 ) {
02153         while ( !*w ) { w++; a++; j++; }
02154         a->number = j;
02155         if ( VarStore((UBYTE *)a, (WORD) (sizeof(struct FuNcTiOn)), a->name,
02156             a->namesize) ) goto ErrToSto;
02157         w++; a++; j++;
02158     }
02159 }
02160 if ( VarStore((UBYTE *)0L, (WORD)0, (WORD)0, (WORD)0) ) goto ErrToSto; /* Flush buffer */
02161 TELLFILE (AR.StoreData.Handle, &(indexent->position));
02162 indexent->size = (WORD)DIFBASE(indexent->position, indexent->variables);
02163 /*
02164     The following code was added when it became apparent (30-jan-2007)
02165     that we need provisions for extra space without upsetting existing
02166     .sav files. Here we can put as much as we want.
02167     Look in GetTable on how to recover numdummies.
02168     Forgetting numdummies has been in there from the beginning.
02169 */
02170 if ( WriteFile(AR.StoreData.Handle, (UBYTE *) (&(e->numdummies)), (LONG)sizeof(WORD)) !=
02171     sizeof(WORD) ) {
02172     MesPrint("Error while writing storage file");
02173     goto ErrReturn;
02174 }
02175 if ( WriteFile(AR.StoreData.Handle, (UBYTE *) (&(e->numfactors)), (LONG)sizeof(WORD)) !=
02176     sizeof(WORD) ) {
02177     MesPrint("Error while writing storage file");
02178     goto ErrReturn;
02179 }
02180 if ( WriteFile(AR.StoreData.Handle, (UBYTE *) (&(e->vflags)), (LONG)sizeof(WORD)) !=
02181     sizeof(WORD) ) {
02182     MesPrint("Error while writing storage file");
02183     goto ErrReturn;
02184 }
02185 if ( WriteFile(AR.StoreData.Handle, (UBYTE *) (&(e->uflags)), (LONG)sizeof(WORD)) !=
02186     sizeof(WORD) ) {
02187     MesPrint("Error while writing storage file");
02188     goto ErrReturn;
02189 }
02190 TELLFILE (AR.StoreData.Handle, &(indexent->position));
02191 if ( f->handle >= 0 ) {
02192     POSITION llength;
02193     llength = *length;
02194     SeekFile(f->handle, &(e->onfile), SEEK_SET);
02195     while ( ISPOSPOS(llength) ) {
02196         SETBASEPOSITION(scrpos, AO.wlen);
02197         if ( ISLESSPOS(llength, scrpos) ) size = BASEPOSITION(llength);
02198         else size = AO.wlen;
02199         if ( ReadFile(f->handle, AO.wpos, size) != size ) {
02200             MesPrint("Error while reading scratch file");
02201             goto ErrReturn;
02202         }
02203         if ( WriteFile(AR.StoreData.Handle, AO.wpos, size) != size ) {
02204             MesPrint("Error while writing storage file");
02205             goto ErrReturn;
02206         }
02207         ADDPOS(llength, -size);
02208     }
02209 }
02210 else {
02211     WORD *ppp;
02212     ppp = (WORD *) ((UBYTE *) (f->PObuffer) + BASEPOSITION(e->onfile));
02213     if ( WriteFile(AR.StoreData.Handle, (UBYTE *) ppp, BASEPOSITION(*length)) !=
02214         BASEPOSITION(*length) ) {
02215         MesPrint("Error while writing storage file");
02216         goto ErrReturn;
02217     }
02218 }
02219 ADD2POS(*length, indexent->position);
02220 e->onfile = indexpos;
02221 /*
02222     AR.StoreData.Fill = SeekFile(AR.StoreData.Handle, &(AM.zeropos), SEEK_END);
02223 */
02224 AR.StoreData.Fill = *length;

```

```

02225     SeekFile (AR.StoreData.Handle, &(AR.StoreData.Fill), SEEK_SET);
02226     scrpos = AR.StoreData.Position;
02227     ADDPOS (scrpos, sizeof(POSITION));
02228     SeekFile (AR.StoreData.Handle, &scrpos, SEEK_SET);
02229     if ( WriteFile (AR.StoreData.Handle, ((UBYTE *) &(AR.StoreData.Index.number))
02230     , (LONG) (sizeof(POSITION))) != sizeof(POSITION) ) goto ErrInSto;
02231     SeekFile (AR.StoreData.Handle, &indexpos, SEEK_SET);
02232     if ( WriteFile (AR.StoreData.Handle, (UBYTE *) indexent, (LONG) (sizeof(INDEXENTRY))) !=
02233     sizeof(INDEXENTRY) ) goto ErrInSto;
02234     FlushFile (AR.StoreData.Handle);
02235     SeekFile (AR.StoreData.Handle, &(AC.StoreFileSize), SEEK_END);
02236     f = AR.infile; AR.infile = AR.outfile; AR.outfile = f;
02237     if ( AO.wpos ) M_free (AO.wpos, "AO.wpos buffer");
02238     AO.wpos = AO.wpoin = 0;
02239     return (0);
02240 ErrToSto:
02241     MesPrint ("---Error while storing namelists");
02242     goto ErrReturn;
02243 ErrInSto:
02244     MesPrint ("Error in storage");
02245 ErrReturn:
02246     if ( AO.wpos ) M_free (AO.wpos, "AO.wpos buffer");
02247     AO.wpos = AO.wpoin = 0;
02248     f = AR.infile; AR.infile = AR.outfile; AR.outfile = f;
02249     return (-1);
02250 }
02251
02252 /*
02253     #] ToStorage :
02254     #[ NextFileIndex :
02255 */
02256 INDEXENTRY *NextFileIndex (POSITION *indexpos)
02257 {
02258     GETIDENTITY
02259     INDEXENTRY *ind;
02260     int i, j = sizeof(FILEINDEX) / (sizeof(LONG));
02261     LONG *lo;
02262     if ( AR.StoreData.Handle <= 0 ) {
02263         if ( SetFileIndex() ) {
02264             MesCall ("NextFileIndex");
02265             return (0);
02266         }
02267     }
02268     SETBASEPOSITION (AR.StoreData.Index.number, 1);
02269 #ifdef SYSDEPENDENTSAVE
02270     SETBASEPOSITION (*indexpos, (2*sizeof(POSITION)));
02271 #else
02272     SETBASEPOSITION (*indexpos, (2*sizeof(POSITION)+sizeof(STOREHEADER)));
02273 #endif
02274     return (AR.StoreData.Index.expression);
02275 }
02276 while ( BASEPOSITION (AR.StoreData.Index.number) >= (LONG) (INFILEINDEX) ) {
02277     if ( ISNOTZEROPOS (AR.StoreData.Index.next) ) {
02278         SeekFile (AR.StoreData.Handle, &(AR.StoreData.Index.next), SEEK_SET);
02279         AR.StoreData.Position = AR.StoreData.Index.next;
02280         if ( ReadFile (AR.StoreData.Handle, (UBYTE
02281 *) (&AR.StoreData.Index), (LONG) (sizeof(FILEINDEX))) !=
02282         (LONG) (sizeof(FILEINDEX)) ) goto ErrNextS;
02283     }
02284     else {
02285         PUTZERO (AR.StoreData.Index.number);
02286         SeekFile (AR.StoreData.Handle, &(AR.StoreData.Position), SEEK_SET);
02287         if ( WriteFile (AR.StoreData.Handle, (UBYTE
02288 *) (&(AR.StoreData.Fill)), (LONG) (sizeof(POSITION)))
02289         != (LONG) (sizeof(POSITION)) ) goto ErrNextS;
02290         PUTZERO (AR.StoreData.Index.next);
02291         SeekFile (AR.StoreData.Handle, &(AR.StoreData.Fill), SEEK_SET);
02292         AR.StoreData.Position = AR.StoreData.Fill;
02293         lo = (LONG *) (&AR.StoreData.Index);
02294         for ( i = 0; i < j; i++ ) *lo++ = 0;
02295         if ( WriteFile (AR.StoreData.Handle, (UBYTE
02296 *) (&AR.StoreData.Index), (LONG) (sizeof(FILEINDEX))) !=
02297         (LONG) (sizeof(FILEINDEX)) ) goto ErrNextS;
02298         ADDPOS (AR.StoreData.Fill, sizeof(FILEINDEX));
02299     }
02300 }
02301 *indexpos = AR.StoreData.Position;
02302 ADDPOS (*indexpos, (2*sizeof(POSITION)) +
02303     BASEPOSITION (AR.StoreData.Index.number) * sizeof(INDEXENTRY));
02304 ind = &AR.StoreData.Index.expression[BASEPOSITION (AR.StoreData.Index.number)];
02305 ADDPOS (AR.StoreData.Index.number, 1);
02306 return (ind);
02307 ErrNextS:
02308 MesPrint ("Error in storage file");
02309 return (0);
02310 }
02311 }
02312 }

```

```

02309 /*
02310      #] NextFileIndex :
02311      #[ SetFileIndex :
02312 */
02313
02320 int SetFileIndex(void)
02321 {
02322     GETIDENTITY
02323     int i, j = sizeof(FILEINDEX) / (sizeof(LONG));
02324     LONG *lo;
02325     if ( AR.StoreData.Handle < 0 ) {
02326         AR.StoreData.Handle = AC.StoreHandle;
02327         PUTZERO(AR.StoreData.Index.next);
02328         PUTZERO(AR.StoreData.Index.number);
02329 #ifdef SYSDEPENDENTSAVE
02330         SETBASEPOSITION(AR.StoreData.Fill, sizeof(FILEINDEX));
02331 #else
02332         if ( WriteStoreHeader(AR.StoreData.Handle) ) return(MesPrint("Error writing storage file
header"));
02333         SETBASEPOSITION(AR.StoreData.Fill, (LONG)sizeof(FILEINDEX)+(LONG)sizeof(STOREHEADER));
02334 #endif
02335         lo = (LONG *)(&AR.StoreData.Index);
02336         for ( i = 0; i < j; i++ ) *lo++ = 0;
02337         if ( WriteFile(AR.StoreData.Handle, (UBYTE *)(&AR.StoreData.Index), (LONG)(sizeof(FILEINDEX)))
!=
02338             (LONG)(sizeof(FILEINDEX)) ) return(MesPrint("Error writing storage file"));
02339     }
02340     else {
02341         POSITION scrpos;
02342 #ifdef SYSDEPENDENTSAVE
02343         PUTZERO(scrpos);
02344 #else
02345         SETBASEPOSITION(scrpos, (LONG)(sizeof(STOREHEADER)));
02346 #endif
02347         SeekFile(AR.StoreData.Handle, &scrpos, SEEK_SET);
02348         if ( ReadFile(AR.StoreData.Handle, (UBYTE *)(&AR.StoreData.Index), (LONG)(sizeof(FILEINDEX))) !=
02349             (LONG)(sizeof(FILEINDEX)) ) return(MesPrint("Error reading storage file"));
02350     }
02351 #ifdef SYSDEPENDENTSAVE
02352     PUTZERO(AR.StoreData.Position);
02353 #else
02354     SETBASEPOSITION(AR.StoreData.Position, (LONG)(sizeof(STOREHEADER)));
02355 #endif
02356     return(0);
02357 }
02358
02359 /*
02360      #] SetFileIndex :
02361      #[ VarStore :
02362
02363      The n -= sizeof(WORD); makes that the real length comes in the
02364      padding space, provided there is padding space (it seems so).
02365      The reading of the information assumes this is the case and hence
02366      things work....
02367 */
02368
02369 int VarStore(UBYTE *s, WORD n, WORD name, WORD namesize)
02370 {
02371     GETIDENTITY
02372     UBYTE *t, *u;
02373     if ( s ) {
02374         n -= sizeof(WORD);
02375         t = (UBYTE *)AO.wpos;
02376     /*
02377         u = (UBYTE *)AT.WorkTop;
02378     */
02379         u = AO.wpos+AO.wlen;
02380         while ( n > 0 && t < u ) { *t++ = *s++; n--; }
02381         while ( t >= u ) {
02382             if ( WriteFile(AR.StoreData.Handle, AO.wpos, AO.wlen) != AO.wlen ) return(-1);
02383             t = AO.wpos;
02384             while ( n > 0 && t < u ) { *t++ = *s++; n--; }
02385         }
02386         s = AC.varnames->namebuffer + name;
02387         n = namesize;
02388         n += sizeof(void *)-1; n &= -(sizeof(void *));
02389         *((WORD *)t) = n;
02390         t += sizeof(WORD);
02391         while ( n > 0 && t < u ) {
02392             if ( namesize > 0 ) { *t++ = *s++; namesize--; }
02393             else { *t++ = 0; }
02394             n--;
02395         }
02396         while ( t >= u ) {
02397             if ( WriteFile(AR.StoreData.Handle, AO.wpos, AO.wlen) != AO.wlen ) return(-1);
02398             t = AO.wpos;
02399             while ( n > 0 && t < u ) {

```

```

02400         if ( namesize > 0 ) { *t++ = *s++; namesize--; }
02401         else { *t++ = 0; }
02402         n--;
02403     }
02404 }
02405 AO.wpoin = t;
02406 }
02407 else {
02408     LONG size;
02409     size = AO.wpoin - AO.wpos;
02410     if ( WriteFile(AR.StoreData.Handle,AO.wpos,size) != size ) return(-1);
02411     AO.wpoin = AO.wpos;
02412 }
02413 return(0);
02414 }
02415
02416 /*
02417     #] VarStore :
02418     #[ TermRenumber :
02419
02420     rennumbers the variables inside term according to the information
02421     in struct renumber.
02422     The search is binary. This avoided having to read/write the
02423     expression twice when it was stored.
02424 */
02425 */
02426
02427 int TermRenumber(WORD *term, RENUMBER renumber, WORD nexpr)
02428 {
02429     WORD *stopper;
02430     WORD *t, *sarg, n;
02431     stopper = term + *term - 1;
02432     stopper = stopper - ABS(*stopper) + 1;
02433     term++;
02434     while ( term < stopper ) {
02435         if ( *term == SYMBOL ) {
02436             t = term + term[1];
02437             term += 2;
02438             do {
02439                 if ( ( n = FindrNumber(*term,&(renumber->symp)) ) < 0 ) goto ErrR;
02440                 *term = renumber->sympnum[n];
02441                 term += 2;
02442             } while ( term < t );
02443         }
02444         else if ( *term == DOTPRODUCT ) {
02445             t = term + term[1];
02446             term += 2;
02447             do {
02448                 if ( ( n = FindrNumber(*term,&(renumber->vect)) ) < 0 ) goto ErrR;
02449                 *term++ = renumber->vecnum[n];
02450                 if ( ( n = FindrNumber(*term,&(renumber->vect)) ) < 0 ) goto ErrR;
02451                 *term = renumber->vecnum[n];
02452                 term += 2;
02453             } while ( term < t );
02454         }
02455         else if ( *term == VECTOR ) {
02456             t = term + term[1];
02457             term += 2;
02458             do {
02459                 if ( ( n = FindrNumber(*term,&(renumber->vect)) ) < 0 ) goto ErrR;
02460                 *term++ = renumber->vecnum[n];
02461                 if ( ( *term >= AM.OffsetIndex ) && ( *term < AM.IndDum ) ) {
02462                     if ( ( n = FindrNumber(*term,&(renumber->indi)) ) < 0 ) goto ErrR;
02463                     *term++ = renumber->indnum[n];
02464                 }
02465                 else term++;
02466             } while ( term < t );
02467         }
02468         else if ( *term == INDEX || *term == LEVICIVITA || *term == GAMMA
02469                 || *term == DELTA ) {
02470             Tensors:
02471             t = term + term[1];
02472             if ( *term == INDEX || *term == DELTA ) term += 2;
02473             else term += FUNHEAD;
02474             /*
02475             term += 2;
02476             */
02477             while ( term < t ) {
02478                 if ( *term >= AM.OffsetIndex + WILDOFFSET ) {
02479                     /*
02480                     Still TOBEDONE
02481                     */
02482                 }
02483             }
02484         }
02485     }
02486 }

```

```

02494         else if ( ( *term >= AM.OffsetIndex ) && ( *term < AM.IndDum ) ) {
02495             if ( ( n = FindrNumber(*term,&(renumber->indi)) )
02496                 < 0 ) goto ErrR;
02497             *term = renumber->indnum[n];
02498         }
02499         else if ( *term < (WILDOFFSET+AM.OffsetVector) ) {
02500             if ( ( n = FindrNumber(*term,&(renumber->vect)) )
02501                 < 0 ) goto ErrR;
02502             *term = renumber->vecnum[n];
02503         }
02504         term++;
02505     }
02506 }
02507 else if ( *term == HAAKJE ) term += term[1];
02508 else {
02509     if ( *term > MAXBUILTINFUNCTION ) {
02510         if ( ( n = FindrNumber(*term,&(renumber->func)) )
02511             < 0 ) goto ErrR;
02512         *term = renumber->funnum[n];
02513     }
02514     if ( *term >= FUNCTION && functions[*term-FUNCTION].spec
02515         >= TENSORFUNCTION && term[1] > FUNHEAD ) goto Tensors;
02516     t = term + term[1];          /* General stopper */
02517     term += FUNHEAD;            /* First argument */
02518     while ( term < t ) {
02519         sarg = term;
02520         NEXTARG(sarg)
02521         if ( *term > 0 ) {
02522             /*
02523              Problem here:
02524              Marking the argument as dirty attacks the heap
02525              very heavily and costs much computer time.
02526             */
02527             ++term = 1;
02528             term += ARGHEAD-1;
02529             while ( term < sarg ) {
02530                 if ( TermRenumber(term,renumber,nexpr) ) goto ErrR;
02531                 term += *term;
02532             }
02533         }
02534         else {
02535             if ( *term <= -MAXBUILTINFUNCTION ) {
02536                 if ( ( n = FindrNumber(*term,&(renumber->func)) )
02537                     < 0 ) goto ErrR;
02538                 *term = -renumber->funnum[n];
02539             }
02540             else if ( *term == -SYMBOL ) {
02541                 term++;
02542                 if ( ( n = FindrNumber(*term,
02543                     &(renumber->symb)) ) < 0 ) goto ErrR;
02544                 *term = renumber->symnum[n];
02545             }
02546             else if ( *term == -INDEX ) {
02547                 term++;
02548                 if ( *term >= AM.OffsetIndex + WILDOFFSET ) {
02549                     /*
02550                      Still TOBEDONE
02551                     */
02552                 }
02553                 else if ( ( *term >= AM.OffsetIndex ) && ( *term < AM.IndDum ) ) {
02554                     if ( ( n = FindrNumber(*term,&(renumber->indi)) )
02555                         < 0 ) goto ErrR;
02556                     *term = renumber->indnum[n];
02557                 }
02558                 else if ( *term < (WILDOFFSET+AM.OffsetVector) ) {
02559                     if ( ( n = FindrNumber(*term,&(renumber->vect)) )
02560                         < 0 ) goto ErrR;
02561                     *term = renumber->vecnum[n];
02562                 }
02563             }
02564             else if ( *term == -VECTOR || *term == -MINVECTOR ) {
02565                 term++;
02566                 if ( ( n = FindrNumber(*term,&(renumber->vect)) )
02567                     < 0 ) goto ErrR;
02568                 *term = renumber->vecnum[n];
02569             }
02570         }
02571         term = sarg;          /* Next argument */
02572     }
02573     term = t;
02574 }
02575 }
02576 return(0);
02577 ErrR:
02578     MesCall("TermRenumber");
02579     SETERROR(-1)
02580 }

```

```

02581
02582 /*
02583     #] TermRenumber :
02584     #[ FindrNumber :
02585 */
02586
02587 WORD FindrNumber(WORD n, VARRENUM *v)
02588 {
02589     WORD *hi,*med,*lo;
02590     hi = v->hi;
02591     lo = v->lo;
02592     med = v->start;
02593     if ( *hi == 0 ) {
02594         if ( n != *hi ) {
02595             MesPrint("Serious problems coming up in FindrNumber");
02596             return(-1);
02597         }
02598         return(*hi);
02599     }
02600     while ( *med != n ) {
02601         if ( *med < n ) {
02602             if ( med == hi ) goto ErrFindr;
02603             lo = med;
02604             med = hi - ((WORDDIFF(hi,med))/2);
02605         }
02606         else {
02607             if ( med == lo ) goto ErrFindr;
02608             hi = med;
02609             med = lo + ((WORDDIFF(med,lo))/2);
02610         }
02611     }
02612     return(WORDDIFF(med,v->lo));
02613 ErrFindr:
02614 /*
02615     Reconstruction:
02616 */
02617 {
02618     int i;
02619     i = WORDDIFF(v->hi,v->lo);
02620     MesPrint("FindrNumber: n = %d, list has %d members",n,i);
02621     while ( i >= 0 ) {
02622         MesPrint("v->lo[%d] = %d",i,v->lo[i]); i--;
02623     }
02624     hi = v->hi;
02625     lo = v->lo;
02626     med = v->start;
02627     MesPrint("Start with %d,%d,%d",0,WORDDIFF(med,v->lo),WORDDIFF(hi,v->lo));
02628     while ( *med != n ) {
02629         if ( *med < n ) {
02630             if ( med == hi ) goto ErrFindr2;
02631             lo = med;
02632             med = hi - ((WORDDIFF(hi,med))/2);
02633         }
02634         else {
02635             if ( med == lo ) goto ErrFindr2;
02636             hi = med;
02637             med = ((WORDDIFF(med,lo))/2) + lo;
02638         }
02639         MesPrint("New: %d,%d,%d, *med = %d",WORDDIFF(lo,v->lo),WORDDIFF(med,v->lo),WORDDIFF(hi,v->lo),*med);
02640     }
02641 }
02642     return(WORDDIFF(med,v->lo));
02643 ErrFindr2:
02644     return(MesPrint("Renumbering problems"));
02645 }
02646
02647 /*
02648     #] FindrNumber :
02649     #[ FindInIndex :
02650
02651     Finds an expression in the storage index if it exists.
02652     If found it returns a pointer to the index entry, otherwise zero.
02653     par = 0     Search by address (--> f == &AR.StoreData, called by GetTable, CoSave )
02654     par = 1     Search by name (--> f == &AO.SaveData, called by CoLoad )
02655
02656     When comparing parameter fields the parameters of the expression
02657     to be searched are in AT.TMaddr. This includes the primary expression
02658     and a possible FROMBRAC information. The FROMBRAC is always last.
02659
02660     The parameter mode tells whether we should worry about arguments of
02661     a stored expression.
02662 */
02663
02664 INDEXENTRY *FindInIndex(WORD expr, FILEDATA *f, WORD par, WORD mode)
02665 {
02666     GETIDENTITY

```

```

02667     INDEXENTRY *ind;
02668     WORD i, hand, *m;
02669     WORD *start, *stop, *stop2, *m2, nomatch = 0;
02670     POSITION stindex, indexpos, scrpos;
02671     LONG number, num;
02672     stindex = f->Position;
02673     m = AT.TMaddr;
02674     stop = m + m[1];
02675     m += SUBEXPSIZE;
02676     start = m;
02677     while ( m < stop ) {
02678         if ( *m == FROMBRAC || *m == WILDCARDS ) break;
02679         m += m[1];
02680     }
02681     stop = m;
02682     if ( !par ) hand = AR.StoreData.Handle;
02683     else hand = AO.SaveData.Handle;
02684     for(;;) {
02685         if ( ( i = (WORD)BASEPOSITION(f->Index.number) ) != 0 ) {
02686             indexpos = f->Position;
02687             ADDPOS(indexpos, (2*sizeof(POSITION)));
02688             ind = f->Index.expression;
02689             do {
02690                 if ( ( !par && ISEQUALPOS(indexpos, Expressions[expr].onfile) )
02691                     || ( par && !StrCmp(EXPRNAME(expr), (UBYTE *) (ind->name)) ) ) {
02692                     nomatch = 1;
02693                     /*
02694                     MesPrint("index: position: %8p", &(ind->position));
02695                     MesPrint("index: length: %8p", &(ind->length));
02696                     MesPrint("index: variables: %8p", &(ind->variables));
02697                     MesPrint("index: nsymbols: %d", ind->nsymbols);
02698                     MesPrint("index: nindices: %d", ind->nindices);
02699                     MesPrint("index: nvectors: %d", ind->nvectors);
02700                     MesPrint("index: nfunctions: %d", ind->nfunctions);
02701                     MesPrint("index: size: %d", ind->size);
02702                     */
02703                     if ( par ) return(ind);
02704                     scrpos = ind->position;
02705                     SeekFile(hand, &scrpos, SEEK_SET);
02706                     if ( ISNOTEQUALPOS(scrpos, ind->position) ) goto ErrGt2;
02707                     if ( ReadFile(hand, (UBYTE *)AT.WorkPointer, (LONG)sizeof(WORD)) !=
02708                         sizeof(WORD) || !*AT.WorkPointer ) goto ErrGt2;
02709                     num = *AT.WorkPointer - 1;
02710                     num *= wsizeof(WORD);
02711                     if ( *AT.WorkPointer < 0 ||
02712                         ReadFile(hand, (UBYTE *) (AT.WorkPointer+1), num) != num ) goto ErrGt2;
02713                     m = start; /* start of parameter field to be searched */
02714                     m2 = AT.WorkPointer + 1;
02715                     stop2 = m2 + m2[1];
02716                     m2 += SUBEXPSIZE;
02717                     while ( m < stop && m2 < stop2 ) {
02718                         if ( *m == SYMBOL ) {
02719                             if ( *m2 != SYMTOSYM ) break;
02720                             m2[3] = m[2];
02721                         }
02722                         else if ( *m == INDEX ) {
02723                             if ( m[2] >= 0 ) {
02724                                 if ( *m2 != INDTOIND ) break;
02725                             }
02726                             else {
02727                                 if ( *m2 != VECTOVEC ) break;
02728                             }
02729                             m2[3] = m[2];
02730                         }
02731                         else if ( *m >= FUNCTION ) {
02732                             if ( *m2 != FUNTOFUN ) break;
02733                             m2[3] = *m;
02734                         }
02735                         else {}
02736                         m += m[1];
02737                         m2 += m2[1];
02738                     }
02739                     if ( ( m >= stop && m2 >= stop2 ) || mode == 0 ) {
02740                         AT.WorkPointer = stop2;
02741                         return(ind);
02742                     }
02743                 }
02744             }
02745             ind++;
02746             ADDPOS(indexpos, sizeof(INDEXENTRY));
02747         } while ( --i > 0 );
02748     }
02749     f->Position = f->Index.next;
02750 #ifndef SYSDEPENDENTSAVE
02751     if ( !ISNOTZEROPOS(f->Position) ) ADDPOS(f->Position, sizeof(STOREHEADER));
02752     number = sizeof(struct FileInDeX);
02753 #endif

```

```

02754         if ( ISEQUALPOS(f->Position,stindex) && !AO.bufferedInd ) goto ErrGetTab;
02755         if ( !par ) {
02756             SeekFile(AR.StoreData.Handle,&(f->Position),SEEK_SET);
02757             if ( ISNOTEQUALPOS(f->Position,AR.StoreData.Position) ) goto ErrGt2;
02758 #ifndef SYSDEPENDENTSAVE
02759             if ( ReadFile(f->Handle, (UBYTE *)(&(f->Index)), number) != number ) goto ErrGt2;
02760 #endif
02761         }
02762         else {
02763             SeekFile(AO.SaveData.Handle,&(f->Position),SEEK_SET);
02764             if ( ISNOTEQUALPOS(f->Position,AO.SaveData.Position) ) goto ErrGt2;
02765 #ifndef SYSDEPENDENTSAVE
02766             if ( ReadSaveIndex(&(f->Index)) ) goto ErrGt2;
02767 #endif
02768         }
02769 #ifdef SYSDEPENDENTSAVE
02770         number = sizeof(struct FileInDeX);
02771         if ( ReadFile(f->Handle, (UBYTE *)(&(f->Index)),number) !=
02772             number ) goto ErrGt2;
02773 #endif
02774     }
02775 ErrGetTab:
02776     if ( nomatch ) {
02777         MesPrint("Parameters of expression %s don't match."
02778             ,EXPRNAME(expr));
02779     }
02780     else {
02781         MesPrint("Cannot find expression %s",EXPRNAME(expr));
02782     }
02783     return(0);
02784 ErrGt2:
02785     MesPrint("Readerror in IndexSearch");
02786     return(0);
02787 }
02788
02789 /*
02790     #] FindInIndex :
02791     #[ GetTable :
02792
02793     Locates stored files and constructs the renumbering tables.
02794     They are allocated in the Workspace.
02795     First the expression data are located. The Index is treated
02796     as a circularly linked buffer which is paged forwardly.
02797     If the indexentry is located (in ind) the two renumber tables
02798     have to be constructed.
02799     Finally the prototype has to be put in the proper buffer, so
02800     that wildcards can be passed. There should be a test with
02801     an already existing prototype that is constructed by the
02802     pattern matcher. This has not been put in yet.
02803
02804     There is a problem with the parallel processing.
02805     Feeding in the variables that were erased by a .store could in
02806     principle happen in different orders (ParFORM) or simultaneously
02807     (TFORM). The proper resolution is to have the compiler call GetTable
02808     when a stored expression is encountered.
02809
02810     This has been mended in development of TFORM by reading the
02811     symbol tables during compilation. See the call to GetTable
02812     in the CodeGenerator.
02813
02814     Next is the problem of FindInIndex which writes in AR.StoreData
02815     Copying this is expensive!
02816
02817     This Doesn't work well for TFORM yet!!!!!!!
02818     e[x1,x2] versus e[x2,x1] messes up.
02819     For the rest is the reloading during execution not thread safe.
02820
02821     The parameter mode tells whether we should worry about arguments
02822     of a stored expression.
02823 */
02824
02825 RENUMBER GetTable(WORD expr, POSITION *position, WORD mode)
02826 {
02827     GETIDENTITY
02828     WORD i, j;
02829     WORD *w;
02830     RENUMBER r;
02831     LONG num, nsize, xx;
02832     WORD jsym, jind, jvec, jfun;
02833     WORD k, type, error = 0, *oldw, *neww, *oldwork = AT.WorkPointer;
02834     struct SyMbOl SyM;
02835     struct InDeX InD;
02836     struct VeCtOr VeC;
02837     struct FuNcTiOn FuN;
02838     INDEXENTRY *ind;
02839
02840     Prepare for FindInIndex to put the prototype in the Workspace.

```



```

02841     oldw will point at the "wildcards"
02842  */
02843  /*
02844       Bug fix. Look also in Generator.
02845  #ifndef WITHPTHREADS
02846
02847       if ( ( r = Expressions[expr].renum ) != 0 ) { }
02848     else {
02849         Expressions[expr].renum =
02850         r = (RENUMBER)Malloca(sizeof(struct ReNuMbEr), "ReNumber");
02851     }
02852  #else
02853       r = (RENUMBER)Malloca(sizeof(struct ReNuMbEr), "ReNumber");
02854  #endif
02855  */
02856       r = (RENUMBER)Malloca(sizeof(struct ReNuMbEr), "ReNumber");
02857
02858     oldw = AT.WorkPointer + 1 + SUBEXPSIZE;
02859  /*
02860     The prototype is loaded in the Workspace by the Index routine.
02861     After all it has to find an occurrence with the proper arguments.
02862     This sets the WorkPointer. Hence be careful now.
02863  */
02864     LOCK(AM.storefilelock);
02865     if ( ( ind = FindInIndex(expr, &AR.StoreData, 0, mode) ) == 0 ) {
02866         UNLOCK(AM.storefilelock);
02867         return(0);
02868     }
02869
02870     xx = ind->nsymbols+ind->nindices+ind->nvectors+ind->nfunctions;
02871     if ( xx == 0 ) {
02872         Expressions[expr].renumlists =
02873         w = AN.dummyrenumlist;
02874     }
02875     else {
02876  /*
02877  #ifndef WITHPTHREADS
02878         Expressions[expr].renumlists =
02879  #endif
02880  */
02881         w = (WORD *)Malloca(sizeof(WORD)*(xx*2), "VarSpace");
02882     }
02883     r->symb.lo = w;
02884     r->symb.start = w + ind->nsymbols/2;
02885     w += ind->nsymbols;
02886     r->symb.hi = w - 1;
02887     r->symnum = w;
02888     w += ind->nsymbols;
02889
02890     r->indi.lo = w;
02891     r->indi.start = w + ind->nindices/2;
02892     w += ind->nindices;
02893     r->indi.hi = w - 1;
02894     r->indnum = w;
02895     w += ind->nindices;
02896
02897     r->vect.lo = w;
02898     r->vect.start = w + ind->nvectors/2;
02899     w += ind->nvectors;
02900     r->vect.hi = w - 1;
02901     r->vecnum = w;
02902     w += ind->nvectors;
02903
02904     r->func.lo = w;
02905     r->func.start = w + ind->nfunctions/2;
02906     w += ind->nfunctions;
02907     r->func.hi = w - 1;
02908     r->funnum = w;
02909  /* w += ind->nfunctions; */
02910
02911     SeekFile(AR.StoreData.Handle, &(ind->variables), SEEK_SET);
02912     *position = ind->position;
02913     jsym = ind->nsymbols;
02914     jvec = ind->nvectors;
02915     jind = ind->nindices;
02916     jfun = ind->nfunctions;
02917  /*
02918         #[ Symbols :
02919  */
02920     {
02921         SYMBOLS s = &SyM;
02922         w = r->symb.lo; j = jsym;
02923         for ( i = 0; i < j; i++ ) {
02924             if ( ReadFile(AR.StoreData.Handle, (UBYTE *)s, (LONG)(sizeof(struct SyMbOl)))
02925                 != sizeof(struct SyMbOl) ) goto ErrGt2;
02926             nsize = s->namesize; nsize += sizeof(void *)-1;
02927             nsize &= -sizeof(void *);

```

```

02928     if ( ReadFile(AR.StoreData.Handle, (UBYTE *) (AT.WorkPointer), nsize)
02929     != nsize ) goto ErrGt2;
02930     *w = s->number;
02931     if ( ( s->flags & INUSE ) != 0 ) {
02932         /* Find the replacement. It must exist! */
02933         neww = oldw;
02934         while ( *neww != SYMTOSYM || neww[2] != *w ) neww += neww[1];
02935         k = neww[3];
02936     }
02937     else if ( GetVar((UBYTE *) AT.WorkPointer, &type, &k, ALLVARIABLES, NOAUTO) != NAMENOTFOUND ) {
02938         if ( type != CSYMBOL ) {
02939             MesPrint("Error: Conflicting types for %s", (AT.WorkPointer));
02940             error = -1;
02941         }
02942         else {
02943             if ( ( s->complex & (VARTYPEIMAGINARY|VARTYPECOMPLEX) ) !=
02944             ( symbols[k].complex & (VARTYPEIMAGINARY|VARTYPECOMPLEX) ) ) {
02945                 MesPrint("Warning: Conflicting complexity for %s", AT.WorkPointer);
02946                 error = -1;
02947             }
02948             if ( ( s->complex & (VARTYPEROOTOFUNITY) ) !=
02949             ( symbols[k].complex & (VARTYPEROOTOFUNITY) ) ) {
02950                 MesPrint("Warning: Conflicting root of unity properties for %s", AT.WorkPointer);
02951                 error = -1;
02952             }
02953             if ( ( s->complex & VARTYPEROOTOFUNITY ) == VARTYPEROOTOFUNITY ) {
02954                 if ( s->maxpower != symbols[k].maxpower ) {
02955                     MesPrint("Warning: Conflicting n in n-th root of unity properties for
02956 %s", AT.WorkPointer);
02957                     error = -1;
02958                 }
02959             }
02960             else if ( ( s->minpower !=
02961             symbols[k].minpower || s->maxpower !=
02962             symbols[k].maxpower ) && AC.WarnFlag ) {
02963                 MesPrint("Warning: Conflicting power restrictions for %s", AT.WorkPointer);
02964             }
02965         }
02966     }
02967     else {
02968         if ( ( k = EntVar(CSYMBOL, (UBYTE *) (AT.WorkPointer), s->complex, s->minpower,
02969         s->maxpower, s->dimension) ) < 0 ) goto GetTcall;
02970     }
02971     * (w+j) = k;
02972     w++;
02973 }
02974 /*
02975     #] Symbols :
02976     #[ Indices :
02977 */
02978 {
02979     INDICES s = &InD;
02980     w = r->indi.lo; j = jind;
02981     for ( i = 0; i < j; i++ ) {
02982         if ( ReadFile(AR.StoreData.Handle, (UBYTE *) s, (LONG) (sizeof(struct InDeX)))
02983         != sizeof(struct InDeX) ) goto ErrGt2;
02984         nsize = s->namesize; nsize += sizeof(void *)-1;
02985         nsize &= -sizeof(void *);
02986         if ( ReadFile(AR.StoreData.Handle, (UBYTE *) (AT.WorkPointer), nsize)
02987         != nsize ) goto ErrGt2;
02988         *w = s->number + AM.OffsetIndex;
02989         if ( s->dimension < 0 ) { /* Relabel the dimension */
02990             s->dimension = -r->symnum[FindrNumber(-s->dimension, &(r->symb))];
02991             if ( s->nmin4 < -NMIN4SHIFT ) { /* Relabel n-4 */
02992                 s->nmin4 = -r->symnum[FindrNumber(-s->nmin4-NMIN4SHIFT
02993                 , &(r->symb))]-NMIN4SHIFT;
02994             }
02995         }
02996         if ( ( s->flags & INUSE ) != 0 ) {
02997             /* Find the replacement. It must exist! */
02998             neww = oldw;
02999             while ( *neww != INDTOIND || neww[2] != *w ) neww += neww[1];
03000             k = neww[3] - AM.OffsetIndex;
03001         }
03002         else if ( s->type == DUMMY ) {
03003             /*
03004             ----->         Here we may have to execute some renumbering
03005             */
03006         }
03007         else if ( GetVar((UBYTE *) (AT.WorkPointer), &type, &k, ALLVARIABLES, NOAUTO) != NAMENOTFOUND ) {
03008             if ( type != CINDEX ) {
03009                 MesPrint("Error: Conflicting types for %s", (AT.WorkPointer));
03010                 error = -1;
03011             }
03012             else {
03013                 if ( s->type !=

```

```

03014         indices[k].type ) {
03015             MesPrint("Warning: %s is also a dummy index", (AT.WorkPointer));
03016             error = -1;
03017             goto GetTb3;
03018         }
03019         if ( s->dimension != indices[k].dimension ) {
03020             MesPrint("Warning: Conflicting dimensions for %s", (AT.WorkPointer));
03021             error = -1;
03022         }
03023     }
03024 }
03025 else {
03026 GetTb3:
03027     if ( ( k = EntVar(CINDEX, (UBYTE *) (AT.WorkPointer),
03028         s->dimension, 0, s->nmin4, 0) ) < 0 ) goto GetTcall;
03029 }
03030 * (w+j) = k + AM.OffsetIndex;
03031 w++;
03032 }
03033 }
03034 }
03035 /*
03036     #] Indices :
03037     #[ Vectors :
03038 */
03039 {
03040     VECTORS s = &Vec;
03041     w = r->vect.lo; j = jvec;
03042     for ( i = 0; i < j; i++ ) {
03043         if ( ReadFile(AR.StoreData.Handle, (UBYTE *)s, (LONG) (sizeof(struct VecTOr)))
03044             != sizeof(struct VecTOr) ) goto ErrGt2;
03045         nsize = s->namesize; nsize += sizeof(void *)-1;
03046         nsize &= -sizeof(void *);
03047         if ( ReadFile(AR.StoreData.Handle, (UBYTE *) (AT.WorkPointer), nsize)
03048             != nsize ) goto ErrGt2;
03049         *w = s->number + AM.OffsetVector;
03050         if ( ( s->flags & INUSE ) != 0 ) {
03051             /* Find the replacement. It must exist! */
03052             neww = oldw;
03053             while ( *neww != VECTOVEC || neww[2] != *w ) neww += neww[1];
03054             k = neww[3] - AM.OffsetVector;
03055         }
03056         else if ( GetVar((UBYTE *) (AT.WorkPointer), &type, &k, ALLVARIABLES, NOAUTO) != NAMENOTFOUND ) {
03057             if ( type != CVECTOR ) {
03058                 MesPrint("Error: Conflicting types for %s", (AT.WorkPointer));
03059                 error = -1;
03060             }
03061             else {
03062                 if ( ( s->complex & (VARTYPEIMAGINARY|VARTYPECOMPLEX) ) !=
03063                     ( vectors[k].complex & (VARTYPEIMAGINARY|VARTYPECOMPLEX) ) ) {
03064                     MesPrint("Warning: Conflicting complexity for %s", (AT.WorkPointer));
03065                     error = -1;
03066                 }
03067             }
03068         }
03069         else {
03070             if ( ( k = EntVar(CVECTOR, (UBYTE *) (AT.WorkPointer),
03071                 s->complex, 0, 0, s->dimension) ) < 0 ) goto GetTcall;
03072         }
03073         * (w+j) = k + AM.OffsetVector;
03074         w++;
03075     }
03076 }
03077 /*
03078     #] Vectors :
03079     #[ Functions :
03080 */
03081 {
03082     FUNCTIONS s = &FuN;
03083     w = r->func.lo; j = jfun;
03084     for ( i = 0; i < j; i++ ) {
03085         if ( ReadFile(AR.StoreData.Handle, (UBYTE *)s, (LONG) (sizeof(struct FuNcTiOn)))
03086             != sizeof(struct FuNcTiOn) ) goto ErrGt2;
03087         nsize = s->namesize; nsize += sizeof(void *)-1;
03088         nsize &= -sizeof(void *);
03089         if ( ReadFile(AR.StoreData.Handle, (UBYTE *) (AT.WorkPointer), nsize)
03090             != nsize ) goto ErrGt2;
03091         *w = s->number + FUNCTION;
03092         if ( ( s->flags & INUSE ) != 0 ) {
03093             /* Find the replacement. It must exist! */
03094             neww = oldw;
03095             while ( *neww != FUNTOFUN || neww[2] != *w ) neww += neww[1];
03096             k = neww[3] - FUNCTION;
03097         }
03098         else if ( GetVar((UBYTE *) (AT.WorkPointer), &type, &k, ALLVARIABLES, NOAUTO) != NAMENOTFOUND ) {
03099             if ( type != CFUNCTION ) {
03100                 MesPrint("Error: Conflicting types for %s", (AT.WorkPointer));

```

```

03101         error = -1;
03102     }
03103     else {
03104         if ( s->complex != functions[k].complex ) {
03105             MesPrint("Warning: Conflicting complexity for %s", (AT.WorkPointer));
03106             error = -1;
03107         }
03108         else if ( s->symmetric != functions[k].symmetric ) {
03109             MesPrint("Warning: Conflicting symmetry properties for %s", (AT.WorkPointer));
03110             error = -1;
03111         }
03112         else if ( ( s->maxnumargs != functions[k].maxnumargs )
03113             || ( s->minnumargs != functions[k].minnumargs ) ) {
03114             MesPrint("Warning: Conflicting argument restriction properties for
03115 %s", (AT.WorkPointer));
03116             error = -1;
03117         }
03118     }
03119     else {
03120         if ( ( k = EntVar(CFUNCTION, (UBYTE *) (AT.WorkPointer),
03121             s->complex, s->commute, s->spec, s->dimension) ) < 0 ) goto GetTcall;
03122         functions[k].symmetric = s->symmetric;
03123         functions[k].maxnumargs = s->maxnumargs;
03124         functions[k].minnumargs = s->minnumargs;
03125     }
03126     *(w+j) = k + FUNCTION;
03127     w++;
03128 }
03129 }
03130 /*
03131     #] Functions :
03132
03133     Now we skip the prototype. This sets the start position at the first term
03134 */
03135     if ( error ) {
03136         UNLOCK(AM.storefilelock);
03137         AT.WorkPointer = oldwork;
03138         return(0);
03139     }
03140     {
03141     /*
03142         For clarity we look where we are.
03143         We want to know: is this position already known?
03144         Could we have inserted extra information here?
03145
03146         nummystery indicates extra words. We have currently in order
03147         (if they exist)
03148             numdummies
03149             numfactors
03150             vflags
03151             uflags
03152     */
03153         POSITION pos;
03154         int nummystery;
03155         TELLFILE(AR.StoreData.Handle, &pos);
03156         nummystery = DIFBASE(ind->position, pos);
03157     /*
03158         MesPrint("---> We are at position      %8p", &pos);
03159         MesPrint("---> The index says      at %8p", &(ind->position));
03160         MesPrint("---> There are %d mystery bytes", nummystery);
03161     */
03162         if ( nummystery > 0 ) {
03163             if ( ReadFile(AR.StoreData.Handle, (UBYTE *) AT.WorkPointer, (LONG) sizeof(WORD)) !=
03164                 sizeof(WORD) ) {
03165                 UNLOCK(AM.storefilelock);
03166                 AT.WorkPointer = oldwork;
03167                 return(0);
03168             }
03169             Expressions[expr].numdummies = *AT.WorkPointer;
03170         /*
03171             MesPrint("---> numdummies = %d", Expressions[expr].numdummies);
03172         */
03173         nummystery -= sizeof(WORD);
03174     }
03175     else {
03176         Expressions[expr].numdummies = 0;
03177     }
03178     if ( nummystery > 0 ) {
03179         if ( ReadFile(AR.StoreData.Handle, (UBYTE *) AT.WorkPointer, (LONG) sizeof(WORD)) !=
03180             sizeof(WORD) ) {
03181             UNLOCK(AM.storefilelock);
03182             AT.WorkPointer = oldwork;
03183             return(0);
03184         }
03185     }
03186     if ( ( AS.OldNumFactors == 0 ) || ( AS.NumOldNumFactors < NumExpressions ) ) {

```

```

03187         WORD *buffer;
03188         int capacity = 20;
03189         if (capacity < NumExpressions) capacity = NumExpressions * 2;
03190
03191         buffer = (WORD *)Malloc1(capacity * sizeof(WORD), "numfactors pointers");
03192         if (AS.OldNumFactors) {
03193             WCOPY(buffer, AS.OldNumFactors, AS.NumOldNumFactors);
03194             M_free(AS.OldNumFactors, "numfactors pointers");
03195         }
03196         AS.OldNumFactors = buffer;
03197
03198         buffer = (WORD *)Mloc1(capacity * sizeof(WORD), "vflags pointers");
03199         if (AS.Oldvflags) {
03200             WCOPY(buffer, AS.Oldvflags, AS.NumOldNumFactors);
03201             M_free(AS.Oldvflags, "vflags pointers");
03202         }
03203         AS.Oldvflags = buffer;
03204
03205         buffer = (WORD *)Mloc1(capacity * sizeof(WORD), "uflags pointers");
03206         if (AS.Olduflags) {
03207             WCOPY(buffer, AS.Olduflags, AS.NumOldNumFactors);
03208             M_free(AS.Olduflags, "uflags pointers");
03209         }
03210         AS.Oldvflags = buffer;
03211
03212         AS.NumOldNumFactors = capacity;
03213     }
03214
03215     AS.OldNumFactors[expr] =
03216     Expressions[expr].numfactors = *AT.WorkPointer;
03217     /*
03218     MesPrint("--> numfactors = %d", Expressions[expr].numfactors);
03219     */
03220     nummystery -= sizeof(WORD);
03221 }
03222 else {
03223     Expressions[expr].numfactors = 0;
03224 }
03225 if ( nummystery > 0 ) {
03226     if ( ReadFile(AR.StoreData.Handle, (UBYTE *)AT.WorkPointer, (LONG)sizeof(WORD)) !=
03227         sizeof(WORD) ) {
03228         UNLOCK(AM.storefilelock);
03229         AT.WorkPointer = oldwork;
03230         return(0);
03231     }
03232     AS.Oldvflags[expr] =
03233     Expressions[expr].vflags = *AT.WorkPointer;
03234     /*
03235     MesPrint("--> vflags = %d", Expressions[expr].vflags);
03236     */
03237     nummystery -= sizeof(WORD);
03238 }
03239 else {
03240     Expressions[expr].vflags = 0;
03241 }
03242 if ( nummystery > 0 ) {
03243     if ( ReadFile(AR.StoreData.Handle, (UBYTE *)AT.WorkPointer, (LONG)sizeof(WORD)) !=
03244         sizeof(WORD) ) {
03245         UNLOCK(AM.storefilelock);
03246         AT.WorkPointer = oldwork;
03247         return(0);
03248     }
03249     AS.Olduflags[expr] =
03250     Expressions[expr].uflags = *AT.WorkPointer;
03251     /*
03252     MesPrint("--> uflags = %d", Expressions[expr].uflags);
03253     */
03254     nummystery -= sizeof(WORD);
03255 }
03256 else {
03257     Expressions[expr].uflags = 0;
03258 }
03259 }
03260
03261 SeekFile(AR.StoreData.Handle, &(ind->position), SEEK_SET);
03262 if ( ReadFile(AR.StoreData.Handle, (UBYTE *)AT.WorkPointer, (LONG)sizeof(WORD)) !=
03263     sizeof(WORD) || !*AT.WorkPointer ) {
03264     UNLOCK(AM.storefilelock);
03265     AT.WorkPointer = oldwork;
03266     return(0);
03267 }
03268 num = *AT.WorkPointer - 1;
03269 num *= sizeof(WORD);
03270 if ( *AT.WorkPointer < 0 ||
03271     ReadFile(AR.StoreData.Handle, (UBYTE *) (AT.WorkPointer+1), num) != num ) {
03272     MesPrint("@Error in stored expressions file at position %10p", *position);
03273     UNLOCK(AM.storefilelock);

```

```

03274         AT.WorkPointer = oldwork;
03275         return(0);
03276     }
03277     UNLOCK(AM.storefilelock);
03278     ADDPOS(*position,num+sizeof(WORD));
03279     r->startposition = *position;
03280     AT.WorkPointer = oldwork;
03281     return(r);
03282 GetTcall:
03283     UNLOCK(AM.storefilelock);
03284     AT.WorkPointer = oldwork;
03285     MesCall("GetTable");
03286     return(0);
03287 ErrGt2:
03288     UNLOCK(AM.storefilelock);
03289     AT.WorkPointer = oldwork;
03290     MesPrint("Readererror in GetTable");
03291     return(0);
03292 }
03293
03294 /*
03295     #] GetTable :
03296     #[ CopyExpression :
03297
03298     Copies from one scratch buffer to another.
03299     We assume here that the complete 'from' scratch buffer is taken.
03300     We also assume that the 'from' buffer is positioned at the end of
03301     the expression.
03302
03303     The locks should be placed in the calling routine. We need basically
03304     AS.outputslock.
03305 */
03306
03307 int CopyExpression(FILEHANDLE *from, FILEHANDLE *to)
03308 {
03309     POSITION posfrom, poscopy;
03310     LONG fullsize,i;
03311     WORD *t1, *t2;
03312     int RetCode;
03313     SeekScratch(from,&posfrom);
03314     if ( from->handle < 0 ) { /* input is in memory */
03315         fullsize = (BASEPOSITION(posfrom))/sizeof(WORD);
03316         if ( ( to->PStop - to->PFull ) >= fullsize ) {
03317             /*
03318              Fits inside the buffer of the output. This will be fast.
03319             */
03320             t1 = from->Pbuffer;
03321             t2 = to->PFull;
03322             NCOPY(t2,t1,fullsize)
03323             to->PFull = to->Pfill = t2;
03324             goto WriteTrailer;
03325         }
03326         if ( to->handle < 0 ) { /* First open the file */
03327             if ( ( RetCode = CreateFile(to->name) ) >= 0 ) {
03328                 to->handle = (WORD)RetCode;
03329                 PUTZERO(to->filesize);
03330                 PUTZERO(to->Pposition);
03331             }
03332             else {
03333                 MLOCK(ErrorMessageLock);
03334                 MesPrint("Cannot create scratch file %s",to->name);
03335                 MUNLOCK(ErrorMessageLock);
03336                 return(-1);
03337             }
03338         }
03339         t1 = from->Pbuffer;
03340         while ( fullsize > 0 ) {
03341             i = to->PStop - to->PFull;
03342             if ( i > fullsize ) i = fullsize;
03343             fullsize -= i;
03344             t2 = to->PFull;
03345             NCOPY(t2,t1,i)
03346             if ( fullsize > 0 ) {
03347                 SeekFile(to->handle,&(to->Pposition),SEEK_SET);
03348                 if ( WriteFile(to->handle,((BYTE *) (to->Pbuffer)),to->Psize) != to->Psize ) {
03349                     MLOCK(ErrorMessageLock);
03350                     MesPrint("Error while writing to disk. Disk full?");
03351                     MUNLOCK(ErrorMessageLock);
03352                     return(-1);
03353                 }
03354                 ADDPOS(to->Pposition,to->Psize);
03355             /*
03356             SeekFile(to->handle,&(to->Pposition),SEEK_CUR); */
03356             to->filesize = to->Pposition;
03357             to->Pfill = to->PFull = to->Pbuffer;
03358         }
03359         else {
03360             to->Pfill = to->PFull = t2;

```

```

03361     }
03362     }
03363     goto WriteTrailer;
03364 }
03365 /*
03366 Now the input involves a file. This needs the use of the PObuffer of from.
03367 First make sure the tail of the buffer has been written
03368 */
03369 if ( ((UBYTE *) (from->POfill)) - ((UBYTE *) (from->PObuffer)) > 0 ) {
03370     if ( WriteFile(from->handle, ((UBYTE *) (from->PObuffer)), ((UBYTE *) (from->POfill)) - ((UBYTE
03371 *) (from->PObuffer)))
03372     != ((UBYTE *) (from->POfill)) - ((UBYTE *) (from->PObuffer)) ) {
03373         MLOCK(ErrorMessageLock);
03374         MesPrint("Error while writing to disk. Disk full?");
03375         MUNLOCK(ErrorMessageLock);
03376         return(-1);
03377     }
03378     SeekFile(from->handle, &(from->POposition), SEEK_CUR);
03379     posfrom = from->filesize = from->POposition;
03380     from->POfill = from->POfull = from->PObuffer;
03381 }
03382 /*
03383 Now copy the complete contents
03384 */
03385 PUTZERO(poscopy);
03386 SeekFile(from->handle, &poscopy, SEEK_SET);
03387 while ( ISLESSPOS(poscopy, posfrom) ) {
03388     fullsize = ReadFile(from->handle, ((UBYTE *) (from->PObuffer)), from->POsize);
03389     if ( fullsize < 0 || ( fullsize % sizeof(WORD) ) != 0 ) {
03390         MLOCK(ErrorMessageLock);
03391         MesPrint("Error while reading from disk while copying expression.");
03392         MUNLOCK(ErrorMessageLock);
03393         return(-1);
03394     }
03395     fullsize /= sizeof(WORD);
03396     from->POfull = from->PObuffer + fullsize;
03397     t1 = from->PObuffer;
03398     if ( ( to->POstop - to->POfull ) >= fullsize ) {
03399         /*
03400 Fits inside the buffer of the output. This will be fast.
03401 */
03402         t2 = to->POfull;
03403         NCOPY(t2, t1, fullsize);
03404         to->POfill = to->POfull = t2;
03405     }
03406     else {
03407         if ( to->handle < 0 ) { /* First open the file */
03408             if ( ( RetCode = CreateFile(to->name) ) >= 0 ) {
03409                 to->handle = (WORD) RetCode;
03410                 PUTZERO(to->POposition);
03411                 PUTZERO(to->filesize);
03412             }
03413             else {
03414                 MLOCK(ErrorMessageLock);
03415                 MesPrint("Cannot create scratch file %s", to->name);
03416                 MUNLOCK(ErrorMessageLock);
03417                 return(-1);
03418             }
03419         }
03420         while ( fullsize > 0 ) {
03421             i = to->POstop - to->POfull;
03422             if ( i > fullsize ) i = fullsize;
03423             fullsize -= i;
03424             t2 = to->POfull;
03425             NCOPY(t2, t1, i);
03426             if ( fullsize > 0 ) {
03427                 SeekFile(to->handle, &(to->POposition), SEEK_SET);
03428                 if ( WriteFile(to->handle, ((UBYTE *) (to->PObuffer)), to->POsize) != to->POsize ) {
03429                     MLOCK(ErrorMessageLock);
03430                     MesPrint("Error while writing to disk. Disk full?");
03431                     MUNLOCK(ErrorMessageLock);
03432                     return(-1);
03433                 }
03434                 ADDPOS(to->POposition, to->POsize);
03435             }
03436             /*
03437 SeekFile(to->handle, &(to->POposition), SEEK_CUR); */
03438             to->filesize = to->POposition;
03439             to->POfill = to->POfull = to->PObuffer;
03440         }
03441         else {
03442             to->POfill = to->POfull = t2;
03443         }
03444     }
03445     SeekFile(from->handle, &poscopy, SEEK_CUR);
03446 }
03447 WriteTrailer:

```

```

03447     if ( ( to->handle >= 0 ) && ( to->POfill > to->PObuffer ) ) {
03448         fullsize = (UBYTE *) (to->POfill) - (UBYTE *) (to->PObuffer);
03449     /*
03450         PUTZERO(to->POposition);
03451         SeekFile(to->handle, &(to->POposition), SEEK_END);
03452     */
03453         SeekFile(to->handle, &(to->filesize), SEEK_SET);
03454         if ( WriteFile(to->handle, ((UBYTE *) (to->PObuffer)), fullsize) != fullsize ) {
03455             MLOCK(ErrorMessageLock);
03456             MesPrint("Error while writing to disk. Disk full?");
03457             MUNLOCK(ErrorMessageLock);
03458             return(-1);
03459         }
03460         ADDPOS(to->filesize, fullsize);
03461         to->POposition = to->filesize;
03462         to->POfill = to->POfull = to->PObuffer;
03463     }
03464
03465     return(0);
03466 }
03467
03468 /*
03469     #] CopyExpression :
03470     #[ ExprStatus :
03471 */
03472
03473 #ifndef HIDEDEBUG
03474
03475 static UBYTE *statusexpr[] = {
03476     (UBYTE *) "LOCALEXPRESSION"
03477 , (UBYTE *) "SKIPLEXPRESSION"
03478 , (UBYTE *) "DROPLEXPRESSION"
03479 , (UBYTE *) "DROPPLEXPRESSION"
03480 , (UBYTE *) "GLOBALEXPRESSION"
03481 , (UBYTE *) "SKIPGEXPRESSION"
03482 , (UBYTE *) "DROPGEXPRESSION"
03483 , (UBYTE *) "UNKNOWN"
03484 , (UBYTE *) "STOREDEXPRESSION"
03485 , (UBYTE *) "HIDDENLEXPRESSION"
03486 , (UBYTE *) "HIDELEXPRESSION"
03487 , (UBYTE *) "DROPHLEXPRESSION"
03488 , (UBYTE *) "UNHIDELEXPRESSION"
03489 , (UBYTE *) "HIDDENGEXPRESSION"
03490 , (UBYTE *) "HIDEGEXPRESSION"
03491 , (UBYTE *) "DROPHGEXPRESSION"
03492 , (UBYTE *) "UNHIDEGEXPRESSION"
03493 , (UBYTE *) "INTOHIDELEXPRESSION"
03494 , (UBYTE *) "INTOHIDEGEXPRESSION"
03495 };
03496
03497 void ExprStatus(EXPRESSIONS e)
03498 {
03499     MesPrint("Expression %s(%d) has status %s(%d,%d). Buffer: %d, Position: %15p",
03500         AC.exprnames->namebuffer+e->name, (WORD) (e->Expressions),
03501         statusexpr[e->status], e->status, e->hidelevel,
03502         e->whichbuffer, &(e->onfile));
03503 }
03504
03505 #endif
03506
03507 /*
03508     #] ExprStatus :
03509     #] StoreExpressions :
03510     #[ System Independent Saved Expressions :
03511
03512     All functions concerned with the system independent reading of save-files
03513     are here. They are called by the functions CoLoad, PutInStore,
03514     SetFileIndex, FindInIndex. In case no translation (endianness flip,
03515     resizing of words, renumbering) has to be done, they just do simple file
03516     reading. The function SaveFileHeader() for writing a header with
03517     information about the system architecture, FORM version, etc. is also
03518     located here.
03519
03520     #[ Flip :
03521 */
03522
03523 static void FlipN(UBYTE *p, int length)
03524 {
03525     UBYTE *q, buf;
03526     q = p + length;
03527     do {
03528         --q;
03529         buf = *p; *p = *q; *q = buf;
03530     } while ( ++p != q );
03531 }
03532
03533 static void Flip16(UBYTE *p)

```



```

03552 {
03553     uint16_t in = *((uint16_t *)p);
03554     uint16_t out = (uint16_t) ( ((in) >> 8) & UINT16_C(0x00FF) | ((in) << 8) & UINT16_C(0xFF00) );
03555     *((uint16_t *)p) = out;
03556 }
03557
03559 static void Flip32(UBYTE *p)
03560 {
03561     uint32_t in = *((uint32_t *)p);
03562     uint32_t out =
03563         ( ((in) >> 24) & UINT32_C(0x000000FF) ) | ( ((in) >> 8) & UINT32_C(0x0000FF00) ) | \
03564         ( ((in) << 8) & UINT32_C(0x00FF0000) ) | ( ((in) << 24) & UINT32_C(0xFF000000) );
03565     *((uint32_t *)p) = out;
03566 }
03567
03569 #ifdef UINT64_C
03570 static void Flip64(UBYTE *p)
03571 {
03572     uint64_t in = *((uint64_t *)p);
03573     uint64_t out =
03574         ( ((in) >> 56) & UINT64_C(0x00000000000000FF) ) | ( ((in) >> 40) & UINT64_C(0x000000000000FF00) )
03575         | \
03576         ( ((in) >> 24) & UINT64_C(0x0000000000FF0000) ) | ( ((in) >> 8) & UINT64_C(0x00000000FF000000) )
03577         | \
03578         ( ((in) << 8) & UINT64_C(0x000000FF00000000) ) | ( ((in) << 24) & UINT64_C(0x0000FF0000000000) )
03579         | \
03580         ( ((in) << 40) & UINT64_C(0x00FF000000000000) ) | ( ((in) << 56) & UINT64_C(0xFF00000000000000) );
03581     *((uint64_t *)p) = out;
03582 }
03583 #else
03584 static void Flip64(UBYTE *p) { FlipN(p, 8); }
03585 #endif /* UINT64_C */
03586
03588 static void Flip128(UBYTE *p) { FlipN(p, 16); }
03589
03591 /*
03592     #] Flip :
03593     #[ Resize :
03594 */
03595
03603 static void ResizeDataBE(UBYTE *src, int slen, UBYTE *dst, int dlen)
03604 {
03605     if ( slen > dlen ) {
03606         src += slen - dlen;
03607         while ( dlen-- ) { *dst++ = *src++; }
03608     }
03609     else {
03610         int i = dlen - slen;
03611         while ( i-- ) { *dst++ = 0; }
03612         while ( slen-- ) { *dst++ = *src++; }
03613     }
03614 }
03615
03619 static void ResizeDataLE(UBYTE *src, int slen, UBYTE *dst, int dlen)
03620 {
03621     if ( slen > dlen ) {
03622         while ( dlen-- ) { *dst++ = *src++; }
03623     }
03624     else {
03625         int i = dlen - slen;
03626         while ( slen-- ) { *dst++ = *src++; }
03627         while ( i-- ) { *dst++ = 0; }
03628     }
03629 }
03630
03643 static void Resize16t16(UBYTE *src, UBYTE *dst)
03644 {
03645     *((int16_t *)dst) = *((int16_t *)src);
03646 }
03647
03649 static void Resize16t32(UBYTE *src, UBYTE *dst)
03650 {
03651     int16_t in = *((int16_t *)src);
03652     int32_t out = (int32_t)in;
03653     *((int32_t *)dst) = out;
03654 }
03655
03657 #ifdef INT64_C
03658 static void Resize16t64(UBYTE *src, UBYTE *dst)
03659 {
03660     int16_t in = *((int16_t *)src);
03661     int64_t out = (int64_t)in;
03662     *((int64_t *)dst) = out;
03663 }
03664 #else
03665 static void Resize16t64(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 2, dst, 8); }

```

```

03666 #endif /* INT64_C */
03667
03669 static void Resize16t128(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 2, dst, 16); }
03670
03672 static void Resize32t32(UBYTE *src, UBYTE *dst)
03673 {
03674     *((int32_t *)dst) = *((int32_t *)src);
03675 }
03676
03678 #ifdef INT64_C
03679 static void Resize32t64(UBYTE *src, UBYTE *dst)
03680 {
03681     int32_t in = *((int32_t *)src);
03682     int64_t out = (int64_t)in;
03683     *((int64_t *)dst) = out;
03684 }
03685 #else
03686 static void Resize32t64(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 4, dst, 8); }
03687 #endif /* INT64_C */
03688
03690 static void Resize32t128(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 4, dst, 16); }
03691
03693 #ifdef INT64_C
03694 static void Resize64t64(UBYTE *src, UBYTE *dst)
03695 {
03696     *((int64_t *)dst) = *((int64_t *)src);
03697 }
03698 #else
03699 static void Resize64t64(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 8, dst, 8); }
03700 #endif /* INT64_C */
03701
03703 static void Resize64t128(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 8, dst, 16); }
03704
03706 static void Resize128t128(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 16); }
03707
03709 static void Resize32t16(UBYTE *src, UBYTE *dst)
03710 {
03711     int32_t in = *((int32_t *)src);
03712     int16_t out = (int16_t)in;
03713     if ( in > (1<<15)-1 || in < -(1<<15)+1 ) AO.resizeFlag |= 1;
03714     *((int16_t *)dst) = out;
03715 }
03716
03723 static void Resize32t16NC(UBYTE *src, UBYTE *dst)
03724 {
03725     int32_t in = *((int32_t *)src);
03726     int16_t out = (int16_t)in;
03727     *((int16_t *)dst) = out;
03728 }
03729
03730 #ifdef INT64_C
03732 static void Resize64t16(UBYTE *src, UBYTE *dst)
03733 {
03734     int64_t in = *((int64_t *)src);
03735     int16_t out = (int16_t)in;
03736     if ( in > (1<<15)-1 || in < -(1<<15)+1 ) AO.resizeFlag |= 1;
03737     *((int16_t *)dst) = out;
03738 }
03740 static void Resize64t16NC(UBYTE *src, UBYTE *dst)
03741 {
03742     int64_t in = *((int64_t *)src);
03743     int16_t out = (int16_t)in;
03744     *((int16_t *)dst) = out;
03745 }
03746 #else
03748 static void Resize64t16(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 8, dst, 2); }
03750 static void Resize64t16NC(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 8, dst, 2); }
03751 #endif /* INT64_C */
03752
03753 #ifdef INT64_C
03755 static void Resize64t32(UBYTE *src, UBYTE *dst)
03756 {
03757     int64_t in = *((int64_t *)src);
03758     int32_t out = (int32_t)in;
03759     if ( in > (INT64_C(1)<<31)-1 || in < -(INT64_C(1)<<31)+1 ) AO.resizeFlag |= 1;
03760     *((int32_t *)dst) = out;
03761 }
03763 static void Resize64t32NC(UBYTE *src, UBYTE *dst)
03764 {
03765     int64_t in = *((int64_t *)src);
03766     int32_t out = (int32_t)in;
03767     *((int32_t *)dst) = out;
03768 }
03769 #else
03771 static void Resize64t32(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 8, dst, 4); }
03773 static void Resize64t32NC(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 8, dst, 4); }
03774 #endif /* INT64_C */

```

```

03775
03777 static void Resize128t16(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 2); }
03778
03780 static void Resize128t16NC(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 2); }
03781
03783 static void Resize128t32(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 4); }
03784
03786 static void Resize128t32NC(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 4); }
03787
03789 static void Resize128t64(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 8); }
03790
03792 static void Resize128t64NC(UBYTE *src, UBYTE *dst) { AO.ResizeData(src, 16, dst, 8); }
03793
03794 /*
03795     #] Resize :
03796     #[ CheckPower and RenumberVec :
03797 */
03798
03805 static void CheckPower32(UBYTE *p)
03806 {
03807     if ( *((int32_t *)p) < -MAXPOWER ) {
03808         AO.powerFlag |= 0x01;
03809         *((int32_t *)p) = -MAXPOWER;
03810     }
03811     p += sizeof(int32_t);
03812     if ( *((int32_t *)p) > MAXPOWER ) {
03813         AO.powerFlag |= 0x02;
03814         *((int32_t *)p) = MAXPOWER;
03815     }
03816 }
03817
03825 static void RenumberVec32(UBYTE *p)
03826 {
03827     /* int32_t wilddoffset = *((int32_t *)AO.SaveHeader.wilddoffset); */
03828     void *dummy = (void *)AO.SaveHeader.wilddoffset; /* to remove a warning about strict-aliasing
rules in gcc */
03829     int32_t wilddoffset = *(int32_t *)dummy;
03830     int32_t in = *((int32_t *)p);
03831     in = in + 2*wilddoffset;
03832     in = in - 2*WILDDOFFSET;
03833     *((int32_t *)p) = in;
03834 }
03835
03836 /*
03837     #] CheckPower and RenumberVec :
03838     #[ ResizeCoeff :
03839 */
03840
03854 static void ResizeCoeff32(UBYTE **bout, UBYTE *bend, UBYTE *top)
03855 {
03856     int i;
03857     int32_t sign;
03858     int32_t *in, *p;
03859     int32_t *out = (int32_t *)*bout;
03860     int32_t *end = (int32_t *)bend;
03861
03862     if ( sizeof(WORD) == 2 ) {
03863         /* 4 -> 2 */
03864         int32_t len = (end - 1 - out) / 2;
03865         int zeros = 2;
03866         p = out + len - 1;
03867
03868         if ( *p & 0xFFFF0000 ) --zeros;
03869         p += len;
03870         if ( *p & 0xFFFF0000 ) --zeros;
03871
03872         in = end - 1;
03873         sign = ( *in-- > 0 ) ? 1 : -1;
03874         p = out + 4*len;
03875         if ( zeros == 2 ) p -= 2;
03876         out = p--;
03877
03878         if ( zeros < 2 ) *p-- = *in >> 16;
03879         *p-- = *in-- & 0x0000FFFF;
03880         for ( i = 1; i < len; ++i ) {
03881             *p-- = *in >> 16;
03882             *p-- = *in-- & 0x0000FFFF;
03883         }
03884         if ( zeros < 2 ) *p-- = *in >> 16;
03885         *p-- = *in-- & 0x0000FFFF;
03886         for ( i = 1; i < len; ++i ) {
03887             *p-- = *in >> 16;
03888             *p-- = *in-- & 0x0000FFFF;
03889         }
03890
03891         *out = (out - p) * sign;
03892         *bout = (UBYTE *) (out+1);

```

```

03893
03894     }
03895     else {
03896         /* 2 -> 4 */
03897         int32_t len = (end - 1 - out) / 2;
03898         if ( len == 1 ) {
03899             *out = *(uint16_t *)out;
03900             ++out;
03901             *out = *(uint16_t *)out;
03902             ++out;
03903             ++out;
03904         }
03905         else {
03906             p = out;
03907             *out = *(uint16_t *)out;
03908             in = out + 1;
03909             for ( i = 1; i < len; ++i ) {
03910                 /* shift */
03911                 *out = (uint32_t)(*(uint16_t *)out)
03912                     + ((uint32_t)(*(uint16_t *)in) << 16);
03913                 ++in;
03914                 if ( ++i == len ) break;
03915                 /* copy */
03916                 ++out;
03917                 *out = *(uint16_t *)in;
03918                 ++in;
03919             }
03920             ++out;
03921             *out = *(uint16_t *)in;
03922             ++in;
03923             for ( i = 1; i < len; ++i ) {
03924                 /* shift */
03925                 *out = (uint32_t)(*(uint16_t *)out)
03926                     + ((uint32_t)(*(uint16_t *)in) << 16);
03927                 ++in;
03928                 if ( ++i == len ) break;
03929                 /* copy */
03930                 ++out;
03931                 *out = *(uint16_t *)in;
03932                 ++in;
03933             }
03934             ++out;
03935             if ( *in < 0 ) *out = -(out - p + 1);
03936             else *out = out - p + 1;
03937             ++out;
03938         }
03939
03940         if ( out > (int32_t *)top ) {
03941             MesPrint("Error in resizing coefficient!");
03942         }
03943
03944         *bout = (UBYTE *)out;
03945     }
03946 }
03947
03948 /*
03949     #] ResizeCoeff :
03950     #[ WriteStoreHeader :
03951 */
03952
03953 #define SAVEREVISION 0x02
03954
03955 LONG WriteStoreHeader(WORD handle)
03956 {
03957     /* template of the STOREHEADER */
03958     static STOREHEADER sh = {
03959         { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF }, /* store header mark */
03960         0, 0, 0, 0, /* sizeof of WORD, LONG, POSITION, void* */
03961         { 0 }, /* endianness check number */
03962         0, 0, 0, 0, /* sizeof variable structs */
03963         { 0 }, /* maxpower */
03964         { 0 }, /* wildoffset */
03965         SAVEREVISION, /* revision */
03966         { 0 } }; /* reserved */
03967     int endian, i;
03968
03969     /* if called for the first time ... */
03970     if ( sh.lenWORD == 0 ) {
03971         sh.lenWORD = sizeof(WORD);
03972         sh.lenLONG = sizeof(LONG);
03973         sh.lenPOS = sizeof(POSITION);
03974         sh.lenPOINTER = sizeof(void *);
03975
03976         endian = 1;
03977         for ( i = 1; i < (int)sizeof(int); ++i ) {
03978             endian <= 8;
03979             endian += i+1;
03980         }
03981     }

```

```

03989     }
03990     for ( i = 0; i < (int)sizeof(int); ++i ) sh.endianness[i] = ((char *)&endian)[i];
03991
03992     sh.sSym = sizeof(struct SyMbol);
03993     sh.sInd = sizeof(struct InDeX);
03994     sh.sVec = sizeof(struct VeCtOr);
03995     sh.sFun = sizeof(struct FuNcTiOn);
03996
03997     /* *((WORD *)sh.maxpower) = MAXPOWER;
03998     *((WORD *)sh.wildoffset) = WILDOFFSET; */
03999     {
04000         WORD dumw[8];
04001         UBYTE *dummy;
04002         for ( i = 0; i < 8; i++ ) dumw[i] = 0;
04003         dummy = (UBYTE *)dumw;
04004         dumw[0] = (WORD)MAXPOWER;
04005         for ( i = 0; i < 16; i++ ) sh.maxpower[i] = dummy[i];
04006         dumw[0] = (WORD)WILDOFFSET;
04007         for ( i = 0; i < 16; i++ ) sh.wildoffset[i] = dummy[i];
04008     }
04009 }
04010
04011 return ( WriteFile(handle, (UBYTE *)(&sh), (LONG)(sizeof(STOREHEADER)))
04012         != (LONG)(sizeof(STOREHEADER)) );
04013 }
04014
04015 /*
04016     #] WriteStoreHeader :
04017     #[ CompactifySizeof :
04018 */
04019
04027 static unsigned int CompactifySizeof(unsigned int size)
04028 {
04029     switch ( size ) {
04030         case 2: return 0;
04031         case 4: return 1;
04032         case 8: return 2;
04033         case 16: return 3;
04034         default: MesPrint("Error compactifying size.");
04035                 return 3;
04036     }
04037 }
04038
04039 /*
04040     #] CompactifySizeof :
04041     #[ ReadSaveHeader :
04042 */
04043
04057 int ReadSaveHeader(void)
04058 {
04059     /* Read-only tables of function pointers for conversions. */
04060     static void (*flipJumpTable[4])(UBYTE *) =
04061     { Flip16, Flip32, Flip64, Flip128 };
04062     static void (*resizeJumpTable[4][4])(UBYTE *, UBYTE *) = /* "own x saved"-sizes */
04063     { { Resize16t16, Resize32t16, Resize64t16, Resize128t16 },
04064       { Resize16t32, Resize32t32, Resize64t32, Resize128t32 },
04065       { Resize16t64, Resize32t64, Resize64t64, Resize128t64 },
04066       { Resize16t128, Resize32t128, Resize64t128, Resize128t128 } };
04067     static void (*resizeNCJumpTable[4][4])(UBYTE *, UBYTE *) = /* "own x saved"-sizes */
04068     { { Resize16t16, Resize32t16NC, Resize64t16NC, Resize128t16NC },
04069       { Resize16t32, Resize32t32, Resize64t32NC, Resize128t32NC },
04070       { Resize16t64, Resize32t64, Resize64t64, Resize128t64NC },
04071       { Resize16t128, Resize32t128, Resize64t128, Resize128t128 } };
04072
04073     int endian, i;
04074     WORD idxW = CompactifySizeof(sizeof(WORD));
04075     WORD idxL = CompactifySizeof(sizeof(LONG));
04076     WORD idxP = CompactifySizeof(sizeof(POSITION));
04077     WORD idxVP = CompactifySizeof(sizeof(void *));
04078
04079     AO.transFlag = 0;
04080     AO.powerFlag = 0;
04081     AO.resizeFlag = 0;
04082     AO.bufferedInd = 0;
04083
04084     if ( ReadFile(AO.SaveData.Handle, (UBYTE *)(&AO.SaveHeader),
04085                 (LONG)sizeof(STOREHEADER)) != (LONG)sizeof(STOREHEADER) )
04086         return (MesPrint("Error reading save file header"));
04087
04088     /* check whether save-file has no header. if yes then it is an old version
04089        of FORM -> go back to position 0 in file which then contains the first
04090        index and skip the rest. */
04091     for ( i = 0; i < 8; ++i ) {
04092         if ( AO.SaveHeader.headermark[i] != 0xFF ) {
04093             POSITION p;
04094             PUTZERO(p);
04095             SeekFile(AO.SaveData.Handle, &p, SEEK_SET);

```

```

04096         return ( 0 );
04097     }
04098 }
04099
04100 if ( AO.SaveHeader.revision != SAVEREVISION ) {
04101     return(MesPrint("Save file header from an old version. Cannot read this file."));
04102 }
04103
04104 endian = 1;
04105 for ( i = 1; i < (int)sizeof(int); ++i ) {
04106     endian <= 8;
04107     endian += i+1;
04108 }
04109 if ( ((char *)&endian)[0] < ((char *)&endian)[1] ) {
04110     /* this machine is big-endian */
04111     AO.ResizeData = ResizeDataBE;
04112 }
04113 else {
04114     /* this machine is little-endian */
04115     AO.ResizeData = ResizeDataLE;
04116 }
04117
04118 /* set AO.transFlag if ANY conversion has to be done later */
04119 if ( AO.SaveHeader.endianness[0] > AO.SaveHeader.endianness[1] ) {
04120     AO.transFlag = ( ((char *)&endian)[0] < ((char *)&endian)[1] );
04121 }
04122 else {
04123     AO.transFlag = ( ((char *)&endian)[0] > ((char *)&endian)[1] );
04124 }
04125 if ( (WORD)AO.SaveHeader.lenWORD != sizeof(WORD) ) AO.transFlag |= 0x02;
04126 if ( (WORD)AO.SaveHeader.lenLONG != sizeof(LONG) ) AO.transFlag |= 0x04;
04127 if ( (WORD)AO.SaveHeader.lenPOS != sizeof(POSITION) ) AO.transFlag |= 0x08;
04128 if ( (WORD)AO.SaveHeader.lenPOINTER != sizeof(void *) ) AO.transFlag |= 0x10;
04129
04130 AO.FlipWORD = flipJumpTable[idxW];
04131 AO.FlipLONG = flipJumpTable[idxL];
04132 AO.FlipPOS = flipJumpTable[idxP];
04133 AO.FlipPOINTER = flipJumpTable[idxVP];
04134
04135 /* Works only for machines where WORD is not greater than 32bit ! */
04136 AO.CheckPower = CheckPower32;
04137 AO.RenumberVec = RenumberVec32;
04138
04139 AO.ResizeWORD = resizeJumpTable[idxW][CompactifySizeof(AO.SaveHeader.lenWORD)];
04140 AO.ResizeNCWORD = resizeNCJumpTable[idxW][CompactifySizeof(AO.SaveHeader.lenWORD)];
04141 AO.ResizeLONG = resizeJumpTable[idxL][CompactifySizeof(AO.SaveHeader.lenLONG)];
04142 AO.ResizePOS = resizeJumpTable[idxP][CompactifySizeof(AO.SaveHeader.lenPOS)];
04143 AO.ResizePOINTER = resizeJumpTable[idxVP][CompactifySizeof(AO.SaveHeader.lenPOINTER)];
04144
04145 {
04146     WORD dumw[8];
04147     UBYTE *dummy;
04148     for ( i = 0; i < 8; i++ ) dumw[i] = 0;
04149     dummy = (UBYTE *)dumw;
04150     for ( i = 0; i < 16; i++ ) dummy[i] = AO.SaveHeader.maxpower[i];
04151     AO.mpower = dumw[0];
04152 }
04153
04154 return ( 0 );
04155 }
04156
04157 /*
04158     #] ReadSaveHeader :
04159     #[ ReadSaveIndex :
04160 */
04161
04175 LONG ReadSaveIndex(FILEINDEX *fileind)
04176 {
04177     /* do we need some translation for the FILEINDEX? */
04178     if ( AO.transFlag ) {
04179         /* if a translated FILEINDEX can hold less entries than the original
04180            FILEINDEX, then we need to buffer the extra entries in this static
04181            variable (can happen going from 32bit to 64bit */
04182         static FILEINDEX sbuffer;
04183
04184         FILEINDEX buffer;
04185         UBYTE *p, *q;
04186         int i;
04187
04188         /* shortcuts */
04189         int lenW = AO.SaveHeader.lenWORD;
04190         int lenL = AO.SaveHeader.lenLONG;
04191         int lenP = AO.SaveHeader.lenPOS;
04192
04193         /* if we have a buffered FILEINDEX then just return it */
04194         if ( AO.bufferedInd ) {
04195             *fileind = sbuffer;

```

```

04196         AO.bufferedInd = 0;
04197         return ( 0 );
04198     }
04199
04200     if ( ReadFile(AO.SaveData.Handle, (UBYTE *)fileind, sizeof(FILEINDEX))
04201         != sizeof(FILEINDEX) ) {
04202         return ( MesPrint("Error(1) reading stored expression.") );
04203     }
04204
04205     /* do we need to flip the endianness? */
04206     if ( AO.transFlag & 1 ) {
04207         LONG number;
04208         /* padding bytes */
04209         int padp = lenL - ((lenW*5+(MAXENAME + 1)) & (lenL-1));
04210         p = (UBYTE *)fileind;
04211         AO.FlipPOS(p); p += lenP;          /* next */
04212         AO.FlipPOS(p);                    /* number */
04213         AO.ResizePOS(p, (UBYTE *)&number);
04214         p += lenP;
04215         for ( i = 0; i < number; ++i ) {
04216             AO.FlipPOS(p); p += lenP;      /* position */
04217             AO.FlipPOS(p); p += lenP;      /* length */
04218             AO.FlipPOS(p); p += lenP;      /* variables */
04219             AO.FlipLONG(p); p += lenL;      /* CompressSize */
04220             AO.FlipWORD(p); p += lenW;      /* nsymbols */
04221             AO.FlipWORD(p); p += lenW;      /* nindices */
04222             AO.FlipWORD(p); p += lenW;      /* nvectors */
04223             AO.FlipWORD(p); p += lenW;      /* nfunctions */
04224             AO.FlipWORD(p); p += lenW;      /* size */
04225             p += padp;
04226         }
04227     }
04228
04229     /* do we need to resize data? */
04230     if ( AO.transFlag > 1 ) {
04231         LONG number, maxnumber;
04232         int n;
04233         /* padding bytes */
04234         int padp = lenL - ((lenW*5+(MAXENAME + 1)) & (lenL-1));
04235         int padq = sizeof(LONG) - ((sizeof(WORD)*5+(MAXENAME + 1)) & (sizeof(LONG)-1));
04236
04237         p = (UBYTE *)fileind; q = (UBYTE *)&buffer;
04238         AO.ResizePOS(p, q);                /* next */
04239         p += lenP; q += sizeof(POSITION);
04240         AO.ResizePOS(p, q);                /* number */
04241         p += lenP;
04242         number = BASEPOSITION(*((POSITION *)q));
04243         /* if FILEINDEX in file contains more entries than the FILEINDEX in
04244            memory can contain, then adjust the numbers and prepare for
04245            buffering */
04246         if ( number > (LONG)INFILEINDEX ) {
04247             AO.bufferedInd = number-INFILEINDEX;
04248             if ( AO.bufferedInd > (WORD)INFILEINDEX ) {
04249                 /* can happen when reading 32bit and writing >=128bit.
04250                    Fix: more than one static buffer for FILEINDEX */
04251                 return ( MesPrint("Too many index entries.") );
04252             }
04253             maxnumber = INFILEINDEX;
04254             SETBASEPOSITION(*((POSITION *)q), INFILEINDEX);
04255         }
04256         else {
04257             maxnumber = number;
04258         }
04259         q += sizeof(POSITION);
04260         /* read all INDEXENTRY that fit into the output buffer */
04261         for ( i = 0; i < maxnumber; ++i ) {
04262             AO.ResizePOS(p, q);                /* position */
04263             p += lenP; q += sizeof(POSITION);
04264             AO.ResizePOS(p, q);                /* length */
04265             p += lenP; q += sizeof(POSITION);
04266             AO.ResizePOS(p, q);                /* variables */
04267             p += lenP; q += sizeof(POSITION);
04268             AO.ResizeLONG(p, q);                /* CompressSize */
04269             p += lenL; q += sizeof(LONG);
04270             AO.ResizeWORD(p, q);                /* nsymbols */
04271             p += lenW; q += sizeof(WORD);
04272             AO.ResizeWORD(p, q);                /* nindices */
04273             p += lenW; q += sizeof(WORD);
04274             AO.ResizeWORD(p, q);                /* nvectors */
04275             p += lenW; q += sizeof(WORD);
04276             AO.ResizeWORD(p, q);                /* nfunctions */
04277             p += lenW; q += sizeof(WORD);
04278             AO.ResizeWORD(p, q);                /* size (unchanged!) */
04279             p += lenW; q += sizeof(WORD);
04280             n = MAXENAME + 1;
04281             NCOPYB(q, p, n)
04282             p += padp;

```

```

04283         q += padq;
04284     }
04285     /* read all the remaining INDEXENTRY and put them into the static buffer */
04286     if ( AO.bufferedInd ) {
04287         sbuffer.next = buffer.next;
04288         SETBASEPOSITION(sbuffer.number, AO.bufferedInd);
04289         q = (UBYTE *)&sbuffer + sizeof(POSITION) + sizeof(LONG);
04290         for ( i = maxnumber; i < number; ++i ) {
04291             AO.ResizePOS(p, q);          /* position */
04292             p += lenP; q += sizeof(POSITION);
04293             AO.ResizePOS(p, q);          /* length */
04294             p += lenP; q += sizeof(POSITION);
04295             AO.ResizePOS(p, q);          /* variables */
04296             p += lenP; q += sizeof(POSITION);
04297             AO.ResizeLONG(p, q);         /* CompressSize */
04298             p += lenL; q += sizeof(LONG);
04299             AO.ResizeWORD(p, q);         /* nsymbols */
04300             p += lenW; q += sizeof(WORD);
04301             AO.ResizeWORD(p, q);         /* nindices */
04302             p += lenW; q += sizeof(WORD);
04303             AO.ResizeWORD(p, q);         /* nvectors */
04304             p += lenW; q += sizeof(WORD);
04305             AO.ResizeWORD(p, q);         /* nfunctions */
04306             p += lenW; q += sizeof(WORD);
04307             AO.ResizeWORD(p, q);         /* size (unchanged!) */
04308             p += lenW; q += sizeof(WORD);
04309             n = MAXENAME + 1;
04310             NCOPYB(q, p, n)
04311             p += padp;
04312             q += padq;
04313         }
04314     }
04315     /* copy to output */
04316     p = (UBYTE *)fileind; q = (UBYTE *)&buffer; n = sizeof(FILEINDEX);
04317     NCOPYB(p, q, n)
04318 }
04319 return ( 0 );
04320 } else {
04321     return ( ReadFile(AO.SaveData.Handle, (UBYTE *)fileind, sizeof(FILEINDEX))
04322             != sizeof(FILEINDEX) );
04323 }
04324 }
04325
04326 /*
04327     #] ReadSaveIndex :
04328     #[ ReadSaveVariables :
04329 */
04330
04361 LONG ReadSaveVariables(UBYTE *buffer, UBYTE *top, LONG *size, LONG *outsize, \
04362                        INDEXENTRY *ind, LONG *stage)
04363 {
04364     /* do we need some translation for the variables? */
04365     if ( AO.transFlag ) {
04366         /* counters for the number of already read symbols, indices, ... that
04367            need to remain valid between different calls to ReadSaveVariables().
04368            are initialized if stage == -1 */
04369         static WORD numReadSym;
04370         static WORD numReadInd;
04371         static WORD numReadVec;
04372         static WORD numReadFun;
04373
04374         POSITION pos;
04375         UBYTE *in, *out, *pp = 0, *end, *outbuf;
04376         LONG numread;
04377         WORD namelen, realnamelen;
04378         /* shortcuts */
04379         WORD lenW = AO.SaveHeader.lenWORD;
04380         WORD lenL = AO.SaveHeader.lenLONG;
04381         WORD lenP = AO.SaveHeader.lenPOINTER;
04382         WORD flip = AO.transFlag & 1;
04383
04384         /* remember file position in case we have to rewind */
04385         TELLFILE(AO.SaveData.Handle, &pos);
04386
04387         /* decide on the position of the in and out buffers.
04388            if the input is "bigger" than the output, we resize in-place, i.e.
04389            we immediately overwrite the source data by the translated data. in
04390            and out buffers start at the same place.
04391            if not, we read from the end of the given buffer and write at the
04392            beginning. */
04393         if ( (lenW > (WORD)sizeof(WORD))
04394             || ( (lenW == (WORD)sizeof(WORD))
04395                 && ( (lenL > (WORD)sizeof(LONG))
04396                     || ( (lenL == (WORD)sizeof(LONG)) && lenP > (WORD)sizeof(void *) )
04397                 ) ) ) {
04398             in = out = buffer;

```



```

04400         end = buffer + *size;
04401     }
04402     else {
04403         /* data will grow roughly by sizeof(WORD)/lenW. the exact value is
04404            not important. if reading and writing areas start to overlap, the
04405            reading will already be near the end of the data and overwriting
04406            doesn't matter. */
04407         LONG newsize = (top - buffer) / (1 + sizeof(WORD)/lenW);
04408         end = top;
04409         out = buffer;
04410         in = end - newsize;
04411         if ( *size > newsize ) *size = newsize;
04412     }
04413
04414     if ( ( numread = ReadFile(AO.SaveData.Handle, in, *size) ) != *size ) {
04415         return ( MesPrint("Error(2) reading stored expression.") );
04416     }
04417
04418     *size = 0;
04419     *outsize = 0;
04420
04421     /* first time in ReadSaveVariables(). initialize counters. */
04422     if ( *stage == -1 ) {
04423         numReadSym = 0;
04424         numReadInd = 0;
04425         numReadVec = 0;
04426         numReadFun = 0;
04427         ++*stage;
04428     }
04429
04430     while ( in < end ) {
04431         /* Symbols */
04432         if ( *stage == 0 ) {
04433             if ( ind->nsymbols <= numReadSym ) {
04434                 ++*stage;
04435                 continue;
04436             }
04437             if ( end - in < AO.SaveHeader.sSym ) {
04438                 goto RSVEnd;
04439             }
04440             if ( flip ) {
04441                 pp = in;
04442                 AO.FlipLONG(pp); pp += lenL;
04443                 while ( pp < in + AO.SaveHeader.sSym ) {
04444                     AO.FlipWORD(pp); pp += lenW;
04445                 }
04446             }
04447             pp = in + AO.SaveHeader.sSym;
04448             AO.ResizeLONG(in, out); in += lenL; out += sizeof(LONG); /* name */
04449             AO.CheckPower(in);
04450             AO.ResizeWORD(in, out); in += lenW;
04451             if ( *((WORD *)out) == -AO.mpower ) *((WORD *)out) = -MAXPOWER;
04452             out += sizeof(WORD); /* minpower */
04453             AO.ResizeWORD(in, out); in += lenW;
04454             if ( *((WORD *)out) == AO.mpower ) *((WORD *)out) = MAXPOWER;
04455             out += sizeof(WORD); /* maxpower */
04456             AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* complex */
04457             AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* number */
04458             AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* flags */
04459             AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* node */
04460             AO.ResizeWORD(in, out); in += lenW; /* namesize */
04461             realnamelen = *((WORD *)out);
04462             realnamelen += sizeof(void *)-1; realnamelen &= -(sizeof(void *));
04463             out += sizeof(WORD);
04464             AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* dimension */
04465             while ( in < pp ) {
04466                 AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD);
04467             }
04468             namelen = *((WORD *)out-1); /* cares for padding "bug" */
04469             if ( end - in < namelen ) {
04470                 goto RSVEnd;
04471             }
04472             *((WORD *)out-1) = realnamelen;
04473             *size += AO.SaveHeader.sSym + namelen;
04474             *outsize += sizeof(struct SyMbol) + realnamelen;
04475             if ( realnamelen > namelen ) {
04476                 int j = namelen;
04477                 NCOPYB(out, in, j);
04478                 out += realnamelen - namelen;
04479             }
04480             else {
04481                 int j = realnamelen;
04482                 NCOPYB(out, in, j);
04483                 in += namelen - realnamelen;
04484             }
04485             ++numReadSym;
04486             continue;

```

```

04487     }
04488     /* Indices */
04489     if ( *stage == 1 ) {
04490         if ( ind->nindices <= numReadInd ) {
04491             ++stage;
04492             continue;
04493         }
04494         if ( end - in < AO.SaveHeader.sInd ) {
04495             goto RSVEnd;
04496         }
04497         if ( flip ) {
04498             pp = in;
04499             AO.FlipLONG(pp); pp += lenL;
04500             while ( pp < in + AO.SaveHeader.sInd ) {
04501                 AO.FlipWORD(pp); pp += lenW;
04502             }
04503         }
04504         pp = in + AO.SaveHeader.sInd;
04505         AO.ResizeLONG(in, out); in += lenL; out += sizeof(LONG); /* name */
04506         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* type */
04507         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* dimension */
04508         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* number */
04509         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* flags */
04510         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* nmin4 */
04511         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* node */
04512         AO.ResizeWORD(in, out); in += lenW; /* namesize */
04513         realnamelen = *((WORD *)out);
04514         realnamelen += sizeof(void *)-1; realnamelen &= -(sizeof(void *));
04515         out += sizeof(WORD);
04516         while ( in < pp ) {
04517             AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD);
04518         }
04519         namelen = *((WORD *)out-1); /* cares for padding "bug" */
04520         if ( end - in < namelen ) {
04521             goto RSVEnd;
04522         }
04523         *((WORD *)out-1) = realnamelen;
04524         *size += AO.SaveHeader.sInd + namelen;
04525         *outsiz += sizeof(struct InDeX) + realnamelen;
04526         if ( realnamelen > namelen ) {
04527             int j = namelen;
04528             NCOPYB(out, in, j);
04529             out += realnamelen - namelen;
04530         }
04531         else {
04532             int j = realnamelen;
04533             NCOPYB(out, in, j);
04534             in += namelen - realnamelen;
04535         }
04536         ++numReadInd;
04537         continue;
04538     }
04539     /* Vectors */
04540     if ( *stage == 2 ) {
04541         if ( ind->nvectors <= numReadVec ) {
04542             ++stage;
04543             continue;
04544         }
04545         if ( end - in < AO.SaveHeader.sVec ) {
04546             goto RSVEnd;
04547         }
04548         if ( flip ) {
04549             pp = in;
04550             AO.FlipLONG(pp); pp += lenL;
04551             while ( pp < in + AO.SaveHeader.sVec ) {
04552                 AO.FlipWORD(pp); pp += lenW;
04553             }
04554         }
04555         pp = in + AO.SaveHeader.sVec;
04556         AO.ResizeLONG(in, out); in += lenL; out += sizeof(LONG); /* name */
04557         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* complex */
04558         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* number */
04559         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* flags */
04560         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* node */
04561         AO.ResizeWORD(in, out); in += lenW; /* namesize */
04562         realnamelen = *((WORD *)out);
04563         realnamelen += sizeof(void *)-1; realnamelen &= -(sizeof(void *));
04564         out += sizeof(WORD);
04565         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* dimension */
04566         while ( in < pp ) {
04567             AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD);
04568         }
04569         namelen = *((WORD *)out-1); /* cares for padding "bug" */
04570         if ( end - in < namelen ) {
04571             goto RSVEnd;
04572         }
04573         *((WORD *)out-1) = realnamelen;

```

```

04574         *size += AO.SaveHeader.sVec + namelen;
04575         *outsize += sizeof(struct VeCtOr) + realnamelen;
04576         if ( realnamelen > namelen ) {
04577             int j = namelen;
04578             NCOPYB(out, in, j)
04579             out += realnamelen - namelen;
04580         }
04581         else {
04582             int j = realnamelen;
04583             NCOPYB(out, in, j)
04584             in += namelen - realnamelen;
04585         }
04586         ++numReadVec;
04587         continue;
04588     }
04589     /* Functions */
04590     if ( *stage == 3 ) {
04591         if ( ind->nfunctions <= numReadFun ) {
04592             ++*stage;
04593             continue;
04594         }
04595         if ( end - in < AO.SaveHeader.sFun ) {
04596             goto RSVEnd;
04597         }
04598         if ( flip ) {
04599             pp = in;
04600             AO.FlipPOINTER(pp); pp += lenP;
04601             AO.FlipLONG(pp); pp += lenL;
04602             AO.FlipLONG(pp); pp += lenL;
04603             while ( pp < in + AO.SaveHeader.sFun ) {
04604                 AO.FlipWORD(pp); pp += lenW;
04605             }
04606         }
04607         pp = in + AO.SaveHeader.sFun;
04608         outbuf = out;
04609         AO.ResizePOINTER(in, out); in += lenP; out += sizeof(void *); /* tabl */
04610         AO.ResizeLONG(in, out); in += lenL; out += sizeof(LONG); /* symminfo */
04611         AO.ResizeLONG(in, out); in += lenL; out += sizeof(LONG); /* name */
04612         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* commute */
04613         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* complex */
04614         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* number */
04615         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* flags */
04616         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* spec */
04617         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* symmetric */
04618         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* numargs */
04619         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* node */
04620         AO.ResizeWORD(in, out); in += lenW; /* namesize */
04621         realnamelen = *((WORD *)out);
04622         realnamelen += sizeof(void *)-1; realnamelen &= -(sizeof(void *));
04623         out += sizeof(WORD);
04624         AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD); /* dimension */
04625         while ( in < pp ) {
04626             AO.ResizeWORD(in, out); in += lenW; out += sizeof(WORD);
04627         }
04628         namelen = *((WORD *)out-1); /* cares for padding "bug" */
04629         if ( end - in < namelen ) {
04630             goto RSVEnd;
04631         }
04632         *((WORD *)out-1) = realnamelen;
04633         *size += AO.SaveHeader.sFun + namelen;
04634         *outsize += sizeof(struct FuNcTiOn) + realnamelen;
04635         if ( realnamelen > namelen ) {
04636             int j = namelen;
04637             NCOPYB(out, in, j);
04638             out += realnamelen - namelen;
04639         }
04640         else {
04641             int j = realnamelen;
04642             NCOPYB(out, in, j);
04643             in += namelen - realnamelen;
04644         }
04645         ++numReadFun;
04646         /* we use the information whether a function is tensorial later in ReadSaveTerm */
04647         AO.tensorList[ ((FUNCTIONS)outbuf)->number+FUNCTION ] =
04648             (UBYTE)((FUNCTIONS)outbuf)->spec == TENSORFUNCTION);
04649         continue;
04650     }
04651     /* handle numdummies */
04652     if ( end - in >= lenW ) {
04653         if ( flip ) AO.FlipWORD(in);
04654         AO.ResizeWORD(in, out);
04655         *size += lenW;
04656         *outsize += sizeof(WORD);
04657     }
04658     /* handle numfactors */
04659     if ( end - in >= lenW ) {
04660         if ( flip ) AO.FlipWORD(in);

```

```

04661         AO.ResizeWORD(in, out);
04662         *size += lenW;
04663         *outsize += sizeof(WORD);
04664     }
04665     /* handle vflags */
04666     if ( end - in >= lenW ) {
04667         if ( flip ) AO.FlipWORD(in);
04668         AO.ResizeWORD(in, out);
04669         *size += lenW;
04670         *outsize += sizeof(WORD);
04671     }
04672     /* handle uflags */
04673     if ( end - in >= lenW ) {
04674         if ( flip ) AO.FlipWORD(in);
04675         AO.ResizeWORD(in, out);
04676         *size += lenW;
04677         *outsize += sizeof(WORD);
04678     }
04679     return ( 0 );
04680 }
04681
04682 RSVEnd:
04683     /* we are here because the remaining buffer cannot hold the next
04684     struct. we position the file behind the last successfully translated
04685     struct and return. */
04686     ADDPOS(pos, *size);
04687     SeekFile(AO.SaveData.Handle, &pos, SEEK_SET);
04688     return ( 0 );
04689 } else {
04690     return ( ReadFile(AO.SaveData.Handle, buffer, *size) != *size );
04691 }
04692 }
04693
04694 /*
04695     #] ReadSaveVariables :
04696     #[ ReadSaveTerm :
04697 */
04698
04727 UBYTE *
04728 ReadSaveTerm32(UBYTE *bin, UBYTE *binend, UBYTE **bout, UBYTE *boutend, UBYTE *top, int terminbuf)
04729 {
04730     GETIDENTITY
04731
04732     UBYTE *boutbuf;
04733     int32_t len, j, id;
04734     int32_t *r, *t, *coeff, *end, *newtermsize, *rend;
04735     int32_t *newsubterm;
04736     int32_t *in = (int32_t *)bin;
04737     int32_t *out = (int32_t *)bout;
04738
04739     /* if called recursively the term is already decompressed in buffer.
04740     is this the case? */
04741     if ( terminbuf ) {
04742         /* don't do any decompression, just adjust the pointers */
04743         len = *out;
04744         end = out + len;
04745         r = in + 1;
04746         rend = (int32_t *)boutend;
04747         coeff = end - ABS(*end-1);
04748         newtermsize = (int32_t *)bout;
04749         out = newtermsize + 1;
04750     }
04751     else {
04752         /* do decompression of necessary. always return if the space in the
04753         buffer is not sufficient */
04754         int32_t rbuf;
04755         r = (int32_t *)AR.CompressBuffer;
04756         rbuf = *r;
04757         len = j = *in;
04758         /* first copy from AR.CompressBuffer if necessary */
04759         if ( j < 0 ) {
04760             ++in;
04761             if ( (UBYTE *)in >= binend ) {
04762                 return ( bin );
04763             }
04764             *out = len = -j + 1 + *in;
04765             end = out + *out;
04766             if ( (UBYTE *)end >= top ) {
04767                 return ( bin );
04768             }
04769             ++out;
04770             *r++ = len;
04771             while ( ++j <= 0 ) {
04772                 int32_t bb = *r++;
04773                 *out++ = bb;
04774             }
04775             j = *in++;

```

```

04776     }
04777     else if ( j == 0 ) {
04778         /* care for padding words */
04779         while ( (UBYTE *)in < binend ) {
04780             *out++ = 0;
04781             if ( (UBYTE *)out > top ) {
04782                 return ( (UBYTE *)bin );
04783             }
04784
04785             *r++ = 0;
04786             ++in;
04787         }
04788         *bout = (UBYTE *)out;
04789         return ( (UBYTE *)in );
04790     }
04791     else {
04792         end = out + len;
04793         if ( (UBYTE *)end >= top ) {
04794             return ( bin );
04795         }
04796     }
04797     if ( (UBYTE *)in + j >= binend ) {
04798         *AR.CompressBuffer = rbuf;
04799         return ( bin );
04800     }
04801     if ( (UBYTE *)out + j >= top ) {
04802         return ( bin );
04803     }
04804     /* second copy from input buffer */
04805     while ( --j >= 0 ) {
04806         int32_t bb = *in++;
04807         *r++ = *out++ = bb;
04808     }
04809
04810     rend = r;
04811     r = (int32_t *)AR.CompressBuffer + 1;
04812     coeff = end - ABS(*end-1);
04813     newtermsize = (int32_t *)*bout;
04814     out = newtermsize + 1;
04815 }
04816
04817 /* iterate over subterms */
04818 while ( out < coeff ) {
04819
04820     id = *out++;
04821     ++r;
04822     t = out + *out - 1;
04823     newsubtermp = out;
04824     ++out; ++r;
04825
04826     if ( id == SYMBOL ) {
04827         while ( out < t ) {
04828             ++out; ++r; /* symbol number */
04829             /* if exponent is too big, rewrite as exponent function */
04830             if ( ABS(*out) >= MAXPOWER ) {
04831                 int32_t *a, *b;
04832                 int32_t n;
04833                 int32_t num = *(out-1);
04834                 int32_t exp = *out;
04835                 coeff += 9;
04836                 end += 9;
04837                 t += 9;
04838                 if ( (UBYTE *)end > top ) return ( bin );
04839                 out -= 3;
04840                 *out++ = EXPONENT; /* id */
04841                 *out++ = 13; /* size */
04842                 *out++ = 1; /* dirtyflag */
04843                 *out++ = -SYMBOL; /* first short arg */
04844                 *out++ = num;
04845                 *out++ = 8; /* second arg, size */
04846                 *out++ = 0; /* dirtyflag */
04847                 *out++ = 6; /* term size */
04848                 *out++ = ABS(exp) & 0x000FFFFF;
04849                 *out++ = ABS(exp) » 16;
04850                 *out++ = 1;
04851                 *out++ = 0;
04852                 *out++ = ( exp < 0 ) ? -5 : 5;
04853                 a = ++r;
04854                 b = out;
04855                 n = rend - r;
04856                 NCOPYI32(b, a, n)
04857             }
04858             else {
04859                 ++out; ++r;
04860             }
04861         }
04862     }

```

```

04863     else if ( id == DOTPRODUCT ) {
04864         while ( out < t ) {
04865             AO.RenumberVec((UBYTE *)out); /* vector 1 */
04866             ++out; ++r;
04867             AO.RenumberVec((UBYTE *)out); /* vector 2 */
04868             ++out; ++r;
04869             /* if exponent is too big, rewrite as exponent function */
04870             if ( ABS(*out) >= MAXPOWER ) {
04871                 int32_t *a, *b;
04872                 int32_t n;
04873                 int32_t num1 = *(out-2);
04874                 int32_t num2 = *(out-1);
04875                 int32_t exp = *out;
04876                 coeff += 17;
04877                 end += 17;
04878                 t += 17;
04879                 if ( (UBYTE *)end > top ) return ( bin );
04880                 out -= 4;
04881                 *out++ = EXPONENT;      /* id */
04882                 *out++ = 22;            /* size */
04883                 *out++ = 1;            /* dirtyflag */
04884                 *out++ = 11;          /* first arg, size */
04885                 *out++ = 0;            /* dirtyflag */
04886                 *out++ = 9;            /* term size */
04887                 *out++ = DOTPRODUCT;  /* p1.p2 */
04888                 *out++ = 5;            /* subterm size */
04889                 *out++ = num1;         /* p1 */
04890                 *out++ = num2;         /* p2 */
04891                 *out++ = 1;            /* exponent */
04892                 *out++ = 1;            /* coeff */
04893                 *out++ = 1;
04894                 *out++ = 3;
04895                 *out++ = 8;            /* second arg, size */
04896                 *out++ = 0;            /* dirtyflag */
04897                 *out++ = 6;            /* term size */
04898                 *out++ = ABS(exp) & 0x000FFFFF;
04899                 *out++ = ABS(exp) » 16;
04900                 *out++ = 1;
04901                 *out++ = 0;
04902                 *out++ = ( exp < 0 ) ? -5 : 5;
04903                 a = ++r;
04904                 b = out;
04905                 n = rend - r;
04906                 NCOPYI32(b, a, n)
04907             }
04908             else {
04909                 ++out; ++r;
04910             }
04911         }
04912     }
04913     else if ( id == VECTOR ) {
04914         while ( out < t ) {
04915             AO.RenumberVec((UBYTE *)out); /* vector number */
04916             ++out; ++r;
04917             ++out; ++r; /* index, do nothing */
04918         }
04919     }
04920     else if ( id == INDEX ) {
04921         /* int32_t vectoroffset = -2 * *((int32_t *)AO.SaveHeader.wildoffset); */
04922         void *dummy = (void *)AO.SaveHeader.wildoffset; /* to remove a warning about
strict-aliasing rules in gcc */
04923         int32_t vectoroffset = -2 * *((int32_t *)dummy);
04924         while ( out < t ) {
04925             /* if there is a vector, renumber it */
04926             if ( *out < vectoroffset ) {
04927                 AO.RenumberVec((UBYTE *)out);
04928             }
04929             ++out; ++r;
04930         }
04931     }
04932     else if ( id == SUBEXPRESSION ) {
04933         /* nothing to translate */
04934         while ( out < t ) {
04935             ++out; ++r;
04936         }
04937     }
04938     else if ( id == DELTA ) {
04939         /* nothing to translate */
04940         r += t - out;
04941         out = t;
04942     }
04943     else if ( id == HAAKJE ) {
04944         /* nothing to translate */
04945         r += t - out;
04946         out = t;
04947     }
04948     else if ( id == GAMMA || id == LEVICIVITA || (id >= FUNCTION && AO.tensorList[id]) ) {

```

```

04949 /*      int32_t vectoroffset = -2 * *((int32_t *)AO.SaveHeader.wildoffset); */
04950 void *dummy = (void *)AO.SaveHeader.wildoffset; /* to remove a warning about
strict-aliasing rules in gcc */
04951 int32_t vectoroffset = -2 * *((int32_t *)dummy);
04952 while ( out < t ) {
04953     /* if there is a vector as an argument, renumber it */
04954     if ( *out < vectoroffset ) {
04955         AO.RenumberVec((UBYTE *)out);
04956     }
04957     ++out; ++r;
04958 }
04959 }
04960 else if ( id >= FUNCTION ) {
04961     int32_t *argEnd;
04962     UBYTE *newbin;
04963
04964     ++out; ++r; /* dirty flags */
04965
04966     /* loop over arguments */
04967     while ( out < t ) {
04968         if ( *out < 0 ) {
04969             /* short notation arguments */
04970             switch ( -*out ) {
04971                 case SYMBOL:
04972                     ++out; ++r;
04973                     ++out; ++r;
04974                     break;
04975                 case SNUMBER:
04976                     argEnd = out+2;
04977                     ++out; ++r;
04978                     if ( sizeof(WORD) == 2 ) {
04979                         /* resize if needed */
04980                         if ( *out > (1<15)-1 || *out < -(1<15)+1 ) {
04981                             int32_t *a, *b;
04982                             int32_t n;
04983                             int32_t num = *out;
04984                             coeff += 6;
04985                             end += 6;
04986                             argEnd += 6;
04987                             t += 6;
04988                             if ( (UBYTE *)end > top ) return ( bin );
04989                             --out;
04990                             *out++ = 8;          /* argument size */
04991                             *out++ = 0;          /* dirtyflag */
04992                             *out++ = 6;          /* term size */
04993                             *out++ = ABS(num) & 0x0000FFFF;
04994                             *out++ = ABS(num) » 16;
04995                             *out++ = 1;
04996                             *out++ = 0;
04997                             *out++ = ( num < 0 ) ? -5 : 5;
04998                             a = ++r;
04999                             b = out;
05000                             n = rend - r;
05001                             NCOPYI32(b, a, n)
05002                         }
05003                     }
05004                     else {
05005                         ++out; ++r;
05006                     }
05007                 }
05008             }
05009             break;
05010         case VECTOR:
05011             ++out; ++r;
05012             AO.RenumberVec((UBYTE *)out);
05013             ++out; ++r;
05014             break;
05015         case INDEX:
05016             ++out; ++r;
05017             ++out; ++r;
05018             break;
05019         case MINVECTOR:
05020             ++out; ++r;
05021             AO.RenumberVec((UBYTE *)out);
05022             ++out; ++r;
05023             break;
05024         default:
05025             if ( -*out >= FUNCTION ) {
05026                 ++out; ++r;
05027                 break;
05028             } else {
05029                 MesPrint("short function code %d not implemented.", *out);
05030                 return ( (UBYTE *)in );
05031             }
05032     }
05033 }
05034 }

```

```

05035         else {
05036             /* long arguments */
05037             int32_t *newargsize = out;
05038             argEnd = out + *out;
05039             ++out; ++r;
05040             ++out; ++r; /* dirty flags */
05041             while ( out < argEnd ) {
05042                 int32_t *keepsizes = out + *out;
05043                 int32_t lenbuf = *out;
05044                 int32_t **ppp = &out; /* to avoid a compiler warning */
05045                 /* recursion */
05046                 newbin = ReadSaveTerm32((UBYTE *)r, binend, (UBYTE **)ppp, (UBYTE *)rend, top,
1);
05047                 r += lenbuf;
05048                 if ( newbin == (UBYTE *)r ) {
05049                     return ( (UBYTE *)in );
05050                 }
05051                 /* if the term done by recursion has changed in size,
05052                    we need to move the rest of the data accordingly */
05053                 if ( out > keepsizes ) {
05054                     int32_t *a, *b;
05055                     int32_t n;
05056                     int32_t extention = out - keepsizes;
05057                     a = r;
05058                     b = out;
05059                     n = rend - r;
05060                     NCOPYI32(b, a, n)
05061                     coeff += extention;
05062                     end += extention;
05063                     argEnd += extention;
05064                     t += extention;
05065                 }
05066                 else if ( out < keepsizes ) {
05067                     int32_t *a, *b;
05068                     int32_t n;
05069                     int32_t extention = keepsizes - out;
05070                     a = keepsizes;
05071                     b = out;
05072                     n = rend - r;
05073                     NCOPYI32(b, a, n)
05074                     coeff -= extention;
05075                     end -= extention;
05076                     argEnd -= extention;
05077                     t -= extention;
05078                 }
05079             }
05080             *newargsize = out - newargsize;
05081         }
05082     }
05083 }
05084 else {
05085     MesPrint("ID %d not recognized.", id);
05086     return ( (UBYTE *)in );
05087 }
05088
05089 *newsubterm = out - newsubterm + 1;
05090 }
05091
05092 if ( (UBYTE *)end >= top ) {
05093     return ( bin );
05094 }
05095
05096 /* do coefficient and adjust term size */
05097 boutbuf = *bout;
05098 *bout = (UBYTE *)out;
05099
05100 ResizeCoeff32(bout, (UBYTE *)end, top);
05101
05102 if ( *bout >= top ) {
05103     *bout = boutbuf;
05104     return ( bin );
05105 }
05106
05107 *newtermsize = (int32_t *)*bout - newtermsize;
05108
05109 return ( (UBYTE *)in );
05110 }
05111
05112 /*
05113     #] ReadSaveTerm :
05114     #[ ReadSaveExpression :
05115 */
05116
05138 LONG ReadSaveExpression(UBYTE *buffer, UBYTE *top, LONG *size, LONG *outsize)
05139 {
05140     if ( AO.transFlag ) {
05141         UBYTE *in, *end, *out, *outend, *p;

```



```

05142     POSITION pos;
05143     LONG half;
05144     WORD lenW = AO.SaveHeader.lenWORD;
05145
05146     /* remember the last file position in case an expression cannot be
05147        fully processed */
05148     TELLFILE(AO.SaveData.Handle,&pos);
05149
05150     /* adjust 'size' depending on whether the translated data is bigger or
05151        smaller */
05152     half = (top-buffer)/2;
05153     if ( *size > half ) *size = half;
05154     if ( lenW < (WORD)sizeof(WORD) ) {
05155         if ( *size * (LONG)sizeof(WORD)/lenW > half ) *size = half*lenW/(LONG)sizeof(WORD);
05156     }
05157     else {
05158         if ( *size > half ) *size = half;
05159     }
05160
05161     /* depending on the necessary resizing we position the input pointer
05162        either at the start of the buffer or in the middle. if the data will
05163        roughly remain the same size, we need only one processing step, so
05164        we put the 'in' at the middle and 'out' and the beginning. in the
05165        other cases we need two processing steps, so first we put 'in' at
05166        the beginning and write at the middle. the second step can then read
05167        from the middle and put its results at the beginning. */
05168     in = out = buffer;
05169     if ( lenW == sizeof(WORD) ) in += half;
05170     else out += half;
05171     end = in + *size;
05172     outend = out + *size;
05173
05174     if ( ReadFile(AO.SaveData.Handle, in, *size) != *size ) {
05175         return ( MesPrint("Error(3) reading stored expression.") );
05176     }
05177
05178     if ( AO.transFlag & 1 ) {
05179         p = in;
05180         end -= lenW;
05181         while ( p <= end ) {
05182             AO.FlipWORD(p); p += lenW;
05183         }
05184         end += lenW;
05185     }
05186
05187     if ( lenW > (WORD)sizeof(WORD) ) {
05188         /* renumber first */
05189         do {
05190             outend = out+*size;
05191             if ( outend > top ) outend = top;
05192             p = ReadSaveTerm32(in, end, &out, outend, top, 0);
05193             if ( p == in ) break;
05194             in = p;
05195         } while ( in <= end - lenW );
05196         /* then resize */
05197         *size = in - buffer;
05198         in = buffer + half;
05199         end = out;
05200         out = buffer;
05201
05202         while ( in < end ) {
05203             /* resize without checking */
05204             AO.ResizeNCWORD(in, out);
05205             in += lenW; out += sizeof(WORD);
05206         }
05207     }
05208     else {
05209         if ( lenW < (WORD)sizeof(WORD) ) {
05210             /* resize first */
05211             while ( in < end ) {
05212                 AO.ResizeWORD(in, out);
05213                 in += lenW; out += sizeof(WORD);
05214             }
05215             in = buffer + half;
05216             end = out;
05217             out = buffer;
05218         }
05219         /* then renumber */
05220         do {
05221             p = ReadSaveTerm32(in, end, &out, buffer+half, buffer+half, 0);
05222             if ( p == in ) break;
05223             in = p;
05224         } while ( in <= end - sizeof(WORD) );
05225         *size = (in - buffer - half) * lenW / (ULONG)sizeof(WORD);
05226     }
05227     *outsize = out - buffer;
05228     ADDPOS(pos, *size);

```

```

05229         SeekFile(AO.SaveData.Handle, &pos, SEEK_SET);
05230
05231         return ( 0 );
05232     }
05233     else {
05234         return ( ReadFile(AO.SaveData.Handle, buffer, *size) != *size );
05235     }
05236 }
05237
05238 /*
05239     #] ReadSaveExpression :
05240     #] System Independent Saved Expressions :
05241 */

```

4.137 structs.h File Reference

This graph shows which files directly or indirectly include this file:



Data Structures

- struct [PoSiTiOn](#)
- struct [STOREHEADER](#)
- struct [InDeXeNtRy](#)
- struct [FiLeInDeX](#)
- struct [FiLeDaTa](#)
- struct [VaRrEnUm](#)
- struct [ReNuMbEr](#)
- struct [LIST](#)
- struct [KEYWORD](#)
- struct [KEYWORDV](#)
- struct [NaMeNode](#)
- struct [NaMeTree](#)
- struct [tree](#)
- struct [MiNmAx](#)
- struct [BrAcKeTiNdEx](#)
- struct [BrAcKeTiNfO](#)
- struct [TaBIes](#)
- struct [ExPrEsSiOn](#)
- struct [SyMbOl](#)
- struct [InDeX](#)
- struct [VeCtOr](#)
- struct [FuNcTiOn](#)
- struct [SeTs](#)
- struct [DuBiOuS](#)
- struct [FaCdOILaR](#)
- struct [DoLIaRS](#)
- struct [MoDoPtDoLIaRS](#)
- struct [fixedset](#)
- struct [TaBIeBaSesUbInDeX](#)
- struct [TaBIeBaSE](#)
- struct [FUN_INFO](#)
- struct [PaRtlcLe](#)

- struct [VeRtEx](#)
- struct [MoDeL](#)
- struct [NaMeSpAcE](#)
- struct [FiLe](#)
- struct [StreaM](#)
- struct [SpecTatoR](#)
- struct [TrAcEs](#)
- struct [TrAcEn](#)
- struct [pReVaR](#)
- struct [INSIDEINFO](#)
- struct [PRELOAD](#)
- struct [PROCEDURE](#)
- struct [DoLoOp](#)
- struct [bit_field](#)
- struct [HANDLERS](#)
- struct [CbUf](#)
- struct [ChAnNeL](#)
- struct [SETUPPARAMETERS](#)
- struct [NeStInG](#)
- struct [StOrEcAcHe](#)
- struct [PeRmUtE](#)
- struct [PeRmUtEp](#)
- struct [DiStRiBuTe](#)
- struct [PaRtl](#)
- struct [sOrT](#)
- struct [POLYMOD](#)
- struct [SHvariables](#)
- struct [COST](#)
- struct [MODNUM](#)
- struct [OPTIMIZE](#)
- struct [OPTIMIZERESULT](#)
- struct [DICTIONARY_ELEMENT](#)
- struct [DICTIONARY](#)
- struct [SWITCHTABLE](#)
- struct [SWITCH](#)
- struct [TERMINFO](#)
- struct [M_const](#)
- struct [P_const](#)
- struct [C_const](#)
- struct [S_const](#)
- struct [R_const](#)
- struct [T_const](#)
- struct [N_const](#)
- struct [O_const](#)
- struct [X_const](#)
- struct [AllGlobals](#)
- struct [FixedGlobals](#)

Macros

- #define [INFILEINDEX](#) ((512-2*sizeof([POSITION](#)))/sizeof([INDEXENTRY](#)))
- #define [EMPTYININDEX](#) (512-2*sizeof([POSITION](#))-[INFILEINDEX](#)*sizeof([INDEXENTRY](#)))
- #define [PHEAD](#)
- #define [PHEAD0](#) void
- #define [BHEAD](#)
- #define [BHEAD0](#)

Typedefs

- typedef struct [PoSiTiOn](#) **POSITION**
- typedef struct [InDeXeNtRy](#) **INDEXENTRY**
- typedef struct [FiLeInDeX](#) **FILEINDEX**
- typedef struct [FiLeDaTa](#) **FILEDATA**
- typedef struct [VaRrEnUm](#) **VARRENUM**
- typedef struct [ReNuMbEr](#) * **RENUMBER**
- typedef struct [NaMeNode](#) **NAMENODE**
- typedef struct [NaMeTree](#) **NAMETREE**
- typedef struct [tree](#) **COMPTREE**
- typedef struct [MiNmAx](#) **MINMAX**
- typedef struct [BrAcKeTiNdEx](#) **BRACKETINDEX**
- typedef struct [BrAcKeTiNfO](#) **BRACKETINFO**
- typedef struct [TaBIes](#) * **TABLES**
- typedef struct [ExPrEsSiOn](#) * **EXPRESSIONS**
- typedef struct [SyMbOl](#) * **SYMBOLS**
- typedef struct [InDeX](#) * **INDICES**
- typedef struct [VeCtOr](#) * **VECTORS**
- typedef struct [FuNcTiOn](#) * **FUNCTIONS**
- typedef struct [SeTs](#) * **SETS**
- typedef struct [DuBiOuS](#) * **DUBIOUSV**
- typedef struct [FaCdOlLaR](#) **FACDOLLAR**
- typedef struct [DoLIaRs](#) * **DOLLARS**
- typedef struct [MoDoPtDoLIaRs](#) **MODEPTDOLLAR**
- typedef struct [fixedset](#) **FIXEDSET**
- typedef struct [TaBIEbAsEsUbInDeX](#) **TABLEBASESUBINDEX**
- typedef struct [TaBIEbAsE](#) **TABLEBASE**
- typedef struct [PaRtlcLe](#) **PARTICLE**
- typedef struct [VeRtEx](#) **VERTEX**
- typedef struct [MoDeL](#) **MODEL**
- typedef struct [NaMeSpAcE](#) **NAMESPACE**
- typedef struct [FiLe](#) **FILEHANDLE**
- typedef struct [StreaM](#) **STREAM**
- typedef struct [SpecTatoR](#) **SPECTATOR**
- typedef struct [TrAcEs](#) **TRACES**
- typedef struct [TrAcEn](#) * **TRACEN**
- typedef struct [pReVaR](#) **PREVAR**
- typedef struct [DoLoOp](#) **DOLOOP**
- typedef struct [bit_field](#) [set_of_char](#)[32]
- typedef struct [bit_field](#) * [one_byte](#)
- typedef int(* [FINISHUFFLE](#)) (WORD *)
- typedef int(* [DO_UFFLE](#)) (WORD *, WORD, WORD, WORD)
- typedef WORD(* [COMPAREDUMMY](#)) (WORD *, WORD *, WORD)
- typedef struct [CbUf](#) **CBUF**
- typedef struct [ChAnNeL](#) **CHANNEL**
- typedef struct [NeStInG](#) * **NESTING**
- typedef struct [StOrEcAcHe](#) * **STORECACHE**
- typedef struct [PeRmUtE](#) **PERM**
- typedef struct [PeRmUtEp](#) **PERMP**
- typedef struct [DiStRiBuTe](#) **DISTRIBUTE**
- typedef struct [PaRtl](#) **PARTI**
- typedef struct [sOrT](#) **SORTING**
- typedef struct [AllGlobals](#) **ALLGLOBALS**
- typedef int(* [WCN](#)) (PHEAD WORD *, WORD *, WORD, WORD)
- typedef int(* [WCN2](#)) (PHEAD WORD *, WORD *)
- typedef WORD(* [COMPARE](#)) (PHEAD WORD *, WORD *, WORD)
- typedef struct [FixedGlobals](#) **FIXEDGLOBALS**

Functions

- **STATIC_ASSERT** (sizeof([STOREHEADER](#))==512)
- **STATIC_ASSERT** (sizeof([FILEINDEX](#))==512)

4.137.1 Detailed Description

Contains definitions for global structs.

!!!CAUTION!!! Changes in this file will most likely have consequences for the recovery mechanism (see [checkpoint.c](#)). You need to care for the code in [checkpoint.c](#) as well and modify the code there accordingly!

The marker [D] is used in comments in this file to mark pointers to which dynamically allocated memory is assigned by a call to malloc() during runtime (in contrast to pointers that point into already allocated memory). This information is especially helpful if one needs to know which pointers need to be freed (cf. [checkpoint.c](#)).

Definition in file [structs.h](#).

4.137.2 Macro Definition Documentation

4.137.2.1 INFILEINDEX

```
#define INFILEINDEX ((512-2*sizeof(POSITION))/sizeof(INDEXENTRY))
```

Maximum number of entries (struct [InDeXeNtRy](#)) in a file index (struct [FiLeInDeX](#)). Number is calculated such that the size of a file index is no more than 512 bytes.

Definition at line 118 of file [structs.h](#).

4.137.2.2 EMPTYININDEX

```
#define EMPTYININDEX (512-2*sizeof(POSITION)-INFILEINDEX*sizeof(INDEXENTRY))
```

Number of empty filling bytes for a file index (struct [FiLeInDeX](#)). It is calculated such that the size of a file index is always 512 bytes.

Definition at line 123 of file [structs.h](#).

4.137.2.3 PHEAD

```
#define PHEAD
```

Definition at line 2494 of file [structs.h](#).

4.137.2.4 PHEAD0

```
#define PHEAD0 void
```

Definition at line 2495 of file [structs.h](#).

4.137.2.5 BHEAD

```
#define BHEAD
```

Definition at line 2496 of file [structs.h](#).

4.137.2.6 BHEAD0

```
#define BHEAD0
```

Definition at line 2497 of file [structs.h](#).

4.137.3 Typedef Documentation

4.137.3.1 INDEXENTRY

```
typedef struct InDeXeNtRy INDEXENTRY
```

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

It represents one expression in the file.

4.137.3.2 FILEINDEX

```
typedef struct FiLeInDeX FILEINDEX
```

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

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

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

This struct is always 512 bytes long.

4.137.3.3 VARRENUM

```
typedef struct VaRrEnUm VARRENUM
```

Contains the pointers to an array in which a binary search will be performed.

4.137.3.4 RENUMBER

```
typedef struct ReNuMbEr * RENUMBER
```

Only symb.lo gets dynamically allocated. All other pointers points into this memory.

4.137.3.5 NAMENODE

```
typedef struct NaMeNode NAMENODE
```

The names of variables are kept in an array. Elements of type [NAMENODE](#) define a tree (that is kept balanced) that make it easy and fast to look for variables. See also [NAMETREE](#).

4.137.3.6 NAMETREE

```
typedef struct NaMeTree NAMETREE
```

A struct of type [NAMETREE](#) controls a complete (balanced) tree of names for the compiler. The compiler maintains several of such trees and the system has been set up in such a way that one could define more of them if we ever want to work with local name spaces.

4.137.3.7 COMPTREE

```
typedef struct tree COMPTREE
```

The subexpressions in the compiler are kept track of in a (balanced) tree to reduce the need for subexpressions and hence save much space in large rhs expressions (like when we have xxxxxx occurrences of objects like $f(x+1, x+1)$ in which each $x+1$ becomes a subexpression. The struct that controls this tree is COMPTREE.

4.137.3.8 TABLES

```
typedef struct TaBleS * TABLES
```

buffers, mm, flags, and prototype are always dynamically allocated, tablepointers only if needed (=0 if unallocated), boomlijst and argtail only for sparse tables.

Allocation is done for both the normal and the stub instance (spare), except for prototype and argtail which share memory.

4.137.3.9 FUNCTIONS

```
typedef struct FuNcTiOn * FUNCTIONS
```

Contains all information about a function. Also used for tables. It is used in the [LIST](#) elements of AC.

4.137.3.10 FILEHANDLE

```
typedef struct File FILEHANDLE
```

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

4.137.3.11 STREAM

```
typedef struct Stream STREAM
```

Input is read from 'streams' which are represented by objects of type STREAM. A stream can be a file, a do-loop, a procedure, the string value of a preprocessor variable When a new stream is opened we have to keep information about where to fall back in the parent stream to allow this to happen even in the middle of reading names etc as would be the case with a`i'b

4.137.3.12 TRACES

```
typedef struct TrAcEs TRACES
```

The struct TRACES keeps track of the progress during the expansion of a 4-dimensional trace. Each time a term gets generated the expansion tree continues in the next statement. When it returns it has to know where to continue. The 4-dimensional traces are more complicated than the n-dimensional traces (see TRACEN) because of the extra tricks that can be used. They are responsible for the shorter final expressions.

4.137.3.13 TRACEN

```
typedef struct TrAcEn * TRACEN
```

The struct TRACEN keeps track of the progress during the expansion of a 4-dimensional trace. Each time a term gets generated the expansion tree continues in the next statement. When it returns it has to know where to continue.

4.137.3.14 PREVAR

```
typedef struct pReVaR PREVAR
```

An element of the type PREVAR is needed for each preprocessor variable.

4.137.3.15 DOLOOP

```
typedef struct DoLoOp DOLOOP
```

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

4.137.3.16 set_of_char

```
typedef struct bit_field set_of_char[32]
```

Used in set_in, set_set, set_del and set_sub.

Definition at line [931](#) of file [structs.h](#).

4.137.3.17 one_byte

```
typedef struct bit_field* one_byte
```

Used in `set_in`, `set_set`, `set_del` and `set_sub`.

Definition at line 937 of file [structs.h](#).

4.137.3.18 FINISHUFFLE

```
typedef int(* FINISHUFFLE) (WORD *)
```

Definition at line 957 of file [structs.h](#).

4.137.3.19 DO_UFFLE

```
typedef int(* DO_UFFLE) (WORD *, WORD, WORD, WORD)
```

Definition at line 958 of file [structs.h](#).

4.137.3.20 COMPAREDUMMY

```
typedef WORD(* COMPAREDUMMY) (WORD *, WORD *, WORD)
```

Definition at line 963 of file [structs.h](#).

4.137.3.21 CBUF

```
typedef struct CbUf CBUF
```

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

4.137.3.22 CHANNEL

```
typedef struct ChAnNeL CHANNEL
```

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

4.137.3.23 NESTING

```
typedef struct NeStInG * NESTING
```

The NESTING struct is used when we enter the argument of functions and there is the possibility that we have to change something there. Because functions can be nested we have to keep track of all levels of functions in case we have to move the outer layers to make room for a larger function argument.

4.137.3.24 STORECACHE

```
typedef struct StOrEcAcHe * STORECACHE
```

The struct of type STORECACHE is used by a caching system for reading terms from stored expressions. Each thread should have its own system of caches.

4.137.3.25 PERM

```
typedef struct PeRmUte PERM
```

The struct PERM is used to generate all permutations when the pattern matcher has to try to match (anti)symmetric functions.

4.137.3.26 PERMP

```
typedef struct PeRmUtEp PERMP
```

Like struct PERM but works with pointers.

4.137.3.27 DISTRIBUTE

```
typedef struct DiStRiBuTe DISTRIBUTE
```

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

4.137.3.28 PARTI

```
typedef struct PaRtI PARTI
```

The struct PARTI is used to help determining whether a partition_ function can be replaced.

4.137.3.29 SORTING

```
typedef struct sOrT SORTING
```

The struct SORTING is used to control a sort operation. It includes a small and a large buffer and arrays for keeping track of various stages of the (merge) sorts. Each sort level has its own struct and different levels can have different sizes for its arrays. Also different threads have their own set of SORTING structs.

4.137.3.30 ALLGLOBALS

```
typedef struct AllGlobals ALLGLOBALS
```

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

4.137.3.31 WCN

```
typedef int(* WCN) (PHEAD WORD *, WORD *, WORD, WORD)
```

Definition at line 2500 of file [structs.h](#).

4.137.3.32 WCN2

```
typedef int(* WCN2) (PHEAD WORD *, WORD *)
```

Definition at line 2501 of file [structs.h](#).

4.137.3.33 COMPARE

```
typedef WORD(* COMPARE) (PHEAD WORD *, WORD *, WORD)
```

Definition at line 2503 of file [structs.h](#).

4.137.3.34 FIXEDGLOBALS

```
typedef struct FixedGlobals FIXEDGLOBALS
```

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

4.138 structs.h

[Go to the documentation of this file.](#)

```
00001
00017 /* # [ License : */
00018 /*
00019 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00020 * When using this file you are requested to refer to the publication
00021 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00022 * This is considered a matter of courtesy as the development was paid
00023 * for by FOM the Dutch physics granting agency and we would like to
00024 * be able to track its scientific use to convince FOM of its value
00025 * for the community.
00026 *
00027 * This file is part of FORM.
00028 *
00029 * FORM is free software: you can redistribute it and/or modify it under the
00030 * terms of the GNU General Public License as published by the Free Software
00031 * Foundation, either version 3 of the License, or (at your option) any later
00032 * version.
00033 *
00034 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00035 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00036 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00037 * details.
00038 *
00039 * You should have received a copy of the GNU General Public License along
00040 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00041 */
00042 /* # ] License : */
00043
00044 #ifndef __STRUCTS__
```

```

00045
00046 #define __STRUCTS__
00047 #ifdef _MSC_VER
00048 #include <wchar.h> /* off_t */
00049 #endif
00050
00051 /*
00052  *#[ sav&store :
00053  */
00054
00055 typedef struct PoSiTiOn {
00060     off_t pl;
00061 } POSITION;
00062
00063 /* Next are the index structs for stored and saved expressions */
00064
00065 typedef struct {
00066     UBYTE headermark[8];
00067     UBYTE lenWORD;
00068     UBYTE lenLONG;
00069     UBYTE lenPOS;
00070     UBYTE lenPOINTER;
00071     UBYTE endianness[16];
00072     UBYTE sSym;
00073     UBYTE sInd;
00074     UBYTE sVec;
00075     UBYTE sFun;
00076     UBYTE maxpower[16];
00077     UBYTE wilddoffset[16];
00078     UBYTE revision;
00079     UBYTE reserved[512-8-4-16-4-16-16-1];
00080 } STOREHEADER;
00081
00082 STATIC_ASSERT(sizeof(STOREHEADER) == 512);
00083
00084 typedef struct InDeXeNtRy {
00085     POSITION position;
00086     POSITION length;
00087     POSITION variables;
00088     LONG CompressSize;
00089     WORD nsymbols;
00090     WORD nindices;
00091     WORD nvectors;
00092     WORD nfunctions;
00093     WORD size;
00094     SBYTE name[MAXENAME+1];
00095 } INDEXENTRY;
00096
00097 #define INFILEINDEX ((512-2*sizeof(POSITION))/sizeof(INDEXENTRY))
00098 #define EMPTYININDEX (512-2*sizeof(POSITION)-INFILEINDEX*sizeof(INDEXENTRY))
00099
00100 typedef struct FileInDeX {
00101     POSITION next;
00102     POSITION number;
00103     INDEXENTRY expression[INFILEINDEX];
00104     SBYTE empty[EMPTYININDEX];
00105 } FILEINDEX;
00106
00107 STATIC_ASSERT(sizeof(FILEINDEX) == 512);
00108
00109 typedef struct FileDaTa {
00110     FILEINDEX Index;
00111     POSITION Fill;
00112     POSITION Position;
00113     WORD Handle;
00114     WORD dirtyflag;
00115 } FILEDATA;
00116
00117 typedef struct VaRrEnUm {
00118     WORD *start;
00119     WORD *lo;
00120     WORD *hi;
00121 } VARRENUM;
00122
00123 typedef struct ReNuMbEr {
00124     POSITION startposition;
00125     /* First stage renumbering */
00126     VARRENUM symb;
00127     VARRENUM indi;
00128     VARRENUM vect;
00129     VARRENUM func;
00130     /* Second stage renumbering */
00131     WORD *symnum;
00132     WORD *indnum;
00133     WORD *vecnum;
00134     WORD *funnum;
00135 } *RENUMBER;

```

```

00189
00190 /*
00191     #] sav&store :
00192     #[ Variables :
00193 */
00194
00202 typedef struct {
00203     void *lijst;
00204     char *message;
00205     int num;
00206     int maxnum;
00207     int size;
00208     int numglobal;
00209     int numtemp;
00210     int numclear;
00211 } LIST;
00212
00218 typedef struct {
00219     char *name;
00220     TFUN func;
00221     int type;
00222     int flags;
00223 } KEYWORD;
00224
00230 typedef struct {
00231     char *name;
00232     int *var;
00233     int type;
00234     int flags;
00235 } KEYWORDV;
00236
00243 typedef struct NaMeNode {
00244     LONG name;
00245     WORD parent;
00246     WORD left;
00247     WORD right;
00248     WORD balance;
00249     WORD type;
00250     WORD number;
00251 } NAMENODE;
00252
00260 typedef struct NaMeTree {
00261     NAMENODE *namenode;
00262     UBYTE *namebuffer;
00263     LONG nodesize;
00264     LONG nodefill;
00265     LONG namesize;
00266     LONG namefill;
00267     LONG oldnamefill;
00268     LONG oldnodefill;
00269     LONG globalnamefill;
00270     LONG globalnodefill;
00271     LONG clearnamefill;
00272     LONG clearnodefill;
00273     WORD headnode;
00274 } NAMETREE;
00275
00282 typedef struct tree {
00283     int parent;
00284     int left;
00285     int right;
00286     int value;
00287     int blnce;
00288     int usage;
00289 } COMPTREE;
00290
00301 typedef struct MiNmAx {
00302     WORD mini;
00303     WORD maxi;
00304     WORD size;
00305 } MINMAX;
00306
00311 typedef struct BrAcKeTiNdEx { /* For indexing brackets in local expressions */
00312     POSITION start; /* Place where bracket starts - start of expr */
00313     POSITION next; /* Place of next indexed bracket in expr */
00314     LONG bracket; /* Offset of position in bracketbuffer */
00315     LONG termsinbracket;
00316 } BRACKETINDEX;
00317
00322 typedef struct BrAcKeTiNfO {
00323     BRACKETINDEX *indexbuffer;
00324     WORD *bracketbuffer;
00325     LONG bracketbuffersize;
00326     LONG indexbuffersize;
00327     LONG bracketfill;
00328     LONG indexfill;
00329     WORD SortType;

```

```

00330 } BRACKETINFO;
00331
00342 typedef struct TaBlEs {
00343     WORD    *tablepointers;
00344     #ifdef WITHPTHREADS
00345     WORD    **prototype;
00346     WORD    **pattern;
00347     #else
00348     WORD    *prototype;
00349     WORD    *pattern;
00350     #endif
00351     MINMAX  *mm;
00352     WORD    *flags;
00353     COMPTREE *boomlijst;
00354     UBYTE   *argtail;
00355     struct TaBlEs *spare;
00356     WORD    *buffers;
00357     LONG    totind;
00358     LONG    reserved;
00359     LONG    defined;
00360     LONG    mdefined;
00361     int     prototypeSize;
00362     int     numind;
00363     int     bounds;
00364     int     strict;
00365     int     sparse;
00366     int     numtree;
00367     int     rootnum;
00368     int     MaxTreeSize;
00369     WORD    bufnum;
00370     WORD    bufferssize;
00371     WORD    buffersfill;
00372     WORD    tablenum;
00373     WORD    mode;
00374     WORD    numdummies;
00375 } *TABLES;
00376
00377
00382 typedef struct ExprEssiOn {
00383     POSITION  onfile;
00384     POSITION  prototype;
00385     POSITION  size;
00386     RENUMBER renum; /* For Renumbering of global stored expressions */
00387     BRACKETINFO *bracketinfo;
00388     BRACKETINFO *newbracketinfo;
00389     WORD    *renumlists;
00390     WORD    *inmem; /* If in memory like e.g. a polynomial */
00391     LONG    counter;
00392     LONG    name;
00393     WORD    hidelevel;
00394     WORD    vflags; /* Various flags */
00395     WORD    printflag;
00396     WORD    status;
00397     WORD    replace;
00398     WORD    node;
00399     WORD    whichbuffer;
00400     WORD    namesize;
00401     WORD    compression;
00402     WORD    numdummies;
00403     WORD    numfactors;
00404     WORD    sizeprototype;
00405     WORD    uflags; /* User flags */
00406     #ifdef PARALLELCODE
00407     WORD    partodo; /* Whether to be done in parallel mode */
00408     #endif
00409 } *EXPRESSIONS;
00410
00411
00416 typedef struct SyMbOl {
00417     LONG    name; /* Don't change unless altering .sav too */
00418     WORD    minpower; /* Location in names buffer */
00419     WORD    maxpower; /* Minimum power admissible */
00420     WORD    complex; /* Maximum power admissible */
00421     WORD    number; /* Properties wrt complex conjugation */
00422     WORD    flags; /* Number when stored in file */
00423     WORD    node; /* Used to indicate usage when storing */
00424     WORD    namesize;
00425     WORD    dimension; /* For dimensionality checks */
00426     #if BITSINWORD == 32
00427     UBYTE d_u_m_m_y[8]; /* Padding for sav compatibility */
00428     #elif BITSINWORD == 16
00429     UBYTE d_u_m_m_y[4]; /* Padding for sav compatibility */
00430     #endif
00431 } *SYMBOLS;
00432
00432 #if BITSINWORD == 32
00433 STATIC_ASSERT(sizeof(struct SyMbOl) == 48);
00434 #elif BITSINWORD == 16
00435 STATIC_ASSERT(sizeof(struct SyMbOl) == 24);
00436 #endif

```

```

00437
00442 typedef struct InDeX { /* Don't change unless altering .sav too */
00443     LONG    name; /* Location in names buffer */
00444     WORD    type; /* Regular or dummy */
00445     WORD    dimension; /* Value of d_(n,n) or -number of symbol */
00446     WORD    number; /* Number when stored in file */
00447     WORD    flags; /* Used to indicate usage when storing */
00448     WORD    nmin4; /* Used for n-4 if dimension < 0 */
00449     WORD    node;
00450     WORD    namesize;
00451 } *INDICES;
00452 #if BITSINWORD == 32
00453 STATIC_ASSERT(sizeof(struct InDeX) == 40);
00454 #elif BITSINWORD == 16
00455 STATIC_ASSERT(sizeof(struct InDeX) == 20);
00456 #endif
00457
00462 typedef struct VeCtOr { /* Don't change unless altering .sav too */
00463     LONG    name; /* Location in names buffer */
00464     WORD    complex; /* Properties under complex conjugation */
00465     WORD    number; /* Number when stored in file */
00466     WORD    flags; /* Used to indicate usage when storing */
00467     WORD    node;
00468     WORD    namesize;
00469     WORD    dimension; /* For dimensionality checks */
00470 #if BITSINWORD == 32
00471     UBYTE d_u_m_m_y[8]; /* Padding for sav compatibility */
00472 #elif BITSINWORD == 16
00473     UBYTE d_u_m_m_y[4]; /* Padding for sav compatibility */
00474 #endif
00475 } *VECTORS;
00476 #if BITSINWORD == 32
00477 STATIC_ASSERT(sizeof(struct VeCtOr) == 40);
00478 #elif BITSINWORD == 16
00479 STATIC_ASSERT(sizeof(struct VeCtOr) == 20);
00480 #endif
00481
00487 typedef struct FuNcTiOn { /* Don't change unless altering .sav too */
00488     TABLES tabl;
00489     LONG    symminfo;
00490     LONG    name;
00491     WORD    commute;
00492     WORD    complex;
00493     WORD    number;
00494     WORD    flags;
00495     WORD    spec;
00496     WORD    symmetric;
00497     WORD    node;
00498     WORD    namesize;
00499     WORD    dimension; /* For dimensionality checks */
00500     WORD    maxnumargs;
00501     WORD    minnumargs;
00502 } *FUNCTIONS;
00503 #if BITSINWORD == 32
00504 STATIC_ASSERT(sizeof(struct FuNcTiOn) == 72);
00505 #elif BITSINWORD == 16
00506 STATIC_ASSERT(sizeof(struct FuNcTiOn) == 36);
00507 #endif
00508
00513 typedef struct SeTs {
00514     LONG    name; /* Location in names buffer */
00515     WORD    type; /* Symbol, vector, index or function */
00516     WORD    first; /* First element in setstore */
00517     WORD    last; /* Last element in setstore (excluding) */
00518     WORD    node;
00519     WORD    namesize;
00520     WORD    dimension; /* For dimensionality checks */
00521     WORD    flags; /* Like: ordered */
00522 } *SETS;
00523
00528 typedef struct DuBiOuS { /* Undeclared objects. Just for compiler. */
00529     LONG    name; /* Location in names buffer */
00530     WORD    node;
00531     WORD    dummy;
00532 } *DUBIOUSV;
00533
00534 typedef struct FaCdOLLaR {
00535     WORD    *where; /* A pointer(!) to the content */
00536     LONG    size;
00537     WORD    type; /* Type can be DOLNUMBER or DOLTERMS */
00538     WORD    value; /* in case it is a (short) number */
00539 } FACDOLLAR;
00540
00541 typedef struct DoLLaRs {
00542     WORD    *where; /* A pointer(!) to the object */
00543     FACDOLLAR *factors; /* an array of factors. nfactors elements */
00544 #ifdef WITHPTHREADS

```

```

00545     pthread_mutex_t pthreadslockread;
00546     pthread_mutex_t pthreadslockwrite;
00547 #endif
00548     LONG     size;                /* The number of words */
00549     LONG     name;
00550     WORD     type;
00551     WORD     node;
00552     WORD     index;
00553     WORD     zero;
00554     WORD     numdummies;
00555     WORD     nfactors;
00556 } *DOLLARS;
00557
00562 typedef struct MoDoPtDoLlArS {
00563 #ifdef WITHPTHREADS
00564     DOLLARS     dstruct;        /* If local dollar: list of DOLLARS for each thread */
00565 #endif
00566     WORD     number;
00567     WORD     type;
00568 } MODOPTDOLLAR;
00569
00574 typedef struct fixedset {
00575     char *name;
00576     char *description;
00577     int type;
00578     int dimension;
00579 } FIXEDSET;
00580
00585 typedef struct TaBlEbAsEsUbInDeX {
00586     POSITION where;
00587     LONG size;
00588 } TABLEBASESUBINDEX;
00589
00594 typedef struct TaBlEbAsE {
00595     POSITION fillpoint;
00596     POSITION current;
00597     UBYTE *name;
00598     int *tablenumbers;          /* Number of each table */
00599     TABLEBASESUBINDEX *subindex; /* For each table */
00600     int numtables;
00601 } TABLEBASE;
00602
00609 typedef struct {
00610     WORD *location;
00611     int numargs;
00612     int numfunnies;
00613     int numwildcards;
00614     int symmet;
00615     int tensor;
00616     int commute;
00617 } FUN_INFO;
00618
00619 typedef struct PaRtIcLe {
00620     WORD number; /* Number of the function */
00621     WORD spin;   /* +/- dimension of SU(2) representation */
00622     WORD mass;   /* Number of symbol or 0 */
00623     WORD type;   /* -1: anti particle, 1: particle, 0: own antiparticle */
00624 } PARTICLE;
00625
00626 typedef struct VeRtEx {
00627     PARTICLE particles[MAXPARTICLES];
00628     WORD couplings[2*MAXCOUPLINGS];
00629     WORD nparticles;
00630     WORD ncouplings;
00631     WORD type;
00632     WORD error;
00633     WORD externonly;
00634     WORD spare;
00635 } VERTEX;
00636
00637 typedef struct MoDeL {
00638     VERTEX **vertices;
00639     WORD *couplings;
00640     UBYTE *name;
00641     void *grccmodel;
00642     WORD legcouple[MAXLEGS+2];
00643     WORD nparticles;
00644     WORD nvertices;
00645     WORD invertices;
00646     WORD sizevertices;
00647     WORD sizecouplings;
00648     WORD ncouplings;
00649     WORD error;
00650     WORD dummy;
00651 } MODEL;
00652
00653 /*

```



```

00654 * The struct NAMESPACE is used for the namespace info. It is a
00655 * doubly linked list of nested namespaces.
00656 */
00657
00658 typedef struct NaMeSpAcE {
00659     struct NaMeSpAcE *previous;
00660     struct NaMeSpAcE *next;
00661     NAMETREE *usenames;
00662     UBYTE *name;
00663 } NAMESPACE;
00664
00665
00666 /*
00667  #] Variables :
00668  #[ Files :
00669  */
00670
00682 typedef struct FiLe {
00683     POSITION PPosition;          /* File position */
00684     POSITION filesize;          /* Because SEEK_END is unsafe on IBM */
00685     WORD *PObuffer;            /* Address of the intermediate buffer */
00686     WORD *POstop;              /* End of the buffer */
00687     WORD *POfill;              /* Fill position of the buffer */
00688     WORD *POfull;              /* Full buffer when only cached */
00689 #ifdef WITHPTHREADS
00690     WORD *wPObuffer;           /* Address of the intermediate worker buffer */
00691     WORD *wPOstop;             /* End of the worker buffer */
00692     WORD *wPOfill;             /* Fill position of the worker buffer */
00693     WORD *wPOfull;             /* Full buffer when only worker cached */
00694 #endif
00695     char *name;                /* name of the file */
00696 #ifdef WITHZLIB
00697     z_stream *zsp;             /* The pointer to the stream struct for gzip */
00698     Bytef *ziobuffer;          /* The output buffer for compression */
00699 #endif
00700     ULONG numblocks;           /* Number of blocks in file */
00701     ULONG inbuffer;            /* Block in the buffer */
00702     LONG PSize;                /* size of the buffer */
00703 #ifdef WITHZLIB
00704     LONG ziosize;              /* size of the zoutbuffer */
00705 #endif
00706 #ifdef WITHPTHREADS
00707     LONG wPSize;               /* size of the worker buffer */
00708     pthread_mutex_t pthreadslock;
00709 #endif
00710     int handle;
00711     int active;                /* File is open or closed. Not used. */
00712 } FILEHANDLE;
00713
00723 typedef struct Stream {
00724     off_t fileposition;
00725     off_t linenumber;
00726     off_t prevline;
00727     UBYTE *buffer;
00728     UBYTE *pointer;
00729     UBYTE *top;
00730     UBYTE *FoldName;
00731     UBYTE *name;
00732     UBYTE *pname;
00733     LONG buffersize;
00734     LONG bufferposition;
00735     LONG inbuffer;
00736     int previous;
00737     int handle;
00738     int type;
00739     int prevs;
00740     int previousNoShowInput;
00741     int eqnum;
00742     int afterwards;
00743     int olddelay;
00744     int oldnoshowinput;
00745     UBYTE isnextchar;
00746     UBYTE nextchar[2];
00747     UBYTE reserved;
00748 } STREAM;
00749
00751 typedef struct SpecTator {
00752     POSITION position;          /* The place where we will be writing */
00753     POSITION readpos;           /* The place from which we read */
00754     FILEHANDLE *fh;
00755     char *name;                /* We identify the spectator by the name of the expression */
00756     WORD exprnumber;           /* During running we use the number. */
00757     WORD flags;                /* local, global? */
00758 } SPECTATOR;
00759
00760 /*
00761  #] Files :

```

```

00762     # [ Traces :
00763     */
00764
00774 typedef struct TrAcEs {           /* For computing 4 dimensional traces */
00775     WORD      *accu;               /* NUMBER * 2 */
00776     WORD      *accup;
00777     WORD      *termp;
00778     WORD      *perm;              /* number */
00779     WORD      *inlist;            /* number */
00780     WORD      *nt3;               /* number/2 */
00781     WORD      *nt4;               /* number/2 */
00782     WORD      *j3;                /* number*2 */
00783     WORD      *j4;                /* number*2 */
00784     WORD      *e3;                /* number*2 */
00785     WORD      *e4;                /* number */
00786     WORD      *eers;              /* number/2 */
00787     WORD      *mepf;              /* number/2 */
00788     WORD      *mdel;              /* number/2 */
00789     WORD      *pepf;              /* number*2 */
00790     WORD      *pdel;              /* number*3/2 */
00791     WORD      sgn;
00792     WORD      stap;
00793     WORD      stepl,kstep,mdum;
00794     WORD      gamm,ad,a3,a4,lc3,lc4;
00795     WORD      sign1,sign2,gamma5,num,level,factor,allsign;
00796     WORD      finalstep;
00797 } TRACES;
00798
00806 typedef struct TrAcEn {           /* For computing n dimensional traces */
00807     WORD      *accu;               /* NUMBER */
00808     WORD      *accup;
00809     WORD      *termp;
00810     WORD      *perm;              /* number */
00811     WORD      *inlist;            /* number */
00812     WORD      sgn,num,level,factor,allsign;
00813 } *TRACEN;
00814
00815 /*
00816     # ] Traces :
00817     # [ Preprocessor :
00818     */
00819
00824 typedef struct pReVaR {
00825     UBYTE *name;
00826     UBYTE *value;
00827     UBYTE *argnames;
00828     int nargs;
00829     int wildarg;
00830 } PREVAR;
00831
00836 typedef struct {
00837     WORD *buffer;
00838     int oldcompiletype;
00839     int oldparallelflag;
00840     int oldnumpotmoddollars;
00841     WORD size;
00842     WORD numdollars;
00843     WORD oldcbuf;
00844     WORD oldrbuf;
00845     WORD inscbuf;
00846     WORD oldcnumlhs;
00847 } INSIDEINFO;
00848
00854 typedef struct {
00855     UBYTE *buffer;
00856     LONG size;
00857 } PRELOAD;
00858
00863 typedef struct {
00864     PRELOAD p;
00865     UBYTE *name;
00866     int loadmode;
00867     int mustfree;
00868 } PROCEDURE;
00869
00877 typedef struct DoLoOp {
00878     PRELOAD p;
00879     UBYTE *name;
00880     UBYTE *vars;                  /* for {} or name of expression */
00881     UBYTE *contents;
00882     UBYTE *dollarname;
00883     LONG startlinenumber;
00884     LONG firstnum;
00885     LONG lastnum;
00886     LONG incnum;
00887     int type;
00888     int NoShowInput;

```

```

00889     int errorsinloop;
00890     int firstloopcall;
00891     WORD firstdollar;    /* When >= 0 we have to get the value from a dollar */
00892     WORD lastdollar;     /* When >= 0 we have to get the value from a dollar */
00893     WORD incdollar;      /* When >= 0 we have to get the value from a dollar */
00894     WORD NumPreTypes;
00895     WORD PreIfLevel;
00896     WORD PreSwitchLevel;
00897 } DOLOOP;
00898
00906 struct bit_field {      /* Assume 8 bits per byte */
00907     UBYTE bit_0          : 1;
00908     UBYTE bit_1          : 1;
00909     UBYTE bit_2          : 1;
00910     UBYTE bit_3          : 1;
00911     UBYTE bit_4          : 1;
00912     UBYTE bit_5          : 1;
00913     UBYTE bit_6          : 1;
00914     UBYTE bit_7          : 1;
00915 /*
00916     UINT bit_0           : 1;
00917     UINT bit_1           : 1;
00918     UINT bit_2           : 1;
00919     UINT bit_3           : 1;
00920     UINT bit_4           : 1;
00921     UINT bit_5           : 1;
00922     UINT bit_6           : 1;
00923     UINT bit_7           : 1;
00924 */
00925 };
00926
00931 typedef struct bit_field set_of_char[32];
00932
00937 typedef struct bit_field *one_byte;
00938
00943 typedef struct {
00944     WORD newlogonly;
00945     WORD newhandle;
00946     WORD oldhandle;
00947     WORD oldlogonly;
00948     WORD oldprinttype;
00949     WORD oldsilent;
00950 } HANDLERS;
00951
00952 /*
00953     #] Preprocessor :
00954     #[ Varia :
00955 */
00956
00957 typedef int (*FINISHUFFLE)(WORD *);
00958 typedef int (*DO_UFFLE)(WORD *,WORD,WORD,WORD);
00959
00960 #ifdef WITHPTHREADS
00961 typedef WORD (*COMPAREDUMMY)(void *,WORD *,WORD *,WORD);
00962 #else
00963 typedef WORD (*COMPAREDUMMY)(WORD *,WORD *,WORD);
00964 #endif
00965
00971 typedef struct CbUf {
00972     WORD *Buffer;
00973     WORD *Top;
00974     WORD *Pointer;
00975     WORD **lhs;
00976     WORD **rhs;
00977     LONG *CanCommu;
00978     LONG *NumTerms;
00979     WORD *numdum;
00980     WORD *dimension;
00981     COMPTREE *boomlijst;
00982     LONG BufferSize;
00983     int numlhs;
00984     int numrhs;
00985     int maxlhs;
00986     int maxrhs;
00987     int mnumlhs;
00988     int mnumrhs;
00989     int numtree;
00990     int rootnum;
00991     int MaxTreeSize;
00992 } CBUF;
00993
01001 typedef struct ChAnNeL {
01002     char *name;
01003     int handle;
01004 } CHANNEL;
01005
01015 typedef struct {

```

```

01016     UBYTE *parameter;
01017     int type;
01018     int flags;
01019     // This ordering is assumed by initializer lists in many places. In this
01020     // case the struct should anyway not have additional padding between the
01021     // two int and the LONG members.
01022     LONG value;
01023 } SETUPPARAMETERS;
01024
01033 typedef struct NeStInG {
01034     WORD *termsize;
01035     WORD *funsize;
01036     WORD *argsize;
01037 } *NESTING;
01038
01045 typedef struct StOrEcAcHe {
01046     POSITION position;
01047     POSITION toppos;
01048     struct StOrEcAcHe *next;
01049     WORD buffer[2];
01050 } *STORECACHE;
01051
01057 typedef struct PeRmUtE {
01058     WORD *objects;
01059     WORD sign;
01060     WORD n;
01061     WORD cycle[MAXMATCH];
01062 } PERM;
01063
01068 typedef struct PeRmUtEp {
01069     WORD **objects;
01070     WORD sign;
01071     WORD n;
01072     WORD cycle[MAXMATCH];
01073 } PERMP;
01074
01080 typedef struct DiStRiBuTe {
01081     WORD *obj1;
01082     WORD *obj2;
01083     WORD *out;
01084     WORD sign;
01085     WORD n1;
01086     WORD n2;
01087     WORD n;
01088     WORD cycle[MAXMATCH];
01089 } DISTRIBUTE;
01090
01096 typedef struct PaRtI {
01097     WORD *psize; /* the sizes of the partitions */
01098     WORD *args; /* the offsets of the arguments to be partitioned */
01099     WORD *nargs; /* argument numbers (different number = different argument) */
01100     WORD *nfun; /* the functions into which the partitions go */
01101     WORD numargs; /* the number of arguments to be partitioned */
01102     WORD numpart; /* the number of partitions */
01103     WORD where; /* offset of the function in the term */
01104 } PARTI;
01105
01115 typedef struct sOrT {
01116     FILEHANDLE file; /* The own sort file */
01117     POSITION SizeInFile[3]; /* Sizes in the various files */
01118     WORD *lBuffer; /* The large buffer */
01119     WORD *lTop; /* End of the large buffer */
01120     WORD *lFill; /* The filling point of the large buffer */
01121     WORD *used; /* auxiliary during actual sort */
01122     WORD *sBuffer; /* The small buffer */
01123     WORD *sTop; /* End of the small buffer */
01124     WORD *sTop2; /* End of the extension of the small buffer */
01125     WORD *sHalf; /* Halfway point in the extension */
01126     WORD *sFill; /* Filling point in the small buffer */
01127     WORD **sPointer; /* Pointers to terms in the small buffer */
01128     WORD **PoinFill; /* Filling point for pointers to the sm.buf */
01129     WORD *cBuffer; /* Compress buffer (if it exists) */
01130     WORD **Patches; /* Positions of patches in large buffer */
01131     WORD **pStop; /* Ends of patches in the large buffer */
01132     WORD **poina; /* auxiliary during actual sort */
01133     WORD **poin2a; /* auxiliary during actual sort */
01134     WORD *ktoi; /* auxiliary during actual sort */
01135     WORD *tree; /* auxiliary during actual sort */
01136 #ifdef WITHZLIB
01137     WORD *fpccompressed; /* is output filepatch compressed? */
01138     WORD *fpincompressed; /* is input filepatch compressed? */
01139     z_stream *zsparray; /* the array of zstreams for decompression */
01140 #endif
01141     POSITION *fPatches; /* Positions of output file patches */
01142     POSITION *inPatches; /* Positions of input file patches */
01143     POSITION *fPatchesStop; /* Positions of output file patches */
01144     POSITION *iPatches; /* Input file patches, Points to fPatches or inPatches */

```

```

01145     FILEHANDLE *f;                /* The actual output file */
01146     FILEHANDLE **ff;              /* Handles for a staged sort */
01147     LONG sTerms;                  /* Terms in small buffer */
01148     LONG LargeSize;               /* Size of large buffer (in words) */
01149     LONG SmallSize;               /* Size of small buffer (in words) */
01150     LONG SmallEsize;              /* Size of small + extension (in words) */
01151     LONG TermsInSmall;            /* Maximum number of terms in small buffer */
01152     LONG Terms2InSmall;           /* with extension for polyfuns etc. */
01153     LONG GenTerms;                /* Number of generated terms */
01154     LONG TermsLeft;               /* Number of terms still in existence */
01155     LONG GenSpace;                /* Amount of space of generated terms */
01156     LONG SpaceLeft;               /* Space needed for still existing terms */
01157     LONG putinsize;                /* Size of buffer in Putin */
01158     LONG ninterms;                /* Which input term ? */
01159     int MaxPatches;               /* Maximum number of patches in large buffer */
01160     int MaxFpatches;              /* Maximum number of patches in one filesort */
01161     int type;                     /* Main, function or sub(routine) */
01162     int lPatch;                   /* Number of patches in the large buffer */
01163     int fPatchNl;                 /* Number of patches in input file */
01164     int PolyWise;                 /* Is there a polyfun and if so, where? */
01165     int PolyFlag;                 /* */
01166     int cBufferSize;              /* Size of the compress buffer */
01167     int maxtermsize;              /* Keeps track for buffer allocations */
01168     int newmaxtermsize;           /* Auxiliary for maxtermsize */
01169     int outputmode;               /* Tells where the output is going */
01170     int stagelevel;               /* In case we have a 'staged' sort */
01171     WORD fPatchN;                 /* Number of patches on file (output) */
01172     WORD inNum;                   /* Number of patches on file (input) */
01173     WORD stage4;                  /* Are we using stage4? */
01174 } SORTING;
01175
01176 #ifdef WITHPTHREADS
01177
01183 typedef struct SoRtBlOcK {
01184     pthread_mutex_t *MasterBlockLock;
01185     WORD **MasterStart;
01186     WORD **MasterFill;
01187     WORD **MasterStop;
01188     int MasterNumBlocks;
01189     int MasterBlock;
01190     int FillBlock;
01191 } SORTBLOCK;
01192 #endif
01193
01194 #ifdef DEBUGGER
01195 typedef struct DeBuGgInG {
01196     int eflag;
01197     int printfFlag;
01198     int logfileFlag;
01199     int stdoutFlag;
01200 } DEBUGSTR;
01201 #endif
01202
01203 #ifdef WITHPTHREADS
01204
01210 typedef struct ThReAdBuCkEt {
01211     POSITION *deferbuffer;          /* For Keep Brackets: remember position */
01212     WORD *threadbuffer;           /* Here are the (primary) terms */
01213     WORD *compressbuffer;         /* For keep brackets we need the compressbuffer */
01214     LONG threadbufferSize;        /* Number of words in threadbuffer */
01215     LONG ddterms;                 /* Number of primary+secondary terms represented */
01216     LONG firstterm;               /* The number of the first term in the bucket */
01217     LONG firstbracket;            /* When doing complete brackets */
01218     LONG lastbracket;             /* When doing complete brackets */
01219     pthread_mutex_t lock;         /* For the load balancing phase */
01220     int free;                     /* Status of the bucket */
01221     int totnum;                   /* Total number of primary terms */
01222     int usenum;                   /* Which is the term being used at the moment */
01223     int busy;                     /* */
01224     int type;                     /* Doing brackets? */
01225 } THREADBUCKET;
01226
01227 #endif
01228
01236 typedef struct {
01237     WORD *coefs;                  /* The array of coefficients */
01238     WORD numsym;                  /* The number of the symbol in the polynomial */
01239     WORD arraysize;               /* The size of the allocation of coefs */
01240     WORD polysize;                /* The maximum power in the polynomial */
01241     WORD modnum;                  /* The prime number of the modulus */
01242 } POLYMOD;
01243
01244 typedef struct {
01245     WORD *outterm;                /* Used in DoShuffle/Merge/FinishShuffle system */
01246     WORD *outfun;
01247     WORD *incoef;
01248     WORD *stopl;

```

```

01249     WORD    *stop2;
01250     WORD    *ststop1;
01251     WORD    *ststop2;
01252     FINISHUFFLE finishuf;
01253     DO_UFFLE    do_uffle;
01254     LONG    combilast;
01255     WORD    nincoef;
01256     WORD    level;
01257     WORD    thefunction;
01258     WORD    option;
01259 } SHvariables;
01260
01261 typedef struct {                /* Used for computing calculational cost in optim.c */
01262     LONG    add;
01263     LONG    mul;
01264     LONG    div;
01265     LONG    pow;
01266 } COST;
01267
01268 typedef struct {
01269     UWORD    *a;        /* The number array */
01270     UWORD    *m;        /* The modulus array */
01271     WORD    na;        /* Size of the number */
01272     WORD    nm;        /* size of the number in the modulus array */
01273 } MODNUM;
01274
01275 /*
01276     Struct for optimizing outputs. If changed, do not forget to change
01277     the padding information in the AO struct.
01278 */
01279 typedef struct {
01280     union { /* we do this to allow padding */
01281         float fval;
01282         int ival[2]; /* This should be enough */
01283     } mctsconstant;
01284     int    horner;
01285     int    hornerdirection;
01286     int    method;
01287     int    mctstimelimit;
01288     int    mctsnumpexpand;
01289     int    mctsnumpkeep;
01290     int    mctsnumprepeat;
01291     int    greedytimelimit;
01292     int    greedyminnum;
01293     int    greedymaxperc;
01294     int    printstats;
01295     int    debugflags;
01296     int    schemeflags;
01297     int    mctsdecaymode;
01298     int    saIter; /* Simulated annealing updates */
01299     union {
01300         float fval;
01301         int ival[2];
01302     } saMaxT; /* Maximum temperature of SA */
01303     union {
01304         float fval;
01305         int ival[2];
01306     } saMinT; /* Minimum temperature of SA */
01307     int    spare;
01308 } OPTIMIZE;
01309
01310 typedef struct {
01311     WORD    *code;
01312     UBYTE    *nameofexpr; /* It is easier to remember an expression by name */
01313     LONG    codesize; /* We need this for the checkpoints */
01314     WORD    exprnr; /* Problem here is: we renumber them in execute.c */
01315     WORD    minvar;
01316     WORD    maxvar;
01317 } OPTIMIZERESULT;
01318
01319 typedef struct {
01320     WORD    *lhs; /* Object to be replaced */
01321     WORD    *rhs; /* Depending on the type it will be UBYTE* or WORD* */
01322     int    type;
01323     int    size; /* Size of the lhs */
01324 } DICTIONARY_ELEMENT;
01325
01326 typedef struct {
01327     DICTIONARY_ELEMENT **elements;
01328     UBYTE    *name;
01329     int    sizeelements;
01330     int    numelements;
01331     int    numbers; /* deal with numbers */
01332     int    variables; /* deal with single variables */
01333     int    characters; /* deal with special characters */
01334     int    funwith; /* deal with functions with arguments */
01335     int    gnumelements; /* if .global shrinks the dictionary */

```

```

01336     int ranges;
01337 } DICTIONARY;
01338
01339 typedef struct {
01340     WORD ncase;
01341     WORD value;
01342     WORD compbuffer;
01343 } SWITCHTABLE;
01344
01345 typedef struct {
01346     SWITCHTABLE *table;
01347     SWITCHTABLE defaultcase;
01348     SWITCHTABLE endswitch;
01349     WORD typetable;
01350     WORD maxcase;
01351     WORD mincase;
01352     WORD numcases;
01353     WORD tablesize;
01354     WORD caseoffset;
01355     WORD iflevel;
01356     WORD whilelevel;
01357     WORD nestingsum;
01358     WORD padding;
01359 } SWITCH;
01360
01361 typedef struct {
01362     WORD *term;
01363     void *currentModel;
01364     void *currentMODEL;
01365     WORD *legcouple[MAXLEGS+1];
01366     LONG numtopo;
01367     LONG numdia;
01368     WORD diaoffset;
01369     WORD level;
01370     WORD externalset;
01371     WORD internalset;
01372     WORD numextern;
01373     WORD flags;
01374 } TERMINFO;
01375
01376 /*
01377 #] Varia :
01378 #[ A :
01379     #[ M : The M struct is for global settings at startup or .clear
01380 */
01381
01382 struct M_const {
01383     POSITION zeropos;           /* (M) is zero */
01384     SORTING *S0;
01385     UWORD *gcmmod;
01386     UWORD *gpowmod;
01387     UBYTE *TempDir;           /* (M) Path with where the temporary files go */
01388     UBYTE *TempSortDir;       /* (M) Path with where the sort files go */
01389     UBYTE *IncDir;            /* (M) Directory path for include files */
01390     UBYTE *InputFileName;     /* (M) */
01391     UBYTE *LogFileName;       /* (M) */
01392     UBYTE *OutBuffer;         /* (M) Output buffer in pre.c */
01393     UBYTE *Path;              /* (M) */
01394     UBYTE *SetupDir;          /* (M) Directory with setup file */
01395     UBYTE *SetupFile;         /* (M) Name of setup file */
01396     UBYTE *gFortran90Kind;
01397     UBYTE *gextrasymp;
01398     UBYTE *ggextrasymp;
01399     UBYTE *oldnumextrasymp;
01400     SPECTATOR *SpectatorFiles;
01401 #ifdef WITHPTHREADS
01402     pthread_rwlock_t handlelock; /* (M) */
01403     pthread_mutex_t storefilelock; /* (M) */
01404     pthread_mutex_t sbuflock;     /* (M) Lock for writing in the AM.sbuffer */
01405     LONG ThreadScratSize;         /* (M) Size of Fscr[0/2] buffers of the workers */
01406     LONG ThreadScratOutSize;     /* (M) Size of Fscr[1] buffers of the workers */
01407 #endif
01408     LONG MaxTer;                 /* (M) Maximum term size. Fixed at setup. In Bytes!!! */
01409     LONG CompressSize;           /* (M) Size of Compress buffer */
01410     LONG ScratSize;              /* (M) Size of Fscr[] buffers */
01411     LONG HideSize;              /* (M) Size of Fscr[2] buffer */
01412     LONG SizeStoreCache;         /* (M) Size of the caches for reading global expr. */
01413     LONG MaxStreamSize;          /* (M) Maximum buffer size in reading streams */
01414     LONG SIOsize;               /* (M) Sort InputOutput buffer size */
01415     LONG SLargeSize;            /* (M) */
01416     LONG SSmallEsize;           /* (M) */
01417     LONG SSmallSize;            /* (M) */
01418     LONG STermsInSmall;         /* (M) */
01419     LONG MaxBracketBufferSize;  /* (M) Max Size for B+ or AB+ per expression */
01420     LONG hProcessBucketSize;    /* (M) */
01421     LONG gProcessBucketSize;    /* (M) */
01422     LONG shmWinSize;            /* (M) size for shared memory window used in communications */

```

```

01430     LONG     OldChildTime;          /* (M) Zero time. Needed in timer. */
01431     LONG     OldSecTime;             /* (M) Zero time for measuring wall clock time */
01432     LONG     OldMilliTime;           /* (M) Same, but milli seconds */
01433     LONG     WorkSize;               /* (M) Size of WorkSpace */
01434     LONG     gThreadBucketSize;      /* (C) */
01435     LONG     ggThreadBucketSize;     /* (C) */
01436     LONG     SumTime;               /* Used in .clear */
01437     LONG     SpectatorSize;          /* Size of the buffer in bytes */
01438     LONG     TimeLimit;              /* Limit in sec to the total real time */
01439 #ifdef WITHFLOAT
01440     LONG     gDefaultPrecision;      /* (M) Default precision in bits for float_ */
01441     LONG     gMaxWeight;             /* (M) Maximum weight for MZV or Euler */
01442     LONG     ggDefaultPrecision;     /* (M) Default precision in bits for float_ */
01443     LONG     ggMaxWeight;            /* (M) Maximum weight for MZV or Euler */
01444 #endif
01445     int      FileOnlyFlag;           /* (M) Writing only to file */
01446     int      Interact;               /* (M) Interactive mode flag */
01447     int      MaxParLevel;            /* (M) Maximum nesting of parentheses */
01448     int      OutBufSize;             /* (M) size of OutBuffer */
01449     int      SMaxFpatches;           /* (M) */
01450     int      SMaxPatches;            /* (M) */
01451     int      StdOut;                 /* (M) Regular output channel */
01452     int      ginsidefirst;           /* (M) Not used yet */
01453     int      gDefDim;               /* (M) */
01454     int      gDefDim4;              /* (M) */
01455     int      NumFixedSets;           /* (M) Number of the predefined sets */
01456     int      NumFixedFunctions;      /* (M) Number of built in functions */
01457     int      rbufnum;               /* (M) startup compiler buffer */
01458     int      dbufnum;               /* (M) dollar variables */
01459     int      sbufnum;               /* (M) subterm variables */
01460     int      zbufnum;               /* (M) special values */
01461     int      SkipClears;             /* (M) Number of .clear to skip at start */
01462     int      gTokensWriteFlag;       /* (M) */
01463     int      gfunpowers;             /* (M) */
01464     int      gStatsFlag;             /* (M) */
01465     int      gNamesFlag;            /* (M) */
01466     int      gCodesFlag;            /* (M) */
01467     int      gSortType;             /* (M) */
01468     int      gproperorderflag;       /* (M) */
01469     int      hparallelflag;          /* (M) */
01470     int      gparallelflag;          /* (M) */
01471     int      totalnumberofthreads;   /* (M) */
01472     int      gSizeCommuteInSet;
01473     int      gThreadStats;
01474     int      ggThreadStats;
01475     int      gFinalStats;
01476     int      ggFinalStats;
01477     int      gThreadsFlag;
01478     int      ggThreadsFlag;
01479     int      gThreadBalancing;
01480     int      ggThreadBalancing;
01481     int      gThreadSortFileSynch;
01482     int      ggThreadSortFileSynch;
01483     int      gProcessStats;
01484     int      ggProcessStats;
01485     int      gOldParallelStats;
01486     int      ggOldParallelStats;
01487     int      maxFlevels;             /* () maximum function levels */
01488     int      resetTimeOnClear;       /* (M) */
01489     int      gcNumDollars;           /* () number of dollars for .clear */
01490     int      MultiRun;
01491     int      gNoSpacesInNumbers;     /* For very long numbers */
01492     int      ggNoSpacesInNumbers;    /* For very long numbers */
01493     int      gIsFortran90;
01494     int      PrintTotalSize;
01495     int      fbuffersize;            /* Size for the AT.fbufnum factorization caches */
01496     int      gOldFactArgFlag;
01497     int      ggOldFactArgFlag;
01498     int      gnumextrasym;
01499     int      ggnumextrasym;
01500     int      NumSpectatorFiles;       /* Elements used in AM.spectatorfiles; */
01501     int      SizeForSpectatorFiles;  /* Size in AM.spectatorfiles; */
01502     int      gOldGCDflag;
01503     int      ggOldGCDflag;
01504     int      gWTimeStatsFlag;
01505     int      ggWTimeStatsFlag;
01506     int      jumpratio;
01507     WORD     MaxTal;                 /* (M) Maximum number of words in a number */
01508     WORD     IndDum;                 /* (M) Basis value for dummy indices */
01509     WORD     DumInd;                 /* (M) */
01510     WORD     WilInd;                 /* (M) Offset for wildcard indices */
01511     WORD     gncmod;                 /* (M) Global setting of modulus. size of gcmmod */
01512     WORD     gnpowmod;               /* (M) Global printing as powers. size gpowmod */
01513     WORD     gmodmode;               /* (M) Global mode for modulus */
01514     WORD     gUnitTrace;             /* (M) Global value of Tr[1] */
01515     WORD     gOutputMode;            /* (M) */
01516     WORD     gOutputSpaces;          /* (M) */

```



```

01517     WORD      gOutNumberType;      /* (M) */
01518     WORD      gCnumpows;           /* (M) */
01519     WORD      gUniTrace[4];        /* (M) */
01520     WORD      MaxWildcards;        /* (M) Maximum number of wildcards */
01521     WORD      mTraceDum;           /* (M) Position/Offset for generated dummies */
01522     WORD      OffsetIndex;         /* (M) */
01523     WORD      OffsetVector;        /* (M) */
01524     WORD      RepMax;              /* (M) Max repeat levels */
01525     WORD      LogType;             /* (M) Type of writing to log file */
01526     WORD      ggStatsFlag;         /* (M) */
01527     WORD      gLineLength;        /* (M) */
01528     WORD      qError;             /* (M) Only error checking {-c option} */
01529     WORD      FortranCont;         /* (M) Fortran Continuation character */
01530     WORD      HoldFlag;           /* (M) Exit on termination? */
01531     WORD      Ordering[15];       /* (M) Auxiliary for ordering wildcards */
01532     WORD      silent;             /* (M) Silent flag. Only results in output. */
01533     WORD      tracebackflag;       /* (M) For tracing errors */
01534     WORD      expnum;             /* (M) internal number of ^ function */
01535     WORD      denomnum;           /* (M) internal number of / function */
01536     WORD      facnum;             /* (M) internal number of fac_ function */
01537     WORD      invfacnum;          /* (M) internal number of invfac_ function */
01538     WORD      sumnum;            /* (M) internal number of sum_ function */
01539     WORD      sumpnum;           /* (M) internal number of sump_ function */
01540     WORD      OldOrderFlag;       /* (M) Flag for allowing old statement order */
01541     WORD      termfunnum;         /* (M) internal number of term_ function */
01542     WORD      matchfunnum;        /* (M) internal number of match_ function */
01543     WORD      countfunnum;        /* (M) internal number of count_ function */
01544     WORD      gPolyFun;           /* (M) global value of PolyFun */
01545     WORD      gPolyFunInv;        /* (M) global value of Inverse of PolyFun */
01546     WORD      gPolyFunType;       /* (M) global value of PolyFun */
01547     WORD      gPolyFunExp;
01548     WORD      gPolyFunVar;
01549     WORD      gPolyFunPow;
01550     WORD      dollarzero;         /* (M) for dollars with zero value */
01551     WORD      atstartup;          /* To protect against DATE_ ending in \n */
01552     WORD      exitflag;           /* (R) For the exit statement */
01553     WORD      NumStoreCaches;     /* (I) Number of storage caches per processor */
01554     WORD      gIndentSpace;       /* For indentation in output */
01555     WORD      ggIndentSpace;
01556     WORD      gShortStatsMax;
01557     WORD      ggShortStatsMax;
01558     WORD      gextrasympols;
01559     WORD      ggextrasympols;
01560     WORD      zerorhs;
01561     WORD      onerhs;
01562     WORD      havesortdir;
01563     WORD      vectorzero;         /* p0_ */
01564     WORD      ClearStore;
01565     WORD      BracketFactors[8];
01566     BOOL      FromStdin;          /* read the input from STDIN */
01567     BOOL      IgnoreDeprecation;  /* ignore deprecation warning */
01568 };
01569 /*
01570     #] M :
01571     #[ P : The P struct defines objects set by the preprocessor
01572 */
01580 struct P_const {
01581     LIST DollarList;             /* (R) Dollar variables. Contains pointers
01582                                to contents of the variables.*/
01583     LIST PreVarList;            /* (R) List of preprocessor variables
01584                                Points to contents. Can be changed */
01585     LIST LoopList;              /* (P) List of do loops */
01586     LIST ProcList;              /* (P) List of procedures */
01587     INSIDEINFO inside;          /* Information during #inside/#endside */
01588     NAMESPACE *firstnamespace; /* is zero when no namespace specified */
01589     NAMESPACE *lastnamespace;  /* is zero when no namespace specified */
01590     UBYTE *fullname;            /* buffer for namespace expanded names */
01591     UBYTE **PreSwitchStrings;   /* (P) The string in a switch */
01592     UBYTE *preStart;            /* (P) Preprocessor instruction buffer */
01593     UBYTE *preStop;             /* (P) end of preStart */
01594     UBYTE *preFill;             /* (P) Filling point in preStart */
01595     UBYTE *procedureExtension;  /* (P) Extension for procedure files (prc) */
01596     UBYTE *cprocedureExtension; /* (P) Extension after .clear */
01597     LONG *PreAssignStack;       /* For nesting $name assignments */
01598     int *PreIfStack;            /* (P) Tracks nesting of #if */
01599     int *PreSwitchModes;        /* (P) Stack of switch status */
01600     int *PreTypes;             /* (P) stack of #call, #do etc nesting */
01601 #ifdef WITHPTHREADS
01602     pthread_mutex_t PreVarLock; /* (P) */
01603 #endif
01604     LONG StopWatchZero;         /* For `timer_' and #reset timer */
01605     LONG InOutBuf;              /* (P) Characters in the output buf in pre.c */
01606     LONG pSize;                 /* (P) size of preStart */
01607     int PreAssignFlag;          /* (C) Indicates #assign -> catch dollar */
01608     int PreContinuation;        /* (C) Indicates whether the statement is new */
01609     int PreproFlag;             /* (P) Internal use to mark work on prepro instr. */
01610     int iBufError;              /* (P) Flag for errors with input buffer */

```

```

01611     int      PreOut;                /* (P) Flag for #+ #- */
01612     int      PreSwitchLevel;        /* (P) Nesting of #switch */
01613     int      NumPreSwitchStrings;    /* (P) Size of PreSwitchStrings */
01614     int      MaxPreTypes;            /* (P) Size of PreTypes */
01615     int      NumPreTypes;            /* (P) Number of nesting objects in PreTypes */
01616     int      MaxPreIfLevel;          /* (C) Maximum number of nested #if. Dynamic */
01617     int      PreIfLevel;             /* (C) Current position if PreIfStack */
01618     int      PreInsideLevel;         /* (C) #inside active? */
01619     int      DelayPrevar;            /* (P) Delaying prevar substitution */
01620     int      AllowDelay;             /* (P) Allow delayed prevar substitution */
01621     int      lhdollarerror;          /* (R) */
01622     int      eat;                    /* ( ) */
01623     int      gNumPre;                /* (P) Number of preprocessor variables for .clear */
01624     int      PreDebug;               /* (C) */
01625     int      OpenDictionary;
01626     int      PreAssignLevel;         /* For nesting # $name = ...; assignments */
01627     int      MaxPreAssignLevel;      /* For nesting # $name = ...; assignments */
01628     int      fullnamesize;           /* size of the fullname buffer */
01629     WORD     DebugFlag;              /* (P) For debugging purposes */
01630     WORD     preError;               /* (P) Blocks certain types of execution */
01631     UBYTE    ComChar;               /* (P) Commentary character */
01632     UBYTE    cComChar;              /* (P) Old commentary character for .clear */
01633 };
01634
01635 /*
01636     #] P :
01637     #[ C : The C struct defines objects changed by the compiler
01638 */
01639 struct C_const {
01640     set_of_char separators;
01641     POSITION StoreFileSize;           /* ( ) Size of store file */
01642     NAMETREE *dollarnames;
01643     NAMETREE *exprnames;
01644     NAMETREE *varnames;
01645     LIST      ChannelList;
01646                                     /* Later also for write? */
01647     LIST      DubiousList;
01648     LIST      FunctionList;
01649     LIST      ExpressionList;
01650     LIST      IndexList;
01651     LIST      SetElementList;
01652     LIST      SetList;
01653     LIST      SymbolList;
01654     LIST      VectorList;
01655     LIST      PotModDollList;
01656     LIST      ModOptDollList;
01657     LIST      TableBaseList;
01658 /*
01659     Compile buffer variables
01660 */
01661     LIST      cbufList;
01662 /*
01663     Objects for auto declarations
01664 */
01665     LIST      AutoSymbolList;        /* (C) */
01666     LIST      AutoIndexList;         /* (C) */
01667     LIST      AutoVectorList;        /* (C) */
01668     LIST      AutoFunctionList;      /* (C) */
01669     NAMETREE *autonames;
01670     LIST      *Symbols;              /* (C) Pointer for autodeclare. Which list is
01671                                     it searching. Later also for subroutines */
01672     LIST      *Indices;              /* (C) id. */
01673     LIST      *Vectors;              /* (C) id. */
01674     LIST      *Functions;            /* (C) id. */
01675     NAMETREE **activenames;
01676     STREAM    *Streams;
01677     STREAM    *CurrentStream;
01678     SWITCH    *SwitchArray;
01679     MODEL     **models;
01680     WORD      *SwitchHeap;
01681     LONG      *termstack;
01682     LONG      *termsortstack;
01683     UWORD     *cmmod;
01684     UWORD     *powmod;
01685     UWORD     *modpowers;
01686     UWORD     *halfmod;              /* (C) half the modulus when not zero */
01687     WORD      *ProtoType;            /* (C) The subexpression prototype {wildcards} */
01688     WORD      *WildC;               /* (C) Filling point for wildcards. */
01689     LONG      *IfHeap;
01690     LONG      *IfCount;
01691     LONG      *IfStack;
01692     UBYTE     *iBuffer;
01693     UBYTE     *iPointer;
01694     UBYTE     *iStop;
01695     UBYTE     **LabelNames;
01696     WORD      *FixIndices;

```

```

01710 WORD      *termsumcheck;
01711 UBYTE     *WildcardNames;
01712 int       *Labels;
01713 SBYTE     *tokens;
01714 SBYTE     *toptokens;
01715 SBYTE     *endoftokens;
01716 WORD      *tokenarglevel;
01717 UWORD     *modinverses;          /* Table for inverses of primes */
01718 UBYTE     *Fortran90Kind;        /* The kind of number in Fortran 90 as in _ki */
01719 WORD      **MultiBracketBuf;     /* Array of buffers for multi-level brackets */
01720 UBYTE     *extrasyms;            /* Array with the name for extra symbols in ToPolynomial */
01721 WORD      *doloopstack;          /* To keep track of begin and end of doloops */
01722 WORD      *doloopnest;           /* To keep track of nesting of doloops etc */
01723 char      *CheckpointRunAfter;
01725 char      *CheckpointRunBefore;
01727 WORD      *IfSumCheck;
01728 WORD      *CommuteInSet;         /* groups of noncommuting functions that can commute */
01729 UBYTE     *TestValue;           /* For debugging */
01730 #ifdef PARALLELCODE
01731 LONG      *inputnumbers;
01732 WORD      *pfirstnum;
01733 #endif
01734 #ifdef WITHPTHREADS
01735 pthread_mutex_t halfmodlock;    /* () Lock for adding buffer for halfmod */
01736 #endif
01737 LONG      argstack[MAXNEST];     /* (C) {contents} Stack for nesting of Argument */
01738 LONG      insidestack[MAXNEST];  /* (C) {contents} Stack for Argument or Inside. */
01739 LONG      inexprstack[MAXNEST];  /* (C) {contents} Stack for Argument or Inside. */
01740 LONG      iBufferSize;           /* (C) Size of the input buffer */
01741 LONG      TransEname;            /* (C) Used when a new definition overwrites
01742                                an old expression. */
01743 LONG      ProcessBucketSize;     /* (C) */
01744 LONG      mProcessBucketSize;    /* (C) */
01745 LONG      CModule;              /* (C) Counter of current module */
01746 LONG      ThreadBucketSize;      /* (C) Roughly the maximum number of input terms */
01747 LONG      CheckpointStamp;
01748 LONG      CheckpointInterval;
01750 #ifdef WITHFLOAT
01751 LONG      DefaultPrecision;       /* (C) Default precision in bits for float_ */
01752 LONG      MaxWeight;             /* (C) Maximum weight for MZV or Euler */
01753 LONG      tDefaultPrecision;     /* (C) Default precision in bits for float_ */
01754 LONG      tMaxWeight;            /* (C) Maximum weight for MZV or Euler */
01755 #endif
01756 int       cbufnum;
01757 int       AutoDeclareFlag;
01759 int       NoShowInput;           /* (C) No listing of input as in .prc, #do */
01760 int       ShortStats;            /* (C) */
01761 int       compiletype;           /* (C) type of statement {DECLARATION etc} */
01762 int       firstconstindex;       /* (C) flag for giving first error message */
01763 int       insidefirst;           /* (C) Not used yet */
01764 int       minsidefirst;          /* (?) Not used yet */
01765 int       wildflag;              /* (C) Flag for marking use of wildcards */
01766 int       NumLabels;             /* (C) Number of labels {in Labels} */
01767 int       MaxLabels;             /* (C) Size of Labels array */
01768 int       lDefDim;               /* (C) */
01769 int       lDefDim4;              /* (C) */
01770 int       NumWildcardNames;       /* (C) Number of ?a variables */
01771 int       WildcardBufferSize;     /* (C) size of WildcardNames buffer */
01772 int       MaxIf;                 /* (C) size of IfHeap, IfSumCheck, IfCount */
01773 int       NumStreams;            /* (C) */
01774 int       MaxNumStreams;         /* (C) */
01775 int       firstctypemessage;     /* (C) Flag for giving first st order error */
01776 int       tablecheck;            /* (C) For table checking */
01777 int       idoption;              /* (C) */
01778 int       BottomLevel;           /* (C) For propercount. Not used!!! */
01779 int       CompileLevel;          /* (C) Subexpression level */
01780 int       TokensWriteFlag;       /* (C) */
01781 int       UnsureDollarMode;      /* (C) ?Controls error messages undefined $'s */
01782 int       outsidefun;            /* (C) Used for writing Tables to file */
01783 int       funpowers;             /* (C) */
01784 int       WarnFlag;              /* (C) */
01785 int       StatsFlag;             /* (C) */
01786 int       NamesFlag;            /* (C) */
01787 int       CodesFlag;            /* (C) */
01788 int       SetupFlag;             /* (C) */
01789 int       SortType;              /* (C) */
01790 int       lSortType;            /* (C) */
01791 int       ThreadStats;           /* (C) */
01792 int       FinalStats;           /* (C) */
01793 int       OldParallelStats;      /* (C) */
01794 int       ThreadsFlag;
01795 int       ThreadBalancing;
01796 int       ThreadSortFileSynch;
01797 int       ProcessStats;          /* (C) */
01798 int       BracketNormalize;      /* (C) Indicates whether the bracket st is normalized */
01799 int       maxtermlevel;          /* (C) Size of termstack */
01800 int       dumnumflag;            /* (C) Where there dummy indices in tokenizer? */

```

```

01801     int    bracketindexflag;    /* (C) Are brackets going to be indexed? */
01802     int    parallelflag;         /* (C) parallel allowed? */
01803     int    mparallelflag;        /* (C) parallel allowed in this module? */
01804     int    inparallelflag;       /* (C) inparallel allowed? */
01805     int    partodoflag;          /* (C) parallel allowed? */
01806     int    properorderflag;      /* (C) clean normalizing. */
01807     int    vetofilling;          /* (C) vetoes overwriting in tablebase stubs */
01808     int    tablefilling;         /* (C) to notify AddrHS we are filling a table */
01809     int    vetotablebasefill;    /* (C) For the load in tablebase */
01810     int    exprfillwarning;      /* (C) Warning has been printed for expressions in fill statements */
    */
01811     int    lhdollarflag;        /* (R) left hand dollar present */
01812     int    NoCompress;          /* (R) Controls native compression */
01813     int    IsFortran90;         /* Tells whether the Fortran is Fortran90 */
01814     int    MultiBracketLevels;  /* Number of elements in MultiBracketBuf */
01815     int    topolynomialflag;     /* To avoid ToPolynomial and FactArg together */
01816     int    ffbufnum;            /* Buffer number for user defined factorizations */
01817     int    OldFactArgFlag;
01818     int    MemDebugFlag;        /* Only used when MALLOCDEBUG in tools.c */
01819     int    OldGCDFlag;
01820     int    WTimeStatsFlag;
01821     int    SortReallocateFlag;  /* Controls reallocation of large+small buffer at module end.
01822                                     0 : Off
01823                                     1 : On, every module (set by On sortreallocate);
01824                                     2 : On, single module (set by #sortreallocate) */
01825     int    PrintBacktraceFlag;  /* Print backtrace on terminate? */
01826     int    FlintPolyFlag;       /* Use Flint for polynomial arithmetic */
01827     int    HumanStatsFlag;      /* Print human-readable stats in the stats print? */
01828     int    doloopstacksize;
01829     int    dolooplevel;
01830     int    CheckpointFlag;
01834     int    SizeCommuteInSet;    /* Size of the CommuteInSet buffer */
01835 #ifdef PARALLELCODE
01836     int    numpfirstnum;        /* For redefine */
01837     int    sizepffirstnum;      /* For redefine */
01838 #endif
01839     int    origin;              /* Determines whether .sort or ModuleOption */
01840     int    vectorlikeLHS;
01841     int    nummodels;
01842     int    modelspace;
01843     int    ModelLevel;
01844     int    ShortStatsMax;       /* For On FewerStatistics 10; */
01845     int    InnerTest;           /* For debugging */
01846     WORD   argsumcheck[MAXNEST]; /* (C) Checking of nesting */
01847     WORD   insidesumcheck[MAXNEST]; /* (C) Checking of nesting */
01848     WORD   inexprsumcheck[MAXNEST]; /* (C) Checking of nesting */
01849     WORD   RepSumCheck[MAXREPEAT]; /* (C) Checks nesting of repeat, if, argument */
01850     WORD   lUnitTrace[4];       /* (C) */
01851     WORD   RepLevel;            /* (C) Tracks nesting of repeat. */
01852     WORD   arglevel;            /* (C) level of nested argument statements */
01853     WORD   insidlevel;          /* (C) level of nested inside statements */
01854     WORD   inexprlevel;         /* (C) level of nested inexpr statements */
01855     WORD   termlevel;           /* (C) level of nested term statements */
01856     WORD   MustTestTable;       /* (C) Indicates whether tables have been changed */
01857     WORD   DumNum;              /* (C) */
01858     WORD   ncmod;               /* (C) Local setting of modulus. size of cmod */
01859     WORD   npowmod;             /* (C) Local printing as powers. size powmod */
01860     WORD   modmode;             /* (C) Mode for modulus calculus */
01861     WORD   nhalfmod;            /* relevant word size of halfmod when defined */
01862     WORD   DirtPow;             /* (C) Flag for changes in printing mod powers */
01863     WORD   lUnitTrace;          /* (C) Local value of Tr[1] */
01864     WORD   NwildC;              /* (C) Wildcard counter */
01865     WORD   ComDefer;            /* (C) defer brackets */
01866     WORD   CollectFun;          /* (C) Collect function number */
01867     WORD   AltCollectFun;       /* (C) Alternate Collect function number */
01868     WORD   OutputMode;          /* (C) */
01869     WORD   Cnumpows;
01870     WORD   OutputSpaces;        /* (C) */
01871     WORD   OutNumberType;       /* (C) Controls RATIONAL/FLOAT output */
01872     WORD   DidClean;            /* (C) Test whether nametree needs cleaning */
01873     WORD   IfLevel;             /* (C) */
01874     WORD   WhileLevel;         /* (C) */
01875     WORD   SwitchLevel;
01876     WORD   SwitchInArray;
01877     WORD   MaxSwitch;
01878     WORD   LogHandle;           /* (C) The Log File */
01879     WORD   LineLength;          /* (C) */
01880     WORD   StoreHandle;         /* (C) Handle of .str file */
01881     WORD   HideLevel;           /* (C) Hiding indicator */
01882     WORD   lPolyFun;            /* (C) local value of PolyFun */
01883     WORD   lPolyFunInv;         /* (C) local value of Inverse of PolyFun */
01884     WORD   lPolyFunType;        /* (C) local value of PolyFunType */
01885     WORD   lPolyFunExp;
01886     WORD   lPolyFunVar;
01887     WORD   lPolyFunPow;
01888     WORD   SymChangeFlag;       /* (C) */
01889     WORD   CollectPercentage;   /* (C) Collect function percentage */

```

```

01890     WORD    extrasymbols;           /* Flag for the extra symbols output mode */
01891     WORD    PolyRatFunChanged;       /* Keeps track whether we changed in the compiler */
01892     WORD    ToBeInFactors;
01893 #ifdef WITHMPI
01894     WORD    RhsExprInModuleFlag;     /* (C) Set by the compiler if RHS expressions exists. */
01895 #endif
01896     UBYTE    Commercial[COMMERCIALSIZE+2]; /* (C) Message to be printed in statistics */
01897     UBYTE    debugFlags[MAXFLAGS+2];   /* On/Off Flag number(s) */
01898 };
01899 /*
01900     #] C :
01901     #[ S : The S struct defines objects changed at the start of the run (Processor)
01902           Basically only set by the master.
01903 */
01911 struct S_const {
01912     POSITION MaxExprSize;             /* ( ) Maximum size of in/out/sort */
01913 #ifdef WITHPTHREADS
01914     pthread_mutex_t inputslock;
01915     pthread_mutex_t outputslock;
01916     pthread_mutex_t MaxExprSizeLock;
01917 #endif
01918     POSITION *OldOnFile;              /* (S) File positions of expressions */
01919     WORD    *OldNumFactors;          /* ( ) NumFactors in (old) expression */
01920     WORD    *Oldvflags;              /* ( ) vflags in (old) expression */
01921     WORD    *Olduflags;              /* ( ) uflags in (old) expression */
01922 #ifdef WITHFLOAT
01923     void    *delta_l;
01924 #endif
01925     int     NumOldOnFile;             /* (S) Number of expressions in OldOnFile */
01926     int     NumOldNumFactors;         /* (S) Number of expressions in OldNumFactors */
01927     int     MultiThreaded;           /* (S) Are we running multi-threaded? */
01928 #ifdef WITHPTHREADS
01929     int     MasterSort;              /* Final stage of sorting to the master */
01930 #endif
01931 #ifdef WITHMPI
01932     int     printflag;              /* controls MesPrint() on each slave */
01933 #endif
01934     int     Balancing;               /* For second stage loadbalancing */
01935     WORD    ExecMode;                /* (S) */
01936
01937     WORD    CollectOverFlag;         /* (R) Indicates overflow at Collect */
01938 #ifdef WITHPTHREADS
01939     WORD    sLevel;                 /* Copy of AR0.sLevel because it can get messy */
01940 #endif
01941 };
01942 /*
01943     #] S :
01944     #[ R : The R struct defines objects changed at run time.
01945           They determine the environment that has to be transferred
01946           together with a term during multithreaded execution.
01947 */
01956 struct R_const {
01957     FILEDATA StoreData;              /* (O) */
01958     FILEHANDLE Fscr[3];              /* (R) Dollars etc play with it too */
01959     FILEHANDLE FoStage4[2];          /* (R) In Sort. Stage 4. */
01960     POSITION DefPosition;             /* (R) Deferred position of keep brackets. */
01961     FILEHANDLE *infile;              /* (R) Points alternately to Fscr[0] or Fscr[1] */
01962     FILEHANDLE *outfile;             /* (R) Points alternately to Fscr[1] or Fscr[0] */
01963     FILEHANDLE *hidefile;            /* (R) Points to Fscr[2] */
01964
01965     WORD    *CompressBuffer;         /* (M) */
01966     WORD    *ComprTop;               /* (M) */
01967     WORD    *CompressPointer;        /* (R) */
01968     COMPAREDUMMY CompareRoutine;
01969     ULONG   *wranfia;
01970     SBYTE    *moebiustable;
01971
01972     LONG    OldTime;                 /* (R) Zero time. Needed in timer. */
01973     LONG    InInBuf;                 /* (R) Characters in input buffer. Scratch files. */
01974     LONG    InHiBuf;                 /* (R) Characters in hide buffer. Scratch file. */
01975     LONG    pWorkSize;               /* (R) Size of pWorkSpace */
01976     LONG    lWorkSize;               /* (R) Size of lWorkSpace */
01977     LONG    posWorkSize;             /* (R) Size of posWorkSpace */
01978     ULONG   wranfseed;
01979     int     NoCompress;               /* (R) Controls native compression */
01980     int     gzipCompress;            /* (R) Controls gzip compression */
01981     int     Cnumlhs;                 /* Local copy of cbuf[rbufnum].numlhs */
01982     int     outtohide;               /* Indicates that output is directly to hide */
01983 #ifdef WITHPTHREADS
01984     int     exprtoto;                /* The expression to do in parallel mode */
01985 #endif
01986     int     wranfcall;
01987     int     wranfnpair1;
01988     int     wranfnpair2;
01989 #if ( BITSINWORD == 32 )
01990     WORD    PrimeList[5000];
01991     WORD    numinprimelist;

```

```

01992     WORD    notfirstprime;
01993 #endif
01994     WORD    GetFile;                /* (R) Where to get the terms {like Hide} */
01995     WORD    KeptInHold;             /* (R) */
01996     WORD    BracketOn;              /* (R) Intensely used in poly_ */
01997     WORD    MaxBracket;             /* (R) Size of BrackBuf. Changed by poly_ */
01998     WORD    CurDum;                 /* (R) Current maximum dummy number */
01999     WORD    DeferFlag;              /* (R) For deferred brackets */
02000     WORD    TePos;                  /* (R) */
02001     WORD    sLevel;                 /* (R) Sorting level */
02002     WORD    Stage4Name;             /* (R) Sorting only */
02003     WORD    GetOneFile;             /* (R) Getting from hide or regular */
02004     WORD    PolyFun;                /* (C) Number of the PolyFun function */
02005     WORD    PolyFunInv;             /* (C) Number of the Inverse of the PolyFun function */
02006     WORD    PolyFunType;            /* (I) value of PolyFunType */
02007     WORD    PolyFunExp;
02008     WORD    PolyFunVar;
02009     WORD    PolyFunPow;
02010     WORD    Eside;                 /* (I) Tells which side of = sign */
02011     WORD    MaxDum;                 /* Maximum dummy value in an expression */
02012     WORD    level;                  /* Running level in Generator */
02013     WORD    expchanged;             /* (R) Info about expression */
02014     WORD    expflags;               /* (R) Info about expression */
02015     WORD    CurExpr;                /* (S) Number of current expression */
02016     WORD    SortType;               /* A copy of AC.SortType to play with */
02017     WORD    ShortSortCount;         /* For On FewerStatistics 10; */
02018     WORD    modeloptions;
02019     WORD    funoffset;
02020     WORD    moebiustablesizes;
02021 };
02022
02023 /*
02024     #] R :
02025     #[ T : These are variables that stay in each thread during multi threaded execution.
02026 */
02027 struct T_const {
02028 #ifdef WITHPTHREADS
02029     SORTBLOCK SB;
02030 #endif
02031     SORTING *S0;                    /* (-) The thread specific sort buffer */
02032     SORTING *SS;                    /* (R) Current sort buffer */
02033     NESTING Nest;                   /* (R) Nesting of function levels etc. */
02034     NESTING NestStop;               /* (R) */
02035     NESTING NestPoin;               /* (R) */
02036     WORD *BrackBuf;                 /* (R) Bracket buffer. Used by poly_ at runtime. */
02037     STORECACHE StoreCache;          /* (R) Cache for picking up stored expr. */
02038     STORECACHE StoreCacheAlloc;     /* (R) Permanent address of StoreCache to keep valgrind happy */
02039     WORD **pWorkSpace;              /* (R) Workspace for pointers. Dynamic. */
02040     LONG *lWorkSpace;               /* (R) Workspace for LONG. Dynamic. */
02041     POSITION *posWorkSpace;           /* (R) Workspace for file positions */
02042     WORD *WorkSpace;                /* (M) */
02043     WORD *WorkTop;                  /* (M) */
02044     WORD *WorkPointer;               /* (R) Pointer in the Workspace heap. */
02045     int *RepCount;                  /* (M) Buffer for repeat nesting */
02046     int *RepTop;                    /* (M) Top of RepCount buffer */
02047     WORD *WildArgTaken;              /* (N) Stack for wildcard pattern matching */
02048     UWORD *factorials;              /* (T) buffer of factorials. Dynamic. */
02049     WORD *small_power_n;             /* length of the number */
02050     UWORD **small_power;             /* the number */
02051     UWORD *bernoullis;              /* (T) The buffer with Bernoulli numbers. Dynamic. */
02052     WORD *primelist;
02053     LONG *pfac;                     /* (T) array of positions of factorials. Dynamic. */
02054     LONG *pBer;                     /* (T) array of positions of Bernoulli's. Dynamic. */
02055     WORD *TMaddr;                   /* (R) buffer for TestSub */
02056     WORD *WildMask;                 /* (N) Wildcard info during pattern matching */
02057     WORD *previousEfactor;           /* (I) Cache for factors in expressions */
02058     WORD **TermMemHeap;              /* For TermMalloc. Set zero in Checkpoint */
02059     UWORD **NumberMemHeap;           /* For NumberMalloc. Set zero in Checkpoint */
02060     UWORD **CacheNumberMemHeap;      /* For CacheNumberMalloc. Set zero in Checkpoint */
02061     BRACKETINFO *bracketinfo;
02062     WORD **ListPoly;
02063     WORD *ListSymbols;
02064     UWORD *NumMem;
02065     WORD *TopologiesTerm;
02066     WORD *TopologiesStart;
02067 #ifdef WITHFLOAT
02068     WORD *indil;
02069     WORD *indi2;
02070     void *mpf_tab1;
02071     void *mpf_tab2;
02072     void *aux_;
02073     void *auxr_;
02074 #endif
02075     PARTI partitions;
02076     LONG sBer;                      /* (T) Size of the Bernoullis buffer */
02077     LONG pWorkPointer;              /* (R) Offset-pointer in pWorkSpace */
02078     LONG lWorkPointer;              /* (R) Offset-pointer in lWorkSpace */

```



```

02087     LONG    posWorkPointer;      /* (R) Offset-pointer in posWorkSpace */
02088     LONG    InNumMem;
02089     int      sfact;               /* (T) size of the factorials buffer */
02090     int      mfac;               /* (T) size of the pfac array. */
02091     int      ebufnum;            /* (R) extra compiler buffer */
02092     int      fbufnum;            /* extra compiler buffer for factorization cache */
02093     int      allbufnum;          /* extra compiler buffer for id,all */
02094     int      aebufnum;          /* extra compiler buffer for id,all */
02095     int      idallflag;          /* indicates use of id,all buffers */
02096     int      idallnum;
02097     int      idallmaxnum;
02098     int      WildcardBufferSize; /* () local copy for updates */
02099 #ifdef WITHPTHREADS
02100     int      identity;           /* () When we work with B->T */
02101     int      LoadBalancing;     /* Needed for synchronization */
02102 #ifdef WITHSORTBOTS
02103     int      SortBotIn1;        /* Input stream 1 for a SortBot */
02104     int      SortBotIn2;        /* Input stream 2 for a SortBot */
02105 #endif
02106 #endif
02107     int      TermMemMax;        /* For TermMalloc. Set zero in Checkpoint */
02108     int      TermMemTop;        /* For TermMalloc. Set zero in Checkpoint */
02109     int      NumberMemMax;      /* For NumberMalloc. Set zero in Checkpoint */
02110     int      NumberMemTop;      /* For NumberMalloc. Set zero in Checkpoint */
02111     int      CacheNumberMemMax; /* For CacheNumberMalloc. Set zero in Checkpoint */
02112     int      CacheNumberMemTop; /* For CacheNumberMalloc. Set zero in Checkpoint */
02113     int      bracketindexflag;  /* Are brackets going to be indexed? */
02114     int      optimitimes;       /* Number of the evaluation of the MCTS tree */
02115     int      ListSymbolsSize;
02116     int      NumListSymbols;
02117     int      numpoly;
02118     int      LeaveNegative;
02119     int      TrimPower;         /* Indicates trimming in polyratfun expansion */
02120     WORD     small_power_maxx;  /* size of the cache for small powers */
02121     WORD     small_power_maxn;  /* size of the cache for small powers */
02122     WORD     dummysubexp[SUBEXPSIZE+4]; /* () used in normal.c */
02123     WORD     comsym[8];         /* () Used in tools.c = {8,SYMBOL,4,0,1,1,1,3} */
02124     WORD     comnum[4];         /* () Used in tools.c = { 4,1,1,3 } */
02125     WORD     comfun[FUNHEAD+4]; /* () Used in tools.c = {7,FUNCTION,3,0,1,1,3} */
02126                                     /* or { 8,FUNCTION,4,0,0,1,1,3 } */
02127     WORD     comind[7];         /* () Used in tools.c = {7,INDEX,3,0,1,1,3} */
02128     WORD     MinVecArg[7+ARGHEAD]; /* (N) but should be more local */
02129     WORD     FunArg[4+ARGHEAD+FUNHEAD]; /* (N) but can be more local */
02130     WORD     locwildvalue[SUBEXPSIZE]; /* () Used in argument.c = {SUBEXPRESSION,SUBEXPSIZE,0,0,0} */
02131     WORD     mulpat[SUBEXPSIZE+5]; /* () Used in argument.c = {TYPEMULT, SUBEXPSIZE+3, 0, */
02132                                     /* SUBEXPRESSION, SUBEXPSIZE, 0, 1, 0, 0, 0 } */
02133     WORD     proexp[SUBEXPSIZE+5]; /* () Used in poly.c */
02134     WORD     TMout[40];         /* (R) Passing info */
02135     WORD     TMbuff;           /* (R) Communication between TestSub and Genera */
02136     WORD     TMdolfac;          /* factor number for dollar */
02137     WORD     nfac;             /* (T) Number of highest stored factorial */
02138     WORD     nBer;             /* (T) Number of highest Bernoulli number. */
02139     WORD     mBer;             /* (T) Size of buffer pBer. */
02140     WORD     PolyAct;           /* (R) Used for putting the PolyFun at end. ini at 0 */
02141     WORD     RecFlag;          /* (R) Used in TestSub. ini at zero. */
02142     WORD     inprimelist;
02143     WORD     sizeprimelist;
02144     WORD     fromindex;        /* Tells the compare routine whether call from index */
02145     WORD     setinterntopo;     /* Set of internal momenta for topogen */
02146     WORD     setexterntopo;     /* Set of external momenta for topogen */
02147     WORD     TopologiesLevel;
02148     WORD     TopologiesOptions[2];
02149 #ifdef WITHFLOAT
02150     WORD     FloatPos;
02151     WORD     SortFloatMode;
02152 #endif
02153 };
02154 /*
02155     #] T :
02156     #[ N : The N struct contains variables used in running information
02157            that is inside blocks that should not be split, like pattern
02158            matching, traces etc. They are local for each thread.
02159            They don't need initializations.
02160 */
02169 struct N_const {
02170     POSITION OldPosIn;          /* (R) Used in sort. */
02171     POSITION OldPosOut;         /* (R) Used in sort */
02172     POSITION theposition;       /* () Used in index.c */
02173     WORD     *EndNest;         /* (R) Nesting of function levels etc. */
02174     WORD     *Frozen;          /* (R) Bracket info */
02175     WORD     *FullProto;       /* (R) Prototype of a subexpression or table */
02176     WORD     *cTerm;           /* (R) Current term for coef_ and term_ */
02177     int      RepPoint;         /* (R) Pointer in RepCount buffer. Tracks repeat */
02178     WORD     *WildValue;       /* (N) Wildcard info during pattern matching */
02179     WORD     *WildStop;        /* (N) Wildcard info during pattern matching */
02180     WORD     *argaddress;      /* (N) Used in pattern matching of arguments */
02181     WORD     *RepFunList;      /* (N) For pattern matching */

```

```

02182 WORD *patstop; /* (N) Used in pattern matching */
02183 WORD *terstop; /* (N) Used in pattern matching */
02184 WORD *terstart; /* (N) Used in pattern matching */
02185 WORD *terfirstcomm; /* (N) Used in pattern matching */
02186 WORD *DumFound; /* (N) For renumbering indices {make local?} */
02187 WORD **DumPlace; /* (N) For renumbering indices {make local?} */
02188 WORD **DumFunPlace; /* (N) For renumbering indices {make local?} */
02189 WORD *UsedSymbol; /* (N) When storing terms of a global expr. */
02190 WORD *UsedVector; /* (N) When storing terms of a global expr. */
02191 WORD *UsedIndex; /* (N) When storing terms of a global expr. */
02192 WORD *UsedFunction; /* (N) When storing terms of a global expr. */
02193 WORD *MaskPointer; /* (N) For wildcard pattern matching */
02194 WORD *ForFindOnly; /* (N) For wildcard pattern matching */
02195 WORD *findTerm; /* (N) For wildcard pattern matching */
02196 WORD *findPattern; /* (N) For wildcard pattern matching */
02197 #ifdef WITHZLIB
02198 Bytef **ziobufnum; /* () Used in compress.c */
02199 Bytef *ziobuffers; /* () Used in compress.c */
02200 #endif
02201 WORD *dummyrenumlist; /* () Used in execute.c and store.c */
02202 int *funargs; /* () Used in lus.c */
02203 WORD **funlocs; /* () Used in lus.c */
02204 int *funinds; /* () Used in lus.c */
02205 UWORD *NoScrat2; /* () Used in normal.c */
02206 WORD *ReplaceScrat; /* () Used in normal.c */
02207 TRACES *tracestack; /* () used in opera.c */
02208 WORD *selecttermundo; /* () Used in pattern.c */
02209 WORD *patternbuffer; /* () Used in pattern.c */
02210 WORD *termbuffer; /* () Used in pattern.c */
02211 WORD **PoinScrat; /* () used in reshuf.c */
02212 WORD **FunScrat; /* () used in reshuf.c */
02213 WORD *RenumScrat; /* () used in reshuf.c */
02214 FUN_INFO *FunInfo; /* () Used in smart.c */
02215 WORD **SplitScrat; /* () Used in sort.c */
02216 WORD **SplitScrat1; /* () Used in sort.c */
02217 SORTING **FunSorts; /* () Used in sort.c */
02218 UWORD *SoScratC; /* () Used in sort.c */
02219 WORD *listinprint; /* () Used in proces.c and message.c */
02220 WORD *currentTerm; /* () Used in proces.c and message.c */
02221 WORD **arglist; /* () Used in function.c */
02222 int *tlistbuf; /* () used in lus.c */
02223 #ifdef WHICHSUBEXPRESSION
02224 UWORD *BinoScrat; /* () Used in proces.c */
02225 #endif
02226 WORD *compressSpace; /* () Used in sort.c */
02227 #ifdef WITHPTHREADS
02228 THREADBUCKET *threadbuck;
02229 EXPRESSIONS expr;
02230 #endif
02231 UWORD *SHcombi;
02232 WORD *poly_vars;
02233 UWORD *cmmod; /* Local setting of modulus. Pointer to value. */
02234 SHvariables SHvar;
02235 LONG deferskipped; /* () Used in proces.c store.c and parallel.c */
02236 LONG InScrat; /* () Used in sort.c */
02237 LONG SplitScratSize; /* () Used in sort.c */
02238 LONG InScrat1; /* () Used in sort.c */
02239 LONG SplitScratSize1; /* () Used in sort.c */
02240 LONG ninterms; /* () Used in proces.c and sort.c */
02241 #ifdef WITHPTHREADS
02242 LONG inputnumber; /* () For use in redefine */
02243 LONG lastinindex;
02244 #endif
02245 #ifdef WHICHSUBEXPRESSION
02246 LONG last2; /* () Used in proces.c */
02247 LONG last3; /* () Used in proces.c */
02248 #endif
02249 LONG SHcombiSize;
02250 int NumTotWildArgs; /* (N) Used in pattern matching */
02251 int UseFindOnly; /* (N) Controls pattern routines */
02252 int UsedOtherFind; /* (N) Controls pattern routines */
02253 int ErrorInDollar; /* (R) */
02254 int numfargs; /* () Used in lus.c */
02255 int numflocs; /* () Used in lus.c */
02256 int nargs; /* () Used in lus.c */
02257 int tohunt; /* () Used in lus.c */
02258 int numoffuns; /* () Used in lus.c */
02259 int funisize; /* () Used in lus.c */
02260 int RSsize; /* () Used in normal.c */
02261 int numracesctack; /* () used in opera.c */
02262 int intracestack; /* () used in opera.c */
02263 int numfuninfo; /* () Used in smart.c */
02264 int NumFunSorts; /* () Used in sort.c */
02265 int MaxFunSorts; /* () Used in sort.c */
02266 int arglistsize; /* () Used in function.c */
02267 int tlistsize; /* () used in lus.c */
02268 int filenum; /* () used in setfile.c */

```



```

02269     int        compressSize;           /* () Used in sort.c */
02270     int        polysortflag;
02271     int        nogroundlevel;          /* () Used to see whether pattern matching at groundlevel */
02272     int        subsubveto;             /* () Sabotage combining subexpressions in TestSub */
02273     WORD       MaxRenumScratch;        /* () used in reshuf.c */
02274     WORD       oldtype;                /* (N) WildCard info at pattern matching */
02275     WORD       oldvalue;               /* (N) WildCard info at pattern matching */
02276     WORD       NumWild;                /* (N) Used in Wildcard */
02277     WORD       RepFunNum;              /* (N) Used in pattern matching */
02278     WORD       DisOrderFlag;           /* (N) Disorder option? Used in pattern matching */
02279     WORD       WildDirt;               /* (N) dirty in wildcard substitution. */
02280     WORD       NumFound;               /* (N) in reshuf only. Local? */
02281     WORD       WildReserve;            /* (N) Used in the wildcards */
02282     WORD       TeInFun;                /* (R) Passing type of action */
02283     WORD       TeSuOut;                /* (R) Passing info. Local? */
02284     WORD       WildArgs;               /* (R) */
02285     WORD       WildEat;                /* (R) */
02286     WORD       PolyNormFlag;           /* (R) For polynomial arithmetic */
02287     WORD       PolyFunTodo;            /* deals with expansions and multiplications */
02288     WORD       sizeselecttermundo;     /* () Used in pattern.c */
02289     WORD       patternbuffersize;      /* () Used in pattern.c */
02290     WORD       numlistinprint;         /* () Used in process.c */
02291     WORD       ncmod;                  /* () used as some type of flag to disable */
02292     WORD       ExpectedSign;
02293     WORD       SignCheck;
02294     WORD       IndDum;                 /* Active dummy indices */
02295     WORD       poly_num_vars;
02296     WORD       idfunctionflag;
02297     WORD       poly_vars_type;         /* type of allocation. For free. */
02298     WORD       tryterm;                /* For EndSort(...,2) */
02299 #ifdef WHICHSUBEXPRESSION
02300     WORD       nbino;                  /* () Used in proces.c */
02301     WORD       last1;                  /* () Used in proces.c */
02302 #endif
02303 };
02304
02305 /*
02306     #] N :
02307     #[ O : The O struct concerns output variables
02308 */
02317 struct O_const {
02318     FILEDATA   SaveData;               /* (O) */
02319     STOREHEADER SaveHeader;            /* () System Independent save-Files */
02320     OPTIMIZERESULT OptimizeResult;
02321     UBYTE      *OutputLine;            /* (O) Sits also in debug statements */
02322     UBYTE      *OutStop;                /* (O) Top of OutputLine buffer */
02323     UBYTE      *OutFill;                /* (O) Filling point in OutputLine buffer */
02324     WORD       *bracket;                /* (O) For writing brackets */
02325     WORD       *termbuf;                /* (O) For writing terms */
02326     WORD       *tabstring;
02327     UBYTE      *wpos;                  /* (O) Only when storing file {local?} */
02328     UBYTE      *wpoin;                 /* (O) Only when storing file {local?} */
02329     UBYTE      *DollarOutBuffer;        /* (O) Outputbuffer for Dollars */
02330     UBYTE      *CurBufWrt;            /* (O) Name of currently written expr. */
02331     void       (*FlipWORD)(UBYTE *);   /* () Function pointers for translations. Initialized by
ReadSaveHeader() */
02332     void       (*FlipLONG)(UBYTE *);
02333     void       (*FlipPOS)(UBYTE *);
02334     void       (*FlipPOINTER)(UBYTE *);
02335     void       (*ResizeData)(UBYTE *,int,UBYTE *,int);
02336     void       (*ResizeWORD)(UBYTE *,UBYTE *);
02337     void       (*ResizeNCWORD)(UBYTE *,UBYTE *);
02338     void       (*ResizeLONG)(UBYTE *,UBYTE *);
02339     void       (*ResizePOS)(UBYTE *,UBYTE *);
02340     void       (*ResizePOINTER)(UBYTE *,UBYTE *);
02341     void       (*CheckPower)(UBYTE *);
02342     void       (*RenumberVec)(UBYTE *);
02343     DICTIONARY **Dictionaries;
02344     UBYTE      *tensorList;            /* Dynamically allocated list with functions that are tensorial. */
02345     WORD       *inscheme;              /* for feeding a Horner scheme to Optimize */
02346 #ifdef WITHFLOAT
02347     UBYTE      *floatspace;
02348     LONG       floatsize;
02349 #endif
02350 /*-----Leave NumInBrack as first non-pointer. This is used by the checkpoints---*/
02351     LONG       NumInBrack;              /* (O) For typing [] option in print */
02352     LONG       wlen;                   /* (O) Used to store files. */
02353     LONG       DollarOutSizeBuffer;     /* (O) Size of DollarOutBuffer */
02354     LONG       DollarInOutBuffer;       /* (O) Characters in DollarOutBuffer */
02355 #if defined(MBSD) && defined(MICROTIME)
02356     LONG       wrap;                   /* (O) For statistics time. wrap around */
02357     LONG       wrapnum;                 /* (O) For statistics time. wrap around */
02358 #endif
02359     OPTIMIZE   Optimize;
02360     int        OutInBuffer;             /* (O) Which routine does the writing */
02361     int        NoSpacesInNumbers;      /* For very long numbers */
02362     int        BlockSpaces;            /* For very long numbers */

```

```

02363     int     CurrentDictionary;
02364     int     SizeDictionaries;
02365     int     NumDictionaries;
02366     int     CurDictNumbers;
02367     int     CurDictVariables;
02368     int     CurDictSpecials;
02369     int     CurDictFunWithArgs;
02370     int     CurDictNumberWarning;
02371     int     CurDictNotInFunctions;
02372     int     CurDictInDollars;
02373     int     gNumDictionaries;
02374     int     IndentSpace;          /* For indentation in output */
02375 #ifdef WITHFLOAT
02376     int     FloatPrec;
02377 #endif
02378     WORD     schemenum;          /* for feeding a Horner scheme to Optimize */
02379     WORD     transFlag;          /* () >0 indicates that translations have to be done */
02380     WORD     powerFlag;          /* () >0 indicates that some exponents/powers had to be adjusted
*/
02381     WORD     mpower;             /* For maxpower adjustment to larger value */
02382     WORD     resizeFlag;        /* () >0 indicates that something went wrong when resizing words
*/
02383     WORD     bufferedInd;        /* () Contains extra INDEXENTRIES, see ReadSaveIndex() for an
explanation */
02384     WORD     OutSkip;            /* (0) How many chars to skip in output line */
02385     WORD     IsBracket;          /* (0) Controls brackets */
02386     WORD     InFbrack;          /* (0) For writing only */
02387     WORD     PrintType;          /* (0) */
02388     WORD     FortFirst;          /* (0) Only in sch.c */
02389     WORD     DoubleFlag;         /* (0) Output in double precision */
02390     WORD     FactorMode;         /* When the output should be written as factors */
02391     WORD     FactorNum;          /* Number of factor currently treated */
02392     WORD     ErrorBlock;
02393     WORD     OptimizationLevel;  /* Level of optimization in the output */
02394     UBYTE     FortDotChar;       /* (0) */
02395 /*
02396 For the padding, please count also the number of int's in the OPTIMIZE struct.
02397 */
02398 };
02399 /*
02400     #] O :
02401     #[ X : The X struct contains variables that deal with the external channel
02402 */
02410 struct X_const {
02411     UBYTE *currentPrompt;
02412     UBYTE *shellname;           /* if !=NULL (default is "/bin/sh -c"), start in
the specified subshell*/
02413     UBYTE *stderrname;         /* If !=NULL (default if "/dev/null"), stderr is
redirected to the specified file*/
02414     int timeout;               /* timeout to initialize preset channels.
If timeout<0, the preset channels are
already initialized*/
02415     int killSignal;            /* signal number, SIGKILL by default*/
02416     int killWholeGroup;        /* if 0, the signal is sent only to a process,
if !=0 (default) is sent to a whole process group*/
02417     int daemonize;             /* if !=0 (default), start in a daemon mode */
02418     int currentExternalChannel;
02419 };
02420 /*
02421     #] X :
02422     #[ Definitions :
02423
02424 Note: we changed the definition from C to Cc in version 5.
02425 The reason is that everywhere the pointer to the compiler
02426 buffer was called C as well. Somehow that gave no problems but
02427 who knows what the future might bring.
02428 */
02429 #ifdef WITHPTHREADS
02430 typedef struct AllGlobals {
02431     struct M_const M;
02432     struct C_const Cc;
02433     struct S_const S;
02434     struct O_const O;
02435     struct P_const P;
02436     struct X_const X;
02437 } ALLGLOBALS;
02438
02439 typedef struct AllPrivates {
02440     struct R_const R;
02441     struct N_const N;
02442     struct T_const T;
02443 } ALLPRIVATES;
02444 #else
02445
02446
02447
02448
02449
02450
02451
02452
02453
02454
02455
02456
02457
02458
02459
02460
02461
02462
02463

```

```

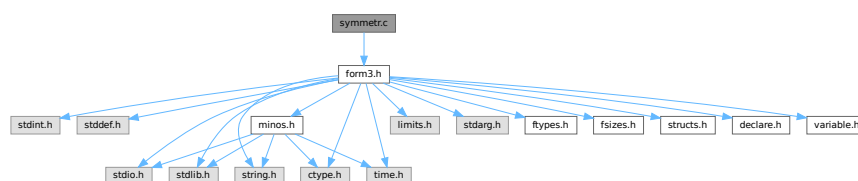
02468 typedef struct AllGlobals {
02469     struct M_const M;
02470     struct C_const Cc;
02471     struct S_const S;
02472     struct R_const R;
02473     struct N_const N;
02474     struct O_const O;
02475     struct P_const P;
02476     struct T_const T;
02477     struct X_const X;
02478 } ALLGLOBALS;
02479
02480 #endif
02481
02482 /*
02483     #] Definitions :
02484     #] A :
02485     #[ FG :
02486 */
02487
02488 #ifdef WITHPTHREADS
02489 #define PHEAD ALLPRIVATES *B,
02490 #define PHEAD0 ALLPRIVATES *B
02491 #define BHEAD B,
02492 #define BHEAD0 B
02493 #else
02494 #define PHEAD
02495 #define PHEAD0 void
02496 #define BHEAD
02497 #define BHEAD0
02498 #endif
02499
02500 typedef int (*WCN) (PHEAD WORD *,WORD *,WORD,WORD);
02501 typedef int (*WCN2) (PHEAD WORD *,WORD *);
02502
02503 typedef WORD (*COMPARE) (PHEAD WORD *,WORD *,WORD);
02504
02516 typedef struct FixedGlobals {
02517     WCN      Operation[8];
02518     WCN2     OperaFind[6];
02519     char     *VarType[10];
02520     char     *ExprStat[21];
02521     char     *FunNam[2];
02522     char     *swmes[3];
02523     char     *fname;
02524     char     *fname2;
02525     UBYTE    *s_one;
02526     WORD     fnamebase;
02527     WORD     fname2base;
02528     WORD     fnamesize;
02529     WORD     fname2size;
02530     UINT     cTable[256];
02531 } FIXEDGLOBALS;
02532
02533 /*
02534     #] FG :
02535 */
02536
02537 #endif

```

4.139 symmetr.c File Reference

#include "form3.h"

Include dependency graph for symmetr.c:



Functions

- int [MatchE](#) (PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
- int [Permute](#) ([PERM](#) *perm, WORD first)
- int [PermuteP](#) ([PERMP](#) *perm, WORD first)
- int [Distribute](#) ([DISTRIBUTE](#) *d, WORD first)
- int [MatchCy](#) (PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
- int [FunMatchCy](#) (PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
- int [FunMatchSy](#) (PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
- int [MatchArgument](#) (PHEAD WORD *arg, WORD *pat)

4.139.1 Detailed Description

The routines that deal with the pattern matching of functions with symmetric properties.

Definition in file [symmetr.c](#).

4.139.2 Function Documentation

4.139.2.1 MatchE()

```
int MatchE (  
    PHEAD WORD * pattern,  
    WORD * fun,  
    WORD * inter,  
    WORD par )
```

Definition at line [56](#) of file [symmetr.c](#).

4.139.2.2 Permute()

```
int Permute (  
    PERM * perm,  
    WORD first )
```

Definition at line [444](#) of file [symmetr.c](#).

4.139.2.3 PermuteP()

```
int PermuteP (  
    PERMP * perm,  
    WORD first )
```

Definition at line [478](#) of file [symmetr.c](#).

4.139.2.4 Distribute()

```
int Distribute (
    DISTRIBUTE * d,
    WORD first )
```

Definition at line 510 of file [symmetr.c](#).

4.139.2.5 MatchCy()

```
int MatchCy (
    PHEAD WORD * pattern,
    WORD * fun,
    WORD * inter,
    WORD par )
```

Definition at line 571 of file [symmetr.c](#).

4.139.2.6 FunMatchCy()

```
int FunMatchCy (
    PHEAD WORD * pattern,
    WORD * fun,
    WORD * inter,
    WORD par )
```

Definition at line 1067 of file [symmetr.c](#).

4.139.2.7 FunMatchSy()

```
int FunMatchSy (
    PHEAD WORD * pattern,
    WORD * fun,
    WORD * inter,
    WORD par )
```

Definition at line 1525 of file [symmetr.c](#).

4.139.2.8 MatchArgument()

```
int MatchArgument (
    PHEAD WORD * arg,
    WORD * pat )
```

Definition at line 2052 of file [symmetr.c](#).

4.140 symmetr.c

[Go to the documentation of this file.](#)

```

00001
00006 /* #[ License : */
00007 /*
00008 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009 *   When using this file you are requested to refer to the publication
00010 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011 *   This is considered a matter of courtesy as the development was paid
00012 *   for by FOM the Dutch physics granting agency and we would like to
00013 *   be able to track its scientific use to convince FOM of its value
00014 *   for the community.
00015 *
00016 *   This file is part of FORM.
00017 *
00018 *   FORM is free software: you can redistribute it and/or modify it under the
00019 *   terms of the GNU General Public License as published by the Free Software
00020 *   Foundation, either version 3 of the License, or (at your option) any later
00021 *   version.
00022 *
00023 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00026 *   details.
00027 *
00028 *   You should have received a copy of the GNU General Public License along
00029 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00030 */
00031 /* #[ License : */
00032 /*
00033     #[ Includes : function.c
00034 */
00035
00036 #include "form3.h"
00037
00038 /*
00039     #[ Includes :
00040     #[ MatchE :                WORD MatchE(pattern,fun,inter,par)
00041
00042         Matches symmetric and antisymmetric tensors.
00043         Pattern and fun point at a tensor.
00044         Problem is the wilddcarding and all its possible permutations.
00045         This routine loops over all of them and calls for each
00046         possible wilddcarding the recursion in ScanFunctions.
00047         Note that this can be very costly.
00048
00049         Originally this routine did only Levi Civita tensors and hence
00050         it dealt only with commuting objects.
00051         Because of the backtracking we cannot fall back to the calling
00052         ScanFunctions routine and check the sequence of functions when
00053         non-commuting objects are involved.
00054 */
00055
00056 int MatchE(PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
00057 {
00058     GETBIDENTITY
00059     WORD *m, *t, *r, i;
00060     int retval;
00061     WORD *mstop, *tstop, j, newvalue, newfun;
00062     WORD fixvec[MAXMATCH], wvec[MAXMATCH], fixind[MAXMATCH], wcind[MAXMATCH];
00063     WORD tfixvec[MAXMATCH], tfixind[MAXMATCH];
00064     WORD vwc, vfix, ifix, iwc, tvfix, tifix, nv, ni;
00065     WORD sign = 0, *rstop, first1, first2, first3, funwild;
00066     WORD *OldWork, nwstore, oRepFunNum;
00067     PERM perm1, perm2;
00068     DISTRIBUTE distr;
00069     WORD *newpat, /* *newter, *instart, */ offset;
00070 /* instart = fun; */
00071     offset = WORDDIF(fun, AN.terstart);
00072     if ( pattern[1] != fun[1] ) return(0);
00073     if ( *pattern >= FUNCTION+WILDOFFSET ) {
00074         if ( CheckWild(BHEAD *pattern-WILDOFFSET, FUNTOFUN, *fun, &newfun) ) return(0);
00075         funwild = 1;
00076     }
00077     else funwild = 0;
00078     mstop = pattern + pattern[1];
00079     tstop = fun + fun[1];
00080     m = pattern + FUNHEAD;
00081     t = fun + FUNHEAD;
00082     while ( m < mstop ) {
00083         if ( *m != *t ) break;
00084         m++; t++;
00085     }
00086     if ( m >= mstop ) {

```

```

00087     AN.RepFunList[AN.RepFunNum++] = offset;
00088     AN.RepFunList[AN.RepFunNum++] = 0;
00089     newpat = pattern + pattern[1];
00090     if ( funwild ) {
00091         m = AN.WildValue;
00092         t = OldWork = AT.WorkPointer;
00093         nwstore = i = (m[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
00094         r = AT.WildMask;
00095         if ( i > 0 ) {
00096             do {
00097                 *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
00098             } while ( --i > 0 );
00099         }
00100         if ( t >= AT.WorkTop ) {
00101             MLOCK(ErrorMessageLock);
00102             MesWork();
00103             MUNLOCK(ErrorMessageLock);
00104             return(-1);
00105         }
00106         AT.WorkPointer = t;
00107         AddWild(BHEAD *pattern-WILDOFFSET,FUNTOFUN,newfun);
00108         if ( newpat >= AN.patstop ) {
00109             if ( AN.UseFindOnly == 0 ) {
00110                 if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
00111                     AN.UsedOtherFind = 1;
00112                     return(1);
00113                 }
00114                 retval = 0;
00115             }
00116             else return(1);
00117         }
00118         else {
00119             /* newter = instart; */
00120             retval = ScanFunctions(BHEAD newpat,inter,par);
00121         }
00122         if ( retval == 0 ) {
00123             m = AN.WildValue;
00124             t = OldWork; r = AT.WildMask; i = nwstore;
00125             if ( i > 0 ) {
00126                 do {
00127                     *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *r++ = *t++;
00128                 } while ( --i > 0 );
00129             }
00130             AT.WorkPointer = OldWork;
00131             return(retval);
00132         }
00133     }
00134     else {
00135         if ( newpat >= AN.patstop ) {
00136             if ( AN.UseFindOnly == 0 ) {
00137                 if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
00138                     AN.UsedOtherFind = 1;
00139                     return(1);
00140                 }
00141                 else return(0);
00142             }
00143             else return(1);
00144         }
00145         /* newter = instart; */
00146         retval = ScanFunctions(BHEAD newpat,inter,par);
00147         return(retval);
00148     }
00149     /*
00150     Now the recursion
00151     */
00152 }
00153 /*
00154 Strategy:
00155 1: match the fixed arguments
00156 2: match, permuting the wildcards if needed.
00157 3: keep track of sign.
00158 */
00159 vwc = 0;
00160 vfix = 0;
00161 ifix = 0;
00162 iwc = 0;
00163 r = pattern+FUNHEAD;
00164 while ( r < mstop ) {
00165     if ( *r < (AM.OffsetVector+WILDOFFSET) ) {
00166         fixvec[vfix++] = *r; /* Fixed vectors */
00167         sign += vwc + ifix + iwc;
00168     }
00169     else if ( *r < MINSPEC ) {
00170         wcvec[vwc++] = *r; /* Wildcard vectors */
00171         sign += ifix + iwc;
00172     }
00173     else if ( *r < (AM.OffsetIndex+WILDOFFSET) ) {

```

```

00174         fixind[ifix++] = *r;          /* Fixed indices */
00175         sign += iwc;
00176     }
00177     else if ( *r < (AM.OffsetIndex+(WILDOFFSET«1)) ) {
00178         wcind[iwc++] = *r;          /* Wildcard indices */
00179     }
00180     else {
00181         fixind[ifix++] = *r;          /* Generated indices ~ fixed */
00182         sign += iwc;
00183     }
00184     r++;
00185 }
00186 if ( iwc == 0 && vwc == 0 ) return(0);
00187 tvfix = tifix = 0;
00188 t = fun + FUNHEAD;
00189 m = fixvec;
00190 mstop = m + vfix;
00191 r = fixind;
00192 rstop = r + ifix;
00193 nv = 0; ni = 0;
00194 while ( t < tstop ) {
00195     if ( *t < 0 ) {
00196         nv++;
00197         if ( m < mstop && *t == *m ) {
00198             m++;
00199         }
00200         else {
00201             sign += WORDDIF(mstop,m);
00202             tfixvec[tfix++] = *t;
00203         }
00204     }
00205     else {
00206         ni++;
00207         if ( r < rstop && *r == *t ) {
00208             r++;
00209         }
00210         else {
00211             sign += WORDDIF(rstop,r);
00212             tfixind[tifix++] = *t;
00213         }
00214     }
00215     t++;
00216 }
00217 if ( m < mstop || r < rstop ) return(0);
00218 if ( tvfix < vwc || (tvfix+tifix) < (vwc+iwc) ) return(0);
00219 sign += ( nv - vfix - vwc ) & ni;
00220 /*
00221     Take now the wildcards that have an assignment already.
00222     See whether they match.
00223 */
00224 {
00225     WORD *wv, *wm, n;
00226     wm = AT.WildMask;
00227     wv = AN.WildValue;
00228     n = AN.NumWild;
00229     do {
00230         if ( *wm ) {
00231             if ( *wv == VECTOVEC ) {
00232                 for ( ni = 0; ni < vwc; ni++ ) {
00233                     if ( wcvec[ni]-WILDOFFSET == wv[2] ) { /* Has been assigned */
00234                         sign += ni;
00235                         vwc--;
00236                         while ( ni < vwc ) {
00237                             wcvec[ni] = wcvec[ni+1];
00238                             ni++;
00239                         }
00240                     /* TryVect: */
00241                     for ( ni = 0; ni < tvfix; ni++ ) {
00242                         if ( tfixvec[ni] == wv[3] ) {
00243                             sign += ni;
00244                             tvfix--;
00245                             while ( ni < tvfix ) {
00246                                 tfixvec[ni] = tfixvec[ni+1];
00247                                 ni++;
00248                             }
00249                             goto NextWV;
00250                         }
00251                     }
00252                     return(0);
00253                 }
00254             }
00255         }
00256         else if ( *wv == INDTOIND ) {
00257             for ( ni = 0; ni < iwc; ni++ ) {
00258                 if ( wcind[ni]-WILDOFFSET == wv[2] ) { /* Has been assigned */
00259                     sign += ni;
00260                     iwc--;

```



```

00261         while ( ni < iwc ) {
00262             wcind[ni] = wcind[ni+1];
00263             ni++;
00264         }
00265         for ( ni = 0; ni < tfix; ni++ ) {
00266             if ( tfixind[ni] == wv[3] ) {
00267                 sign += ni;
00268                 tfix--;
00269                 while ( ni < tfix ) {
00270                     tfixind[ni] = tfixind[ni+1];
00271                     ni++;
00272                 }
00273                 goto NextWV;
00274             }
00275         }
00276         /* goto TryVect; */
00277         return(0);
00278     }
00279 }
00280 }
00281 }
00282 else if ( *wv == VECTOSUB ) {
00283     for ( ni = 0; ni < vwc; ni++ ) {
00284         if ( wcvect[ni]-WILDOFFSET == wv[2] ) return(0);
00285     }
00286 }
00287 else if ( *wv == INDTO SUB ) {
00288     for ( ni = 0; ni < iwc; ni++ ) {
00289         if ( wcind[ni]-WILDOFFSET == wv[2] ) return(0);
00290     }
00291 }
00292 }
00293 NextWV:
00294     wm++;
00295     wv += wv[1];
00296     n--;
00297     if ( n > 0 ) {
00298         while ( n > 0 && ( *wv == FROMSET || *wv == SETTONUM
00299             || *wv == LOADDOLLAR ) ) { wv += wv[1]; wm++; n--; }
00300     /*
00301         Freak problem: doesn't test for n and ran into a remaining
00302         code equal to SETTONUM followed by a big number and then
00303         ran out of the memory.
00304
00305         while ( *wv == FROMSET || *wv == SETTONUM
00306             || ( *wv == LOADDOLLAR && n > 0 ) ) { wv += wv[1]; wm++; n--; }
00307     */
00308     }
00309     } while ( n > 0 );
00310 }
00311 /*
00312     Now there are only free wildcards left.
00313     Possibly the assigned values ate too many vectors.
00314     The rest has to be done the 'hard way' via permutations.
00315     This is too bad when there are 10 indices.
00316     This could cause 10! tries.
00317     We try to avoid the worst case by using a very special
00318     (somewhat slow) permutation routine that has as its worst
00319     cases some rather unlikely configurations, rather than some
00320     common ones (as would have been the case with the conventional
00321     permutation routine).
00322     assume:
00323         vvvvvvvvvvvviiiiiii      (tvfix in tfixvec and tfix in tfixind)
00324         VVVVVVVVVVIIIIIIIIIII   (vwc in wcvect and iwc in wcind)
00325     Note: all further assignments are possible at this point!
00326     Strategy:
00327         permute v
00328         permute i
00329         loop over the ordered distribution of the leftover v's
00330         through the i's.
00331 */
00332     if ( tvfix < vwc ) { return(0); }
00333     perml.n = tvfix;
00334     perml.sign = 0;
00335     perml.objects = tfixvec;
00336     perm2.n = tfix;
00337     perm2.sign = 0;
00338     perm2.objects = tfixind;
00339     distr.n1 = tvfix - vwc;
00340     distr.n2 = tfix;
00341     distr.obj1 = tfixvec + vwc;
00342     distr.obj2 = tfixind;
00343     distr.out = fixvec;          /* For scratch */
00344     first1 = 1;
00345     /*
00346         Store the current Wildcard assignments
00347     */

```

```

00348     m = AN.WildValue;
00349     t = OldWork = AT.WorkPointer;
00350     nwstore = i = (m[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
00351     r = AT.WildMask;
00352     if ( i > 0 ) {
00353         do {
00354             *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
00355         } while ( --i > 0 );
00356     }
00357     if ( t >= AT.WorkTop ) {
00358         MLOCK(ErrorMessageLock);
00359         MesWork();
00360         MUNLOCK(ErrorMessageLock);
00361         return(-1);
00362     }
00363     AT.WorkPointer = t;
00364     while ( (first1 = Permute(&perm1,first1) ) == 0 ) {
00365         first2 = 1;
00366         while ( (first2 = Permute(&perm2,first2) ) == 0 ) {
00367             first3 = 1;
00368             while ( (first3 = Distribute(&distr,first3) ) == 0 ) {
00369 /*
00370                 Make now the wildcard assignments
00371 */
00372                 for ( i = 0; i < vwc; i++ ) {
00373                     j = wcvec[i] - WILDOFFSET;
00374                     if ( CheckWild(BHEAD j,VECTOVEC,tfixvec[i],&newvalue) )
00375                         goto NoCaseB;
00376                     AddWild(BHEAD j,VECTOVEC,newvalue);
00377                 }
00378                 for ( i = 0; i < iwc; i++ ) {
00379                     j = wcind[i] - WILDOFFSET;
00380                     if ( CheckWild(BHEAD j,INDTOIND,fixvec[i],&newvalue) )
00381                         goto NoCaseB;
00382                     AddWild(BHEAD j,INDTOIND,newvalue);
00383                 }
00384 /*
00385                 Go into the recursion
00386 */
00387                 oRepFunNum = AN.RepFunNum;
00388                 AN.RepFunList[AN.RepFunNum++] = offset;
00389                 AN.RepFunList[AN.RepFunNum++] =
00390                     ( perm1.sign + perm2.sign + distr.sign + sign ) & 1;
00391                 newpat = pattern + pattern[1];
00392                 if ( funwild ) AddWild(BHEAD *pattern-WILDOFFSET,FUNTOFUN,newfun);
00393                 if ( newpat >= AN.patstop ) {
00394                     if ( AN.UseFindOnly == 0 ) {
00395                         if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
00396                             AN.UsedOtherFind = 1;
00397                             return(1);
00398                         }
00399                     }
00400                     else return(1);
00401                 }
00402                 else {
00403 /*
00404                     newter = instart; */
00405                     if ( ScanFunctions(BHEAD newpat,inter,par) ) { return(1); }
00406 /*
00407                     Restore the old Wildcard assignments
00408 */
00409                     AN.RepFunNum = oRepFunNum;
00410 NoCaseB:
00411                     m = AN.WildValue;
00412                     t = OldWork; r = AT.WildMask; i = nwstore;
00413                     if ( i > 0 ) {
00414                         do {
00415                             *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
00416                         } while ( --i > 0 );
00417                     }
00418                     AT.WorkPointer = t;
00419                 }
00420             }
00421             AT.WorkPointer = OldWork;
00422             return(0);
00423         }
00424     }
00425 /*
00426     #] MatchE :
00427     #[ Permute :          WORD Permute(perm,first)
00428
00429     Special permutation function.
00430     Works recursively.
00431     The aim is to cycle through in as fast a way as possible,
00432     to take care that each object hits the various positions
00433     already early in the game.
00434

```

```

00435      Start at two: -> cycle of two
00436      then three -> cycle of three
00437      etc;
00438      The innermost cycle is the longest. This is the opposite
00439      of the usual way of generating permutations and it is
00440      certainly not the fastest one. It allows for the fastest
00441      hit in the assignment of wildcards though.
00442  */
00443
00444  int Permute(PERM *perm, WORD first)
00445  {
00446      WORD *s, c, i, j;
00447      if ( first ) {
00448          perm->sign = ( perm->sign <= 1 ) ? 0: 1;
00449          for ( i = 0; i < perm->n; i++ ) perm->cycle[i] = 0;
00450          return(0);
00451      }
00452      i = perm->n;
00453      while ( --i > 0 ) {
00454          s = perm->objects;
00455          c = s[0];
00456          j = i;
00457          while ( --j >= 0 ) { *s = s[1]; s++; }
00458          *s = c;
00459          if ( ( i & 1 ) != 0 ) perm->sign ^= 1;
00460          if ( perm->cycle[i] < i ) {
00461              (perm->cycle[i])++;
00462              return(0);
00463          }
00464          else {
00465              perm->cycle[i] = 0;
00466          }
00467      }
00468      return(1);
00469  }
00470
00471  /*
00472      #] Permute :
00473      #[ PermuteP :          WORD PermuteP(perm,first)
00474
00475      Like Permute, but works on an array of pointers
00476  */
00477
00478  int PermuteP(PERMP *perm, WORD first)
00479  {
00480      WORD **s, *c, i, j;
00481      if ( first ) {
00482          perm->sign = ( perm->sign <= 1 ) ? 0: 1;
00483          for ( i = 0; i < perm->n; i++ ) perm->cycle[i] = 0;
00484          return(0);
00485      }
00486      i = perm->n;
00487      while ( --i > 0 ) {
00488          s = perm->objects;
00489          c = s[0];
00490          j = i;
00491          while ( --j >= 0 ) { *s = s[1]; s++; }
00492          *s = c;
00493          if ( ( i & 1 ) != 0 ) perm->sign ^= 1;
00494          if ( perm->cycle[i] < i ) {
00495              (perm->cycle[i])++;
00496              return(0);
00497          }
00498          else {
00499              perm->cycle[i] = 0;
00500          }
00501      }
00502      return(1);
00503  }
00504
00505  /*
00506      #] PermuteP :
00507      #[ Distribute :
00508  */
00509
00510  int Distribute(DISTRIBUTE *d, WORD first)
00511  {
00512      WORD *to, *from, *inc, *from2, i, j;
00513      if ( first ) {
00514          d->n = d->n1 + d->n2;
00515          to = d->out;
00516          from = d->obj2;
00517          for ( i = 0; i < d->n2; i++ ) {
00518              d->cycle[i] = 0;
00519              *to++ = *from++;
00520          }
00521          from = d->obj1;

```

```

00522         while ( i < d->n ) {
00523             d->cycle[i++] = 1;
00524             *to++ = *from++;
00525         }
00526         d->sign = 0;
00527         return(0);
00528     }
00529     if ( d->n1 == 0 || d->n2 == 0 ) return(1);
00530     j = 0;
00531     i = 0;
00532     inc = d->cycle;
00533     from = inc + d->n;
00534     while ( *inc ) { j++; inc++; }
00535     while ( inc < from && !*inc ) { i++; inc++; }
00536     if ( inc >= from ) return(1);
00537     d->sign ^= ((i&j)-j+1) & 1;
00538     *inc = 0;
00539     *--inc = 1;
00540     while ( --j >= 0 ) *--inc = 1;
00541     while ( --i > 0 ) *--inc = 0;
00542     to = d->out;
00543     from = d->obj1;
00544     from2 = d->obj2;
00545     for ( i = 0; i < d->n; i++ ) {
00546         if ( *inc++ ) {
00547             *to++ = *from++;
00548         }
00549         else {
00550             *to++ = *from2++;
00551         }
00552     }
00553     return(0);
00554 }
00555
00556 /*
00557 #] Distribute :
00558 #[ MatchCy :
00559
00560     Matching of (r)cyclic tensors.
00561     Parameters like in MatchE.
00562     The structure of the routine is much simpler, because the number
00563     of possibilities is much more limited.
00564     The major complication is the ?a-type wildcards
00565     We need a strategy for T(il?,?a,il?,?b). Which is the shorter
00566     match: ?a or ?b ? (if possible of course)
00567     This is also relevant in the case of the shortest match if there
00568     is more than one choice for il.
00569 */
00570
00571 int MatchCy(PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
00572 {
00573     GETBIDENTITY
00574     WORD *t, *tstop, *p, *pstop, *m, *r, *oldworkpointer = AT.WorkPointer;
00575     WORD *thewildcards, *multiplicity, *renum, wc, newvalue, oldwilval = 0;
00576     WORD *params, *lowlevel = 0;
00577     int argcount = 0, funnycount = 0, tcount = fun[1] - FUNHEAD;
00578     int type = 0, pnun, i, j, k, nwstore, iraise, itop, sumeat;
00579     CBUF *C = cbuf+AT.ebufnum;
00580     int ntw = 3*AN.NumTotWildArgs+1;
00581     LONG oldcpointer = C->Pointer - C->Buffer;
00582     WORD offset = fun-AN.terstart, *newpat;
00583
00584     if ( (functions[fun[0]-FUNCTION].symmetric & ~REVERSEORDER) == RCYCLESYMMETRIC ) type = 1;
00585     pnun = pattern[0];
00586     nwstore = (AN.WildValue[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
00587     if ( pnun > FUNCTION + WILDOFFSET ) {
00588         pnun -= WILDOFFSET;
00589         if ( CheckWild(BHEAD pnun,FUNTOFUN,fun[0],&newvalue) ) return(0);
00590         oldwilval = 1;
00591         t = lowlevel = AT.WorkPointer;
00592         m = AN.WildValue;
00593         i = nwstore;
00594         r = AT.WildMask;
00595         if ( i > 0 ) {
00596             do {
00597                 *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
00598             } while ( --i > 0 );
00599         }
00600         *t++ = C->numrhs;
00601         if ( t >= AT.WorkTop ) {
00602             MLOCK(ErrorMessageLock);
00603             MesWork();
00604             MUNLOCK(ErrorMessageLock);
00605             return(-1);
00606         }
00607         AT.WorkPointer = t;
00608         AddWild(BHEAD pnun,FUNTOFUN,newvalue);

```

```

00609     }
00610     if ( (functions[pnum-FUNCTION].symmetric & ~REVERSEORDER) == RCYCLESYMMETRIC ) type = 1;
00611
00612     /* First we have to make an inventory. Are there FUNNYWILD pointers? */
00613
00614     p = pattern + FUNHEAD;
00615     pstop = pattern + pattern[1];
00616     while ( p < pstop ) {
00617         if ( *p == FUNNYWILD ) { p += 2; funnycount++; }
00618         else { p++; argcount++; }
00619     }
00620     if ( argcount > tcount ) goto NoSuccess;
00621     if ( argcount < tcount && funnycount == 0 ) goto NoSuccess;
00622     if ( argcount == 0 && tcount == 0 && funnycount == 0 ) {
00623         AN.RepFunList[AN.RepFunNum++] = offset;
00624         AN.RepFunList[AN.RepFunNum++] = 0;
00625         newpat = pattern + pattern[1];
00626         if ( newpat >= AN.patstop ) {
00627             if ( AN.UseFindOnly == 0 ) {
00628                 if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
00629                     AT.WorkPointer = oldworkpointer;
00630                     AN.UsedOtherFind = 1;
00631                     return(1);
00632                 }
00633                 j = 0;
00634             }
00635             else {
00636                 AT.WorkPointer = oldworkpointer;
00637                 return(1);
00638             }
00639         }
00640         else j = ScanFunctions(BHEAD newpat,inter,par);
00641         if ( j ) return(j);
00642         goto NoSuccess;
00643     }
00644     tstop = fun + fun[1];
00645
00646     /* Store the wildcard assignments */
00647
00648     params = AT.WorkPointer;
00649     thewildcards = t = params + tcount;
00650     t += ntwa;
00651     if ( oldwilval ) lowlevel = oldworkpointer;
00652     else lowlevel = t;
00653     m = AN.WildValue;
00654     i = nwstore;
00655     if ( i > 0 ) {
00656         r = AT.WildMask;
00657         do {
00658             *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
00659         } while ( --i > 0 );
00660         *t++ = C->numrhs;
00661     }
00662     if ( t >= AT.WorkTop ) {
00663         MLOCK(ErrorMessageLock);
00664         MesWork();
00665         MUNLOCK(ErrorMessageLock);
00666         return(-1);
00667     }
00668     AT.WorkPointer = t;
00669     /*
00670     #[ Case 1: no funnies or all funnies must be empty. We just cycle through.
00671     */
00672     if ( argcount == tcount ) {
00673         if ( funnycount > 0 ) { /* Test all funnies first */
00674             p = pattern + FUNHEAD;
00675             t = fun + FUNHEAD;
00676             while ( p < pstop ) {
00677                 if ( *p != FUNNYWILD ) { p++; continue; }
00678                 AN.argaddress = t;
00679                 if ( CheckWild(BHEAD p[1],ARGTOARG,0,t) ) goto nomatch;
00680                 AddWild(BHEAD p[1],ARGTOARG,0);
00681                 p += 2;
00682             }
00683             oldwilval = 1;
00684         }
00685         for ( k = 0; k <= type; k++ ) {
00686             if ( k == 0 ) {
00687                 p = params; t = fun + FUNHEAD;
00688                 while ( t < tstop ) *p++ = *t++;
00689             }
00690             else {
00691                 p = params+tcount; t = fun + FUNHEAD;
00692                 while ( t < tstop ) *--p = *t++;
00693             }
00694             for ( i = 0; i < tcount; i++ ) { /* The various cycles */
00695                 p = pattern + FUNHEAD;

```

```

00696         wc = 0;
00697         for ( j = 0; j < tcount; j++, p++ ) { /* The arguments */
00698             while ( *p == FUNNYWILD ) p += 2;
00699             t = params + (i+j)%tcount;
00700             if ( *t == *p ) continue;
00701             if ( *p >= AM.OffsetIndex + WILDOFFSET
00702                 && *p < AM.OffsetIndex + 2*WILDOFFSET ) {
00703
00704                 /* Test wildcard index */
00705
00706                 wc = *p - WILDOFFSET;
00707                 if ( CheckWild(BHEAD wc,INDTOIND,*t,&newvalue) ) break;
00708                 AddWild(BHEAD wc,INDTOIND,newvalue);
00709             }
00710             else if ( *t < MINSPEC && p[j] < MINSPEC
00711                 && *p >= AM.OffsetVector + WILDOFFSET ) {
00712
00713                 /* Test wildcard vector */
00714
00715                 wc = *p - WILDOFFSET;
00716                 if ( CheckWild(BHEAD wc,VECTOVEC,*t,&newvalue) ) break;
00717                 AddWild(BHEAD wc,VECTOVEC,newvalue);
00718             }
00719             else break;
00720         }
00721         if ( j >= tcount ) { /* Match! */
00722
00723             /* Continue with other functions. Make sure of the funnies */
00724
00725             AN.RepFunList[AN.RepFunNum++] = offset;
00726             AN.RepFunList[AN.RepFunNum++] = 0;
00727
00728             if ( funnycount > 0 ) {
00729                 p = pattern + FUNHEAD;
00730                 t = fun + FUNHEAD;
00731                 while ( p < pstop ) {
00732                     if ( *p != FUNNYWILD ) { p++; continue; }
00733                     AN.argaddress = t;
00734                     AddWild(BHEAD p[1],ARGTOARG,0);
00735                     p += 2;
00736                 }
00737             }
00738             newpat = pattern + pattern[1];
00739             if ( newpat >= AN.patstop ) {
00740                 if ( AN.UseFindOnly == 0 ) {
00741                     if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
00742                         AT.WorkPointer = oldworkpointer;
00743                         AN.UsedOtherFind = 1;
00744                         return(1);
00745                     }
00746                     j = 0;
00747                 }
00748                 else {
00749                     AT.WorkPointer = oldworkpointer;
00750                     return(1);
00751                 }
00752             }
00753             else j = ScanFunctions(BHEAD newpat,inter,par);
00754             if ( j ) {
00755                 AT.WorkPointer = oldworkpointer;
00756                 return(j); /* Full match. Return our success */
00757             }
00758             AN.RepFunNum -= 2;
00759         }
00760
00761         /* No (deeper) match. -> reset wildcards and continue */
00762
00763         if ( wc && nwstore > 0 ) {
00764             j = nwstore;
00765             m = AN.WildValue;
00766             t = thewildcards + ntw; r = AT.WildMask;
00767             if ( j > 0 ) {
00768                 do {
00769                     *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
00770                 } while ( --j > 0 );
00771             }
00772             C->numrhs = *t++;
00773             C->Pointer = C->Buffer + oldcpointer;
00774         }
00775     }
00776 }
00777 goto NoSuccess;
00778 }
00779 /*
00780 #] Case 1:
00781 #[ Case 2: One FUNNYWILD. Fix its length.
00782 */

```

```

00783     if ( funnycount == 1 ) {
00784         funnycount = tcount - argcount; /* Number or arguments to be eaten */
00785         for ( k = 0; k <= type; k++ ) {
00786             if ( k == 0 ) {
00787                 p = params; t = fun + FUNHEAD;
00788                 while ( t < tstop ) *p++ = *t++;
00789             }
00790             else {
00791                 p = params+tcount; t = fun + FUNHEAD;
00792                 while ( t < tstop ) *--p = *t++;
00793             }
00794             for ( i = 0; i < tcount; i++ ) { /* The various cycles */
00795                 p = pattern + FUNHEAD;
00796                 t = params;
00797                 wc = 0;
00798                 for ( j = 0; j < tcount; j++, p++, t++ ) { /* The arguments */
00799                     if ( *t == *p ) continue;
00800                     if ( *p == FUNNYWILD ) {
00801                         p++; wc = 1;
00802                         AN.argaddress = t;
00803                         if ( CheckWild(BHEAD *p,ARGTOARG|EATTENSOR,funnycount,t) ) break;
00804                         AddWild(BHEAD *p,ARGTOARG|EATTENSOR,funnycount);
00805                         j += funnycount-1; t += funnycount-1;
00806                     }
00807                     else if ( *p >= AM.OffsetIndex + WILDOFFSET
00808                             && *p < AM.OffsetIndex + 2*WILDOFFSET ) {
00809
00810                         /* Test wildcard index */
00811
00812                         wc = *p - WILDOFFSET;
00813                         if ( CheckWild(BHEAD wc,INDTOIND,*t,&newvalue) ) break;
00814                         AddWild(BHEAD wc,INDTOIND,newvalue);
00815                     }
00816                     else if ( *t < MINSPEC && *p < MINSPEC
00817                             && *p >= AM.OffsetVector + WILDOFFSET ) {
00818
00819                         /* Test wildcard vector */
00820
00821                         wc = *p - WILDOFFSET;
00822                         if ( CheckWild(BHEAD wc,VECTOVEC,*t,&newvalue) ) break;
00823                         AddWild(BHEAD wc,VECTOVEC,newvalue);
00824                     }
00825                     else break;
00826                 }
00827                 if ( j >= tcount ) { /* Match! */
00828
00829                     /* Continue with other functions. Make sure of the funnies */
00830
00831                     AN.RepFunList[AN.RepFunNum++] = offset;
00832                     AN.RepFunList[AN.RepFunNum++] = 0;
00833                     newpat = pattern + pattern[1];
00834                     if ( newpat >= AN.patstop ) {
00835                         if ( AN.UseFindOnly == 0 ) {
00836                             if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
00837                                 AT.WorkPointer = oldworkpointer;
00838                                 AN.UsedOtherFind = 1;
00839                                 return(1);
00840                             }
00841                             j = 0;
00842                         }
00843                         else {
00844                             AT.WorkPointer = oldworkpointer;
00845                             return(1);
00846                         }
00847                     }
00848                     else j = ScanFunctions(BHEAD newpat,inter,par);
00849                     if ( j ) {
00850                         AT.WorkPointer = oldworkpointer;
00851                         return(j); /* Full match. Return our success */
00852                     }
00853                     AN.RepFunNum -= 2;
00854                 }
00855             }
00856
00857             /* No (deeper) match. -> reset wildcards and continue */
00858             if ( wc ) {
00859                 j = nwstore;
00860                 m = AN.WildValue;
00861                 t = thewildcards + ntwa; r = AT.WildMask;
00862                 if ( j > 0 ) {
00863                     do {
00864                         *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
00865                     } while ( --j > 0 );
00866                 }
00867                 C->numrhs = *t++;
00868                 C->Pointer = C->Buffer + oldcpointer;
00869             }

```

```

00870         t = params;
00871         wc = *t;
00872         for ( j = 1; j < tcount; j++ ) { *t = t[1]; t++; }
00873         *t = wc;
00874     }
00875 }
00876     goto NoSuccess;
00877 }
00878 /*
00879 #] Case 2:
00880 #[ Case 3: More than one FUNNYWILD. Complicated.
00881 */
00882
00883     sumeat = tcount - argcount; /* Total number to be eaten by Funnies */
00884 /*
00885     In the first funnycount elements of 'thewildcards' we arrange
00886     for the summing over the various possibilities.
00887     The renumbering table is in thewildcards[2*funnycount]
00888     The multiplicity table is in thewildcards[funnycount]
00889     The number of arguments for each is in thewildcards[]
00890 */
00891     p = pattern+FUNHEAD;
00892     for ( i = funnycount; i < ntw; i++ ) thewildcards[i] = -1;
00893     multiplicity = thewildcards + funnycount;
00894     renum = multiplicity + funnycount;
00895     j = 0;
00896     while ( p < pstop ) {
00897         if ( *p != FUNNYWILD ) { p++; continue; }
00898         p++;
00899         if ( renum[*p] < 0 ) {
00900             renum[*p] = j;
00901             multiplicity[j] = 1;
00902             j++;
00903         }
00904         else multiplicity[renum[*p]]++;
00905         p++;
00906     }
00907 /*
00908     Strategy: First 'declared' has a tendency to be smaller
00909 */
00910     for ( i = 1; i < AN.NumTotWildArgs; i++ ) {
00911         if ( renum[i] < 0 ) continue;
00912         for ( j = i+1; j <= AN.NumTotWildArgs; j++ ) {
00913             if ( renum[j] < 0 ) continue;
00914             if ( renum[i] < renum[j] ) continue;
00915             k = multiplicity[renum[i]];
00916             multiplicity[renum[i]] = multiplicity[renum[j]];
00917             multiplicity[renum[j]] = k;
00918             k = renum[i]; renum[i] = renum[j]; renum[j] = k;
00919         }
00920     }
00921     for ( i = 0; i < funnycount; i++ ) thewildcards[i] = 0;
00922     iraise = funnycount-1;
00923     for ( ;; ) {
00924         for ( i = 0, j = sumeat; i < iraise; i++ )
00925             j -= thewildcards[i]*multiplicity[i];
00926         if ( j < 0 || j % multiplicity[iraise] != 0 ) {
00927             if ( j > 0 ) {
00928                 thewildcards[iraise-1]++;
00929                 continue;
00930             }
00931             itop = iraise-1;
00932             while ( itop > 0 && j < 0 ) {
00933                 j += thewildcards[itop]*multiplicity[itop];
00934                 thewildcards[itop] = 0;
00935                 itop--;
00936             }
00937             if ( itop <= 0 && j <= 0 ) break;
00938             thewildcards[itop]++;
00939             continue;
00940         }
00941         thewildcards[iraise] = j / multiplicity[iraise];
00942
00943         for ( k = 0; k <= type; k++ ) {
00944             if ( k == 0 ) {
00945                 p = params; t = fun + FUNHEAD;
00946                 while ( t < tstop ) *p++ = *t++;
00947             }
00948             else {
00949                 p = params+tcount; t = fun + FUNHEAD;
00950                 while ( t < tstop ) *--p = *t++;
00951             }
00952             for ( i = 0; i < tcount; i++ ) { /* The various cycles */
00953                 p = pattern + FUNHEAD;
00954                 t = params;
00955                 wc = 0;
00956                 for ( j = 0; j < tcount; j++, p++, t++ ) { /* The arguments */

```



```

00957         if ( *t == *p ) continue;
00958     if ( *p == FUNNYWILD ) {
00959         p++; wc = thewildcards[renum[*p]];
00960         AN.argaddress = t;
00961         if ( CheckWild(BHEAD *p,ARGTOARG|EATTENSOR,wc,t) ) break;
00962         AddWild(BHEAD *p,ARGTOARG|EATTENSOR,wc);
00963         j += wc-1; t += wc-1; wc = 1;
00964     }
00965     else if ( *p >= AM.OffsetIndex + WILDOFFSET
00966     && *p < AM.OffsetIndex + 2*WILDOFFSET ) {
00967
00968         /* Test wildcard index */
00969
00970         wc = *p - WILDOFFSET;
00971         if ( CheckWild(BHEAD wc,INDTOIND,*t,&newvalue) ) break;
00972         AddWild(BHEAD wc,INDTOIND,newvalue);
00973     }
00974     else if ( *t < MINSPEC && *p < MINSPEC
00975     && *p >= AM.OffsetVector + WILDOFFSET ) {
00976
00977         /* Test wildcard vector */
00978
00979         wc = *p - WILDOFFSET;
00980         if ( CheckWild(BHEAD wc,VECTOVEC,*t,&newvalue) ) break;
00981         AddWild(BHEAD wc,VECTOVEC,newvalue);
00982     }
00983     else break;
00984 }
00985 if ( j >= tcount ) { /* Match! */
00986
00987     /* Continue with other functions. Make sure of the funnies */
00988
00989     AN.RepFunList[AN.RepFunNum++] = offset;
00990     AN.RepFunList[AN.RepFunNum++] = 0;
00991     newpat = pattern + pattern[1];
00992     if ( newpat >= AN.patstop ) {
00993         if ( AN.UseFindOnly == 0 ) {
00994             if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
00995                 AT.WorkPointer = oldworkpointer;
00996                 AN.UsedOtherFind = 1;
00997                 return(1);
00998             }
00999             j = 0;
01000         }
01001         else {
01002             AT.WorkPointer = oldworkpointer;
01003             return(1);
01004         }
01005     }
01006     else j = ScanFunctions(BHEAD newpat,inter,par);
01007     if ( j ) {
01008         AT.WorkPointer = oldworkpointer;
01009         return(j); /* Full match. Return our success */
01010     }
01011     AN.RepFunNum -= 2;
01012 }
01013
01014 /* No (deeper) match. -> reset wildcards and continue */
01015
01016 if ( wc ) {
01017     j = nwstore;
01018     m = AN.WildValue;
01019     t = thewildcards + ntwa; r = AT.WildMask;
01020     if ( j > 0 ) {
01021         do {
01022             *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01023         } while ( --j > 0 );
01024     }
01025     C->numrhs = *t++;
01026     C->Pointer = C->Buffer + oldcpointer;
01027 }
01028 t = params;
01029 wc = *t;
01030 for ( j = 1; j < tcount; j++ ) { *t = t[1]; t++; }
01031 *t = wc;
01032 }
01033 }
01034 (thewildcards[iraise-1])++;
01035 }
01036 /*
01037 #] Case 3:
01038 */
01039 NoSuccess:
01040 if ( oldwilval > 0 ) {
01041 nomatch:;
01042     j = nwstore;
01043     if ( j > 0 ) {

```

```

01044         m = AN.WildValue;
01045         t = lowlevel; r = AT.WildMask;
01046         if ( j > 0 ) {
01047             do {
01048                 *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01049             } while ( --j > 0 );
01050         }
01051         C->numrhs = *t++;
01052         C->Pointer = C->Buffer + oldcpointer;
01053     }
01054 }
01055 AT.WorkPointer = oldworkpointer;
01056 return(0);
01057 }
01058
01059 /*
01060 #] MatchCy :
01061 #[ FunMatchCy :
01062
01063     Matching of (r)cyclic functions.
01064     Like MatchCy, but now for general functions.
01065 */
01066 int FunMatchCy(PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
01067 {
01068     GETBIDENTITY
01069     WORD *t, *tstop, *p, *pstop, *m, *r, *oldworkpointer = AT.WorkPointer;
01070     WORD **a, *thewildcards, *multiplicity, *renum, wc, wcc, oldwilval = 0;
01071     LONG ow = AT.pWorkPointer;
01072     WORD newvalue, *lowlevel = 0;
01073     int argcount = 0, funnycount = 0, tcount = 0;
01074     int type = 0, pnum, i, j, k, nwstore, iraise, itop, sumeat;
01075     CBUF *C = cbuf+AT.ebufnum;
01076     int ntw = 3*AN.NumTotWildArgs+1;
01077     LONG oldcpointer = C->Pointer - C->Buffer;
01078     WORD offset = fun-AN.terstart, *newpat;
01079
01080     if ( (functions[fun[0]-FUNCTION].symmetric & ~REVERSEORDER) == RCYCLESYMMETRIC ) type = 1;
01081     pnum = pattern[0];
01082     nwstore = (AN.WildValue[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
01083     if ( pnum > FUNCTION + WILDOFFSET ) {
01084         pnum -= WILDOFFSET;
01085         if ( CheckWild(BHEAD pnum,FUNTOFUN,fun[0],&newvalue) ) return(0);
01086         oldwilval = 1;
01087         t = lowlevel = oldworkpointer;
01088         m = AN.WildValue;
01089         i = nwstore;
01090         r = AT.WildMask;
01091         if ( i > 0 ) {
01092             do {
01093                 *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
01094             } while ( --i > 0 );
01095         }
01096         *t++ = C->numrhs;
01097         if ( t >= AT.WorkTop ) {
01098             MLOCK(ErrorMessageLock);
01099             MesWork();
01100             MUNLOCK(ErrorMessageLock);
01101             return(-1);
01102         }
01103         AT.WorkPointer = t;
01104         AddWild(BHEAD pnum,FUNTOFUN,newvalue);
01105     }
01106     if ( (functions[pnum-FUNCTION].symmetric & ~REVERSEORDER) == RCYCLESYMMETRIC ) type = 1;
01107
01108     /* First we have to make an inventory. Are there -ARGWILD pointers? */
01109
01110     p = pattern + FUNHEAD;
01111     pstop = pattern + pattern[1];
01112     while ( p < pstop ) {
01113         if ( *p == -ARGWILD ) { p += 2; funnycount++; }
01114         else { NEXTARG(p); argcount++; }
01115     }
01116     t = fun + FUNHEAD;
01117     tstop = fun + fun[1];
01118     while ( t < tstop ) { NEXTARG(t); tcount++; }
01119
01120     if ( argcount > tcount ) return(0);
01121     if ( argcount < tcount && funnycount == 0 ) return(0);
01122     if ( argcount == 0 && tcount == 0 && funnycount == 0 ) {
01123         AN.RepFunList[AN.RepFunNum++] = offset;
01124         AN.RepFunList[AN.RepFunNum++] = 0;
01125         newpat = pattern + pattern[1];
01126         if ( newpat >= AN.patstop ) {
01127             if ( AN.UseFindOnly == 0 ) {
01128                 if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
01129                     AT.WorkPointer = oldworkpointer;
01130

```

```

01131             AN.UsedOtherFind = 1;
01132             return(1);
01133         }
01134         j = 0;
01135     }
01136     else {
01137         AT.WorkPointer = oldworkpointer;
01138         return(1);
01139     }
01140 }
01141 else j = ScanFunctions(BHEAD newpat,inter,par);
01142 if ( j ) return(j);
01143 goto NoSuccess;
01144 }
01145
01146 /* Store the wildcard assignments */
01147
01148 WantAddPointers(tcount);
01149 AT.pWorkPointer += tcount;
01150 thewildcards = t = AT.WorkPointer;
01151 t += ntw;
01152 if ( oldwilval ) lowlevel = oldworkpointer;
01153 else lowlevel = t;
01154 m = AN.WildValue;
01155 i = nwstore;
01156 if ( i > 0 ) {
01157     r = AT.WildMask;
01158     do {
01159         *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
01160     } while ( --i > 0 );
01161     *t++ = C->numrhs;
01162 }
01163 if ( t >= AT.WorkTop ) {
01164     MLOCK(ErrorMessageLock);
01165     MesWork();
01166     MUNLOCK(ErrorMessageLock);
01167     return(-1);
01168 }
01169 AT.WorkPointer = t;
01170 /*
01171 #[ Case 1: no funnies or all funnies must be empty. We just cycle through.
01172 */
01173 if ( argcount == tcount ) {
01174     if ( funnycount > 0 ) { /* Test all funnies first */
01175         p = pattern + FUNHEAD;
01176         t = fun + FUNHEAD;
01177         while ( p < pstop ) {
01178             if ( *p != -ARGWILD ) { p++; continue; }
01179             AN.argaddress = t;
01180             if ( CheckWild(BHEAD p[1],ARGTOARG,0,t) ) goto nomatch;
01181             AddWild(BHEAD p[1],ARGTOARG,0);
01182             p += 2;
01183         }
01184         oldwilval = 1;
01185     }
01186     for ( k = 0; k <= type; k++ ) {
01187         if ( k == 0 ) {
01188             a = AT.pWorkSpace+oww; t = fun + FUNHEAD;
01189             while ( t < tstop ) { *a++ = t; NEXTARG(t); }
01190         }
01191         else {
01192             a = AT.pWorkSpace+oww+tcount; t = fun + FUNHEAD;
01193             while ( t < tstop ) { *--a = t; NEXTARG(t); }
01194         }
01195         for ( i = 0; i < tcount; i++ ) { /* The various cycles */
01196             p = pattern + FUNHEAD;
01197             wc = 0;
01198             for ( j = 0; j < tcount; j++ ) { /* The arguments */
01199                 while ( *p == -ARGWILD ) p += 2;
01200                 t = AT.pWorkSpace[oww+((i+j)%tcount)];
01201                 if ( ( wcc = MatchArgument(BHEAD t,p) ) == 0 ) break;
01202                 if ( wcc > 1 ) wc = 1;
01203                 NEXTARG(p);
01204             }
01205             if ( j >= tcount ) { /* Match! */
01206
01207                 /* Continue with other functions. Make sure of the funnies */
01208
01209                 AN.RepFunList[AN.RepFunNum++] = offset;
01210                 AN.RepFunList[AN.RepFunNum++] = 0;
01211
01212                 if ( funnycount > 0 ) {
01213                     p = pattern + FUNHEAD;
01214                     t = fun + FUNHEAD;
01215                     while ( p < pstop ) {
01216                         if ( *p != -ARGWILD ) { p++; continue; }
01217                         AN.argaddress = t;

```

```

01218             AddWild(BHEAD p[1],ARGTOARG,0);
01219             p += 2;
01220         }
01221     }
01222     newpat = pattern + pattern[1];
01223     if ( newpat >= AN.patstop ) {
01224         if ( AN.UseFindOnly == 0 ) {
01225             if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
01226                 AT.WorkPointer = oldworkpointer;
01227                 AT.pWorkPointer = oww;
01228                 AN.UsedOtherFind = 1;
01229                 return(1);
01230             }
01231             j = 0;
01232         }
01233         else {
01234             AT.WorkPointer = oldworkpointer;
01235             AT.pWorkPointer = oww;
01236             return(1);
01237         }
01238     }
01239     else j = ScanFunctions(BHEAD newpat,inter,par);
01240     if ( j ) {
01241         AT.WorkPointer = oldworkpointer;
01242         AT.pWorkPointer = oww;
01243         return(j); /* Full match. Return our success */
01244     }
01245     AN.RepFunNum -= 2;
01246 }
01247
01248 /* No (deeper) match. -> reset wildcards and continue */
01249
01250 if ( wc && nwstore > 0 ) {
01251     j = nwstore;
01252     m = AN.WildValue;
01253     t = thewildcards + ntw; r = AT.WildMask;
01254     if ( j > 0 ) {
01255         do {
01256             *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01257         } while ( --j > 0 );
01258     }
01259     C->numrhs = *t++;
01260     C->Pointer = C->Buffer + oldcpointer;
01261 }
01262 }
01263 }
01264     goto NoSuccess;
01265 }
01266 /*
01267 #] Case 1:
01268 #[ Case 2: One -ARGWILD. Fix its length.
01269 */
01270 if ( funnycount == 1 ) {
01271     funnycount = tcount - argcount; /* Number of arguments to be eaten */
01272     for ( k = 0; k <= type; k++ ) {
01273         if ( k == 0 ) {
01274             a = AT.pWorkSpace+oww; t = fun + FUNHEAD;
01275             while ( t < tstop ) { *a++ = t; NEXTARG(t); }
01276         }
01277         else {
01278             a = AT.pWorkSpace+oww+tcount; t = fun + FUNHEAD;
01279             while ( t < tstop ) { *--a = t; NEXTARG(t); }
01280         }
01281     }
01282     for ( i = 0; i < tcount; i++ ) { /* The various cycles */
01283         p = pattern + FUNHEAD;
01284         a = AT.pWorkSpace+oww;
01285         wc = 0;
01286         for ( j = 0; j < tcount; j++, a++ ) { /* The arguments */
01287             t = *a;
01288             if ( *p == -ARGWILD ) {
01289                 wc = 1;
01290                 AN.argaddress = (WORD *)a;
01291                 if ( CheckWild(BHEAD p[1],ARLTOARL,funnycount,(WORD *)a) ) break;
01292                 AddWild(BHEAD p[1],ARLTOARL,funnycount);
01293                 j += funnycount-1; a += funnycount-1;
01294             }
01295             else if ( MatchArgument(BHEAD t,p) == 0 ) break;
01296             NEXTARG(p);
01297         }
01298         if ( j >= tcount ) { /* Match! */
01299             /* Continue with other functions. Make sure of the funnies */
01300
01301             AN.RepFunList[AN.RepFunNum++] = offset;
01302             AN.RepFunList[AN.RepFunNum++] = 0;
01303             newpat = pattern + pattern[1];
01304             if ( newpat >= AN.patstop ) {

```

```

01305         if ( AN.UseFindOnly == 0 ) {
01306             if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
01307                 AT.WorkPointer = oldworkpointer;
01308                 AT.pWorkPointer = oww;
01309                 AN.UsedOtherFind = 1;
01310                 return(1);
01311             }
01312             j = 0;
01313         }
01314         else {
01315             AT.WorkPointer = oldworkpointer;
01316             AT.pWorkPointer = oww;
01317             return(1);
01318         }
01319     }
01320     else j = ScanFunctions(BHEAD newpat,inter,par);
01321     if ( j ) {
01322         AT.WorkPointer = oldworkpointer;
01323         AT.pWorkPointer = oww;
01324         return(j); /* Full match. Return our success */
01325     }
01326     AN.RepFunNum -= 2;
01327 }
01328
01329 /* No (deeper) match. -> reset wildcards and continue */
01330
01331 if ( wc ) {
01332     j = nwstore;
01333     m = AN.WildValue;
01334     t = thewildcards + ntw; r = AT.WildMask;
01335     if ( j > 0 ) {
01336         do {
01337             *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01338         } while ( --j > 0 );
01339     }
01340     C->numrhs = *t++;
01341     C->Pointer = C->Buffer + oldcpointer;
01342 }
01343 a = AT.pWorkSpace+oww;
01344 t = *a;
01345 for ( j = 1; j < tcount; j++ ) { *a = a[1]; a++; }
01346 *a = t;
01347 }
01348 }
01349 goto NoSuccess;
01350 }
01351 /*
01352 #] Case 2:
01353 #[ Case 3: More than one -ARGWILD. Complicated.
01354 */
01355
01356 sumeat = tcount - argcount; /* Total number to be eaten by Funnies */
01357 /*
01358 In the first funnycount elements of 'thewildcards' we arrange
01359 for the summing over the various possibilities.
01360 The renumbering table is in thewildcards[2*funnycount]
01361 The multiplicity table is in thewildcards[funnycount]
01362 The number of arguments for each is in thewildcards[]
01363 */
01364 p = pattern+FUNHEAD;
01365 for ( i = funnycount; i < ntw; i++ ) thewildcards[i] = -1;
01366 multiplicity = thewildcards + funnycount;
01367 renum = multiplicity + funnycount;
01368 j = 0;
01369 while ( p < pstop ) {
01370     if ( *p != -ARGWILD ) { p++; continue; }
01371     p++;
01372     if ( renum[*p] < 0 ) {
01373         renum[*p] = j;
01374         multiplicity[j] = 1;
01375         j++;
01376     }
01377     else multiplicity[renum[*p]]++;
01378     p++;
01379 }
01380 /*
01381 Strategy: First 'declared' has a tendency to be smaller
01382 */
01383 for ( i = 1; i < AN.NumTotWildArgs; i++ ) {
01384     if ( renum[i] < 0 ) continue;
01385     for ( j = i+1; j <= AN.NumTotWildArgs; j++ ) {
01386         if ( renum[j] < 0 ) continue;
01387         if ( renum[i] < renum[j] ) continue;
01388         k = multiplicity[renum[i]];
01389         multiplicity[renum[i]] = multiplicity[renum[j]];
01390         multiplicity[renum[j]] = k;
01391         k = renum[i]; renum[i] = renum[j]; renum[j] = k;

```

```

01392     }
01393 }
01394 for ( i = 0; i < funnycount; i++ ) thewildcards[i] = 0;
01395 iraise = funnycount-1;
01396 for ( ;; ) {
01397     for ( i = 0, j = sumeat; i < iraise; i++ )
01398         j -= thewildcards[i]*multiplicity[i];
01399     if ( j < 0 || j % multiplicity[iraise] != 0 ) {
01400         if ( j > 0 ) {
01401             thewildcards[iraise-1]++;
01402             continue;
01403         }
01404         itop = iraise-1;
01405         while ( itop > 0 && j < 0 ) {
01406             j += thewildcards[itop]*multiplicity[itop];
01407             thewildcards[itop] = 0;
01408             itop--;
01409         }
01410         if ( itop <= 0 && j <= 0 ) break;
01411         thewildcards[itop]++;
01412         continue;
01413     }
01414     thewildcards[iraise] = j / multiplicity[iraise];
01415
01416     for ( k = 0; k <= type; k++ ) {
01417         if ( k == 0 ) {
01418             a = AT.pWorkSpace+oww; t = fun + FUNHEAD;
01419             while ( t < tstop ) { *a++ = t; NEXTARG(t); }
01420         }
01421         else {
01422             a = AT.pWorkSpace+oww+tcount; t = fun + FUNHEAD;
01423             while ( t < tstop ) { *--a = t; NEXTARG(t); }
01424         }
01425         for ( i = 0; i < tcount; i++ ) { /* The various cycles */
01426             p = pattern + FUNHEAD;
01427             a = AT.pWorkSpace+oww;
01428             wc = 0;
01429             for ( j = 0; j < tcount; j++, a++ ) { /* The arguments */
01430                 t = *a;
01431                 if ( *p == -ARGWILD ) {
01432                     wc = thewildcards[renum[p[1]]];
01433                     AN.argaddress = (WORD *)a;
01434                     if ( CheckWild(BHEAD p[1],ARLTOARL,wc,(WORD *)a) ) break;
01435                     AddWild(BHEAD p[1],ARLTOARL,wc);
01436                     j += wc-1; a += wc-1; wc = 1;
01437                 }
01438                 else if ( MatchArgument(BHEAD t,p) == 0 ) break;
01439                 NEXTARG(p);
01440             }
01441             if ( j >= tcount ) { /* Match! */
01442
01443                 /* Continue with other functions. Make sure of the funnies */
01444
01445                 AN.RepFunList[AN.RepFunNum++] = offset;
01446                 AN.RepFunList[AN.RepFunNum++] = 0;
01447                 newpat = pattern + pattern[1];
01448                 if ( newpat >= AN.patstop ) {
01449                     if ( AN.UseFindOnly == 0 ) {
01450                         if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
01451                             AT.WorkPointer = oldworkpointer;
01452                             AT.pWorkPointer = oww;
01453                             AN.UsedOtherFind = 1;
01454                             return(1);
01455                         }
01456                         j = 0;
01457                     }
01458                     else {
01459                         AT.WorkPointer = oldworkpointer;
01460                         AT.pWorkPointer = oww;
01461                         return(1);
01462                     }
01463                 }
01464                 else j = ScanFunctions(BHEAD newpat,inter,par);
01465                 if ( j ) {
01466                     AT.WorkPointer = oldworkpointer;
01467                     AT.pWorkPointer = oww;
01468                     return(j); /* Full match. Return our success */
01469                 }
01470                 AN.RepFunNum -= 2;
01471             }
01472
01473             /* No (deeper) match. -> reset wildcards and continue */
01474
01475             if ( wc ) {
01476                 j = nwstore;
01477                 m = AN.WildValue;
01478                 t = thewildcards + ntw; r = AT.WildMask;

```

```

01479         if ( j > 0 ) {
01480             do {
01481                 *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01482             } while ( --j > 0 );
01483         }
01484         C->numrhs = *t++;
01485         C->Pointer = C->Buffer + oldcpointer;
01486     }
01487     a = AT.pWorkSpace+oww;
01488     t = *a;
01489     for ( j = 1; j < tcount; j++ ) { *a = a[1]; a++; }
01490     *a = t;
01491 }
01492 }
01493 (thewildcards[iraise-1])++;
01494 }
01495 /*
01496 #] Case 3:
01497 */
01498 NoSuccess:
01499     if ( oldwilval > 0 ) {
01500 nomatch:;
01501         j = nwstore;
01502         m = AN.WildValue;
01503         t = lowlevel; r = AT.WildMask;
01504         if ( j > 0 ) {
01505             do {
01506                 *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01507             } while ( --j > 0 );
01508         }
01509         C->numrhs = *t++;
01510         C->Pointer = C->Buffer + oldcpointer;
01511     }
01512     AT.WorkPointer = oldworkpointer;
01513     AT.pWorkPointer = oww;
01514     return(0);
01515 }
01516
01517 /*
01518 #] FunMatchCy :
01519 #[ FunMatchSy :
01520
01521     Matching of (anti)symmetric functions.
01522     Like MatchE, but now for general functions.
01523 */
01524
01525 int FunMatchSy(PHEAD WORD *pattern, WORD *fun, WORD *inter, WORD par)
01526 {
01527     GETBIDENTITY
01528     WORD *t, *tstop, *p, *pstop, *m, *r, *oldworkpointer = AT.WorkPointer;
01529     WORD **a, *thewildcards, oldwilval = 0;
01530     WORD newvalue, *lowlevel = 0, num, assig;
01531     WORD *cycles;
01532     LONG oww = AT.pWorkPointer, lhpar, lhfunnies;
01533     int argcount = 0, funnycount = 0, tcount = 0, signs = 0, signfun = 0, signo;
01534     int type = 0, pn timer, i, j, k, nwstore, iraise, cou2;
01535     CBUF *C = cbuf+AT.ebufnum;
01536     int ntw = 3*AN.NumTotWildArgs+1;
01537     LONG oldcpointer = C->Pointer - C->Buffer;
01538     WORD offset = fun-AN.terstart, *newpat;
01539
01540     if ( (functions[fun[0]-FUNCTION].symmetric & ~REVERSEORDER) == RCYCLESYMMETRIC ) type = 1;
01541     pn timer = pattern[0];
01542     nwstore = (AN.WildValue[-SUBEXPSIZE+1]-SUBEXPSIZE)/4;
01543     if ( pn timer > FUNCTION + WILDOFFSET ) {
01544         pn timer -= WILDOFFSET;
01545         if ( CheckWild(BHEAD pn timer,FUNTOFUN,fun[0],&newvalue) ) return(0);
01546         oldwilval = 1;
01547         t = lowlevel = oldworkpointer;
01548         m = AN.WildValue;
01549         i = nwstore;
01550         r = AT.WildMask;
01551         if ( i > 0 ) {
01552             do {
01553                 *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
01554             } while ( --i > 0 );
01555         }
01556         *t++ = C->numrhs;
01557         if ( t >= AT.WorkTop ) {
01558             MLOCK(ErrorMessageLock);
01559             MesWork();
01560             MUNLOCK(ErrorMessageLock);
01561             return(-1);
01562         }
01563         AT.WorkPointer = t;
01564         AddWild(BHEAD pn timer,FUNTOFUN,newvalue);
01565     }

```

```

01566     if ( (functions[pnum-FUNCTION].symmetric & ~REVERSEORDER) == RCYCLESYMMETRIC ) type = 1;
01567
01568     /* Try for a straight match. After all, both have been normalized */
01569
01570     if ( fun[1] == pattern[1] ) {
01571         i = fun[1]-FUNHEAD; p = pattern+FUNHEAD; t = fun + FUNHEAD;
01572         while ( --i >= 0 ) { if ( *p++ != *t++ ) break; }
01573         if ( i < 0 ) goto quicky;
01574     }
01575
01576     /* First we have to make an inventory. Are there -ARGWILD pointers? */
01577
01578     p = pattern + FUNHEAD;
01579     pstop = pattern + pattern[1];
01580     while ( p < pstop ) {
01581         if ( *p == -ARGWILD ) { p += 2; funnycount++; }
01582         else { NEXTARG(p); argcount++; }
01583     }
01584     t = fun + FUNHEAD;
01585     tstop = fun + fun[1];
01586     while ( t < tstop ) { NEXTARG(t); tcount++; }
01587
01588     if ( argcount > tcount ) return(0);
01589     if ( argcount < tcount && funnycount == 0 ) return(0);
01590     if ( argcount == 0 && tcount == 0 && funnycount == 0 ) {
01591 quicky:
01592         if ( AN.SignCheck && signs != AN.ExpectedSign ) goto NoSuccess;
01593         AN.RepFunList[AN.RepFunNum++] = offset;
01594         AN.RepFunList[AN.RepFunNum++] = signs;
01595         newpat = pattern + pattern[1];
01596         if ( newpat >= AN.patstop ) {
01597             if ( AN.UseFindOnly == 0 ) {
01598                 if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
01599                     AT.WorkPointer = oldworkpointer;
01600                     AN.UsedOtherFind = 1;
01601                     return(1);
01602                 }
01603                 j = 0;
01604             }
01605             else {
01606                 AT.WorkPointer = oldworkpointer;
01607                 return(1);
01608             }
01609         }
01610         else j = ScanFunctions(BHEAD newpat,inter,par);
01611         if ( j ) {
01612             AT.WorkPointer = oldworkpointer;
01613             return(j);
01614         }
01615         goto NoSuccess;
01616     }
01617
01618     /* Store the wildcard assignments */
01619
01620     WantAddPointers(tcount+argcount+funnycount);
01621     AT.pWorkPointer += tcount+argcount+funnycount;
01622     thewildcards = t = AT.WorkPointer;
01623     t += ntw;
01624     if ( oldwilval ) lowlevel = oldworkpointer;
01625     else lowlevel = t;
01626     m = AN.WildValue;
01627     i = nwstore; assig = 0;
01628     if ( i > 0 ) {
01629         r = AT.WildMask;
01630         do {
01631             assig += *r;
01632             *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *m++; *t++ = *r++;
01633         } while ( --i > 0 );
01634         *t++ = C->numrhs;
01635     }
01636     if ( t >= AT.WorkTop ) {
01637         MLOCK(ErrorMessageLock);
01638         MesWork();
01639         MUNLOCK(ErrorMessageLock);
01640         return(-1);
01641     }
01642     AT.WorkPointer = t;
01643
01644     /* Store pointers to the arguments */
01645
01646     t = fun + FUNHEAD; a = AT.pWorkSpace+oww;
01647     while ( t < tstop ) { *a++ = t; NEXTARG(t) }
01648     lhpar = a-AT.pWorkSpace;
01649     t = pattern + FUNHEAD;
01650     while ( t < pstop ) {
01651         if ( *t != -ARGWILD ) *a++ = t;
01652         NEXTARG(t)

```



```

01653     }
01654     lhfunnies = a-AT.pWorkSpace;
01655     t = pattern + FUNHEAD; cou2 = 0;
01656     while ( t < pstop ) {
01657         cou2++;
01658         if ( *t == -ARGWILD ) {
01659             *a++ = t;
01660         /*
01661          signfun: last ?a: tcount-argcount: number of arguments in ?a (assume one ?a)
01662          argcount+funnycount-cou2: arguments after ?a.
01663          Together tells whether moving ?a to end of list is even or odd
01664         */
01665             signfun = ((argcount+funnycount-cou2)*(tcount-argcount)) & 1;
01666         }
01667         NEXTARG(t)
01668     }
01669     signs += signfun;
01670     if ( funnycount > 0 ) {
01671         if ( ( (functions[fun[0]-FUNCTION].symmetric & ~REVERSEORDER) == SYMMETRIC )
01672             || ( (functions[fun[0]-FUNCTION].symmetric & ~REVERSEORDER) == ANTISYMMETRIC )
01673             || ( (functions[pnum-FUNCTION].symmetric & ~REVERSEORDER) == SYMMETRIC )
01674             || ( (functions[pnum-FUNCTION].symmetric & ~REVERSEORDER) == ANTISYMMETRIC ) ) {
01675             AT.WorkPointer = oldworkpointer;
01676             AT.pWorkPointer = oww;
01677             MLOCK(ErrorMessageLock);
01678             MesPrint("Sorry: no argument field wildcards yet in (anti)symmetric functions");
01679             MUNLOCK(ErrorMessageLock);
01680             Terminate(-1);
01681         }
01682     }
01683     /*
01684     Sort the regular arguments by
01685     1: no wildcards, fast.
01686     2: wildcards that have been assigned.
01687     3: general arguments.
01688     4: wildcards without an assignment.
01689     */
01690     iraise = argcount;
01691     for ( i = 0; i < iraise; i++ ) {
01692         t = AT.pWorkSpace[i+lhpars];
01693         if ( *t > 0 ) { /* Category 3: general argument */
01694             continue;
01695         }
01696         else if ( *t <= -FUNCTION ) {
01697             if ( *t > -FUNCTION - WILDOFFSET ) goto cat1;
01698             type = FUNTOFUN; num = -*t - WILDOFFSET;
01699         }
01700         else if ( *t == -SYMBOL ) {
01701             if ( t[1] < 2*MAXPOWER ) goto cat1;
01702             type = SYMTOSYM; num = t[1] - 2*MAXPOWER;
01703         }
01704         else if ( *t == -INDEX ) {
01705             if ( t[1] < AM.OffsetIndex + WILDOFFSET ) goto cat1;
01706             type = INDTOIND; num = t[1] - WILDOFFSET;
01707         }
01708         else if ( *t == -VECTOR || *t == -MINVECTOR ) {
01709             if ( t[1] < AM.OffsetVector + WILDOFFSET ) goto cat1;
01710             type = VECTOVEC; num = t[1] - WILDOFFSET;
01711         }
01712         else goto cat1; /* Things like -SNUMBER etc. */
01713     /*
01714     Now we have a wildcard and have to see whether it was assigned
01715     */
01716     m = AN.WildValue;
01717     j = nwstore;
01718     r = AT.WildMask;
01719     while ( --j >= 0 ) {
01720         if ( m[2] == num && *r ) {
01721             if ( type == *m ) break;
01722             if ( type == SYMTOSYM ) {
01723                 if ( *m == SYMTONUM || *m == SYMTOSUB ) break;
01724             }
01725             else if ( type == INDTOIND ) {
01726                 if ( *m == INDTOSUB ) break;
01727             }
01728             else if ( type == VECTOVEC ) {
01729                 if ( *m == VECTOMIN || *m == VECTOSUB ) break;
01730             }
01731         }
01732         m += 4; r++;
01733     }
01734     if ( j < 0 ) { /* Category 4: Wildcard that was not assigned */
01735         a = AT.pWorkSpace+lhpars;
01736         iraise--;
01737         if ( iraise != i ) signs++;
01738         m = a[iraise];
01739         a[iraise] = a[i];

```

```

01740         a[i] = m; i--;
01741     }
01742     else { /* Category 2: Wildcard that was assigned */
01743         for ( j = 0; j < tcount; j++ ) {
01744             if ( MatchArgument(BHEAD AT.pWorkSpace[oww+j],t) ) {
01745                 k = nwstore;
01746                 r = AT.WildMask;
01747                 num = 0;
01748                 while ( --k >= 0 ) num += *r++;
01749                 if ( num == assig ) { /* no wildcards were changed */
01750                     goto oneless;
01751                 }
01752                 break;
01753             }
01754         }
01755         if ( j >= tcount ) goto NoSuccess;
01756         j = nwstore;
01757         m = AN.WildValue;
01758         t = thewildcards + ntw; r = AT.WildMask;
01759         if ( j > 0 ) {
01760             do { /* undo assignment */
01761                 *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01762             } while ( --j > 0 );
01763         }
01764         C->numrhs = *t++;
01765     }
01766     continue;
01767 cat1:
01768     for ( j = 0; j < tcount; j++ ) {
01769         m = AT.pWorkSpace[j+oww];
01770         if ( *t != *m ) continue;
01771         if ( *t < 0 ) {
01772             if ( *t <= -FUNCTION ) break;
01773             if ( t[1] == m[1] ) break;
01774         }
01775         else {
01776             k = *t; r = t;
01777             while ( --k >= 0 && *m++ == *r++ ) {}
01778             if ( k < 0 ) break;
01779         }
01780     }
01781     if ( j >= tcount ) goto NoSuccess; /* Even the fixed ones don't match */
01782 oneless:
01783     signs += j - i;
01784 /*
01785     The next statements replace the one that is commented out
01786 */
01787     tcount--;
01788     while ( j < tcount ) {
01789         AT.pWorkSpace[oww+j] = AT.pWorkSpace[oww+j+1]; j++;
01790     }
01791 /*
01792     AT.pWorkSpace[oww+j] = AT.pWorkSpace[oww+(-tcount)];
01793 */
01794     argcount--; j = i;
01795     while ( j < argcount ) {
01796         AT.pWorkSpace[lhpar+j] = AT.pWorkSpace[lhpar+j+1]; j++;
01797     }
01798     iraise--; i--;
01799 }
01800 /*
01801     Now we see whether there are any ARGWILD objects that have been
01802     assigned already. In that case the work simplifies considerably.
01803     Currently (12-nov-2001) only in (R)CYCLIC functions; hence we do not
01804     test the sign!
01805 */
01806     for ( i = 0; i < funnycount; i++ ) {
01807         k = AT.pWorkSpace[lhfunnies+i][1];
01808         m = AN.WildValue;
01809         j = nwstore;
01810         r = AT.WildMask;
01811         while ( --j >= 0 ) {
01812             if ( *m == ARGTOARG && m[2] == k ) break;
01813             m += 4; r++;
01814         }
01815         if ( *r == 0 ) continue; /* not assigned yet */
01816         m = cbuf[AT.ebufnum].rhs[m[3]];
01817         if ( *m > 0 ) { /* Tensor arguments */
01818             j = *m;
01819             if ( j > tcount - argcount ) goto NoSuccess;
01820             while ( --j >= 0 ) {
01821                 m++;
01822                 if ( *m < 0 ) type = -VECTOR;
01823                 else if ( *m < AM.OffsetIndex ) type = -SNUMBER;
01824                 else type = -INDEX;
01825                 a = AT.pWorkSpace+oww;
01826                 for ( k = 0; k < tcount; k++ ) {

```

```

01827             if ( a[k][0] != type || a[k][1] != *m ) continue;
01828             a[k] = a[--tcount];
01829             goto nextjarg;
01830         }
01831         goto NoSuccess;
01832 nextjarg;;
01833     }
01834 }
01835 else {
01836     m++;
01837     while ( *m ) {
01838         for ( k = 0; k < tcount; k++ ) {
01839             t = AT.pWorkSpace[oww+k];
01840             if ( *t != *m ) continue;
01841             r = m;
01842             if ( *r < 0 ) {
01843                 if ( *r < -FUNCTION ) goto nextargw;
01844                 else if ( r[1] == t[1] ) goto nextargw;
01845             }
01846             else {
01847                 j = *r;
01848                 while ( --j >= 0 && *r++ == *t++ ) {}
01849                 if ( j < 0 ) goto nextargw;
01850             }
01851         }
01852         goto NoSuccess;
01853 nextargw;;
01854         AT.pWorkSpace[oww+k] = AT.pWorkSpace[oww+(--tcount)];
01855         NEXTARG(m)
01856     }
01857     AT.pWorkSpace[lhfunnies+i] = AT.pWorkSpace[lhfunnies+(--funnycount)];
01858 }
01859 if ( tcount == 0 ) {
01860     if ( argcount > 0 ) goto NoSuccess;
01861     for ( i = 0; i < funnycount; i++ ) {
01862         AddWild(BHEAD AT.pWorkSpace[lhfunnies+i][1],ARGTOARG,0);
01863     }
01864     goto quicky;
01865 }
01866 }
01867 /*
01868 We have now in lhpars first iraise elements with a dubious nature.
01869 Then argcount-iraise wildcards that have not been assigned.
01870 In lhfunnies we have funnycount ARGTOARG objects. ( (R)CYCLIC only )
01871
01872 First work our way through the 'dubious' objects
01873 We check whether assig changes.
01874 */
01875 for ( i = 0; i < iraise; i++ ) {
01876     for ( j = 0; j < tcount; j++ ) {
01877         if ( MatchArgument(BHEAD AT.pWorkSpace[oww+j],AT.pWorkSpace[lhpars+i]) ) {
01878             k = nwstore;
01879             r = AT.WildMask;
01880             num = 0;
01881             while ( --k >= 0 ) num += *r++;
01882             if ( num == assig ) { /* no wildcards were changed */
01883                 signs += j-i;
01884                 AT.pWorkSpace[oww+j] = AT.pWorkSpace[oww+(--tcount)];
01885                 if ( tcount > j ) signs += tcount-j-1;
01886                 argcount--;
01887                 a = AT.pWorkSpace + lhpars;
01888                 for ( j = i; j < argcount; j++ ) a[j] = a[j+1];
01889                 iraise--;
01890                 goto nexttiraise;
01891             }
01892             else { /* We cannot use this yet */
01893                 j = nwstore;
01894                 m = AN.WildValue;
01895                 t = thewildcards + ntwa; r = AT.WildMask;
01896                 if ( j > 0 ) {
01897                     do { /* undo assignment */
01898                         *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
01899                     } while ( --j > 0 );
01900                 }
01901                 C->numrhs = *t++;
01902                 C->Pointer = C->Buffer + oldcpointer;
01903                 goto nexttiraise;
01904             }
01905         }
01906     }
01907     goto NoSuccess;
01908 nexttiraise;;
01909 }
01910 /*
01911 Now all leftover patterns have unassigned wildcards in them.
01912 From now on we are in potential factorial territory.
01913

```

```

01914     Strategy:
01915         1: cycle through the regular objects.
01916         2: save wildcard settings
01917         3: divide the ARGWILDS
01918         4: make permutations of leftover arguments
01919         5: try them all
01920 */
01921     cycles = AT.WorkPointer;
01922     for ( i = 0; i < tcount; i++ ) cycles[i] = tcount-i;
01923     AT.WorkPointer += tcount;
01924     signo = 0;
01925     /*MesPrint("<1> signs = %d",signs);*/
01926     for ( ;; ) {
01927         WORD oRepFunNum = AN.RepFunNum;
01928         for ( j = 0; j < argcount; j++ ) {
01929             if ( MatchArgument (BHEAD AT.pWorkSpace[oww+j],AT.pWorkSpace[lhpars+j]) == 0 ) {
01930                 break;
01931             }
01932         }
01933         if ( j >= argcount ) {
01934             /*
01935             Thus far we have a match. Now the funnies
01936             */
01937             if ( funnycount ) {
01938                 AT.WorkPointer = oldworkpointer;
01939                 AT.pWorkPointer = oww;
01940                 MLOCK(ErrorMessageLock);
01941                 MesPrint("Sorry: no argument field wildcards yet in (anti)symmetric functions");
01942                 MUNLOCK(ErrorMessageLock);
01943             /*
01944             Bugfix 31-oct-2001, reported by Kasper Peeters
01945             We returned here with value -1 but that is not caught.
01946             Extra note (12-nov-2001): the sign becomes a bit problematic
01947             if we have funnies. No more than one allowed in antisymmetric
01948             functions, or we have serious problems.
01949             */
01950             Terminate(-1);
01951         }
01952         AN.RepFunList[AN.RepFunNum++] = offset;
01953         if ( ( ( functions[fun[0]-FUNCTION].symmetric & ~REVERSEORDER) == ANTISYMMETRIC )
01954             || ( functions[pnum-FUNCTION].symmetric & ~REVERSEORDER) == ANTISYMMETRIC ) ) {
01955             AN.RepFunList[AN.RepFunNum++] = ( signs + signo ) & 1;
01956         }
01957         else {
01958             AN.RepFunList[AN.RepFunNum++] = 0;
01959         }
01960         newpat = pattern + pattern[1];
01961         if ( newpat >= AN.patstop ) {
01962             WORD countsgn, sgn = 0;
01963             for ( countsgn = oRepFunNum+1; countsgn < AN.RepFunNum; countsgn += 2 ) {
01964                 if ( AN.RepFunList[countsgn] ) sgn ^= 1;
01965             }
01966             if ( AN.SignCheck == 0 || sgn == AN.ExpectedSign ) {
01967                 AT.WorkPointer = oldworkpointer;
01968                 AT.pWorkPointer = oww;
01969                 return(1);
01970             }
01971             if ( AN.UseFindOnly == 0 ) {
01972                 if ( FindOnce(BHEAD AN.findTerm,AN.findPattern) ) {
01973                     AT.WorkPointer = oldworkpointer;
01974                     AT.pWorkPointer = oww;
01975                     AN.UsedOtherFind = 1;
01976                     return(1);
01977                 }
01978             }
01979             j = 0;
01980         }
01981         else j = ScanFunctions(BHEAD newpat,inter,par);
01982         if ( j ) {
01983             WORD countsgn, sgn = 0;
01984             for ( countsgn = oRepFunNum+1; countsgn < AN.RepFunNum; countsgn += 2 ) {
01985                 if ( AN.RepFunList[countsgn] ) sgn ^= 1;
01986             }
01987             if ( AN.SignCheck == 0 || sgn == AN.ExpectedSign ) {
01988                 AT.WorkPointer = oldworkpointer;
01989                 AT.pWorkPointer = oww;
01990                 return(j);
01991             }
01992         }
01993     }
01994     AN.RepFunNum = oRepFunNum;
01995     i = argcount - 1;
01996 }
01997 else i = j;
01998 j = nwstore;
01999 m = AN.WildValue;
02000 t = thewildcards + ntw; r = AT.WildMask;

```

```

02001         if ( j > 0 ) {
02002             do {
02003                 *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
02004             } while ( --j > 0 );
02005         }
02006         C->numrhs = *t++;
02007         C->Pointer = C->Buffer + oldcpointer;
02008     /*
02009         On to the next cycle
02010     */
02011     a = AT.pWorkSpace + oww;
02012     for ( j = i+1, t = a[i]; j < tcount; j++ ) a[j-1] = a[j];
02013     a[tcount-1] = t; cycles[i]--;
02014     signo += tcount - i - 1;
02015     while ( cycles[i] <= 0 ) {
02016         cycles[i] = tcount - i;
02017         i--;
02018         if ( i < 0 ) goto NoSuccess;
02019     /*
02020         MLOCK(ErrorMessageLock);
02021         MesPrint("Cycle i = %d",i);
02022         MUNLOCK(ErrorMessageLock);
02023     */
02024     for ( j = i+1, t = a[i]; j < tcount; j++ ) a[j-1] = a[j];
02025     a[tcount-1] = t; cycles[i]--;
02026     signo += tcount - i - 1;
02027     }
02028 }
02029 NoSuccess:
02030     if ( oldwilval > 0 ) {
02031         j = nwstore;
02032         m = AN.WildValue;
02033         t = lowlevel; r = AT.WildMask;
02034         if ( j > 0 ) {
02035             do {
02036                 *m++ = *t++; *m++ = *t++; *m++ = *t++; *m++ = *t++; *r++ = *t++;
02037             } while ( --j > 0 );
02038         }
02039         C->numrhs = *t++;
02040         C->Pointer = C->Buffer + oldcpointer;
02041     }
02042     AT.WorkPointer = oldworkpointer;
02043     AT.pWorkPointer = oww;
02044     return(0);
02045 }
02046
02047 /*
02048 #] FunMatchSy :
02049 #[ MatchArgument :
02050 */
02051
02052 int MatchArgument(PHEAD WORD *arg, WORD *pat)
02053 {
02054     GETBIDENTITY
02055     WORD *m = pat, *t = arg, i, j, newvalue;
02056     WORD *argmstop = pat, *argtstop = arg;
02057     WORD *cto, *cfrom, *csav, ci;
02058     WORD oRepFunNum, *oRepFunList;
02059     WORD *oterstart, *oterstop, *opatstop;
02060     WORD wildargs, wildeat;
02061     WORD *mtrmstop, *ttrmstop, *msubstop, msizcoef;
02062     WORD *wildargtaken;
02063     int wc = 1;
02064
02065     NEXTARG(argmstop);
02066     NEXTARG(argtstop);
02067 /*
02068 #[ Both fast :
02069 */
02070     if ( *m < 0 && *t < 0 ) {
02071         if ( *t <= -FUNCTION ) {
02072             if ( *t == *m ) {}
02073             else if ( *m <= -FUNCTION-WILDOFFSET
02074                 && functions[-*t-FUNCTION].spec
02075                 == functions[-*m-FUNCTION-WILDOFFSET].spec ) {
02076                 i = -*m - WILDOFFSET; wc = 2;
02077                 if ( CheckWild(BHEAD i,FUNTOFUN,-*t,&newvalue) ) {
02078                     return(0);
02079                 }
02080                 AddWild(BHEAD i,FUNTOFUN,newvalue);
02081             }
02082             else if ( *m == -SYMBOL && m[1] >= 2*MAXPOWER ) {
02083                 i = m[1] - 2*MAXPOWER;
02084                 AN.argaddress = AT.FunArg;
02085                 AT.FunArg[ARGHEAD+1] = -*t;
02086                 if ( CheckWild(BHEAD i,SYMTOSUB,1,AN.argaddress) ) return(0);
02087                 AddWild(BHEAD i,SYMTOSUB,0);

```

```

02088         }
02089         else return(0);
02090     }
02091     else if ( *t == *m ) {
02092         if ( t[1] == m[1] ) {}
02093         else if ( *t == -SYMBOL ) {
02094             j = SYMTOSYM;
02095 SymAll:         if ( ( i = m[1] - 2*MAXPOWER ) < 0 ) return(0);
02096                 wc = 2;
02097                 if ( CheckWild(BHEAD i,j,t[1],&newvalue) ) return(0);
02098                 AddWild(BHEAD i,j,newvalue);
02099         }
02100         else if ( *t == -INDEX ) {
02101 IndAll:         i = m[1] - WILDOFFSET;
02102                 if ( i < AM.OffsetIndex || i >= WILDOFFSET+AM.OffsetIndex )
02103                     return(0);
02104                 /* We kill the summed over indices here */
02105                 wc = 2;
02106                 if ( CheckWild(BHEAD i,INDTOIND,t[1],&newvalue) ) return(0);
02107                 AddWild(BHEAD i,INDTOIND,newvalue);
02108         }
02109         else if ( *t == -VECTOR || *t == -MINVECTOR ) {
02110             i = m[1] - WILDOFFSET;
02111             if ( i < AM.OffsetVector ) return(0);
02112             wc = 2;
02113             if ( CheckWild(BHEAD i,VECTOVEC,t[1],&newvalue) ) return(0);
02114             AddWild(BHEAD i,VECTOVEC,newvalue);
02115         }
02116         else return(0);
02117     }
02118     else if ( *m == -INDEX && m[1] >= AM.OffsetIndex+WILDOFFSET
02119 && m[1] < AM.OffsetIndex+(WILDOFFSET<<1) ) {
02120         if ( *t == -VECTOR ) goto IndAll;
02121         if ( *t == -SNUMBER && t[1] >= 0 && t[1] < AM.OffsetIndex ) goto IndAll;
02122         if ( *t == -MINVECTOR ) {
02123             i = m[1] - WILDOFFSET;
02124             AN.argaddress = AT.MinVecArg;
02125             AT.MinVecArg[ARGHEAD+3] = t[1];
02126             wc = 2;
02127             if ( CheckWild(BHEAD i,INDTOSUB,1,AN.argaddress) ) return(0);
02128             AddWild(BHEAD i,INDTOSUB,(WORD)0);
02129         }
02130         else return(0);
02131     }
02132     else if ( *m == -SYMBOL && m[1] >= 2*MAXPOWER && *t == -SNUMBER ) {
02133         j = SYMTONUM;
02134         goto SymAll;
02135     }
02136     else if ( *m == -VECTOR && *t == -MINVECTOR &&
02137 ( i = m[1] - WILDOFFSET ) >= AM.OffsetVector ) {
02138         wc = 2;
02139 /*
02140         AN.argaddress = AT.MinVecArg;
02141         AT.MinVecArg[ARGHEAD+3] = t[1];
02142         if ( CheckWild(BHEAD i,VECTOSUB,1,AN.argaddress) ) return(0);
02143         AddWild(BHEAD i,VECTOSUB,(WORD)0);
02144 */
02145         if ( CheckWild(BHEAD i,VECTOMIN,t[1],&newvalue) ) return(0);
02146         AddWild(BHEAD i,VECTOMIN,newvalue);
02147     }
02148 }
02149 else if ( *m == -MINVECTOR && *t == -VECTOR &&
02150 ( i = m[1] - WILDOFFSET ) >= AM.OffsetVector ) {
02151     wc = 2;
02152 /*
02153     AN.argaddress = AT.MinVecArg;
02154     AT.MinVecArg[ARGHEAD+3] = t[1];
02155     if ( CheckWild(BHEAD i,VECTOSUB,1,AN.argaddress) ) return(0);
02156     AddWild(BHEAD i,VECTOSUB,(WORD)0);
02157 */
02158     if ( CheckWild(BHEAD i,VECTOMIN,t[1],&newvalue) ) return(0);
02159     AddWild(BHEAD i,VECTOMIN,newvalue);
02160 }
02161 else return(0);
02162 }
02163 /*
02164 #] Both fast :
02165 #[ Fast arg :
02166 */
02167 else if ( *m > 0 && *t <= -FUNCTION ) {
02168     if ( ( m[ARGHEAD]+ARGHEAD == *m ) && m[*m-1] == 3
02169 && m[*m-2] == 1 && m[*m-3] == 1 && m[ARGHEAD+1] >= FUNCTION
02170 && m[ARGHEAD+2] == *m-ARGHEAD-4 ) { /* Check for f(?a) etc */
02171         WORD *mmmmst, *mmmm, mmmi;
02172         if ( m[ARGHEAD+1] >= FUNCTION+WILDOFFSET ) {
02173             mmmi = *m - WILDOFFSET;
02174             wc = 2;

```

```

02175         if ( CheckWild(BHEAD mmmi,FUNTOFUN,-*t,&newvalue) ) return(0);
02176         AddWild(BHEAD mmmi,FUNTOFUN,newvalue);
02177     }
02178     else if ( m[ARGHEAD+1] != -*t ) return(0);
02179 /*
02180         Only arguments allowed are ?a etc.
02181 */
02182     mmmst = m+m-3;
02183     mmm = m + ARGHEAD + FUNHEAD + 1;
02184     while ( mmm < mmmst ) {
02185         if ( *mmm != -ARGWILD ) return(0);
02186         mmmi = 0;
02187         AN.argaddress = t; wc = 2;
02188         if ( CheckWild(BHEAD mmm[1],ARGTOARG,mmmi,t) ) return(0);
02189         AddWild(BHEAD mmm[1],ARGTOARG,mmmi);
02190         mmm += 2;
02191     }
02192 }
02193     else return(0);
02194 }
02195 /*
02196 #] Fast arg :
02197 #[ Fast pat :
02198 */
02199     else if ( *m < 0 && *t > 0 ) {
02200         if ( *m == -SYMBOL ) { /* SYMTOSUB */
02201             if ( m[1] < 2*MAXPOWER ) return(0);
02202             i = m[1] - 2*MAXPOWER;
02203             AN.argaddress = t; wc = 2;
02204             if ( CheckWild(BHEAD i,SYMTOSUB,1,AN.argaddress) ) return(0);
02205             AddWild(BHEAD i,SYMTOSUB,0);
02206         }
02207         else if ( *m == -VECTOR ) {
02208             if ( ( i = m[1] - WILDOFFSET ) < AM.OffsetVector ) return(0);
02209             AN.argaddress = t; wc = 2;
02210             if ( CheckWild(BHEAD i,VECTOSUB,1,t) ) return(0);
02211             AddWild(BHEAD i,VECTOSUB,(WORD)0);
02212         }
02213         else if ( *m == -INDEX ) {
02214             if ( ( i = m[1] - WILDOFFSET ) < AM.OffsetIndex ) return(0);
02215             if ( i >= AM.OffsetIndex + WILDOFFSET ) return(0);
02216             AN.argaddress = t; wc = 2;
02217             if ( CheckWild(BHEAD i,INDTOSUB,1,AN.argaddress) ) return(0);
02218             AddWild(BHEAD i,INDTOSUB,(WORD)0);
02219         }
02220         else return(0);
02221     }
02222 /*
02223 #] Fast pat :
02224 #[ Both general :
02225 */
02226     else if ( *m > 0 && *t > 0 ) {
02227         i = *m;
02228         do { if ( *m++ != *t++ ) break; } while ( --i > 0 );
02229         if ( i > 0 ) {
02230 /*
02231             Not an exact match here.
02232             We have to hope that the pattern contains a composite wildcard.
02233 */
02234             m = pat; t = arg;
02235             m += ARGHEAD; t += ARGHEAD; /* Point at (first?) term */
02236             mtrmstop = m + *m;
02237             ttrmstop = t + *t;
02238             if ( mtrmstop < argmstop ) return(0); /* More than one term */
02239             msizcoef = mtrmstop[-1];
02240             if ( msizcoef < 0 ) msizcoef = -msizcoef;
02241             msubstop = mtrmstop - msizcoef;
02242             m++;
02243             if ( m >= msubstop ) return(0); /* Only coefficient */
02244 /*
02245             Here we have a composite term. It can match provided it
02246             matches the entire argument. This argument must be a
02247             single term also and the coefficients should match
02248             (more or less).
02249             The matching takes:
02250             1: Match the functions etc. Nothing can be left.
02251             2: Match dotproducts and symbols. ONLY must match
02252                and nothing may be left.
02253             For safety it is best to take the term out and put it
02254             in workspace.
02255 */
02256             if ( argtstop > ttrmstop ) return(0);
02257             m--;
02258
02259             oterstart = AN.terstart;
02260             oterstop = AN.terstop;
02261             opatstop = AN.patstop;

```

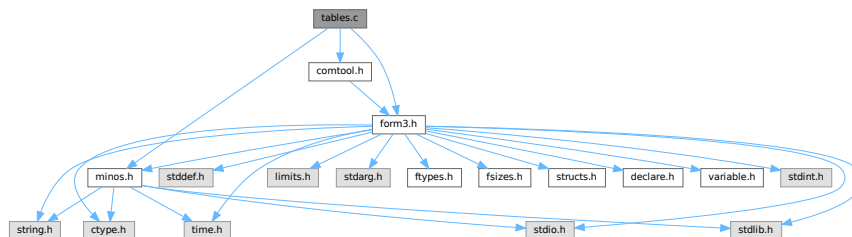
```

02262         oRepFunList = AN.RepFunList;
02263         oRepFunNum = AN.RepFunNum;
02264         AN.RepFunNum = 0;
02265         wildargtaken = AT.WorkPointer;
02266         AN.RepFunList = wildargtaken + AN.NumTotWildArgs;
02267         AT.WorkPointer = (WORD *) ((UBYTE *) (AN.RepFunList)) + AM.MaxTer/2;
02268         csav = cto = AT.WorkPointer;
02269         cfrom = t;
02270         ci = *t;
02271         while ( --ci >= 0 ) *cto++ = *cfrom++;
02272         AT.WorkPointer = cto;
02273         ci = msizcoef;
02274         cfrom = mtrmstop;
02275         while ( --ci >= 0 ) {
02276             if ( *--cfrom != *--cto ) {
02277                 AT.WorkPointer = wildargtaken;
02278                 AN.RepFunList = oRepFunList;
02279                 AN.RepFunNum = oRepFunNum;
02280                 AN.terstart = oterstart;
02281                 AN.terstop = oterstop;
02282                 AN.patstop = opatstop;
02283                 return(0);
02284             }
02285         }
02286         *m -= msizcoef;
02287         wildargs = AN.WildArgs;
02288         wildeat = AN.WildEat;
02289         for ( i = 0; i < wildargs; i++ ) wildargtaken[i] = AT.WildArgTaken[i];
02290         AN.ForFindOnly = 0; AN.UseFindOnly = 1;
02291         AN.nogroundlevel++;
02292         if ( FindRest(BHEAD csav,m) && ( AN.UsedOtherFind || FindOnly(BHEAD csav,m) ) ) { }
02293         else {
02294             *m += msizcoef;
02295             AT.WorkPointer = wildargtaken;
02296             AN.RepFunList = oRepFunList;
02297             AN.RepFunNum = oRepFunNum;
02298             AN.terstart = oterstart;
02299             AN.terstop = oterstop;
02300             AN.patstop = opatstop;
02301             AN.WildArgs = wildargs;
02302             AN.WildEat = wildeat;
02303             AN.nogroundlevel--;
02304             for ( i = 0; i < wildargs; i++ ) AT.WildArgTaken[i] = wildargtaken[i];
02305             return(0);
02306         }
02307         AN.nogroundlevel--;
02308         AN.WildArgs = wildargs;
02309         AN.WildEat = wildeat;
02310         for ( i = 0; i < wildargs; i++ ) AT.WildArgTaken[i] = wildargtaken[i];
02311         Substitute(BHEAD csav,m,1);
02312         cto = csav;
02313         cfrom = cto + *cto - msizcoef;
02314         cto++;
02315         *m += msizcoef;
02316         AT.WorkPointer = wildargtaken;
02317         AN.RepFunList = oRepFunList;
02318         AN.RepFunNum = oRepFunNum;
02319         AN.terstart = oterstart;
02320         AN.terstop = oterstop;
02321         AN.patstop = opatstop;
02322         if ( *cto != SUBEXPRESSION ) return(0);
02323         cto += cto[1];
02324         if ( cto < cfrom ) return(0);
02325     }
02326 }
02327 /*
02328 #] Both general :
02329 */
02330 else return(0);
02331 /*
02332 And now the success: (wc = 2 means that there was a wildcard involved)
02333 */
02334 return(wc);
02335 }
02336 /*
02337 #] MatchArgument :
02338 */

```


4.141 tables.c File Reference

```
#include "form3.h"
#include "minos.h"
#include "comtool.h"
Include dependency graph for tables.c:
```



Functions

- void [ClearTableTree](#) (TABLES T)
- int [InsTableTree](#) (TABLES T, WORD *tp)
- void [RedoTableTree](#) (TABLES T, int newsize)
- int [FindTableTree](#) (TABLES T, WORD *tp, int inc)
- int [DoTableExpansion](#) (WORD *term, WORD level)
- int [CoTableBase](#) (UBYTE *s)
- int [FlipTable](#) (FUNCTIONS f, int type)
- int [SpareTable](#) (TABLES TT)
- DBASE * [FindTB](#) (UBYTE *name)
- int [CoTBcreate](#) (UBYTE *s)
- int [CoTBopen](#) (UBYTE *s)
- int [CoTBaddto](#) (UBYTE *s)
- int [CoTBenter](#) (UBYTE *s)
- int [CoTestUse](#) (UBYTE *s)
- int [CheckTableDeclarations](#) (DBASE *d)
- int [CoTBload](#) (UBYTE *ss)
- int [TestUse](#) (WORD *term, WORD level)
- int [CoTBaudit](#) (UBYTE *s)
- int [CoTBon](#) (UBYTE *s)
- int [CoTBoff](#) (UBYTE *s)
- int [CoTBcleanup](#) (UBYTE *s)
- int [CoTBreplace](#) (UBYTE *s)
- int [CoTBuse](#) (UBYTE *s)
- int [CoApply](#) (UBYTE *s)
- int [CoTBhelp](#) (UBYTE *s)
- void [ReWorkT](#) (WORD *term, WORD *funs, WORD numfuns)
- void [Apply](#) (WORD *term, WORD level)
- int [ApplyExec](#) (WORD *term, int maxtogo, WORD level)
- void [ApplyReset](#) (WORD level)
- void [TableReset](#) (void)
- int [ReleaseTB](#) (void)

Variables

- char * [helptb](#) []

4.141.1 Detailed Description

Contains all functions that deal with the table bases on the 'FORM level' The low level database routines are in [minos.c](#)

Definition in file [tables.c](#).

4.141.2 Function Documentation

4.141.2.1 ClearTableTree()

```
void ClearTableTree (
    TABLES T )
```

Definition at line 72 of file [tables.c](#).

4.141.2.2 InsTableTree()

```
int InsTableTree (
    TABLES T,
    WORD * tp )
```

Definition at line 103 of file [tables.c](#).

4.141.2.3 RedoTableTree()

```
void RedoTableTree (
    TABLES T,
    int newsize )
```

Definition at line 266 of file [tables.c](#).

4.141.2.4 FindTableTree()

```
int FindTableTree (
    TABLES T,
    WORD * tp,
    int inc )
```

Definition at line 291 of file [tables.c](#).

4.141.2.5 DoTableExpansion()

```
int DoTableExpansion (
    WORD * term,
    WORD level )
```

Definition at line 334 of file [tables.c](#).

4.141.2.6 CoTableBase()

```
int CoTableBase (
    UBYTE * s )
```

Definition at line 627 of file [tables.c](#).

4.141.2.7 FlipTable()

```
int FlipTable (
    FUNCTIONS f,
    int type )
```

Definition at line 671 of file [tables.c](#).

4.141.2.8 SpareTable()

```
int SpareTable (
    TABLES TT )
```

Definition at line 694 of file [tables.c](#).

4.141.2.9 FindTB()

```
DBASE * FindTB (
    UBYTE * name )
```

Definition at line 734 of file [tables.c](#).

4.141.2.10 CoTBcreate()

```
int CoTBcreate (
    UBYTE * s )
```

Definition at line 756 of file [tables.c](#).

4.141.2.11 CoTBopen()

```
int CoTBopen (
    UBYTE * s )
```

Definition at line 772 of file [tables.c](#).

4.141.2.12 CoTBaddto()

```
int CoTBaddto (
    UBYTE * s )
```

Definition at line 801 of file [tables.c](#).

4.141.2.13 CoTBenter()

```
int CoTBenter (
    UBYTE * s )
```

Definition at line 974 of file [tables.c](#).

4.141.2.14 CoTestUse()

```
int CoTestUse (
    UBYTE * s )
```

Definition at line 1133 of file [tables.c](#).

4.141.2.15 CheckTableDeclarations()

```
int CheckTableDeclarations (
    DBASE * d )
```

Definition at line 1178 of file [tables.c](#).

4.141.2.16 CoTBload()

```
int CoTBload (
    UBYTE * ss )
```

Definition at line 1252 of file [tables.c](#).

4.141.2.17 TestUse()

```
int TestUse (
    WORD * term,
    WORD level )
```

Definition at line 1414 of file [tables.c](#).

4.141.2.18 CoTBaudit()

```
int CoTBaudit (
    UBYTE * s )
```

Definition at line 1474 of file [tables.c](#).

4.141.2.19 CoTBon()

```
int CoTBon (
    UBYTE * s )
```

Definition at line 1514 of file [tables.c](#).

4.141.2.20 CoTBoff()

```
int CoTBoff (
    UBYTE * s )
```

Definition at line 1545 of file [tables.c](#).

4.141.2.21 CoTBcleanup()

```
int CoTBcleanup (
    UBYTE * s )
```

Definition at line 1576 of file [tables.c](#).

4.141.2.22 CoTBreplace()

```
int CoTBreplace (
    UBYTE * s )
```

Definition at line 1588 of file [tables.c](#).

4.141.2.23 CoTBuse()

```
int CoTBuse (
    UBYTE * s )
```

Definition at line 1603 of file [tables.c](#).

4.141.2.24 CoApply()

```
int CoApply (
    UBYTE * s )
```

Definition at line 1797 of file [tables.c](#).

4.141.2.25 CoTBhelp()

```
int CoTBhelp (
    UBYTE * s )
```

Definition at line 1884 of file [tables.c](#).

4.141.2.26 ReWorkT()

```
void ReWorkT (
    WORD * term,
    WORD * funcs,
    WORD numfuncs )
```

Definition at line 1900 of file [tables.c](#).

4.141.2.27 Apply()

```
void Apply (
    WORD * term,
    WORD level )
```

Definition at line 1981 of file [tables.c](#).

4.141.2.28 ApplyExec()

```
int ApplyExec (
    WORD * term,
    int maxtogo,
    WORD level )
```

Definition at line 2039 of file [tables.c](#).

4.141.2.29 ApplyReset()

```
void ApplyReset (
    WORD level )
```

Definition at line 2256 of file [tables.c](#).

4.141.2.30 TableReset()

```
void TableReset (
    void )
```

Definition at line 2291 of file [tables.c](#).

4.141.2.31 ReleaseTB()

```
int ReleaseTB (
    void )
```

Definition at line 2317 of file [tables.c](#).

4.141.3 Variable Documentation

4.141.3.1 helptb

```
char* helptb[]
```

Definition at line 1852 of file [tables.c](#).

4.142 tables.c

[Go to the documentation of this file.](#)

```
00001
00006 /* #[ License : */
00007 /*
00008 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009 *   When using this file you are requested to refer to the publication
00010 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011 *   This is considered a matter of courtesy as the development was paid
00012 *   for by FOM the Dutch physics granting agency and we would like to
00013 *   be able to track its scientific use to convince FOM of its value
00014 *   for the community.
00015 *
00016 *   This file is part of FORM.
00017 *
00018 *   FORM is free software: you can redistribute it and/or modify it under the
00019 *   terms of the GNU General Public License as published by the Free Software
00020 *   Foundation, either version 3 of the License, or (at your option) any later
00021 *   version.
00022 *
00023 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00026 *   details.
00027 *
00028 *   You should have received a copy of the GNU General Public License along
00029 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00030 */
00031 /* #[ License : */
00032 /*
00033 *   #[ Includes :
00034 *
00035 *   File contains the routines for the tree structure of sparse tables
00036 *   We insert elements by
00037 *   InsTableTree(T,tp) with T the TABLES element and tp the pointer
00038 *   to the indices.
00039 *   We look for elements with
00040 *   FindTableTree(T,tp,inc) with T the TABLES element, tp the pointer to the
00041 *   indices or the function arguments and inc tells which of these options.
00042 *   The tree is cleared with ClearTableTree(T) and we rebuild the tree
00043 *   after a .store in which we lost a part of the table with
00044 *   RedoTableTree(T,newsize)
00045 *
00046 *   In T->tablepointers we have the lists of indices for each element.
00047 *   Additionally for each element there is an extension. There are
00048 *   TABLEEXTENSION WORDs reserved for that. The old system had two words
00049 *   One for the element in the rhs of the compile buffer and one for
00050 *   an additional rhs in case the original would be overwritten by a new
00051 *   definition, but the old was fixed by .global and hence it should be possible
00052 *   to restore it.
00053 *   New use (new = 24-sep-2001)
00054 *   rhs1,numCompBuffer1,rhs2,numCompBuffer2,usage
00055 *   Hence TABLEEXTENSION will be 5. Note that for 64 bits the use of the
00056 *   compiler buffer is overdoing it a bit, but it would be too complicated
00057 *   to try to give it special code.
00058 */
00059
00060 #include "form3.h"
00061 #include "minos.h"
00062 #include "comtool.h"
00063
00064 /* static UBYTE *sparse = (UBYTE *) "sparse"; */
00065 static UBYTE *tablebase = (UBYTE *) "tablebase";
00066
00067 /*
00068 *   #[ Includes :
00069 *   #[ ClearTableTree :
00070 */
```

```

00071
00072 void ClearTableTree(TABLES T)
00073 {
00074     COMPTREE *root;
00075     if ( T->boomlijst == 0 ) {
00076         T->MaxTreeSize = 125;
00077         T->boomlijst = (COMPTREE *)Malloc1(T->MaxTreeSize*sizeof(COMPTREE),
00078             "ClearTableTree");
00079     }
00080     root = T->boomlijst;
00081     T->numtree = 0;
00082     T->rootnum = 0;
00083     root->left = -1;
00084     root->right = -1;
00085     root->parent = -1;
00086     root->blnce = 0;
00087     root->value = -1;
00088     root->usage = 0;
00089 }
00090
00091 /*
00092 #] ClearTableTree :
00093 #[ InsTableTree :
00094
00095 int InsTableTree(TABLES T,WORD *,arglist)
00096     Searches for the element specified by the list of arguments.
00097     If found, it returns -(the offset in T->tablepointers)
00098     If not found, it will allocate a new element, balance the tree if
00099     necessary and return the number of the element in the boomlijst
00100     This number is always > 0, because we start from 1.
00101 */
00102
00103 int InsTableTree(TABLES T, WORD *tp)
00104 {
00105     COMPTREE *boomlijst, *q, *p, *s;
00106     WORD *v1, *v2, *v3, xstop;
00107     int ip, iq, is;
00108     if ( T->numtree + 1 >= T->MaxTreeSize ) {
00109         if ( T->MaxTreeSize == 0 ) ClearTableTree(T);
00110         else {
00111             is = T->MaxTreeSize * 2;
00112             s = (COMPTREE *)Malloc1(is*sizeof(COMPTREE),"InsTableTree");
00113             for ( ip = 0; ip < T->MaxTreeSize; ip++ ) { s[ip] = T->boomlijst[ip]; }
00114             if ( T->boomlijst ) M_free(T->boomlijst,"InsTableTree");
00115             T->boomlijst = s;
00116             T->MaxTreeSize = is;
00117         }
00118     }
00119     boomlijst = T->boomlijst;
00120     q = boomlijst + T->rootnum;
00121     if ( q->right == -1 ) { /* First element */
00122         T->numtree++;
00123         s = boomlijst+T->numtree;
00124         q->right = T->numtree;
00125         s->parent = T->rootnum;
00126         s->left = s->right = -1;
00127         s->blnce = 0;
00128         s->value = tp - T->tablepointers;
00129         s->usage = 0;
00130         return(T->numtree);
00131     }
00132     ip = q->right;
00133     if ( T->numind >= 0 ) xstop = T->numind;
00134     else xstop = *tp + 1;
00135     while ( ip >= 0 ) {
00136         p = boomlijst + ip;
00137         v1 = T->tablepointers + p->value;
00138         v2 = tp; v3 = tp + xstop;
00139         while ( *v1 == *v2 && v2 < v3 ) { v1++; v2++; }
00140         if ( v2 >= v3 ) return(-p->value);
00141         if ( *v1 > *v2 ) {
00142             iq = p->right;
00143             if ( iq >= 0 ) { ip = iq; }
00144             else {
00145                 T->numtree++;
00146                 is = T->numtree;
00147                 p->right = is;
00148                 s = boomlijst + is;
00149                 s->parent = ip; s->left = s->right = -1;
00150                 s->blnce = 0; s->value = tp - T->tablepointers;
00151                 s->usage = 0;
00152                 p->blnce++;
00153                 if ( p->blnce == 0 ) return(T->numtree);
00154                 goto balance;
00155             }
00156         }
00157         else if ( *v1 < *v2 ) {

```



```

00158         iq = p->left;
00159         if ( iq >= 0 ) { ip = iq; }
00160         else {
00161             T->numtree++;
00162             is = T->numtree;
00163             s = boomlijst+is;
00164             p->left = is;
00165             s->parent = ip; s->left = s->right = -1;
00166             s->blnce = 0; s->value = tp - T->tablepointers;
00167             s->usage = 0;
00168             p->blnce--;
00169             if ( p->blnce == 0 ) return(T->numtree);
00170             goto balance;
00171         }
00172     }
00173 }
00174 MesPrint("Serious problems in InsTableTree!\n");
00175 Terminate(-1);
00176 return(0);
00177 balance:;
00178 for (;;) {
00179     p = boomlijst + ip;
00180     iq = p->parent;
00181     if ( iq == T->rootnum ) break;
00182     q = boomlijst + iq;
00183     if ( ip == q->left ) q->blnce--;
00184     else q->blnce++;
00185     if ( q->blnce == 0 ) break;
00186     if ( q->blnce == -2 ) {
00187         if ( p->blnce == -1 ) { /* single rotation */
00188             q->left = p->right;
00189             p->right = iq;
00190             p->parent = q->parent;
00191             q->parent = ip;
00192             if ( boomlijst[p->parent].left == iq ) boomlijst[p->parent].left = ip;
00193             else boomlijst[p->parent].right = ip;
00194             if ( q->left >= 0 ) boomlijst[q->left].parent = iq;
00195             q->blnce = p->blnce = 0;
00196         }
00197         else { /* double rotation */
00198             s = boomlijst + is;
00199             q->left = s->right;
00200             p->right = s->left;
00201             s->right = iq;
00202             s->left = ip;
00203             if ( p->right >= 0 ) boomlijst[p->right].parent = ip;
00204             if ( q->left >= 0 ) boomlijst[q->left].parent = iq;
00205             s->parent = q->parent;
00206             q->parent = is;
00207             p->parent = is;
00208             if ( boomlijst[s->parent].left == iq )
00209                 boomlijst[s->parent].left = is;
00210             else boomlijst[s->parent].right = is;
00211             if ( s->blnce > 0 ) { q->blnce = s->blnce = 0; p->blnce = -1; }
00212             else if ( s->blnce < 0 ) { p->blnce = s->blnce = 0; q->blnce = 1; }
00213             else { p->blnce = s->blnce = q->blnce = 0; }
00214         }
00215         break;
00216     }
00217     else if ( q->blnce == 2 ) {
00218         if ( p->blnce == 1 ) { /* single rotation */
00219             q->right = p->left;
00220             p->left = iq;
00221             p->parent = q->parent;
00222             q->parent = ip;
00223             if ( boomlijst[p->parent].left == iq ) boomlijst[p->parent].left = ip;
00224             else boomlijst[p->parent].right = ip;
00225             if ( q->right >= 0 ) boomlijst[q->right].parent = iq;
00226             q->blnce = p->blnce = 0;
00227         }
00228         else { /* double rotation */
00229             s = boomlijst + is;
00230             q->right = s->left;
00231             p->left = s->right;
00232             s->left = iq;
00233             s->right = ip;
00234             if ( p->left >= 0 ) boomlijst[p->left].parent = ip;
00235             if ( q->right >= 0 ) boomlijst[q->right].parent = iq;
00236             s->parent = q->parent;
00237             q->parent = is;
00238             p->parent = is;
00239             if ( boomlijst[s->parent].left == iq ) boomlijst[s->parent].left = is;
00240             else boomlijst[s->parent].right = is;
00241             if ( s->blnce < 0 ) { q->blnce = s->blnce = 0; p->blnce = 1; }
00242             else if ( s->blnce > 0 ) { p->blnce = s->blnce = 0; q->blnce = -1; }
00243             else { p->blnce = s->blnce = q->blnce = 0; }
00244         }
00245     }

```

```

00245         break;
00246     }
00247     is = ip; ip = iq;
00248 }
00249 return(T->numtree);
00250 }
00251
00252 /*
00253 #] InsTableTree :
00254 #[ RedoTableTree :
00255
00256     To be used when a sparse table is trimmed due to a .store
00257     We rebuild the tree. In the future one could try to become faster
00258     at the cost of quite some complexity.
00259     We need to keep the first 'size' elements in the boomlijst.
00260     Kill all others and reconstruct the tree with the original ordering.
00261     This is very complicated! Because .store will either keep the whole
00262     table or remove the whole table we should not come here often.
00263     Hence we choose the slow solution for now.
00264 */
00265
00266 void RedoTableTree(TABLES T, int newsz)
00267 {
00268     WORD *tp;
00269     int i;
00270     ClearTableTree(T);
00271     for ( i = 0, tp = T->tablepointers; i < newsz; i++ ) {
00272         InsTableTree(T, tp);
00273         tp += ABS(T->numind)+TABLEEXTENSION;
00274     }
00275 }
00276
00277 /*
00278 #] RedoTableTree :
00279 #[ FindTableTree :
00280
00281 int FindTableTree(TABLES T, WORD *, arglist, int, inc)
00282     Searches for the element specified by the list of arguments.
00283     If found, it returns the offset in T->tablepointers
00284     If not found, it will return -1
00285     The list here is from the list of function arguments. Hence it
00286     has pairs of numbers -SNUMBER, index
00287     Actually inc says how many numbers there are and the above case is
00288     for inc = 2. For inc = 1 we have just a list of indices.
00289 */
00290
00291 int FindTableTree(TABLES T, WORD *tp, int inc)
00292 {
00293     COMPTREE *boomlijst = T->boomlijst, *q = boomlijst + T->rootnum, *p;
00294     WORD *v1, *v2, *v3, xstop;
00295     int ip, iq;
00296     if ( q->right == -1 ) return(-1);
00297     ip = q->right;
00298     if ( inc > 1 ) tp += inc-1;
00299     if ( T->numind >= 0 ) xstop = T->numind;
00300     else { /* We have to read the number of arguments first */
00301         if ( *tp <= 0 ) return(-1); /* Cannot be! */
00302         xstop = *tp+1;
00303     }
00304     while ( ip >= 0 ) {
00305         p = boomlijst + ip;
00306         v1 = T->tablepointers + p->value;
00307         v2 = tp; v3 = v1 + xstop;
00308         while ( *v1 == *v2 && v1 < v3 ) { v1++; v2 += inc; }
00309         if ( v1 == v3 ) {
00310             p->usage++;
00311             return(p->value);
00312         }
00313         if ( *v1 > *v2 ) {
00314             iq = p->right;
00315             if ( iq >= 0 ) { ip = iq; }
00316             else return(-1);
00317         }
00318         else if ( *v1 < *v2 ) {
00319             iq = p->left;
00320             if ( iq >= 0 ) { ip = iq; }
00321             else return(-1);
00322         }
00323     }
00324     MesPrint("Serious problems in FindTableTree\n");
00325     Terminate(-1);
00326     return(-1);
00327 }
00328
00329 /*
00330 #] FindTableTree :
00331 #[ DoTableExpansion :

```

```

00332 */
00333
00334 int DoTableExpansion(WORD *term, WORD level)
00335 {
00336     GETIDENTITY
00337     WORD *t, *tstop, *stopper, *termout, *m, *mm, *tp, *r, xx;
00338     WORD numsubexp, numbuf;
00339     TABLES T = 0;
00340     int i, j, num;
00341     AN.TeInFun = AR.TePos = 0;
00342     tstop = term + *term;
00343     stopper = tstop - ABS(tstop[-1]);
00344     t = term+1;
00345     while ( t < stopper ) {
00346         if ( *t != TABLEFUNCTION ) { t += t[1]; continue; }
00347         if ( t[FUNHEAD] > -FUNCTION ) { t += t[1]; continue; }
00348         T = functions[-t[FUNHEAD]-FUNCTION].tabl;
00349         if ( T == 0 ) { t += t[1]; continue; }
00350         if ( T->sparse ) T = T->sparse;
00351         if ( t[1] == FUNHEAD+2 && t[FUNHEAD+1] <= -FUNCTION ) break;
00352         if ( t[1] < FUNHEAD+1+2*ABS(T->numind) ) { t += t[1]; continue; }
00353         for ( i = 0; i < ABS(T->numind); i++ ) {
00354             if ( t[FUNHEAD+1+2*i] != -SYMBOL ) break;
00355         }
00356         if ( i >= ABS(T->numind) ) break;
00357         t += t[1];
00358     }
00359     if ( t >= stopper ) {
00360         MesPrint("Internal error: Missing table_ function");
00361         Terminate(-1);
00362     }
00363     /*
00364     Table in T. Now collect the numbers of the symbols;
00365     */
00366     termout = AT.WorkPointer;
00367     if ( T->sparse ) {
00368         for ( i = 0; i < T->totind; i++ ) {
00369             /*
00370             Loop over all table elements
00371             */
00372             m = termout + 1; mm = term + 1;
00373             while ( mm < t ) *m++ = *mm++;
00374             r = m;
00375             if ( t[1] == FUNHEAD+2 && t[FUNHEAD+1] <= -FUNCTION ) {
00376                 *m++ = -t[FUNHEAD+1];
00377                 tp = T->tablepointers + (ABS(T->numind)+TABLEEXTENSION)*i;
00378                 if ( T->numind < 0 ) {
00379                     xx = tp[0]+1;
00380                     *m++ = FUNHEAD+xx*2;
00381                     for ( j = 2; j < FUNHEAD; j++ ) *m++ = 0;
00382                     for ( j = 0; j < xx; j++ ) {
00383                         *m++ = -SNUMBER; *m++ = *tp++;
00384                     }
00385                 }
00386                 else {
00387                     *m++ = FUNHEAD+T->numind*2;
00388                     for ( j = 2; j < FUNHEAD; j++ ) *m++ = 0;
00389                     for ( j = 0; j < T->numind; j++ ) {
00390                         *m++ = -SNUMBER; *m++ = *tp++;
00391                     }
00392                 }
00393             }
00394             else if ( T->numind < 0 ) {
00395                 tp = T->tablepointers + (ABS(T->numind)+TABLEEXTENSION)*i;
00396                 xx = tp[0]+1;
00397                 *m++ = SYMBOL; *m++ = 2+xx*2; mm = t + FUNHEAD+1;
00398                 for ( j = 0; j < xx; j++, mm += 2, tp++ ) {
00399                     if ( *tp != 0 ) { *m++ = mm[1]; *m++ = *tp; }
00400                 }
00401                 r[1] = m-r;
00402                 if ( r[1] == 2 ) m = r;
00403             }
00404             else {
00405                 *m++ = SYMBOL; *m++ = 2+T->numind*2; mm = t + FUNHEAD+1;
00406                 tp = T->tablepointers + (T->numind+TABLEEXTENSION)*i;
00407                 for ( j = 0; j < T->numind; j++, mm += 2, tp++ ) {
00408                     if ( *tp != 0 ) { *m++ = mm[1]; *m++ = *tp; }
00409                 }
00410                 r[1] = m-r;
00411                 if ( r[1] == 2 ) m = r;
00412             }
00413             /*
00414             The next code replaces this old code
00415
00416             *m++ = SUBEXPRESSION;
00417             *m++ = SUBEXPSIZE;
00418             *m++ = *tp;

```

```

00419         *m++ = 1;
00420         *m++ = T->bufnum;
00421         FILLSUB(m);
00422         mm = t + t[1];
00423
00424         We had forgotten to take the parameters into account.
00425         Hence the subexpression prototype for wildcards was missed
00426         Now we slow things down a little bit, but we do not run
00427         any risks. There is still one problem. We have not checked
00428         that the prototype matches.
00429     */
00430     tp = T->tablepointers + (ABS(T->numind)+TABLEEXTENSION)*i
00431           +ABS(T->numind);
00432     numsubexp = tp[0]; numbuf = tp[1];
00433     r = m;
00434 #ifdef WITHPTHREADS
00435     tp = T->prototype[identity];
00436 #else
00437     tp = T->prototype;
00438 #endif
00439     for ( j = 0; j < tp[1]; j++ ) *m++ = tp[j];
00440     r[2] = numsubexp; r[4] = numbuf;
00441 /*
00442     r = m;
00443     tp = T->tablepointers + (ABS(T->numind)+TABLEEXTENSION)*i;
00444     *m++ = -t[FUNHEAD];
00445     if ( T->numind < 0 ) {
00446         xx = tp[0]+1;
00447         *m++ = t[1] - xx - T->numind - 1;
00448     }
00449     else {
00450         xx = T->numind;
00451         *m++ = t[1] - 1;
00452     }
00453     for ( j = 2; j < FUNHEAD; j++ ) *m++ = t[j];
00454     for ( j = 0; j < xx; j++ ) {
00455         *m++ = -SNUMBER; *m++ = *tp++;
00456     }
00457     tp = t + FUNHEAD + 1 + 2*T->numind;
00458     mm = t + t[1];
00459     while ( tp < mm ) *m++ = *tp++;
00460     r[1] = m-r;
00461 */
00462 /*
00463     From now on is old code
00464 */
00465     mm = t + t[1];
00466     while ( mm < tstop ) *m++ = *mm++;
00467     *termout = m - termout;
00468     AT.WorkPointer = m;
00469     if ( Generator(BHEAD termout,level) ) {
00470         MesCall("DoTableExpand");
00471         return(-1);
00472     }
00473     AT.WorkPointer = termout;
00474 }
00475 }
00476 else {
00477     for ( i = 0; i < T->totind; i++ ) {
00478 #if TABLEEXTENSION == 2
00479         if ( T->tablepointers[i] < 0 ) continue;
00480 #else
00481         if ( T->tablepointers[TABLEEXTENSION*i] < 0 ) continue;
00482 #endif
00483         m = termout + 1; mm = term + 1;
00484         while ( mm < t ) *m++ = *mm++;
00485         r = m;
00486         if ( t[1] == FUNHEAD+2 && t[FUNHEAD+1] <= -FUNCTION ) {
00487             *m++ = -t[FUNHEAD+1];
00488             *m++ = FUNHEAD+T->numind*2;
00489             for ( j = 2; j < FUNHEAD; j++ ) *m++ = 0;
00490             tp = T->tablepointers + (T->numind+TABLEEXTENSION)*i;
00491             for ( j = 0; j < T->numind; j++ ) {
00492                 if ( j > 0 ) {
00493                     num = T->mm[j].mini + ( i % T->mm[j-1].size ) / T->mm[j].size;
00494                 }
00495                 else {
00496                     num = T->mm[j].mini + i / T->mm[j].size;
00497                 }
00498                 *m++ = -SNUMBER; *m++ = num;
00499             }
00500         }
00501         else {
00502             *m++ = SYMBOL; *m++ = 2+T->numind*2; mm = t + FUNHEAD+1;
00503             for ( j = 0; j < T->numind; j++, mm += 2 ) {
00504                 if ( j > 0 ) {
00505                     num = T->mm[j].mini + ( i % T->mm[j-1].size ) / T->mm[j].size;

```

```

00506         }
00507         else {
00508             num = T->mm[j].mini + i / T->mm[j].size;
00509         }
00510         if ( num != 0 ) { *m++ = mm[1]; *m++ = num; }
00511     }
00512     r[1] = m-r;
00513     if ( r[1] == 2 ) m = r;
00514 }
00515 /*
00516     The next code replaces this old code
00517
00518     *m++ = SUBEXPRESSION;
00519     *m++ = SUBEXPSIZE;
00520     *m++ = *tp;
00521     *m++ = 1;
00522     *m++ = T->bufnum;
00523     FILLSUB(m);
00524     mm = t + t[1];
00525
00526     We had forgotten to take the parameters into account.
00527     Hence the subexpression prototype for wildcards was missed
00528     Now we slow things down a little bit, but we do not run
00529     any risks. There is still one problem. We have not checked
00530     that the prototype matches.
00531 */
00532     r = m;
00533     *m++ = -t[FUNHEAD];
00534     *m++ = t[1] - 1;
00535     for ( j = 2; j < FUNHEAD; j++ ) *m++ = t[j];
00536     for ( j = 0; j < T->numind; j++ ) {
00537         if ( j > 0 ) {
00538             num = T->mm[j].mini + ( i % T->mm[j-1].size ) / T->mm[j].size;
00539         }
00540         else {
00541             num = T->mm[j].mini + i / T->mm[j].size;
00542         }
00543         *m++ = -SNUMBER; *m++ = num;
00544     }
00545     tp = t + FUNHEAD + 1 + 2*T->numind;
00546     mm = t + t[1];
00547     while ( tp < mm ) *m++ = *tp++;
00548     r[1] = m - r;
00549 /*
00550     From now on is old code
00551 */
00552     while ( mm < tstop ) *m++ = *mm++;
00553     *termout = m - termout;
00554     AT.WorkPointer = m;
00555     if ( Generator(BHEAD termout, level) ) {
00556         MesCall("DoTableExpand");
00557         return(-1);
00558     }
00559 }
00560 }
00561 return(0);
00562 }
00563
00564 /*
00565 #] DoTableExpansion :
00566 #[ TableBase :
00567
00568 File with all the database related things.
00569 We have the routines for the generic database command
00570 TableBase,options;
00571 TB,options;
00572 Options are:
00573     Open "File.tbl";           Open for R/W
00574     Open "File.tbl", readonly; Open for R
00575     Create "File.tbl";         Create for write
00576     Load "File.tbl", tablename; Loads stubs of table
00577     Load "File.tbl";           Loads stubs of all tables
00578     Enter "File.tbl", tablename; Loads whole table
00579     Enter "File.tbl";          Loads all tables
00580     Audit "File.tbl", options; Print list of contents
00581     Replace "File.tbl", tablename; Saves a table (with overwrite)
00582     Replace "File.tbl", table element; Saves a table element
00583     Cleanup "File.tbl";        Makes tables contingent
00584     AddTo "File.tbl" tablename; Add if not yet there.
00585     AddTo "File.tbl" table element; Add if not yet there.
00586     Delete "File.tbl" tablename;
00587     Delete "File.tbl" table element;
00588
00589     On/Off substitute;
00590     On/Off compress "File.tbl";
00591     id tbl_(f?,?a) = f(?a);
00592     When a tbl_ is used, automatically the corresponding element is compiled

```

```

00593     at the start of the next module.
00594     if TB,On,substitute [tablename], use of table RHS (if loaded)
00595     if TB,Off,substitute [tablename], use of tbl_(table,...);
00596
00597
00598     Still needed: Something like OverLoad to allow loading parts of a table
00599     from more than one file. Date stamps needed? In that case we need a touch
00600     command as well.
00601
00602     If we put all our diagrams inside, we have to go outside the concept
00603     of tables.
00604
00605     #] TableBase :
00606     #[ CoTableBase :
00607
00608     To be followed by ,subkey
00609 */
00610 static KEYWORD tboptions[] = {
00611     {"addto",          (TFUN)CoTBaddto,      0,    PARTEST}
00612     ,{"audit",         (TFUN)CoTBaudit,      0,    PARTEST}
00613     ,{"cleanup",       (TFUN)CoTBCleanup,    0,    PARTEST}
00614     ,{"create",        (TFUN)CoTBcreate,     0,    PARTEST}
00615     ,{"enter",         (TFUN)CoTBenter,      0,    PARTEST}
00616     ,{"help",          (TFUN)CoTBhelp,       0,    PARTEST}
00617     ,{"load",          (TFUN)CoTBload,       0,    PARTEST}
00618     ,{"off",           (TFUN)CoTBoff,        0,    PARTEST}
00619     ,{"on",            (TFUN)CoTBon,         0,    PARTEST}
00620     ,{"open",          (TFUN)CoTBopen,       0,    PARTEST}
00621     ,{"replace",       (TFUN)CoTBreplace,    0,    PARTEST}
00622     ,{"use",           (TFUN)CoTBuse,        0,    PARTEST}
00623 };
00624
00625 static UBYTE *tablebasename = 0;
00626
00627 int CoTableBase(UBYTE *s)
00628 {
00629     UBYTE *option, c, *t;
00630     int i,optlistsize = sizeof(tboptions)/sizeof(KEYWORD), error = 0;
00631     while ( *s == ' ' ) s++;
00632     if ( *s != '"' ) {
00633         if ( ( tolower(*s) == 'h' ) && ( tolower(s[1]) == 'e' )
00634             && ( tolower(s[2]) == 'l' ) && ( tolower(s[3]) == 'p' )
00635             && ( FG.cTable[s[4]] > 1 ) ) {
00636             CoTBhelp(s);
00637             return(0);
00638         }
00639         proper;;
00640         MesPrint("&Proper syntax: TableBase \"filename\" options");
00641         return(1);
00642     }
00643     s++; tablebasename = s;
00644     while ( *s && *s != '"' ) s++;
00645     if ( *s != '"' ) goto proper;
00646     t = s; s++; *t = 0;
00647     while ( *s == ' ' || *s == '\\t' || *s == ',' ) s++;
00648     option = s;
00649     while ( FG.cTable[*s] == 0 ) s++;
00650     c = *s; *s = 0;
00651     for ( i = 0; i < optlistsize; i++ ) {
00652         if ( StrICmp(option, (UBYTE *) (tboptions[i].name)) == 0 ) {
00653             *s = c;
00654             while ( *s == ',' ) s++;
00655             error = (tboptions[i].func)(s);
00656             *t = '"';
00657             return(error);
00658         }
00659     }
00660     MesPrint("&Unrecognized option %s in TableBase statement",option);
00661     return(1);
00662 }
00663
00664 /*
00665     #] CoTableBase :
00666     #[ FlipTable :
00667
00668     Flips the table between use as 'stub' and regular use
00669 */
00670
00671 int FlipTable(FUNCTIONS f, int type)
00672 {
00673     TABLES T, TT;
00674     T = f->tabl;
00675     if ( ( TT = T->spare ) == 0 ) {
00676         MesPrint("Error: trying to change mode on a table that has no tablebase");
00677         return(-1);
00678     }
00679     if ( TT->mode == type ) f->tabl = TT;

```

```

00680     return(0);
00681 }
00682
00683 /*
00684 #] FlipTable :
00685 #[ SpareTable :
00686
00687     Creates a spare element for a table. This is used in the table bases.
00688     It is a (thus far) empty copy of the TT table.
00689     By using FlipTable we can switch between them and alter which version of
00690     a table we will be using. Note that this also causes some extra work in the
00691     ResetVariables and the Globalize routines.
00692 */
00693
00694 int SpareTable(TABLES TT)
00695 {
00696     TABLES T;
00697     T = (TABLES)Malloc1(sizeof(struct TableEs),"table");
00698     T->defined = T->mdefined = 0; T->sparse = TT->sparse; T->mm = 0; T->flags = 0;
00699     T->numtree = 0; T->rootnum = 0; T->MaxTreeSize = 0;
00700     T->boomlijst = 0;
00701     T->strict = TT->strict;
00702     T->bounds = TT->bounds;
00703     T->bufnum = inicbufs();
00704     T->argtail = TT->argtail;
00705     T->spare = TT;
00706     T->bufferssize = 8;
00707     T->buffers = (WORD *)Malloc1(sizeof(WORD)*T->bufferssize,"SpareTable buffers");
00708     T->buffersfill = 0;
00709     T->buffers[T->buffersfill++] = T->bufnum;
00710     T->mode = 0;
00711     T->numind = TT->numind;
00712     T->totind = 0;
00713     T->prototype = TT->prototype;
00714     T->pattern = TT->pattern;
00715     T->tablepointers = 0;
00716     T->reserved = 0;
00717     T->tablenum = 0;
00718     T->numdummies = 0;
00719     T->mm = (MINMAX *)Malloc1(ABS(T->numind)*sizeof(MINMAX),"table dimensions");
00720     T->flags = (WORD *)Malloc1(ABS(T->numind)*sizeof(WORD),"table flags");
00721     ClearTableTree(T);
00722     TT->spare = T;
00723     TT->mode = 1;
00724     return(0);
00725 }
00726
00727 /*
00728 #] SpareTable :
00729 #[ FindTB :
00730
00731     Looks for a tablebase with the given name in the active tablebases.
00732 */
00733
00734 DBASE *FindTB(UBYTE *name)
00735 {
00736     DBASE *d;
00737     int i;
00738     for ( i = 0; i < NumTableBases; i++ ) {
00739         d = tablebases+i;
00740         if ( d->name && ( StrCmp(name,(UBYTE *) (d->name)) == 0 ) ) { return(d); }
00741     }
00742     return(0);
00743 }
00744
00745 /*
00746 #] FindTB :
00747 #[ CoTBcreate :
00748
00749     Creates a new tablebase.
00750     Error is when there is already an active tablebase by this name.
00751     If a file with the given name exists already, but it does not correspond
00752     to an active table base, its contents will be lost.
00753     Note that tablebasename is a static variable, defined in CoTableBase
00754 */
00755
00756 int CoTBcreate(UBYTE *s)
00757 {
00758     DUMMYUSE(s);
00759     if ( FindTB(tablebasename) != 0 ) {
00760         MesPrint("&There is already an open TableBase with the name %s",tablebasename);
00761         return(-1);
00762     }
00763     NewDbase((char *)tablebasename,0);
00764     return(0);
00765 }
00766

```

```

00767 /*
00768 #] CoTBcreate :
00769 #[ CoTBopen :
00770 */
00771
00772 int CoTBopen(UBYTE *s)
00773 {
00774     DBASE *d;
00775     MLONG rw = 1;
00776
00777     SkipSpaces(&s);
00778
00779     if ( *s ) {
00780         if ( ConsumeOption(&s,"readonly") != 0 ) {
00781             rw = 0;
00782         } else {
00783             MesPrint("&Invalid option for TableBase open: %s, ignoring", s);
00784         }
00785     }
00786
00787     if ( ( d = FindTB(tablebasename) ) != 0 ) {
00788         MesPrint("&There is already an open TableBase with the name %s",tablebasename);
00789         return(-1);
00790     }
00791     d = GetDbase((char *)tablebasename, rw);
00792     if ( CheckTableDeclarations(d) ) return(-1);
00793     return(0);
00794 }
00795
00796 /*
00797 #] CoTBopen :
00798 #[ CoTBaddto :
00799 */
00800
00801 int CoTBaddto(UBYTE *s)
00802 {
00803     GETIDENTITY
00804     DBASE *d;
00805     UBYTE *tablename, c, *t, elementstring[ELEMENTSIZE+20], *ss, *es;
00806     WORD type, funnum, lbrac, first, num, *expr, *w;
00807     TABLES T = 0;
00808     MLONG basenumber;
00809     LONG x;
00810     int i, j, error = 0, sum;
00811     if ( ( d = FindTB(tablebasename) ) == 0 ) {
00812         MesPrint("&No open tablebase with the name %s",tablebasename);
00813         return(-1);
00814     }
00815
00816     if ( ( d->rwmode ) == 0 ) {
00817         MesPrint("&Tablebase with the name %s opened in read only mode",tablebasename);
00818         return(-1);
00819     }
00820     AO.DollarOutSizeBuffer = 32;
00821     AO.DollarOutBuffer = (UBYTE *)Malloc1(AO.DollarOutSizeBuffer,
00822         "TableOutBuffer");
00823
00824     /*
00825     Now loop through the names and start adding
00826     */
00827     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
00828     while ( *s ) {
00829         tablename = s;
00830         if ( ( s = SkipAName(s) ) == 0 ) goto tableabort;
00831         c = *s; *s = 0;
00832         if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
00833             || ( T = functions[funnum].tabl ) == 0 ) {
00834             MesPrint("&%s should be a previously declared table",tablename);
00835             *s = c; goto tableabort;
00836         }
00837         if ( T->sparse == 0 ) {
00838             MesPrint("&%s should be a sparse table",tablename);
00839             *s = c; goto tableabort;
00840         }
00841         basenumber = AddTableName(d,(char *)tablename,T);
00842         if ( T->sparse && ( T->mode == 1 ) ) T = T->sparse;
00843         if ( basenumber < 0 ) basenumber = -basenumber;
00844         else if ( basenumber == 0 ) { *s = c; goto tableabort; }
00845         *s = c;
00846         if ( *s == '(' ) { /* Addition of single element */
00847             s++; es = s;
00848             for ( i = 0, w = AT.WorkPointer; i < ABS(T->numind); i++ ) {
00849                 ParseSignedNumber(x,s);
00850                 if ( FG.cTable[s[-1]] != 1 || ( *s != ',' && *s != ')' ) ) {
00851                     MesPrint("&Table arguments in TableBase addto statement should be numbers");
00852                     return(1);
00853                 }
00854                 *w++ = x;

```



```

00854         if ( *s == ' ' ) break;
00855         s++;
00856     }
00857     if ( *s != ' ' || i < ( ABS(T->numind) - 1 ) ) {
00858         MesPrint("&Incorrect number of table arguments in TableBase addto statement. Should be
%d"
00859             ,ABS(T->numind));
00860         error = 1;
00861     }
00862     c = *s; *s = 0;
00863     i = FindTableTree(T,AT.WorkPointer,1);
00864     if ( i < 0 ) {
00865         MesPrint("&Element %s has not been defined",es);
00866         error = 1;
00867         *s++ = c;
00868     }
00869     else if ( ExistsObject(d,basenumber,(char *)es) ) {}
00870     else {
00871         int dict = AO.CurrentDictionary;
00872         AO.CurrentDictionary = 0;
00873         sum = i + ABS(T->numind);
00874         /*
00875             See also commentary below
00876         */
00877         AO.DollarInOutBuffer = 1;
00878         AO.PrintType = 1;
00879         ss = AO.DollarOutBuffer;
00880         *ss = 0;
00881         AO.OutInBuffer = 1;
00882         #if ( TABLEEXTENSION == 2 )
00883             expr = cbuf[T->bufnum].rhs[T->tablepointers[sum]];
00884         #else
00885             expr = cbuf[T->tablepointers[sum+1]].rhs[T->tablepointers[sum]];
00886         #endif
00887         lbrac = 0; first = 0;
00888         while ( *expr ) {
00889             if ( WriteTerm(expr,&lbrac,first,PRINTON,0) ) {
00890                 error = 1; break;
00891             }
00892             expr += *expr;
00893         }
00894         AO.OutInBuffer = 0;
00895         AddObject(d,basenumber,(char *)es,(char *) (AO.DollarOutBuffer));
00896         *s++ = c;
00897         AO.CurrentDictionary = dict;
00898     }
00899 }
00900 else {
00901     /*
00902         Now we have to start looping through all defined elements of this table.
00903         We have to construct the arguments in text format.
00904     */
00905     for ( i = 0; i < T->totind; i++ ) {
00906         #if ( TABLEEXTENSION == 2 )
00907             if ( !T->sparse && T->tablepointers[i] < 0 ) continue;
00908         #else
00909             if ( !T->sparse && T->tablepointers[TABLEEXTENSION+i] < 0 ) continue;
00910         #endif
00911         sum = i * ( ABS(T->numind) + TABLEEXTENSION );
00912         t = elementstring;
00913         for ( j = 0; j < ABS(T->numind); j++, sum++ ) {
00914             if ( j > 0 ) *t++ = ',';
00915             num = T->tablepointers[sum];
00916             t = NumCopy(num,t);
00917             if ( ( t - elementstring ) >= ELEMENTSIZE ) {
00918                 MesPrint("&Table element specification takes more than %ld characters and cannot
be handled",
00919                     (MLONG) ELEMENTSIZE);
00920                 goto tableabort;
00921             }
00922         }
00923         if ( ExistsObject(d,basenumber,(char *)elementstring) ) { continue; }
00924         /*
00925             We have the number in basenumber and the element in elementstring.
00926             Now we need the rhs. We can use the code from WriteDollarToBuffer.
00927             Main complication: in the table compiler buffer there can be
00928             brackets. The dollars do not have those.....
00929         */
00930         AO.DollarInOutBuffer = 1;
00931         AO.PrintType = 1;
00932         ss = AO.DollarOutBuffer;
00933         *ss = 0;
00934         AO.OutInBuffer = 1;
00935         #if ( TABLEEXTENSION == 2 )
00936             expr = cbuf[T->bufnum].rhs[T->tablepointers[sum]];
00937         #else
00938             expr = cbuf[T->tablepointers[sum+1]].rhs[T->tablepointers[sum]];

```

```

00939 #endif
00940         lbrac = 0; first = 0;
00941         while ( *expr ) {
00942             if ( WriteTerm(expr,&lbrac,first,PRINTON,0) ) {
00943                 error = 1; break;
00944             }
00945             expr += *expr;
00946         }
00947         AO.OutInBuffer = 0;
00948         AddObject(d,basenumber, (char *)elementstring, (char *) (AO.DollarOutBuffer));
00949     }
00950 }
00951 while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
00952 }
00953 if ( WriteIniInfo(d) ) goto tableabort;
00954 M_free(AO.DollarOutBuffer,"DollarOutBuffer");
00955 AO.DollarOutBuffer = 0;
00956 AO.DollarOutSizeBuffer = 0;
00957 return(error);
00958 tableabort:;
00959 M_free(AO.DollarOutBuffer,"DollarOutBuffer");
00960 AO.DollarOutBuffer = 0;
00961 AO.DollarOutSizeBuffer = 0;
00962 AO.OutInBuffer = 0;
00963 return(1);
00964 }
00965
00966 /*
00967 #] CoTBaddto :
00968 #[ CoTBenter :
00969
00970 Loads the elements of the tables specified into memory and sends them
00971 one by one to the compiler as Fill statements.
00972 */
00973
00974 int CoTBenter(UBYTE *s)
00975 {
00976     DBASE *d;
00977     MLONG basenumber;
00978     UBYTE *arguments, *rhs, *buffer, *t, *u, c, *tablename;
00979     LONG size;
00980     int i, j, error = 0, error1 = 0, printall = 0;
00981     TABLES T = 0;
00982     WORD type, funnum;
00983     int dict = AO.CurrentDictionary;
00984     AO.CurrentDictionary = 0;
00985     if ( ( d = FindTB(tablebasename) ) == 0 ) {
00986         MesPrint("&No open tablebase with the name %s, check for existence of file or try readonly
mode when opening.",tablebasename);
00987         error = -1;
00988         goto Endofall;
00989     }
00990     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
00991     if ( *s == '!' ) { printall = 1; s++; }
00992     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
00993     if ( *s ) {
00994         while ( *s ) {
00995             tablename = s;
00996             if ( ( s = SkipAName(s) ) == 0 ) { error = 1; goto Endofall; }
00997             c = *s; *s = 0;
00998             if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
|| ( T = functions[funnum].tabl ) == 0 ) {
01000                 MesPrint("&%s should be a previously declared table",tablename);
01001                 basenumber = 0;
01002             }
01003             else if ( T->sparse == 0 ) {
01004                 MesPrint("&%s should be a sparse table",tablename);
01005                 basenumber = 0;
01006             }
01007             else { basenumber = GetTableName(d,(char *)tablename); }
01008             if ( T->sparse == 0 ) { SpareTable(T); }
01009             if ( basenumber > 0 ) {
01010                 for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01011                     for ( j = 0; j < NUMOBJECTS; j++ ) {
01012                         if ( basenumber != d->iblocks[i]->objects[j].tablename )
01013                             continue;
01014                         arguments = (UBYTE *) (d->iblocks[i]->objects[j].element);
01015                         rhs = (UBYTE *) ReadObject(d,basenumber, (char *)arguments);
01016                         if ( printall ) {
01017                             if ( rhs ) {
01018                                 MesPrint("%s(%s) = %s",tablename,arguments,rhs);
01019                             }
01020                             else {
01021                                 MesPrint("%s(%s) = 0",tablename,arguments);
01022                             }
01023                         }
01024                     }
01025                 }
01026             }
01027         }
01028     }
01029     if ( rhs ) {

```

```

01025         u = rhs; while ( *u ) u++;
01026         size = u-rhs;
01027         u = arguments; while ( *u ) u++;
01028         size += u-arguments;
01029         u = tablename; while ( *u ) u++;
01030         size += u-tablename;
01031         size += 6;
01032         buffer = (UBYTE *)Malloc1(size,"TableBase copy");
01033         t = tablename; u = buffer;
01034         while ( *t ) *u++ = *t++;
01035         *u++ = '(';
01036         t = arguments;
01037         while ( *t ) *u++ = *t++;
01038         *u++ = ')'; *u++ = '=';
01039         t = rhs;
01040         while ( *t ) *u++ = *t++;
01041         if ( t == rhs ) *u++ = '0';
01042         *u++ = 0; *u = 0;
01043         M_free(rhs,"rhs in TBenter");
01044
01045         error1 = CoFill(buffer);
01046
01047         if ( error1 < 0 ) goto Endofall;
01048         if ( error1 != 0 ) error = error1;
01049         M_free(buffer,"TableBase copy");
01050     }
01051 }
01052 }
01053 }
01054 *s = c;
01055 while ( *s == ',' || *s == ' ' || *s == '\\t' ) s++;
01056 }
01057 }
01058 else {
01059     s = (UBYTE *) (d->tablenames); basenumber = 0;
01060     while ( *s ) {
01061         basenumber++;
01062         tablename = s; while ( *s ) s++; s++;
01063         while ( *s ) s++;
01064         s++;
01065         if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
01066             || ( T = functions[funnum].tabl ) == 0 ) {
01067             MesPrint("%s should be a previously declared table",tablename);
01068         }
01069         else if ( T->sparse == 0 ) {
01070             MesPrint("%s should be a sparse table",tablename);
01071         }
01072         if ( T->sparse == 0 ) { SpareTable(T); }
01073         for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01074             for ( j = 0; j < NUMOBJECTS; j++ ) {
01075                 if ( d->iblocks[i]->objects[j].tablenumber == basenumber ) {
01076                     arguments = (UBYTE *) (d->iblocks[i]->objects[j].element);
01077                     rhs = (UBYTE *) ReadObject(d,basenumber,(char *)arguments);
01078                     if ( printall ) {
01079                         if ( rhs ) {
01080                             MesPrint("%s%s = %s",tablename,arguments,rhs);
01081                         }
01082                         else {
01083                             MesPrint("%s%s = 0",tablename,arguments);
01084                         }
01085                     }
01086                     if ( rhs ) {
01087                         u = rhs; while ( *u ) u++;
01088                         size = u-rhs;
01089                         u = arguments; while ( *u ) u++;
01090                         size += u-arguments;
01091                         u = tablename; while ( *u ) u++;
01092                         size += u-tablename;
01093                         size += 6;
01094                         buffer = (UBYTE *) Malloc1(size,"TableBase copy");
01095                         t = tablename; u = buffer;
01096                         while ( *t ) *u++ = *t++;
01097                         *u++ = '(';
01098                         t = arguments;
01099                         while ( *t ) *u++ = *t++;
01100                         *u++ = ')'; *u++ = '=';
01101                         t = rhs;
01102                         while ( *t ) *u++ = *t++;
01103                         if ( t == rhs ) *u++ = '0';
01104                         *u++ = 0; *u = 0;
01105                         M_free(rhs,"rhs in TBenter");
01106
01107                         error1 = CoFill(buffer);
01108
01109                         if ( error1 < 0 ) goto Endofall;
01110                         if ( error1 != 0 ) error = error1;
01111                         M_free(buffer,"TableBase copy");

```

```

01112     }
01113     }
01114     }
01115     }
01116     }
01117 }
01118 Endofall;;
01119 AO.CurrentDictionary = dict;
01120 return(error);
01121 }
01122
01123 /*
01124 #] CoTBenter :
01125 #[ CoTestUse :
01126
01127     Possibly to be followed by names of tables.
01128     We make an array of TABLES structs to be tested in AC.usedtables.
01129     Note: only sparse tables are allowed.
01130     No arguments means all tables.
01131 */
01132
01133 int CoTestUse(UBYTE *s)
01134 {
01135     GETIDENTITY
01136     UBYTE *tablename, c;
01137     WORD type, funnum, *w;
01138     TABLES T;
01139     int error = 0;
01140     w = AT.WorkPointer;
01141     *w++ = TYPETESTUSE; *w++ = 2;
01142     while ( *s ) {
01143         tablename = s;
01144         if ( ( s = SkipAName(s) ) == 0 ) return(1);
01145         c = *s; *s = 0;
01146         if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
01147             || ( T = functions[funnum].tabl ) == 0 ) {
01148             MesPrint("&%s should be a previously declared table",tablename);
01149             error = 1;
01150         }
01151         else if ( T->sparse == 0 ) {
01152             MesPrint("&%s should be a sparse table",tablename);
01153             error = 1;
01154         }
01155         *w++ = funnum + FUNCTION;
01156         *s = c;
01157         while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
01158     }
01159     AT.WorkPointer[1] = w - AT.WorkPointer;
01160 /*
01161     if ( AT.WorkPointer[1] > 2 ) {
01162         AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
01163     }
01164 */
01165     AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
01166     return(error);
01167 }
01168
01169 /*
01170 #] CoTestUse :
01171 #[ CheckTableDeclarations :
01172
01173     Checks that all tables in a tablebase have identical properties to
01174     possible previous declarations. If they have not been declared
01175     before, they are declared here.
01176 */
01177
01178 int CheckTableDeclarations(DBASE *d)
01179 {
01180     WORD type, funnum;
01181     UBYTE *s, *ss, *t, *command = 0;
01182     int k, error = 0, error1, i;
01183     TABLES T;
01184     LONG commandsize = 0;
01185
01186     s = (UBYTE *) (d->tablenames);
01187     for ( k = 0; k < d->topnumber; k++ ) {
01188         if ( GetVar(s,&type,&funnum,ANYTYPE,NOAUTO) == NAMENOTFOUND ) {
01189             /*
01190                 We have to declare the table
01191             */
01192             ss = s; i = 0; while ( *ss ) { ss++; i++; } /* name */
01193             ss++; while ( *ss ) { ss++; i++; } /* tail */
01194             if ( commandsize == 0 ) {
01195                 commandsize = i + 15;
01196                 if ( commandsize < 100 ) commandsize = 100;
01197             }
01198             if ( (i+11) > commandsize ) {

```

```

01199         if ( command ) { M_free(command,"table command"); command = 0; }
01200         commandsize = i+10;
01201     }
01202     if ( command == 0 ) {
01203         command = (UBYTE *)Malloc1(commandsize,"table command");
01204     }
01205     t = command; ss = tablebase; while ( *ss ) *t++ = *ss++;
01206     *t++ = ','; while ( *s ) *t++ = *s++;
01207     s++; while ( *s ) *t++ = *s++;
01208     *t++ = ')'; *t = 0; s++;
01209     error1 = DoTable(command,1);
01210     if ( error1 ) error = error1;
01211 }
01212 else if ( ( type != CFUNCTION )
01213         || ( T = functions[funnum].tabl ) == 0 )
01214         || ( T->sparse == 0 ) ) {
01215     MesPrint("%s has been declared previously, but not as a sparse table.",s);
01216     error = 1;
01217     while ( *s ) s++;
01218     s++;
01219     while ( *s ) s++;
01220     s++;
01221 }
01222 else {
01223 /*
01224     Test dimension and argtail. There should be an exact match.
01225     We are not going to rename arguments when reading the elements.
01226 */
01227     ss = s;
01228     while ( *s ) s++;
01229     s++;
01230     if ( StrCmp(s,T->argtail) ) {
01231         MesPrint("&Declaration of table %s in %s different from previous
declaration",ss,d->name);
01232         error = 1;
01233     }
01234     while ( *s ) s++;
01235     s++;
01236 }
01237 }
01238 if ( command ) { M_free(command,"table command"); }
01239 return(error);
01240 }
01241
01242 /*
01243 #] CheckTableDeclarations :
01244 #[ CoTbload :
01245
01246     Loads the table stubbs of the specified tables in the indicated
01247     tablebase. Syntax:
01248     TableBase "tablebasename.tbl" load [tablename(s)];
01249     If no tables are specified all tables are taken.
01250 */
01251
01252 int CoTbload(UBYTE *ss)
01253 {
01254     DBASE *d;
01255     UBYTE *s, *name, *t, *r, *command, *arguments, *tail;
01256     LONG commandsize;
01257     int num, cs, es, ns, ts, i, j, error = 0, error1;
01258     if ( ( d = FindTB(tablebasename) ) == 0 ) {
01259         MesPrint("&No open tablebase with the name %s",tablebasename);
01260         return(-1);
01261     }
01262     commandsize = 120;
01263     command = (UBYTE *)Malloc1(commandsize,"Fill command");
01264     AC.vetofilling = 1;
01265     if ( *ss ) {
01266         while ( *ss == ',' || *ss == ' ' || *ss == '\t' ) ss++;
01267         while ( *ss ) {
01268             name = ss; ss = SkipAName(ss); *ss = 0;
01269             s = (UBYTE *) (d->tablenames);
01270             num = 0; ns = 0;
01271             while ( *s ) {
01272                 num++;
01273                 if ( StrCmp(s,name) ) {
01274                     while ( *s ) s++;
01275                     s++;
01276                     while ( *s ) s++;
01277                     s++;
01278                     num++;
01279                     continue;
01280                 }
01281                 name = s; while ( *s ) s++; ns = s-name; s++;
01282                 tail = s; while ( *s ) s++; ts = s-tail; s++;
01283                 tail++; while ( FG.cTable[*tail] == 1 ) tail++;
01284 /*

```

```

01285         Go through all elements
01286     */
01287     for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01288         for ( j = 0; j < NUMOBJECTS; j++ ) {
01289             if ( d->iblocks[i]->objects[j].tablenumber == num ) {
01290                 t = arguments = (UBYTE *) (d->iblocks[i]->objects[j].element);
01291                 while ( *t ) t++;
01292                 es = t - arguments;
01293                 cs = 2*es + 2*ns + ts + 10;
01294                 if ( cs > commandsize ) {
01295                     commandsize = 2*cs;
01296                     if ( command ) M_free(command,"Fill command");
01297                     command = (UBYTE *) Malloc1(commandsize,"Fill command");
01298                 }
01299                 r = command; t = name; while ( *t ) *r++ = *t++;
01300                 *r++ = '('; t = arguments; while ( *t ) *r++ = *t++;
01301                 *r++ = '='; *r++ = '='; *r++ = 't'; *r++ = 'b'; *r++ = 'l';
01302                 *r++ = '_'; *r++ = '('; t = name; while ( *t ) *r++ = *t++;
01303                 *r++ = ','; t = arguments; while ( *t ) *r++ = *t++;
01304                 t = tail; while ( *t ) {
01305                     if ( *t == '?' && r[-1] != ',' ) {
01306                         t++;
01307                         if ( FG.cTable[*t] == 0 || *t == '$' || *t == '[' ) {
01308                             t = SkipAName(t);
01309                             if ( *t == '[' ) {
01310                                 SKIPBRA1(t);
01311                             }
01312                         }
01313                     } else if ( *t == '{' ) {
01314                         SKIPBRA2(t);
01315                     }
01316                     else if ( *t ) { *r++ = *t++; continue; }
01317                 }
01318                 else *r++ = *t++;
01319             }
01320             *r++ = ')'; *r = 0;
01321         /*
01322         Still to do: replacemode or no replacemode?
01323     */
01324         AC.vetotablebasefill = 1;
01325         error1 = CoFill(command);
01326         AC.vetotablebasefill = 0;
01327         if ( error1 < 0 ) goto finishup;
01328         if ( error1 != 0 ) error = error1;
01329     }
01330 }
01331 }
01332 break;
01333 }
01334 while ( *ss == ',' || *ss == ' ' || *ss == '\t' ) ss++;
01335 }
01336 }
01337 else { /* do all of them */
01338     s = (UBYTE *) (d->tablenames);
01339     num = 0; ns = 0;
01340     while ( *s ) {
01341         num++;
01342         name = s; while ( *s ) s++; ns = s - name; s++;
01343         tail = s; while ( *s ) s++; ts = s - tail; s++;
01344         tail++; while ( FG.cTable[*tail] == 1 ) tail++;
01345     /*
01346     Go through all elements
01347     */
01348     for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01349         for ( j = 0; j < NUMOBJECTS; j++ ) {
01350             if ( d->iblocks[i]->objects[j].tablenumber == num ) {
01351                 t = arguments = (UBYTE *) (d->iblocks[i]->objects[j].element);
01352                 while ( *t ) t++;
01353                 es = t - arguments;
01354                 cs = 2*es + 2*ns + ts + 10;
01355                 if ( cs > commandsize ) {
01356                     commandsize = 2*cs;
01357                     if ( command ) M_free(command,"Fill command");
01358                     command = (UBYTE *) Malloc1(commandsize,"Fill command");
01359                 }
01360                 r = command; t = name; while ( *t ) *r++ = *t++;
01361                 *r++ = '('; t = arguments; while ( *t ) *r++ = *t++;
01362                 *r++ = '='; *r++ = '='; *r++ = 't'; *r++ = 'b'; *r++ = 'l';
01363                 *r++ = '_'; *r++ = '('; t = name; while ( *t ) *r++ = *t++;
01364                 *r++ = ','; t = arguments; while ( *t ) *r++ = *t++;
01365                 t = tail; while ( *t ) {
01366                     if ( *t == '?' && r[-1] != ',' ) {
01367                         t++;
01368                         if ( FG.cTable[*t] == 0 || *t == '$' || *t == '[' ) {
01369                             t = SkipAName(t);
01370                             if ( *t == '[' ) {
01371                                 SKIPBRA1(t);

```

```

01372             }
01373         }
01374         else if ( *t == '{' ) {
01375             SKIPBRA2(t);
01376         }
01377         else if ( *t ) { *r++ = *t++; continue; }
01378     }
01379     else *r++ = *t++;
01380 }
01381 *r++ = '}'; *r = 0;
01382 /*
01383     Still to do: replacemode or no replacemode?
01384 */
01385     AC.vetotablebasefill = 1;
01386     error1 = CoFill(command);
01387     AC.vetotablebasefill = 0;
01388     if ( error1 < 0 ) goto finishup;
01389     if ( error1 != 0 ) error = error1;
01390 }
01391 }
01392 }
01393 }
01394 }
01395 finishup:;
01396     AC.vetofilling = 0;
01397     if ( command ) M_free(command,"Fill command");
01398     return(error);
01399 }
01400
01401 /*
01402     #] CoTBlod :
01403     #[ TestUse :
01404
01405     Look for tbl_(tablename,arguments)
01406     if tablename is encountered, check first whether the element is in
01407     use already. If not, check in the tables in AC.usedtables.
01408     If the element is not there, add it to AC.usedtables.
01409
01410
01411     We need the arguments of TestUse to see for which tables it is to be done
01412 */
01413
01414 int TestUse(WORD *term, WORD level)
01415 {
01416     WORD *tstop, *t, *m, *tstart, tabnum;
01417     WORD *funs, numfuns;
01418     int error = 0;
01419     TABLES T;
01420     LONG i;
01421     CBUF *C = cbuf+AM.rbufnum;
01422     int isp;
01423
01424     numfuns = C->lhs[level][1] - 2;
01425     funs = C->lhs[level] + 2;
01426     GETSTOP(term,tstop);
01427     t = term+1;
01428     while ( t < tstop ) {
01429         if ( *t != TABLESTUB ) { t += t[1]; continue; }
01430         tstart = t;
01431         m = t + FUNHEAD;
01432         t += t[1];
01433         if ( *m >= -FUNCTION ) continue;
01434         tabnum = -*m;
01435         if ( ( T = functions[tabnum-FUNCTION].tbl ) == 0 ) continue;
01436         if ( T->sparse == 0 ) continue;
01437 /*
01438         Check whether we have to test this one
01439 */
01440         if ( numfuns > 0 ) {
01441             for ( i = 0; i < numfuns; i++ ) {
01442                 if ( tabnum == funs[i] ) break;
01443             }
01444             if ( i >= numfuns && numfuns > 0 ) continue;
01445         }
01446 /*
01447         Test whether the element has been defined already.
01448         If not, mark it as used.
01449         Note: we only allow sparse tables (for now)
01450 */
01451         m++;
01452         for ( i = 0; i < ABS(T->numind); i++, m += 2 ) {
01453             if ( m >= t || *m != -SNUMBER ) break;
01454         }
01455         if ( ( i == ABS(T->numind) ) &&
01456             ( ( isp = FindTableTree(T,tstart+FUNHEAD+1,2) ) >= 0 ) ) {
01457             if ( ( T->tablepointers[isp+ABS(T->numind)+4] & ELEMENTLOADED ) == 0 ) {
01458                 T->tablepointers[isp+ABS(T->numind)+4] |= ELEMENTUSED;

```

```

01459         }
01460     }
01461     else {
01462         MesPrint("TestUse: Encountered a table element inside tbl_ that does not correspond to a
tablebase element");
01463         error = -1;
01464     }
01465 }
01466 return(error);
01467 }
01468
01469 /*
01470 #] TestUse :
01471 #[ CoTBaudit :
01472 */
01473
01474 int CoTBaudit(UBYTE *s)
01475 {
01476     DBASE *d;
01477     UBYTE *name, *tail;
01478     int i, j, error = 0, num;
01479
01480     if ( ( d = FindTB(tablebasename) ) == 0 ) {
01481         MesPrint("&No open tablebase with the name %s",tablebasename);
01482         return(-1);
01483     }
01484     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
01485     while ( *s ) {
01486         /*
01487             Get the options here
01488             They will mainly involve the sorting of the output.
01489         */
01490         s++;
01491     }
01492     s = (UBYTE *) (d->tablenames); num = 0;
01493     while ( *s ) {
01494         num++;
01495         name = s; while ( *s ) s++; s++;
01496         tail = s; while ( *s ) s++; s++;
01497         MesPrint("Table,sparse,%s%s",name,tail);
01498         for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01499             for ( j = 0; j < NUMOBJECTS; j++ ) {
01500                 if ( d->iblocks[i]->objects[j].tablenumber == num ) {
01501                     MesPrint("    %s(%s)",name,d->iblocks[i]->objects[j].element);
01502                 }
01503             }
01504         }
01505     }
01506     return(error);
01507 }
01508
01509 /*
01510 #] CoTBaudit :
01511 #[ CoTBon :
01512 */
01513
01514 int CoTBon(UBYTE *s)
01515 {
01516     DBASE *d;
01517     UBYTE *ss, c;
01518     int error = 0;
01519     if ( ( d = FindTB(tablebasename) ) == 0 ) {
01520         MesPrint("&No open tablebase with the name %s",tablebasename);
01521         return(-1);
01522     }
01523     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
01524     while ( *s ) {
01525         ss = SkipAName(s);
01526         c = *ss; *ss = 0;
01527         if ( StrICmp(s,(UBYTE *) ("compress")) == 0 ) {
01528             d->mode &= ~NOCOMPRESS;
01529         }
01530         else {
01531             MesPrint("&subkey %s not defined in TableBase On statement");
01532             error = 1;
01533         }
01534         *ss = c; s = ss;
01535         while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
01536     }
01537     return(error);
01538 }
01539
01540 /*
01541 #] CoTBon :
01542 #[ CoTBoff :
01543 */
01544

```



```

01545 int CoTBoff(UBYTE *s)
01546 {
01547     DBASE *d;
01548     UBYTE *ss, c;
01549     int error = 0;
01550     if ( ( d = FindTB(tablebasename) ) == 0 ) {
01551         MesPrint("&No open tablebase with the name %s",tablebasename);
01552         return(-1);
01553     }
01554     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
01555     while ( *s ) {
01556         ss = SkipAName(s);
01557         c = *ss; *ss = 0;
01558         if ( StrICmp(s,(UBYTE *)("compress")) == 0 ) {
01559             d->mode |= NOCOMPRESS;
01560         }
01561         else {
01562             MesPrint("&subkey %s not defined in TableBase Off statement");
01563             error = 1;
01564         }
01565         *ss = c; s = ss;
01566         while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
01567     }
01568     return(error);
01569 }
01570
01571 /*
01572 #] CoTBoff :
01573 #[ CoTBcleanup :
01574 */
01575
01576 int CoTBcleanup(UBYTE *s)
01577 {
01578     DUMMYUSE(s);
01579     MesPrint("&TableBase Cleanup statement not yet implemented");
01580     return(1);
01581 }
01582
01583 /*
01584 #] CoTBcleanup :
01585 #[ CoTBreplace :
01586 */
01587
01588 int CoTBreplace(UBYTE *s)
01589 {
01590     DUMMYUSE(s);
01591     MesPrint("&TableBase Replace statement not yet implemented");
01592     return(1);
01593 }
01594
01595 /*
01596 #] CoTBreplace :
01597 #[ CoTBuse :
01598
01599 Here the actual table use as determined in TestUse causes the needed
01600 table elements to be loaded
01601 */
01602
01603 int CoTBuse(UBYTE *s)
01604 {
01605     GETIDENTITY
01606     DBASE *d;
01607     MLONG basenumber;
01608     UBYTE *arguments, *rhs, *buffer, *t, *u, c, *tablename, *p;
01609     LONG size, sum, x;
01610     int i, j, error = 0, error1 = 0, k;
01611     TABLES T = 0;
01612     WORD type, funnum, mode, *w;
01613     if ( ( d = FindTB(tablebasename) ) == 0 ) {
01614         MesPrint("&No open tablebase with the name %s",tablebasename);
01615         return(-1);
01616     }
01617     while ( *s == ',' || *s == ' ' || *s == '\t' ) s++;
01618     if ( *s ) {
01619         while ( *s ) {
01620             tablename = s;
01621             if ( ( s = SkipAName(s) ) == 0 ) return(1);
01622             c = *s; *s = 0;
01623             if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
01624                 || ( T = functions[funnum].tabl ) == 0 ) {
01625                 MesPrint("&%s should be a previously declared table",tablename);
01626                 basenumber = 0;
01627             }
01628             else if ( T->sparse == 0 ) {
01629                 MesPrint("&%s should be a sparse table",tablename);
01630                 basenumber = 0;
01631             }

```

```

01632         else { basenumber = GetTableName(d, (char *)tablename); }
01633 /*      if ( T->sparse == 0 ) { SpareTable(T); } */
01634 if ( basenumber > 0 ) {
01635     for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01636         for ( j = 0; j < NUMOBJECTS; j++ ) {
01637             if ( d->iblocks[i]->objects[j].tablenumber != basenumber ) continue;
01638             arguments = p = (UBYTE *) (d->iblocks[i]->objects[j].element);
01639 /*
01640             Now translate the arguments and see whether we need
01641             this one....
01642 */
01643             for ( k = 0, w = AT.WorkPointer; k < ABS(T->numind); k++ ) {
01644                 ParseSignedNumber(x,p);
01645                 *w++ = x; p++;
01646             }
01647             sum = FindTableTree(T,AT.WorkPointer,1);
01648             if ( sum < 0 ) {
01649                 MesPrint("Table %s in tablebase %s has not been loaded properly"
01650                     ,tablename,tablebasename);
01651                 error = 1;
01652                 continue;
01653             }
01654             sum += ABS(T->numind) + 4;
01655             mode = T->tablepointers[sum];
01656             if ( ( mode & ELEMENTLOADED ) == ELEMENTLOADED ) {
01657                 T->tablepointers[sum] &= ~ELEMENTUSED;
01658                 continue;
01659             }
01660             if ( ( mode & ELEMENTUSED ) == 0 ) continue;
01661 /*
01662             We need this one!
01663 */
01664             rhs = (UBYTE *)ReadijObject(d,i,j, (char *)arguments);
01665             if ( rhs ) {
01666                 u = rhs; while ( *u ) u++;
01667                 size = u-rhs;
01668                 u = arguments; while ( *u ) u++;
01669                 size += u-arguments;
01670                 u = tablename; while ( *u ) u++;
01671                 size += u-tablename;
01672                 size += 6;
01673                 buffer = (UBYTE *)Malloca1(size,"TableBase copy");
01674                 t = tablename; u = buffer;
01675                 while ( *t ) *u++ = *t++;
01676                 *u++ = '(';
01677                 t = arguments;
01678                 while ( *t ) *u++ = *t++;
01679                 *u++ = ')'; *u++ = '=';
01680                 t = rhs;
01681                 while ( *t ) *u++ = *t++;
01682                 if ( t == rhs ) { *u++ = '0'; }
01683                 *u++ = 0; *u = 0;
01684                 M_free(rhs,"rhs in TBuse xxx");
01685
01686                 error1 = CoFill(buffer);
01687
01688                 if ( error1 < 0 ) { return(error); }
01689                 if ( error1 != 0 ) error = error1;
01690                 M_free(buffer,"TableBase copy");
01691             }
01692             T->tablepointers[sum] &= ~ELEMENTUSED;
01693             T->tablepointers[sum] |= ELEMENTLOADED;
01694         }
01695     }
01696 }
01697 *s = c;
01698 while ( *s == ',' || *s == ' ' || *s == '\\t' ) s++;
01699 }
01700 }
01701 else {
01702     s = (UBYTE *) (d->tablenames); basenumber = 0;
01703     while ( *s ) {
01704         basenumber++;
01705         tablename = s;
01706         while ( *s ) s++;
01707         s++;
01708         while ( *s ) s++;
01709         s++;
01710         if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
01711             || ( T = functions[funnum].tabl ) == 0 ) {
01712             MesPrint("%s should be a previously declared table",tablename);
01713         }
01714         else if ( T->sparse == 0 ) {
01715             MesPrint("%s should be a sparse table",tablename);
01716         }
01717         if ( T->sparse && T->mode == 0 ) {
01718             MesPrint("In table %s we have a problem with stubb orders in CoTBuse",tablename);

```

```

01719     error = -1;
01720 }
01721 /*      if ( T->spare == 0 ) { SpareTable(T); } */
01722 for ( i = 0; i < d->info.numberofindexblocks; i++ ) {
01723     for ( j = 0; j < NUMOBJECTS; j++ ) {
01724         if ( d->iblocks[i]->objects[j].tablename == basenumber ) {
01725             arguments = p = (UBYTE *) (d->iblocks[i]->objects[j].element);
01726 /*
01727             Now translate the arguments and see whether we need
01728             this one....
01729 */
01730             for ( k = 0, w = AT.WorkPointer; k < ABS(T->numind); k++ ) {
01731                 ParseSignedNumber(x,p);
01732                 *w++ = x; p++;
01733             }
01734             sum = FindTableTree(T,AT.WorkPointer,1);
01735             if ( sum < 0 ) {
01736                 MesPrint("Table %s in tablebase %s has not been loaded properly"
01737                     ,tablename,tablebasename);
01738                 error = 1;
01739                 continue;
01740             }
01741             sum += ABS(T->numind) + 4;
01742             mode = T->tablepointers[sum];
01743             if ( ( mode & ELEMENTLOADED ) == ELEMENTLOADED ) {
01744                 T->tablepointers[sum] &= ~ELEMENTUSED;
01745                 continue;
01746             }
01747             if ( ( mode & ELEMENTUSED ) == 0 ) continue;
01748 /*
01749             We need this one!
01750 */
01751             rhs = (UBYTE *)ReadijObject(d,i,j,(char *)arguments);
01752             if ( rhs ) {
01753                 u = rhs; while ( *u ) u++;
01754                 size = u-rhs;
01755                 u = arguments; while ( *u ) u++;
01756                 size += u-arguments;
01757                 u = tablename; while ( *u ) u++;
01758                 size += u-tablename;
01759                 size += 6;
01760                 buffer = (UBYTE *)Malloc1(size,"TableBase copy");
01761                 t = tablename; u = buffer;
01762                 while ( *t ) *u++ = *t++;
01763                 *u++ = '(';
01764                 t = arguments;
01765                 while ( *t ) *u++ = *t++;
01766                 *u++ = ')'; *u++ = '=';
01767
01768                 t = rhs;
01769                 while ( *t ) *u++ = *t++;
01770                 if ( t == rhs ) { *u++ = '0'; }
01771                 *u++ = 0; *u = 0;
01772                 M_free(rhs,"rhs in TBuse");
01773
01774                 error1 = CoFill(buffer);
01775
01776                 if ( error1 < 0 ) { return(error); }
01777                 if ( error1 != 0 ) error = error1;
01778                 M_free(buffer,"TableBase copy");
01779             }
01780             T->tablepointers[sum] &= ~ELEMENTUSED;
01781             T->tablepointers[sum] |= ELEMENTLOADED;
01782         }
01783     }
01784 }
01785 }
01786 }
01787 return(error);
01788 }
01789
01790 /*
01791 #] CoTBuse :
01792 #[ CoApply :
01793
01794 Possibly to be followed by names of tables.
01795 */
01796
01797 int CoApply(UBYTE *s)
01798 {
01799     GETIDENTITY
01800     UBYTE *tablename, c;
01801     WORD type, funnum, *w;
01802     TABLES T;
01803     LONG maxtogo = MAXPOSITIVE;
01804     int error = 0;
01805     w = AT.WorkPointer;

```

```

01806     if ( FG.cTable[*s] == 1 ) {
01807         maxtogo = 0;
01808         while ( FG.cTable[*s] == 1 ) {
01809             maxtogo = maxtogo*10 + (*s-'0');
01810             s++;
01811         }
01812         while ( *s == ',' ) s++;
01813         if ( maxtogo > MAXPOSITIVE || maxtogo < 0 ) maxtogo = MAXPOSITIVE;
01814     }
01815     *w++ = TYPEAPPLY; *w++ = 3; *w++ = maxtogo;
01816     while ( *s ) {
01817         tablename = s;
01818         if ( ( s = SkipAName(s) ) == 0 ) return(1);
01819         c = *s; *s = 0;
01820         if ( ( GetVar(tablename,&type,&funnum,CFUNCTION,NOAUTO) == NAMENOTFOUND )
01821             || ( T = functions[funnum].tbl ) == 0 ) {
01822             MesPrint("%s should be a previously declared table",tablename);
01823             error = 1;
01824         }
01825         else if ( T->sparse == 0 ) {
01826             MesPrint("%s should be a sparse table",tablename);
01827             error = 1;
01828         }
01829         *w++ = funnum + FUNCTION;
01830         *s = c;
01831         while ( *s == ' ' || *s == ',' || *s == '\t' ) s++;
01832     }
01833     AT.WorkPointer[1] = w - AT.WorkPointer;
01834     /*
01835     if ( AT.WorkPointer[1] > 2 ) {
01836         AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
01837     }
01838     */
01839     AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
01840     /*
01841     AT.WorkPointer[0] = TYPEAPPLYRESET;
01842     AddNtoL(AT.WorkPointer[1],AT.WorkPointer);
01843     */
01844     return(error);
01845 }
01846
01847 /*
01848 #] CoApply :
01849 #[ CoTBhelp :
01850 */
01851
01852 char *helptb[] = {
01853     "The TableBase statement is used as follows:"
01854     ,"TableBase \"file.tbl\" keyword subkey(s)"
01855     ,"    in which we have"
01856     ,"Keyword    Subkey(s)    Action"
01857     ,"open                Opens file.tbl for R/W"
01858     ,"create              Creates file.tbl for R/W. Old contents are lost"
01859     ,"load                Loads all stubs of all tables"
01860     ,"load    tablename(s) Loads all stubs the tables mentioned"
01861     ,"enter              Loads all stubs and rhs of all tables"
01862     ,"enter    tablename(s) Loads all stubs and rhs of the tables mentioned"
01863     ,"audit             Prints list of contents"
01864     /* ,"replace tablename saves a table (with overwrite)" */
01865     /* ,"replace tableelement saves a table element (with overwrite)" */
01866     /* ,"cleanup          makes tables contingent" */
01867     ,"addto    tablename adds all elements if not yet there"
01868     ,"addto    tableelement adds element if not yet there"
01869     /* ,"delete    tablename removes table from tablebase" */
01870     /* ,"delete    tableelement removes element from tablebase" */
01871     ,"on        compress elements are stored in gzip format (default)"
01872     ,"off       compress elements are stored in uncompressed format"
01873     ,"use                compiles all needed elements"
01874     ,"use    tablename(s) compiles all needed elements of these tables"
01875     ," "
01876     ,"Related commands are:"
01877     ,"testuse          marks which tbl_ elements occur for all tables"
01878     ,"testuse tablename(s) marks which tbl_ elements occur for given tables"
01879     ,"apply            replaces tbl_ if rhs available"
01880     ,"apply    tablename(s) replaces tbl_ for given tables if rhs available"
01881     ," "
01882     };
01883
01884 int CoTBhelp(UBYTE *s)
01885 {
01886     int i, ii = sizeof(helptb)/sizeof(char *);
01887     DUMMYUSE(s);
01888     for ( i = 0; i < ii; i++ ) MesPrint("%s",helptb[i]);
01889     return(0);
01890 }
01891
01892 /*

```

```

01893     #] CoTBhelp :
01894     #[ ReWorkT :
01895
01896     Replaces the STUBBS of the functions in the list.
01897     This gains one space. Hence we have to be very careful
01898 */
01899
01900 void ReWorkT(WORD *term, WORD *funcs, WORD numfuncs)
01901 {
01902     WORD *tstop, *tend, *m, *t, *tt, *mm, *mmm, *r, *rr;
01903     int i, j;
01904     tend = term + *term; tstop = tend - ABS(tend[-1]);
01905     m = t = term+1;
01906     while ( t < tstop ) {
01907         if ( *t == TABLESTUB ) {
01908             for ( i = 0; i < numfuncs; i++ ) {
01909                 if ( -t[FUNHEAD] == funcs[i] ) break;
01910             }
01911             if ( numfuncs == 0 || i < numfuncs ) { /* Hit */
01912                 i = t[1] - 1;
01913                 *m++ = -t[FUNHEAD]; *m++ = i; t += 2; i -= FUNHEAD;
01914                 if ( m < t ) { for ( j = 0; j < FUNHEAD-2; j++ ) *m++ = *t++; }
01915                 else { m += FUNHEAD-2; t += FUNHEAD-2; }
01916                 t++;
01917                 while ( i-- > 0 ) { *m++ = *t++; }
01918                 tt = t; mm = m;
01919                 if ( mm < tt ) {
01920                     while ( tt < tend ) *mm++ = *tt++;
01921                     *term = mm - term;
01922                     tend = term + *term; tstop = tend - ABS(tend[-1]);
01923                     t = m;
01924                 }
01925             }
01926             else { goto inc; }
01927         }
01928         else if ( *t >= FUNCTION ) {
01929             tt = t + t[1];
01930             mm = m;
01931             for ( j = 0; j < FUNHEAD; j++ ) {
01932                 if ( m == t ) { m++; t++; }
01933                 else *m++ = *t++;
01934             }
01935             while ( t < tt ) {
01936                 if ( *t <= -FUNCTION ) {
01937                     if ( m == t ) { m++; t++; }
01938                     else *m++ = *t++;
01939                 }
01940                 else if ( *t < 0 ) {
01941                     if ( m == t ) { m += 2; t += 2; }
01942                     else { *m++ = *t++; *m++ = *t++; }
01943                 }
01944                 else {
01945                     rr = t + *t; mmm = m;
01946                     for ( j = 0; j < ARGHEAD; j++ ) {
01947                         if ( m == t ) { m++; t++; }
01948                         else *m++ = *t++;
01949                     }
01950                     while ( t < rr ) {
01951                         r = t + *t;
01952                         ReWorkT(t, funcs, numfuncs);
01953                         j = *t;
01954                         if ( m == t ) { m += j; t += j; }
01955                         else { while ( j-- >= 0 ) *m++ = *t++; }
01956                         t = r;
01957                     }
01958                     *mmm = m - mmm;
01959                 }
01960             }
01961             mm[1] = m - mm;
01962             t = tt;
01963         }
01964         else {
01965             inc:
01966                 j = t[1];
01967                 if ( m < t ) { while ( j-- >= 0 ) *m++ = *t++; }
01968                 else { m += j; t += j; }
01969             }
01970         if ( m < t ) {
01971             while ( t < tend ) *m++ = *t++;
01972             *term = m - term;
01973         }
01974     }
01975
01976 /*
01977     #] ReWorkT :
01978     #[ Apply :
01979 */

```

```

01980
01981 void Apply(WORD *term, WORD level)
01982 {
01983     WORD *funcs, numfuncs;
01984     TABLES T;
01985     int i, j;
01986     CBUF *C = cbuf+AM.rbufnum;
01987     /*
01988     Point the tables in the proper direction
01989     */
01990     numfuncs = C->lhs[level][1] - 2;
01991     funcs = C->lhs[level] + 2;
01992     if ( numfuncs > 0 ) {
01993         for ( i = MAXBUILTINFUNCTION-FUNCTION; i < AC.FunctionList.num; i++ ) {
01994             if ( ( T = functions[i].tabl ) != 0 ) {
01995                 for ( j = 0; j < numfuncs; j++ ) {
01996                     if ( i == (funcs[j]-FUNCTION) && T->spare ) {
01997                         FlipTable(&(functions[i]),0);
01998                         break;
01999                     }
02000                 }
02001             }
02002         }
02003     }
02004     else {
02005         for ( i = MAXBUILTINFUNCTION-FUNCTION; i < AC.FunctionList.num; i++ ) {
02006             if ( ( T = functions[i].tabl ) != 0 ) {
02007                 if ( T->spare ) FlipTable(&(functions[i]),0);
02008             }
02009         }
02010     }
02011     /*
02012     Now the replacements everywhere of
02013     id tbl_(table,?a) = table(?a);
02014     Actually, this has to be done recursively.
02015     Note that we actually gain one space.
02016     */
02017     ReWorkT(term, funcs, numfuncs);
02018 }
02019
02020 /*
02021 #] Apply :
02022 #[ ApplyExec :
02023
02024 Replaces occurrences of tbl_(table,indices,pattern) by the proper
02025 rhs of table(indices,pattern). It does this up to maxtogo times
02026 in the given term. It starts with the occurrences inside the
02027 arguments of functions. If necessary it finishes at groundlevel.
02028 An infinite number of tries is indicated by maxtogo = 2^15-1 or 2^31-1.
02029 The occurrences are replaced by subexpressions. This allows TestSub
02030 to finish the job properly.
02031
02032 The main trick here is T = T->spare which turns to the proper rhs.
02033
02034 The return value is the number of substitutions that can still be made
02035 based on maxtogo. Hence, if the returnvalue is different from maxtogo
02036 there was a substitution.
02037 */
02038
02039 int ApplyExec(WORD *term, int maxtogo, WORD level)
02040 {
02041     GETIDENTITY
02042     WORD rhsnumber, *Tpattern, *funcs, numfuncs, funnum;
02043     WORD ii, *t, *tl, *w, *p, *m, *ml, *u, *r, tbufnum, csize, wilds;
02044     NESTING NN;
02045     int i, j, isp, stilltogo;
02046     CBUF *C;
02047     TABLES T;
02048     /*
02049     Startup. We need NestPoin for when we have to replace something deep down.
02050     */
02051     t = term;
02052     m = t + *t;
02053     csize = ABS(m[-1]);
02054     m -= csize;
02055     AT.NestPoin->termsize = t;
02056     if ( AT.NestPoin == AT.Nest ) AN.EndNest = t + *t;
02057     t++;
02058     /*
02059     First we look inside function arguments. Also when clean!
02060     */
02061     while ( t < m ) {
02062         if ( *t < FUNCTION ) { t += t[1]; continue; }
02063         if ( functions[*t-FUNCTION].spec > 0 ) { t += t[1]; continue; }
02064         AT.NestPoin->funsize = t;
02065         r = t + t[1];
02066         t += FUNHEAD;

```

```

02067     while ( t < r ) {
02068         if ( *t < 0 ) { NEXTARG(t); continue; }
02069         AT.NestPoin->argsize = t1 = t;
02070         u = t + *t;
02071         t += ARGHEAD;
02072         AT.NestPoin++;
02073         while ( t < u ) {
02074             /*
02075                 Now we loop over the terms inside a function argument
02076                 This defines a recursion and we have to call ApplyExec again.
02077                 The real problem is when we catch something and we have
02078                 to insert a subexpression pointer. This may use more or
02079                 less space and the whole term has to be readjusted.
02080                 This is why we have the NestPoin variables. They tell us
02081                 where the sizes of the term, the function and the arguments
02082                 are sitting, and also where the dirty flags are.
02083                 This readjusting is of course done in the groundlevel code.
02084                 Here we worry about the maxtogo count.
02085             */
02086             stilltogo = ApplyExec(t,maxtogo,level);
02087             if ( stilltogo != maxtogo ) {
02088                 if ( stilltogo <= 0 ) {
02089                     AT.NestPoin--;
02090                     return(stilltogo);
02091                 }
02092                 maxtogo = stilltogo;
02093                 u = t1 + *t1;
02094                 m = term + *term - csize;
02095             }
02096             t += *t;
02097         }
02098         AT.NestPoin--;
02099     }
02100 }
02101 /*
02102     Now we look at the ground level
02103 */
02104 C = cbuf+AM.rbufnum;
02105 t = term + 1;
02106 while ( t < m ) {
02107     if ( *t != TABLESTUB ) { t += t[1]; continue; }
02108     funnum = -t[FUNHEAD];
02109     if ( ( funnum < FUNCTION )
02110         || ( funnum >= FUNCTION+WILDOffset )
02111         || ( ( T = functions[funnum-FUNCTION].tabl ) == 0 )
02112         || ( T->sparse == 0 )
02113         || ( T->spare == 0 ) ) { t += t[1]; continue; }
02114     numfuns = C->lhs[level][1] - 3;
02115     funs = C->lhs[level] + 3;
02116     if ( numfuns > 0 ) {
02117         for ( i = 0; i < numfuns; i++ ) {
02118             if ( funs[i] == funnum ) break;
02119         }
02120         if ( i >= numfuns ) { t += t[1]; continue; }
02121     }
02122     r = t + t[1];
02123     AT.NestPoin->funsize = t + 1;
02124     t1 = t;
02125     t += FUNHEAD + 1;
02126     /*
02127         Test whether the table catches
02128         Test 1: index arguments and range. isp will be the number
02129         of the element in the table.
02130     */
02131     T = T->sparse;
02132     #ifdef WITHPTHREADS
02133     Tpattern = T->pattern[identity];
02134     #else
02135     Tpattern = T->pattern;
02136     #endif
02137     p = Tpattern+FUNHEAD+1;
02138     for ( i = 0; i < ABS(T->numind); i++, t += 2 ) {
02139         if ( *t != -SNUMBER ) break;
02140     }
02141     if ( i < ABS(T->numind) ) { t = r; continue; }
02142     isp = FindTableTree(T,t1+FUNHEAD+1,2);
02143     if ( isp < 0 ) { t = r; continue; }
02144     rhsnumber = T->tablepointers[isp+ABS(T->numind)];
02145     #if ( TABLEEXTENSION == 2 )
02146     tbufnum = T->bunum;
02147     #else
02148     tbufnum = T->tablepointers[isp+ABS(T->numind)+1];
02149     #endif
02150     t = t1+FUNHEAD+2;
02151     ii = ABS(T->numind);
02152     while ( --ii >= 0 ) {
02153         *p = *t; t += 2; p += 2;

```

```

02154     }
02155     /*
02156         If there are more arguments we have to do some
02157         pattern matching. This should be easy. We adapted the
02158         pattern, so that the array indices match already.
02159     */
02160     #ifdef WITHPTHREADS
02161         AN.FullProto = T->prototype[identity];
02162     #else
02163         AN.FullProto = T->prototype;
02164     #endif
02165     AN.WildValue = AN.FullProto + SUBEXPSIZE;
02166     AN.WildStop = AN.FullProto+AN.FullProto[1];
02167     ClearWild(BHEAD0);
02168     AN.RepFunNum = 0;
02169     AN.RepFunList = AN.EndNest;
02170     AT.WorkPointer = (WORD *)(((UBYTE *) (AN.EndNest)) + AM.MaxTer/2);
02171     /*
02172         The RepFunList is after the term but not very relevant.
02173         We need because MatchFunction uses it
02174     */
02175     if ( AT.WorkPointer + t1[1] >= AT.WorkTop ) { MesWork(); }
02176     wilds = 0;
02177     w = AT.WorkPointer;
02178     *w++ = -t1[FUNHEAD];
02179     *w++ = t1[1] - 1;
02180     for ( i = 2; i < FUNHEAD; i++ ) *w++ = t1[i];
02181     t = t1 + FUNHEAD+1;
02182     while ( t < r ) *w++ = *t++;
02183     t = AT.WorkPointer;
02184     AT.WorkPointer = w;
02185     if ( MatchFunction(BHEAD Tpattern,t,&wilds) > 0 ) {
02186     /*
02187         Here we caught one. Now we should worry about:
02188         1: inserting the subexpression pointer with its wildcards
02189         2: NestPoin because we may not be at the lowest level
02190         The function starts at t1.
02191     */
02192     #ifdef WITHPTHREADS
02193         m1 = T->prototype[identity];
02194     #else
02195         m1 = T->prototype;
02196     #endif
02197     m1[2] = rhsnumber;
02198     m1[4] = tbufnum;
02199     t = t1;
02200     j = t[1];
02201     i = m1[1];
02202     if ( j > i ) {
02203         j = i - j;
02204         NCOPY(t,m1,i);
02205         m1 = AN.EndNest;
02206         while ( r < m1 ) *t++ = *r++;
02207         AN.EndNest = t;
02208         *term += j;
02209         NN = AT.NestPoin;
02210         while ( NN > AT.Nest ) {
02211             NN--;
02212             NN->termsize[0] += j;
02213             NN->funsize[1] += j;
02214             NN->argsize[0] += j;
02215             NN->funsize[2] |= DIRTYFLAG;
02216             NN->argsize[1] |= DIRTYFLAG;
02217         }
02218         m += j;
02219     }
02220     else if ( j < i ) {
02221         j = i-j;
02222         t = AN.EndNest;
02223         while ( t >= r ) { t[j] = *t; t--; }
02224         t = t1;
02225         NCOPY(t,m1,i);
02226         AN.EndNest += j;
02227         *term += j;
02228         NN = AT.NestPoin;
02229         while ( NN > AT.Nest ) {
02230             NN--;
02231             NN->termsize[0] += j;
02232             NN->funsize[1] += j;
02233             NN->argsize[0] += j;
02234             NN->funsize[2] |= DIRTYFLAG;
02235             NN->argsize[1] |= DIRTYFLAG;
02236         }
02237         m += j;
02238     }
02239     else {
02240         NCOPY(t,m1,j);

```



```

02241         }
02242         r = t1 + t1[1];
02243         maxtogo--;
02244         if ( maxtogo <= 0 ) return(maxtogo);
02245     }
02246     t = r;
02247 }
02248 return(maxtogo);
02249 }
02250
02251 /*
02252 #] ApplyExec :
02253 #[ ApplyReset :
02254 */
02255
02256 void ApplyReset(WORD level)
02257 {
02258     WORD *funcs, numfuncs;
02259     TABLES T;
02260     int i, j;
02261     CBUF *C = cbuf+AM.rbufnum;
02262
02263     numfuncs = C->lhs[level][1] - 2;
02264     funcs = C->lhs[level] + 2;
02265     if ( numfuncs > 0 ) {
02266         for ( i = MAXBUILTINFUNCTION-FUNCTION; i < AC.FunctionList.num; i++ ) {
02267             if ( ( T = functions[i].tabl ) != 0 ) {
02268                 for ( j = 0; j < numfuncs; j++ ) {
02269                     if ( i == (funcs[j]-FUNCTION) && T->spare ) {
02270                         FlipTable(&(functions[i]),1);
02271                         break;
02272                     }
02273                 }
02274             }
02275         }
02276     }
02277     else {
02278         for ( i = MAXBUILTINFUNCTION-FUNCTION; i < AC.FunctionList.num; i++ ) {
02279             if ( ( T = functions[i].tabl ) != 0 ) {
02280                 if ( T->spare ) FlipTable(&(functions[i]),1);
02281             }
02282         }
02283     }
02284 }
02285
02286 /*
02287 #] ApplyReset :
02288 #[ TableReset :
02289 */
02290
02291 void TableReset(void)
02292 {
02293     TABLES T;
02294     int i;
02295
02296     for ( i = MAXBUILTINFUNCTION-FUNCTION; i < AC.FunctionList.num; i++ ) {
02297         if ( ( T = functions[i].tabl ) != 0 && T->spare && T->mode == 0 ) {
02298             functions[i].tabl = T->spare;
02299         }
02300     }
02301 }
02302
02303 /*
02304 #] TableReset :
02305 #[ LoadTableElement :
02306 ?????
02307 int LoadTableElement(DBASE *d, TABLE *T, WORD num)
02308 {
02309 }
02310
02311 #] LoadTableElement :
02312 #[ ReleaseTB :
02313
02314 Releases all TableBases
02315 */
02316
02317 int ReleaseTB(void)
02318 {
02319     DBASE *d;
02320     int i;
02321     for ( i = NumTableBases - 1; i >= 0; i-- ) {
02322         d = tablebases+i;
02323         fclose(d->handle);
02324         FreeTableBase(d);
02325     }
02326     return(0);
02327 }

```

```

02328
02329 /*
02330      #] ReleaseTB :
02331 */

```

4.143 threads.c File Reference

4.143.1 Detailed Description

Routines for the interface of FORM with the pthreads library

This is the main part of the parallelization of TFORM. It is important to also look in the files [structs.h](#) and [variable.h](#) because the treatment of the A and B structs is essential (these structs are used by means of the macros AM, AP, AC, AS, AR, AT, AN, AO and AX). Also the definitions and use of the macros BHEAD and PHEAD should be looked up.

The sources are set up in such a way that if WITHPTHREADS isn't defined there is no trace of pthread parallelization. The reason is that TFORM is far more memory hungry than sequential FORM.

Special attention should also go to the locks. The proper use of the locks is essential and determines whether TFORM can work at all. We use the LOCK/UNLOCK macros which are empty in the case of sequential FORM. These locks are at many places in the source files when workers can interfere with each others data or with the data of the master.

Definition in file [threads.c](#).

4.144 threads.c

[Go to the documentation of this file.](#)

```

00001
00022 /* #] License : */
00023 /*
00024 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00025 *   When using this file you are requested to refer to the publication
00026 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00027 *   This is considered a matter of courtesy as the development was paid
00028 *   for by FOM the Dutch physics granting agency and we would like to
00029 *   be able to track its scientific use to convince FOM of its value
00030 *   for the community.
00031 *
00032 *   This file is part of FORM.
00033 *
00034 *   FORM is free software: you can redistribute it and/or modify it under the
00035 *   terms of the GNU General Public License as published by the Free Software
00036 *   Foundation, either version 3 of the License, or (at your option) any later
00037 *   version.
00038 *
00039 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00040 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00041 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00042 *   details.
00043 *
00044 *   You should have received a copy of the GNU General Public License along
00045 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00046 */
00047 /* #] License : */
00048
00049 #ifdef WITHPTHREADS
00050
00051 #define WHOLEBRACKETS
00052 /*
00053 *   #] Variables :
00054 *
00055 *   The sortbot additions are from 17-may-2007 and after. They constitute
00056 *   an attempt to make the final merge sorting faster for the master.
00057 *   This way the master has only one compare per term.

```

```

00058     It does add some complexity, but the final merge routine (MasterMerge)
00059     is much simpler for the sortbots. On the other hand the original merging is
00060     for a large part a copy of the MergePatches routine in sort.c and hence
00061     even though complex the bad part has been thoroughly debugged.
00062 */
00063
00064 #include "form3.h"
00065
00066 #ifdef WITH_ALARM
00067 // This is only required if we are blocking SIG_ALARM in the worker threads.
00068 #include <signal.h>
00069 #endif
00070
00071 #ifdef WITHFLOAT
00072 #include <gmp.h>
00073
00074 int PackFloat(WORD *,mpf_t);
00075 int UnpackFloat(mpf_t, WORD *);
00076 void RatToFloat(mpf_t result, UWORD *format, int ratsize);
00077 #endif
00078
00079 static int numberofthreads;
00080 static int numberofworkers;
00081 static int identityofthreads = 0;
00082 static int *listofavailables;
00083 static int topofavailables = 0;
00084 static pthread_key_t identitykey;
00085 static INILOCK(numberofthreadslock)
00086 static INILOCK(availabilitylock)
00087 static pthread_t *threadpointers = 0;
00088 static pthread_mutex_t *wakeuplocks;
00089 static pthread_mutex_t *wakeupmasterthreadlocks;
00090 static pthread_cond_t *wakeupconditions;
00091 static pthread_condattr_t *wakeupconditionattributes;
00092 static int *wakeup;
00093 static int *wakeupmasterthread;
00094 static INILOCK(wakeupmasterlock)
00095 static pthread_cond_t wakeupmasterconditions = PTHREAD_COND_INITIALIZER;
00096 static pthread_cond_t *wakeupmasterthreadconditions;
00097 static int wakeupmaster = 0;
00098 static int identityretval;
00099 /* static INILOCK(clearclocklock) */
00100 static LONG *timerinfo;
00101 static LONG *sumtimerinfo;
00102 static int numberclaimed;
00103
00104 static THREADBUCKET **threadbuckets, **freebuckets;
00105 static int numthreadbuckets;
00106 static int numberoffullbuckets;
00107
00108 /* static int numberbusy = 0; */
00109
00110 INILOCK(dummylock)
00111 INIRWLOCK(dummyrwlock)
00112 static pthread_cond_t dummywakeupcondition = PTHREAD_COND_INITIALIZER;
00113
00114 #ifdef WITHSORTBOTS
00115 static POSITION SortBotPosition;
00116 static int numberofsortbots;
00117 static INILOCK(wakeupsortbotlock)
00118 static pthread_cond_t wakeupsortbotconditions = PTHREAD_COND_INITIALIZER;
00119 static int topsortbotavailables = 0;
00120 static LONG numberofterms;
00121 #endif
00122
00123 /*
00124     #] Variables :
00125     #[ Identity :
00126         #[ StartIdentity :
00127 */
00128 void StartIdentity(void)
00129 {
00130     pthread_key_create(&identitykey,FinishIdentity);
00131 }
00132
00133 /*
00134     #] StartIdentity :
00135     #[ FinishIdentity :
00136 */
00137 void FinishIdentity(void *keyp)
00138 {
00139     DUMMYUSE(keyp);
00140     free(keyp);
00141 }
00142
00143 /*
00144     #] FinishIdentity :

```

```

00154         #] SetIdentity :
00155     */
00160 int SetIdentity(int *identityretval)
00161 {
00162     /*
00163     #ifdef _MSC_VER
00164         printf("addr %d\n",&numberofthreadslock);
00165         printf("size %d\n",sizeof(numberofthreadslock));
00166     #endif
00167     */
00168     LOCK(numberofthreadslock);
00169     *identityretval = identityofthreads++;
00170     UNLOCK(numberofthreadslock);
00171     pthread_setspecific(identitykey, (void *)identityretval);
00172     return(*identityretval);
00173 }
00174
00175 /*
00176     #] SetIdentity :
00177     #] WhoAmI :
00178 */
00179
00190 int WhoAmI(void)
00191 {
00192     int *identity;
00193     /*
00194     First a fast exit for when there is at most one thread
00195     */
00196     if ( identityofthreads <= 1 ) return(0);
00197     /*
00198     Now the reading of the key.
00199     According to the book the statement should read:
00200
00201     pthread_getspecific(identitykey, (void **)(&identity));
00202
00203     but according to the information in pthread.h it is:
00204     */
00205     identity = (int *)pthread_getspecific(identitykey);
00206     return(*identity);
00207 }
00208
00209 /*
00210     #] WhoAmI :
00211     #] BeginIdentities :
00212 */
00213
00218 void BeginIdentities(void)
00219 {
00220     StartIdentity();
00221     SetIdentity(&identityretval);
00222 }
00223
00224 /*
00225     #] BeginIdentities :
00226     #] Identity :
00227     #] StartHandleLock :
00228 */
00235 void StartHandleLock(void)
00236 {
00237     AM.handlelock = dummyrwlock;
00238 }
00239
00240 /*
00241     #] StartHandleLock :
00242     #] StartAllThreads :
00243 */
00261 int StartAllThreads(int number)
00262 {
00263     int identity, j, dummy, mul;
00264     ALLPRIVATES *B;
00265     pthread_t thethread;
00266     identity = WhoAmI();
00267
00268     #ifdef WITHSORTBOTS
00269     timerinfo = (LONG *)Malloca(sizeof(LONG)*number*2,"timerinfo");
00270     sumtimerinfo = (LONG *)Malloca(sizeof(LONG)*number*2,"sumtimerinfo");
00271     for ( j = 0; j < number*2; j++ ) { timerinfo[j] = 0; sumtimerinfo[j] = 0; }
00272     mul = 2;
00273     #else
00274     timerinfo = (LONG *)Malloca(sizeof(LONG)*number,"timerinfo");
00275     sumtimerinfo = (LONG *)Malloca(sizeof(LONG)*number,"sumtimerinfo");
00276     for ( j = 0; j < number; j++ ) { timerinfo[j] = 0; sumtimerinfo[j] = 0; }
00277     mul = 1;
00278     #endif
00279
00280     listofavailables = (int *)Malloca(sizeof(int)*(number+1),"listofavailables");
00281     threadpointers = (pthread_t *)Malloca(sizeof(pthread_t)*number*mul,"threadpointers");
00282     AB = (ALLPRIVATES **)Malloca(sizeof(ALLPRIVATES *)*number*mul,"Private structs");

```

```

00283
00284     wakeup = (int *)Malloc1(sizeof(int)*number*mul,"wakeup");
00285     wakeuplocks = (pthread_mutex_t *)Malloc1(sizeof(pthread_mutex_t)*number*mul,"wakeuplocks");
00286     wakeupconditions = (pthread_cond_t
*)Malloc1(sizeof(pthread_cond_t)*number*mul,"wakeupconditions");
00287     wakeupconditionattributes = (pthread_condattr_t *)
00288         Malloc1(sizeof(pthread_condattr_t)*number*mul,"wakeupconditionattributes");
00289
00290     wakeupmasterthread = (int *)Malloc1(sizeof(int)*number*mul,"wakeupmasterthread");
00291     wakeupmasterthreadlocks = (pthread_mutex_t
*)Malloc1(sizeof(pthread_mutex_t)*number*mul,"wakeupmasterthreadlocks");
00292     wakeupmasterthreadconditions = (pthread_cond_t
*)Malloc1(sizeof(pthread_cond_t)*number*mul,"wakeupmasterthread");
00293
00294     numberofthreads = number;
00295     numberofworkers = number - 1;
00296     threadpointers[identity] = pthread_self();
00297     topofavailables = 0;
00298
00299 #ifdef WITH_ALARM
00300     /* During thread creation, we block SIGALRM on the main thread. The created
00301        threads will inherit this. This is required for #timeout to work properly
00302        in TFORM: only the main thread should receive SIGALRM. */
00303     sigset_t sig_set;
00304     sigemptyset(&sig_set);
00305     sigaddset(&sig_set, SIGALRM);
00306     pthread_sigmask(SIG_BLOCK, &sig_set, NULL);
00307 #endif
00308
00309     for ( j = 1; j < number; j++ ) {
00310         if ( pthread_create(&thethread,NULL,RunThread,(void *)(&dummy)) )
00311             goto failure;
00312     }
00313     /*
00314     Now we initialize the master at the same time that the workers are doing so.
00315     */
00316     B = InitializeOneThread(identity);
00317     AR.infile = &(AR.Fscr[0]);
00318     AR.outfile = &(AR.Fscr[1]);
00319     AR.hidefile = &(AR.Fscr[2]);
00320     AM.sbuflock = dummylock;
00321     AS.inputslock = dummylock;
00322     AS.outputslock = dummylock;
00323     AS.MaxExprSizeLock = dummylock;
00324     AP.PreVarLock = dummylock;
00325     AC.halfmodlock = dummylock;
00326     MakeThreadBuckets(number,0);
00327     /*
00328     Now we wait for the workers to finish their startup.
00329     We don't want to initialize the sortbots yet and run the risk that
00330     some of them may end up with a lower number than one of the workers.
00331     */
00332     MasterWaitAll();
00333 #ifdef WITHSORTBOTS
00334     if ( numberofworkers > 2 ) {
00335         numberofsortbots = numberofworkers-2;
00336         for ( j = numberofworkers+1; j < 2*numberofworkers-1; j++ ) {
00337             if ( pthread_create(&thethread,NULL,RunSortBot,(void *)(&dummy)) )
00338                 goto failure;
00339         }
00340     }
00341     else {
00342         numberofsortbots = 0;
00343     }
00344     MasterWaitAllSortBots();
00345     DefineSortBotTree();
00346 #endif
00347     IniSortBlocks(number-1);
00348     AS.MasterSort = 0;
00349     AM.storefilelock = dummylock;
00350
00351 #ifdef WITH_ALARM
00352     /* Now we allow the main thread to receive SIGALRM again. */
00353     pthread_sigmask(SIG_UNBLOCK, &sig_set, NULL);
00354 #endif
00355
00356     /*
00357     MesPrint("AB = %x %x %x  %d",AB[0],AB[1],AB[2], identityofthreads);
00358     */
00359     return(0);
00360 failure:
00361     MesPrint("Cannot start %d threads",number);
00362     Terminate(-1);
00363     return(-1);
00364 }
00365
00366 /*

```

```

00367     #] StartAllThreads :
00368     #[ InitializeOneThread :
00369     */
00373     UBYTE *scratchname[] = { (UBYTE *)"scratchsize",
00374                             (UBYTE *)"scratchsize",
00375                             (UBYTE *)"hidesize" };
00402     ALLPRIVATES *InitializeOneThread(int identity)
00403     {
00404         WORD *t, *ScratchBuf;
00405         int i, j, bsize, *bp;
00406         LONG ScratchSize[3], IOSize;
00407         ALLPRIVATES *B;
00408         UBYTE *s;
00409
00410         wakeup[identity] = 0;
00411         wakeuplocks[identity] = dummylock;
00412         pthread_condattr_init(&(wakeupconditionattributes[identity]));
00413         pthread_condattr_setpshared(&(wakeupconditionattributes[identity]),PTHREAD_PROCESS_PRIVATE);
00414         wakeupconditions[identity] = dummywakeupcondition;
00415         pthread_cond_init(&(wakeupconditions[identity]),&(wakeupconditionattributes[identity]));
00416         wakeupmasterthread[identity] = 0;
00417         wakeupmasterthreadlocks[identity] = dummylock;
00418         wakeupmasterthreadconditions[identity] = dummywakeupcondition;
00419
00420         bsize = sizeof(ALLPRIVATES);
00421         bsize = (bsize+sizeof(int)-1)/sizeof(int);
00422         B = (ALLPRIVATES *)Malloc1(sizeof(int)*bsize,"B struct");
00423         for ( bp = (int *)B, j = 0; j < bsize; j++ ) *bp++ = 0;
00424
00425         AB[identity] = B;
00426     /*
00427         12-jun-2007 JV:
00428         For the timing one has to know a few things:
00429         The POSIX standard is that there is only a single process ID and that
00430         getrusage returns the time of all the threads together.
00431         Under Linux there are two methods though: The older LinuxThreads and NPTL.
00432         LinuxThreads gives each thread its own process id. This makes that we
00433         can time the threads with getrusage, and hence this was done. Under NPTL
00434         this has been 'corrected' and suddenly getrusage doesn't work anymore the
00435         way it used to. Now we need
00436         clock_gettime(CLOCK_THREAD_CPUTIME_ID,&timing)
00437         which is declared in <time.h> and we need -lrt extra in the link statement.
00438         (this is at least the case on blade02 at DESY-Zeuthen).
00439         See also the code in tools.c at the routine Timer.
00440         We may still have to include more stuff there.
00441     */
00442     if ( identity > 0 ) TimeCPU(0);
00443
00444     #ifdef WITHSORTBOTS
00445
00446     if ( identity > numberofworkers ) {
00447     /*
00448         Some workspace is needed when we have a PolyFun and we have to add
00449         two terms and the new result is going to be longer than the old result.
00450     */
00451         LONG length = AM.WorkSize*sizeof(WORD)/8+AM.MaxTer*2;
00452         AT.WorkSpace = (WORD *)Malloc1(length,"WorkSpace");
00453         AT.WorkTop = AT.WorkSpace + length/sizeof(WORD);
00454         AT.WorkPointer = AT.WorkSpace;
00455         AT.identity = identity;
00456     /*
00457         The SB struct gets treated in IniSortBlocks.
00458         The SortBotIn variables will be defined DefineSortBotTree.
00459     */
00460         if ( AN.SoScratC == 0 ) {
00461             AN.SoScratC = (UWORD *)Malloc1(2*(AM.MaxTal+2)*sizeof(UWORD),"Scratch in SortBot");
00462         }
00463         AT.SS = (SORTING *)Malloc1(sizeof(SORTING),"dummy sort buffer");
00464         AT.SS->PolyFlag = 0;
00465
00466         AT.comsym[0] = 8;
00467         AT.comsym[1] = SYMBOL;
00468         AT.comsym[2] = 4;
00469         AT.comsym[3] = 0;
00470         AT.comsym[4] = 1;
00471         AT.comsym[5] = 1;
00472         AT.comsym[6] = 1;
00473         AT.comsym[7] = 3;
00474         AT.comnum[0] = 4;
00475         AT.comnum[1] = 1;
00476         AT.comnum[2] = 1;
00477         AT.comnum[3] = 3;
00478         AT.comfun[0] = FUNHEAD+4;
00479         AT.comfun[1] = FUNCTION;
00480         AT.comfun[2] = FUNHEAD;
00481         AT.comfun[3] = 0;
00482     #if FUNHEAD > 3

```

```

00483         for ( i = 4; i <= FUNHEAD; i++ )
00484             AT.comfun[i] = 0;
00485     #endif
00486     AT.comfun[FUNHEAD+1] = 1;
00487     AT.comfun[FUNHEAD+2] = 1;
00488     AT.comfun[FUNHEAD+3] = 3;
00489     AT.comind[0] = 7;
00490     AT.comind[1] = INDEX;
00491     AT.comind[2] = 3;
00492     AT.comind[3] = 0;
00493     AT.comind[4] = 1;
00494     AT.comind[5] = 1;
00495     AT.comind[6] = 3;
00496
00497     AT.inprimelist = -1;
00498     AT.sizeprimelist = 0;
00499     AT.primelist = 0;
00500     AT.bracketinfo = 0;
00501
00502     AR.CompareRoutine = (COMPAREDUMMY) (&Compare1);
00503
00504     AR.sLevel = 0;
00505     AR.wranfia = 0;
00506     AR.wranfcall = 0;
00507     AR.wranfnpair1 = NPAIR1;
00508     AR.wranfnpair2 = NPAIR2;
00509     AN.NumFunSorts = 5;
00510     AN.MaxFunSorts = 5;
00511     AN.SplitScratch = 0;
00512     AN.SplitScratchSize = AN.InScratch = 0;
00513     AN.SplitScratch1 = 0;
00514     AN.SplitScratchSize1 = AN.InScratch1 = 0;
00515
00516     AN.FunSorts = (SORTING **)Malloc1((AN.NumFunSorts+1)*sizeof(SORTING *),"FunSort pointers");
00517     for ( i = 0; i <= AN.NumFunSorts; i++ ) AN.FunSorts[i] = 0;
00518     AN.FunSorts[0] = AT.S0 = AT.SS;
00519     AN.idfunctionflag = 0;
00520     AN.tryterm = 0;
00521
00522     return(B);
00523 }
00524 if ( identity == 0 && AN.SoScratC == 0 ) {
00525     AN.SoScratC = (UWORD *)Malloc1(2*(AM.MaxTal+2)*sizeof(UWORD),"Scratch in SortBot");
00526 }
00527 #endif
00528 AR.CurDum = AM.IndDum;
00529 for ( j = 0; j < 3; j++ ) {
00530     if ( identity == 0 ) {
00531         if ( j == 2 ) {
00532             ScratchSize[j] = AM.HideSize;
00533         }
00534         else {
00535             ScratchSize[j] = AM.ScratSize;
00536         }
00537         if ( ScratchSize[j] < 10*AM.MaxTer ) ScratchSize[j] = 10 * AM.MaxTer;
00538     }
00539     else {
00540         /*
00541             ScratchSize[j] = AM.ScratSize / (numberofthreads-1);
00542             ScratchSize[j] = ScratchSize[j] / 20;
00543             if ( ScratchSize[j] < 10*AM.MaxTer ) ScratchSize[j] = 10 * AM.MaxTer;
00544         */
00545         if ( j == 1 ) ScratchSize[j] = AM.ThreadScratOutSize;
00546         else ScratchSize[j] = AM.ThreadScratSize;
00547         if ( ScratchSize[j] < 4*AM.MaxTer ) ScratchSize[j] = 4 * AM.MaxTer;
00548         AR.Fscr[j].name = 0;
00549     }
00550     ScratchSize[j] = ( ScratchSize[j] + 255 ) / 256;
00551     ScratchSize[j] = ScratchSize[j] * 256;
00552     ScratchBuf = (WORD *)Malloc1(ScratchSize[j]*sizeof(WORD),(char *) (scratchname[j]));
00553     AR.Fscr[j].POsize = ScratchSize[j] * sizeof(WORD);
00554     AR.Fscr[j].POfull = AR.Fscr[j].POfill = AR.Fscr[j].PObuffer = ScratchBuf;
00555     AR.Fscr[j].POstop = AR.Fscr[j].PObuffer + ScratchSize[j];
00556     PUTZERO(AR.Fscr[j].POposition);
00557     AR.Fscr[j].pthreadlock = dummylock;
00558     AR.Fscr[j].wPOsize = AR.Fscr[j].POsize;
00559     AR.Fscr[j].wPObuffer = AR.Fscr[j].PObuffer;
00560     AR.Fscr[j].wPOfill = AR.Fscr[j].POfill;
00561     AR.Fscr[j].wPOfull = AR.Fscr[j].POfull;
00562     AR.Fscr[j].wPOstop = AR.Fscr[j].POstop;
00563 }
00564 AR.InInBuf = 0;
00565 AR.InHiBuf = 0;
00566 AR.Fscr[0].handle = -1;
00567 AR.Fscr[1].handle = -1;
00568 AR.Fscr[2].handle = -1;
00569 AR.FoStage4[0].handle = -1;

```

```

00570     AR.FoStage4[1].handle = -1;
00571     IOsize = AM.S0->file.POSize;
00572 #ifdef WITHZLIB
00573     AR.FoStage4[0].ziosize = IOsize;
00574     AR.FoStage4[1].ziosize = IOsize;
00575     AR.FoStage4[0].ziobuffer = 0;
00576     AR.FoStage4[1].ziobuffer = 0;
00577 #endif
00578     AR.FoStage4[0].POsize = ((IOsize+sizeof(WORD)-1)/sizeof(WORD))*sizeof(WORD);
00579     AR.FoStage4[1].POsize = ((IOsize+sizeof(WORD)-1)/sizeof(WORD))*sizeof(WORD);
00580
00581     AR.hidefile = &(AR.Fscr[2]);
00582     AR.StoreData.Handle = -1;
00583     AR.SortType = AC.SortType;
00584
00585     AN.IndDum = AM.IndDum;
00586
00587     if ( identity > 0 ) {
00588         s = (UBYTE *) (FG.fname); i = 0;
00589         while ( *s ) { s++; i++; }
00590         s = (UBYTE *) Malloc1(sizeof(char)*(i+12),"name for Fscr[0] file");
00591         snprintf((char *)s,i+12,"%s.%d",FG.fname,identity);
00592         s[i-3] = 's'; s[i-2] = 'c'; s[i-1] = '0';
00593         AR.Fscr[0].name = (char *)s;
00594         s = (UBYTE *) (FG.fname); i = 0;
00595         while ( *s ) { s++; i++; }
00596         s = (UBYTE *) Malloc1(sizeof(char)*(i+12),"name for Fscr[1] file");
00597         snprintf((char *)s,i+12,"%s.%d",FG.fname,identity);
00598         s[i-3] = 's'; s[i-2] = 'c'; s[i-1] = '1';
00599         AR.Fscr[1].name = (char *)s;
00600     }
00601
00602     AR.CompressBuffer = (WORD *) Malloc1((AM.CompressSize+10)*sizeof(WORD),"compresssize");
00603     AR.ComprTop = AR.CompressBuffer + AM.CompressSize;
00604     AR.CompareRoutine = (COMPAREDUMMY) (&Compare1);
00605 /*
00606     Here we make all allocations for the struct AT
00607     (which is AB[identity].T or B->T with B = AB+identity).
00608 */
00609     AT.WorkSpace = (WORD *) Malloc1(AM.WorkSize*sizeof(WORD),"WorkSpace");
00610     AT.WorkTop = AT.WorkSpace + AM.WorkSize;
00611     AT.WorkPointer = AT.WorkSpace;
00612
00613     AT.Nest = (NESTING) Malloc1((LONG) sizeof(struct NeStIng)*AM.maxFlevels,"functionlevels");
00614     AT.NestStop = AT.Nest + AM.maxFlevels;
00615     AT.NestPoin = AT.Nest;
00616
00617     AT.WildMask = (WORD *) Malloc1((LONG) AM.MaxWildcards*sizeof(WORD),"maxwildcards");
00618
00619     LOCK(availabilitylock);
00620     AT.ebufnum = inicbufs(); /* Buffer for extras during execution */
00621     AT.fbufnum = inicbufs(); /* Buffer for caching in factorization */
00622     AT.allbufnum = inicbufs(); /* Buffer for id,all */
00623     AT.aebufnum = inicbufs(); /* Buffer for id,all */
00624     UNLOCK(availabilitylock);
00625
00626     AT.RepCount = (int *) Malloc1((LONG) ((AM.RepMax+3)*sizeof(int)),"repeat buffers");
00627     AN.RepPoint = AT.RepCount;
00628     AN.polysortflag = 0;
00629     AN.subsubveto = 0;
00630     AN.tryterm = 0;
00631     AT.RepTop = AT.RepCount + AM.RepMax;
00632
00633     AT.WildArgTaken = (WORD *) Malloc1((LONG) AC.WildcardBufferSize*sizeof(WORD)/2
00634         ,"argument list names");
00635     AT.WildcardBufferSize = AC.WildcardBufferSize;
00636     AT.previousEfactor = 0;
00637
00638     AT.identity = identity;
00639     AT.LoadBalancing = 0;
00640 /*
00641     Still to do: the SS stuff.
00642         the Fscr[3]
00643         the FoStage4[2]
00644 */
00645     if ( AT.WorkSpace == 0 ||
00646         AT.Nest == 0 ||
00647         AT.WildMask == 0 ||
00648         AT.RepCount == 0 ||
00649         AT.WildArgTaken == 0 ) goto OnError;
00650 /*
00651     And initializations
00652 */
00653     AT.comsym[0] = 8;
00654     AT.comsym[1] = SYMBOL;
00655     AT.comsym[2] = 4;
00656     AT.comsym[3] = 0;

```



```

00657     AT.comsym[4] = 1;
00658     AT.comsym[5] = 1;
00659     AT.comsym[6] = 1;
00660     AT.comsym[7] = 3;
00661     AT.comnum[0] = 4;
00662     AT.comnum[1] = 1;
00663     AT.comnum[2] = 1;
00664     AT.comnum[3] = 3;
00665     AT.comfun[0] = FUNHEAD+4;
00666     AT.comfun[1] = FUNCTION;
00667     AT.comfun[2] = FUNHEAD;
00668     AT.comfun[3] = 0;
00669     #if FUNHEAD > 3
00670     for ( i = 4; i <= FUNHEAD; i++ )
00671         AT.comfun[i] = 0;
00672     #endif
00673     AT.comfun[FUNHEAD+1] = 1;
00674     AT.comfun[FUNHEAD+2] = 1;
00675     AT.comfun[FUNHEAD+3] = 3;
00676     AT.comind[0] = 7;
00677     AT.comind[1] = INDEX;
00678     AT.comind[2] = 3;
00679     AT.comind[3] = 0;
00680     AT.comind[4] = 1;
00681     AT.comind[5] = 1;
00682     AT.comind[6] = 3;
00683     AT.locwildvalue[0] = SUBEXPRESSION;
00684     AT.locwildvalue[1] = SUBEXPSIZE;
00685     for ( i = 2; i < SUBEXPSIZE; i++ ) AT.locwildvalue[i] = 0;
00686     AT.mulpat[0] = TYPMULT;
00687     AT.mulpat[1] = SUBEXPSIZE+3;
00688     AT.mulpat[2] = 0;
00689     AT.mulpat[3] = SUBEXPRESSION;
00690     AT.mulpat[4] = SUBEXPSIZE;
00691     AT.mulpat[5] = 0;
00692     AT.mulpat[6] = 1;
00693     for ( i = 7; i < SUBEXPSIZE+5; i++ ) AT.mulpat[i] = 0;
00694     AT.proexp[0] = SUBEXPSIZE+4;
00695     AT.proexp[1] = EXPRESSION;
00696     AT.proexp[2] = SUBEXPSIZE;
00697     AT.proexp[3] = -1;
00698     AT.proexp[4] = 1;
00699     for ( i = 5; i < SUBEXPSIZE+1; i++ ) AT.proexp[i] = 0;
00700     AT.proexp[SUBEXPSIZE+1] = 1;
00701     AT.proexp[SUBEXPSIZE+2] = 1;
00702     AT.proexp[SUBEXPSIZE+3] = 3;
00703     AT.proexp[SUBEXPSIZE+4] = 0;
00704     AT.dummysubexp[0] = SUBEXPRESSION;
00705     AT.dummysubexp[1] = SUBEXPSIZE+4;
00706     for ( i = 2; i < SUBEXPSIZE; i++ ) AT.dummysubexp[i] = 0;
00707     AT.dummysubexp[SUBEXPSIZE] = WILDDUMMY;
00708     AT.dummysubexp[SUBEXPSIZE+1] = 4;
00709     AT.dummysubexp[SUBEXPSIZE+2] = 0;
00710     AT.dummysubexp[SUBEXPSIZE+3] = 0;
00711
00712     AT.MinVecArg[0] = 7+ARGHEAD;
00713     AT.MinVecArg[ARGHEAD] = 7;
00714     AT.MinVecArg[1+ARGHEAD] = INDEX;
00715     AT.MinVecArg[2+ARGHEAD] = 3;
00716     AT.MinVecArg[3+ARGHEAD] = 0;
00717     AT.MinVecArg[4+ARGHEAD] = 1;
00718     AT.MinVecArg[5+ARGHEAD] = 1;
00719     AT.MinVecArg[6+ARGHEAD] = -3;
00720     t = AT.FunArg;
00721     *t++ = 4+ARGHEAD+FUNHEAD;
00722     for ( i = 1; i < ARGHEAD; i++ ) *t++ = 0;
00723     *t++ = 4+FUNHEAD;
00724     *t++ = 0;
00725     *t++ = FUNHEAD;
00726     for ( i = 2; i < FUNHEAD; i++ ) *t++ = 0;
00727     *t++ = 1; *t++ = 1; *t++ = 3;
00728
00729     AT.inprimelist = -1;
00730     AT.sizeprimelist = 0;
00731     AT.primelist = 0;
00732     AT.nfac = AT.nBer = 0;
00733     AT.factorials = 0;
00734     AT.bernoullis = 0;
00735     AR.wranfia = 0;
00736     AR.wranfcall = 0;
00737     AR.wranfnpair1 = NPAIR1;
00738     AR.wranfnpair2 = NPAIR2;
00739     AR.wranfseed = 0;
00740     AN.SplitScratch = 0;
00741     AN.SplitScratchSize = AN.InScratch = 0;
00742     AN.SplitScratch1 = 0;
00743     AN.SplitScratchSize1 = AN.InScratch1 = 0;

```

```

00744 /*
00745     Now the sort buffers. They depend on which thread. The master
00746     inherits the sortbuffer from AM.S0
00747 */
00748     if ( identity == 0 ) {
00749         AT.S0 = AM.S0;
00750     }
00751     else {
00752 /*
00753         For the moment we don't have special settings.
00754         They may become costly in virtual memory.
00755 */
00756         AT.S0 = AllocSort(AM.S0->LargeSize*sizeof(WORD)/numberofworkers
00757             ,AM.S0->SmallSize*sizeof(WORD)/numberofworkers
00758             ,AM.S0->SmallEsize*sizeof(WORD)/numberofworkers
00759             ,AM.S0->TermsInSmall
00760             ,AM.S0->MaxPatches
00761 /*
00762             ,AM.S0->MaxPatches/numberofworkers */
00763             ,AM.S0->MaxFpatches/numberofworkers
00764             ,AM.S0->file.POsize
00765             ,0);
00766     }
00767     AR.CompressPointer = AR.CompressBuffer;
00768 /*
00769     Install the store caches (15-aug-2006 JV)
00770 */
00771     AT.StoreCache = AT.StoreCacheAlloc = 0;
00772     if ( AM.NumStoreCaches > 0 ) {
00773         STORECACHE sa, sb;
00774         LONG size;
00775         size = sizeof(struct StOrEcAcHe)+AM.SizeStoreCache;
00776         size = ((size-1)/sizeof(size_t)+1)*sizeof(size_t);
00777         AT.StoreCacheAlloc = (STORECACHE)Malloc1(size*AM.NumStoreCaches,"StoreCaches");
00778         sa = AT.StoreCache = AT.StoreCacheAlloc;
00779         for ( i = 0; i < AM.NumStoreCaches; i++ ) {
00780             sb = (STORECACHE)(void *) ((UBYTE *)sa+size);
00781             if ( i == AM.NumStoreCaches-1 ) {
00782                 sa->next = 0;
00783             }
00784             else {
00785                 sa->next = sb;
00786             }
00787             SETBASEPOSITION(sa->position,-1);
00788             SETBASEPOSITION(sa->toppos,-1);
00789             sa = sb;
00790         }
00791     }
00792     ReserveTempFiles(2);
00793     return(B);
00794 OnError:;
00795     MLOCK(ErrorMessageLock);
00796     MesPrint("Error initializing thread %d",identity);
00797     MUNLOCK(ErrorMessageLock);
00798     Terminate(-1);
00799     return(B);
00800 }
00801
00802 /*
00803     #] InitializeOneThread :
00804     #[ FinalizeOneThread :
00805 */
00816 void FinalizeOneThread(int identity)
00817 {
00818     timerinfo[identity] = TimeCPU(1);
00819 }
00820
00821 /*
00822     #] FinalizeOneThread :
00823     #[ ClearAllThreads :
00824 */
00831 void ClearAllThreads(void)
00832 {
00833     int i;
00834     MasterWaitAll();
00835     for ( i = 1; i <= numberofworkers; i++ ) {
00836         WakeupThread(i,CLEARCLOCK);
00837     }
00838 #ifndef WITHSORTBOTS
00839     for ( i = numberofworkers+1; i <= numberofworkers+numberofsortbots; i++ ) {
00840         WakeupThread(i,CLEARCLOCK);
00841     }
00842 #endif
00843 }
00844
00845 /*
00846     #] ClearAllThreads :

```

```

00847     # [ TerminateAllThreads :
00848     /*
00855 void TerminateAllThreads(void)
00856 {
00857     int i;
00858     for ( i = 1; i <= numberofworkers; i++ ) {
00859         GetThread(i);
00860         WakeupThread(i,TERMINATETHREAD);
00861     }
00862 #ifdef WITHSORTBOTS
00863     for ( i = numberofworkers+1; i <= numberofworkers+numberofsortbots; i++ ) {
00864         WakeupThread(i,TERMINATETHREAD);
00865     }
00866 #endif
00867     for ( i = 1; i <= numberofworkers; i++ ) {
00868         pthread_join(threadpointers[i],NULL);
00869     }
00870 #ifdef WITHSORTBOTS
00871     for ( i = numberofworkers+1; i <= numberofworkers+numberofsortbots; i++ ) {
00872         pthread_join(threadpointers[i],NULL);
00873     }
00874 #endif
00875 }
00876
00877 /*
00878     #] TerminateAllThreads :
00879     # [ MakeThreadBuckets :
00880     /*
00906 int MakeThreadBuckets(int number, int par)
00907 {
00908     int i;
00909     LONG sizethreadbuckets;
00910     THREADBUCKET *thr;
00911     /*
00912     First we need a decent estimate. Not all terms should be maximal.
00913     Note that AM.MaxTer is in bytes!!!
00914     Maybe we should try to limit the size here a bit more effectively.
00915     This is a great consumer of memory.
00916     /*
00917     sizethreadbuckets = ( AC.ThreadBucketSize + 1 ) * AM.MaxTer + 2*sizeof(WORD);
00918     if ( AC.ThreadBucketSize >= 250 )     sizethreadbuckets /= 4;
00919     else if ( AC.ThreadBucketSize >= 90 )     sizethreadbuckets /= 3;
00920     else if ( AC.ThreadBucketSize >= 40 )     sizethreadbuckets /= 2;
00921     sizethreadbuckets /= sizeof(WORD);
00922
00923     if ( par == 0 ) {
00924         numthreadbuckets = 2*(number-1);
00925         threadbuckets = (THREADBUCKET **)Malloc1(numthreadbuckets*sizeof(THREADBUCKET
00926 *),"threadbuckets");
00927         freebuckets = (THREADBUCKET **)Malloc1(numthreadbuckets*sizeof(THREADBUCKET
00928 *),"threadbuckets");
00929     }
00930     if ( par > 0 ) {
00931         if ( sizethreadbuckets <= threadbuckets[0]->threadbuffersize ) return(0);
00932         for ( i = 0; i < numthreadbuckets; i++ ) {
00933             thr = threadbuckets[i];
00934             M_free(thr->deferbuffer,"deferbuffer");
00935         }
00936     }
00937     else {
00938         for ( i = 0; i < numthreadbuckets; i++ ) {
00939             threadbuckets[i] = (THREADBUCKET *)Malloc1(sizeof(THREADBUCKET),"threadbuckets");
00940             threadbuckets[i]->lock = dummylock;
00941         }
00942     }
00943     for ( i = 0; i < numthreadbuckets; i++ ) {
00944         thr = threadbuckets[i];
00945         thr->threadbuffersize = sizethreadbuckets;
00946         thr->free = BUCKETFREE;
00947         thr->deferbuffer = (POSITION *)Malloc1(2*sizethreadbuckets*sizeof(WORD)
00948             + (AC.ThreadBucketSize+1)*sizeof(POSITION),"deferbuffer");
00949         thr->threadbuffer = (WORD *) (thr->deferbuffer+AC.ThreadBucketSize+1);
00950         thr->compressbuffer = (WORD *) (thr->threadbuffer+sizethreadbuckets);
00951         thr->busy = BUCKETPREPARINGTERM;
00952         thr->usenum = thr->totnum = 0;
00953         thr->type = BUCKETDOINGTERMS;
00954     }
00955     return(0);
00956 }
00957
00958     #] MakeThreadBuckets :
00959     # [ GetTimerInfo :
00960     /*
00966 int GetTimerInfo(LONG** ti,LONG** sti)
00967 {

```

```

00968     *ti = timerinfo;
00969     *sti = sumtimerinfo;
00970 #ifdef WITHSORTBOTS
00971     return AM.totalnumberofthreads*2;
00972 #else
00973     return AM.totalnumberofthreads;
00974 #endif
00975 }
00976
00977 /*
00978     #] GetTimerInfo :
00979     #[ WriteTimerInfo :
00980 */
00981
00982 void WriteTimerInfo(LONG* ti, LONG* sti)
00983 {
00984     int i;
00985 #ifdef WITHSORTBOTS
00986     int max = AM.totalnumberofthreads*2;
00987 #else
00988     int max = AM.totalnumberofthreads;
00989 #endif
00990     for ( i=0; i<max; ++i ) {
00991         timerinfo[i] = ti[i];
00992         sumtimerinfo[i] = sti[i];
00993     }
00994 }
00995
01000 /*
01001     #] WriteTimerInfo :
01002     #[ GetWorkerTimes :
01003 */
01004 LONG GetWorkerTimes(void)
01005 {
01006     LONG retval = 0;
01007     int i;
01008     for ( i = 1; i <= numberofworkers; i++ ) retval += timerinfo[i] + sumtimerinfo[i];
01009 #ifdef WITHSORTBOTS
01010     for ( i = numberofworkers+1; i <= numberofworkers+numberofsortbots; i++ )
01011         retval += timerinfo[i] + sumtimerinfo[i];
01012 #endif
01013     return(retval);
01014 }
01015
01021 /*
01022     #] GetWorkerTimes :
01023     #[ UpdateOneThread :
01024 */
01025 int UpdateOneThread(int identity)
01026 {
01027     ALLPRIVATES *B = AB[identity], *B0 = AB[0];
01028     AR.GetFile = AR0.GetFile;
01029     AR.KeptInHold = AR0.KeptInHold;
01030     AR.CurExpr = AR0.CurExpr;
01031     AR.SortType = AC.SortType;
01032     if ( AT.WildcardBufferSize < AC.WildcardBufferSize ) {
01033         M_free(AT.WildArgTaken, "argument list names");
01034         AT.WildcardBufferSize = AC.WildcardBufferSize;
01035         AT.WildArgTaken = (WORD *)Malloc1((LONG)AC.WildcardBufferSize*sizeof(WORD)/2
01036             , "argument list names");
01037         if ( AT.WildArgTaken == 0 ) return(-1);
01038     }
01039     return(0);
01040 }
01041
01048 /*
01049     #] UpdateOneThread :
01050     #[ LoadOneThread :
01051 */
01052 int LoadOneThread(int from, int identity, THREADBUCKET *thr, int par)
01053 {
01054     WORD *t1, *t2;
01055     ALLPRIVATES *B = AB[identity], *B0 = AB[from];
01056
01057     AR.DefPosition = AR0.DefPosition;
01058     AR.NoCompress = AR0.NoCompress;
01059     AR.gzipCompress = AR0.gzipCompress;
01060     AR.BracketOn = AR0.BracketOn;
01061     AR.CurDum = AR0.CurDum;
01062     AR.DeferFlag = AR0.DeferFlag;
01063     AR.TePos = 0;
01064     AR.sLevel = AR0.sLevel;
01065     AR.Stage4Name = AR0.Stage4Name;
01066     AR.GetOneFile = AR0.GetOneFile;
01067     AR.PolyFun = AR0.PolyFun;
01068     AR.PolyFunInv = AR0.PolyFunInv;
01069     AR.PolyFunType = AR0.PolyFunType;

```

```

01083     AR.PolyFunExp = AR0.PolyFunExp;
01084     AR.PolyFunVar = AR0.PolyFunVar;
01085     AR.PolyFunPow = AR0.PolyFunPow;
01086     AR.Eside = AR0.Eside;
01087     AR.Cnumlhs = AR0.Cnumlhs;
01088 /*
01089     AR.MaxBracket = AR0.MaxBracket;
01090
01091     The compressbuffer contents are mainly relevant for keep brackets
01092     We should do this only if there is a keep brackets statement
01093     We may however still need the compressbuffer for expressions in the rhs.
01094 */
01095     if ( par >= 1 ) {
01096 /*
01097         We may not need this %%%% 7-apr-2006
01098 */
01099         t1 = AR.CompressBuffer; t2 = AR0.CompressBuffer;
01100         while ( t2 < AR0.CompressPointer ) *t1++ = *t2++;
01101         AR.CompressPointer = t1;
01102     }
01103     else {
01104         AR.CompressPointer = AR.CompressBuffer;
01105     }
01106     if ( AR.DeferFlag ) {
01107         if ( AR.infile->handle < 0 ) {
01108             AR.infile->POfill = AR0.infile->POfill;
01109         }
01110         else {
01111 /*
01112             We have to set the value of PPosition to something that will
01113             force a read in the first try.
01114 */
01115             AR.infile->POfull = AR.infile->POfill = AR.infile->PObuffer;
01116         }
01117     }
01118     if ( par == 0 ) {
01119         AN.threadbuck = thr;
01120         AN.ninterms = thr->firstterm;
01121     }
01122     else if ( par == 1 ) {
01123         WORD *tstop;
01124         t1 = thr->threadbuffer; tstop = t1 + *t1;
01125         t2 = AT.WorkPointer;
01126         while ( t1 < tstop ) *t2++ = *t1++;
01127         AN.ninterms = thr->firstterm;
01128     }
01129     AN.TeInFun = 0;
01130     AN.ncmod = AC.ncmod;
01131     AT.BrackBuf = AT0.BrackBuf;
01132     AT.bracketindexflag = AT0.bracketindexflag;
01133     AN.PolyFunTodo = 0;
01134 /*
01135     The relevant variables and the term are in their place.
01136     There is nothing more to do.
01137 */
01138     return(0);
01139 }
01140
01141 /*
01142 #] LoadOneThread :
01143 #[ BalanceRunThread :
01144 */
01145 int BalanceRunThread(PHEAD int identity, WORD *term, WORD level)
01146 {
01147     GETBIDENTITY
01148     ALLPRIVATES *BB;
01149     WORD *t, *tt;
01150     int i, *ti, *tti;
01151
01152     LoadOneThread(AT.identity,identity,0,2);
01153 /*
01154     Extra loading if needed. Quantities changed in Generator.
01155     Like the level that has to be passed.
01156 */
01157     BB = AB[identity];
01158     BB->R.level = level;
01159     BB->T.TMbuff = AT.TMbuff;
01160     ti = AT.RepCount; tti = BB->T.RepCount;
01161     i = AN.RepPoint - AT.RepCount;
01162     BB->N.RepPoint = BB->T.RepCount + i;
01163     for ( ; i >= 0; i-- ) tti[i] = ti[i];
01164
01165     t = term; i = *term;
01166     tt = BB->T.WorkSpace;
01167     NCOPY(tt,t,i);
01168     BB->T.WorkPointer = tt;

```

```

01185
01186     WakeupThread(identity,HIGHERLEVELGENERATION);
01187
01188     return(0);
01189 }
01190
01191 /*
01192     #] BalanceRunThread :
01193     #[ SetWorkerFiles :
01194 */
01199 void SetWorkerFiles(void)
01200 {
01201     int id;
01202     ALLPRIVATES *B, *B0 = AB[0];
01203     for ( id = 1; id < AM.totalnumberofthreads; id++ ) {
01204         B = AB[id];
01205         AR.infile = &(AR.Fscr[0]);
01206         AR.outfile = &(AR.Fscr[1]);
01207         AR.hidefile = &(AR.Fscr[2]);
01208         AR.infile->handle = AR0.infile->handle;
01209         AR.hidefile->handle = AR0.hidefile->handle;
01210         if ( AR.infile->handle < 0 ) {
01211             AR.infile->PObuffer = AR0.infile->PObuffer;
01212             AR.infile->PStop = AR0.infile->PStop;
01213             AR.infile->POfill = AR0.infile->POfill;
01214             AR.infile->POfull = AR0.infile->POfull;
01215             AR.infile->POsize = AR0.infile->POsize;
01216             AR.InInBuf = AR0.InInBuf;
01217             AR.infile->POposition = AR0.infile->POposition;
01218             AR.infile->filesize = AR0.infile->filesize;
01219         }
01220         else {
01221             AR.infile->PObuffer = AR.infile->wPObuffer;
01222             AR.infile->PStop = AR.infile->wPStop;
01223             AR.infile->POfill = AR.infile->wPOfill;
01224             AR.infile->POfull = AR.infile->wPOfull;
01225             AR.infile->POsize = AR.infile->wPOsize;
01226             AR.InInBuf = 0;
01227             PUTZERO(AR.infile->POposition);
01228         }
01229     /*
01230         If there is some writing, it better happens to ones own outfile.
01231         Currently this is to be done only for InParallel.
01232         Merging of the outputs is then done by the CopyExpression routine.
01233     */
01234     {
01235         AR.outfile->PObuffer = AR.outfile->wPObuffer;
01236         AR.outfile->PStop = AR.outfile->wPStop;
01237         AR.outfile->POfill = AR.outfile->wPOfill;
01238         AR.outfile->POfull = AR.outfile->wPOfull;
01239         AR.outfile->POsize = AR.outfile->wPOsize;
01240         PUTZERO(AR.outfile->POposition);
01241     }
01242     if ( AR.hidefile->handle < 0 ) {
01243         AR.hidefile->PObuffer = AR0.hidefile->PObuffer;
01244         AR.hidefile->PStop = AR0.hidefile->PStop;
01245         AR.hidefile->POfill = AR0.hidefile->POfill;
01246         AR.hidefile->POfull = AR0.hidefile->POfull;
01247         AR.hidefile->POsize = AR0.hidefile->POsize;
01248         AR.InHiBuf = AR0.InHiBuf;
01249         AR.hidefile->POposition = AR0.hidefile->POposition;
01250         AR.hidefile->filesize = AR0.hidefile->filesize;
01251     }
01252     else {
01253         AR.hidefile->PObuffer = AR.hidefile->wPObuffer;
01254         AR.hidefile->PStop = AR.hidefile->wPStop;
01255         AR.hidefile->POfill = AR.hidefile->wPOfill;
01256         AR.hidefile->POfull = AR.hidefile->wPOfull;
01257         AR.hidefile->POsize = AR.hidefile->wPOsize;
01258         AR.InHiBuf = 0;
01259         PUTZERO(AR.hidefile->POposition);
01260     }
01261 }
01262 if ( AR0.StoreData.dirtyflag ) {
01263     for ( id = 1; id < AM.totalnumberofthreads; id++ ) {
01264         B = AB[id];
01265         AR.StoreData = AR0.StoreData;
01266     }
01267 }
01268 }
01269
01270 /*
01271     #] SetWorkerFiles :
01272     #[ RunThread :
01273 */
01281 void *RunThread(void *dummy)
01282 {

```

```

01283     WORD *term, *ttin, *tt, *ttco, *oldwork;
01284     int identity, wakeupsignal, identityretv, i, toberelased, errorcode;
01285     ALLPRIVATES *B;
01286     THREADBUCKET *thr;
01287     POSITION *ppdef;
01288     EXPRESSIONS e;
01289     DUMMYUSE(dummy);
01290     identity = SetIdentity(&identityretv);
01291     threadpointers[identity] = pthread_self();
01292     B = InitializeOneThread(identity);
01293     while ( ( wakeupsignal = ThreadWait(identity) ) > 0 ) {
01294         switch ( wakeupsignal ) {
01295             /*
01296              *      #[] STARTNEWEXPRESSION :
01297              */
01298             case STARTNEWEXPRESSION:
01299                 /*
01300                  *      Set up the sort routines etc.
01301                  *      Start with getting some buffers synchronized with the compiler
01302                  */
01303                 if ( UpdateOneThread(identity) ) {
01304                     MLOCK(ErrorMessageLock);
01305                     MesPrint("Update error in starting expression in thread %d in module
01306 %d",identity,AC.CModule);
01307                     MUNLOCK(ErrorMessageLock);
01308                     Terminate(-1);
01309                 }
01310                 AR.DeferFlag = AC.ComDefer;
01311                 AR.sLevel = AS.sLevel;
01312                 AR.MaxDum = AM.IndDum;
01313                 AR.expchanged = AB[0]->R.expchanged;
01314                 AR.expflags = AB[0]->R.expflags;
01315                 AR.PolyFun = AB[0]->R.PolyFun;
01316                 AR.PolyFunInv = AB[0]->R.PolyFunInv;
01317                 AR.PolyFunType = AB[0]->R.PolyFunType;
01318                 AR.PolyFunExp = AB[0]->R.PolyFunExp;
01319                 AR.PolyFunVar = AB[0]->R.PolyFunVar;
01320                 AR.PolyFunPow = AB[0]->R.PolyFunPow;
01321             /*
01322              *      Now fire up the sort buffer.
01323              */
01324             NewSort (BHEAD0);
01325             break;
01326             #] STARTNEWEXPRESSION :
01327             #[] LOWESTLEVELGENERATION :
01328             /*
01329              *      case LOWESTLEVELGENERATION:
01330              *      #ifndef INNERTEST
01331              *      if ( AC.InnerTest ) {
01332              *          if ( StrCmp(AC.TestValue, (UBYTE *)INNERTEST) == 0 ) {
01333              *              MesPrint("Testing(Worker%d): value = %s",AT.identity,AC.TestValue);
01334              *          }
01335              *      }
01336              *      #endif
01337              *      e = Expressions + AR.CurExpr;
01338              *      thr = AN.threadbuck;
01339              *      ppdef = thr->deferbuffer;
01340              *      ttin = thr->threadbuffer;
01341              *      ttco = thr->compressbuffer;
01342              *      term = AT.WorkPointer;
01343              *      thr->usenum = 0;
01344              *      toberelased = 0;
01345              *      AN.inputnumber = thr->firstterm;
01346              *      AN.ninterms = thr->firstterm;
01347              *      do {
01348              *          thr->usenum++; /* For if the master wants to steal the bucket */
01349              *          tt = term; i = *ttin;
01350              *          NCOPY(tt,ttin,i);
01351              *          AT.WorkPointer = tt;
01352              *          if ( AR.DeferFlag ) {
01353              *              tt = AR.CompressBuffer; i = *ttco;
01354              *              NCOPY(tt,ttco,i);
01355              *              AR.CompressPointer = tt;
01356              *              AR.DefPosition = ppdef[0]; ppdef++;
01357              *          }
01358              *          if ( thr->free == BUCKETTERMINATED ) {
01359              *              /*
01360              *              The next statement allows the master to steal the bucket
01361              *              for load balancing purposes. We do still execute the current
01362              *              term, but afterwards we drop out.
01363              *              Once we have written the release code, we cannot use this
01364              *              bucket anymore. Hence the exit to the label bucketstolen.
01365              */
01366              *              if ( thr->usenum == thr->totnum ) {
01367              *                  thr->free = BUCKETCOMINGFREE;
01368              *              }

```

```

01369         else {
01370             thr->free = BUCKETRELEASED;
01371             tobereleased = 1;
01372         }
01373     }
01374     /*
01375         What if we want to steal and we set thr->free while
01376         the thread is inside the next code for a long time?
01377     if ( AT.LoadBalancing ) {
01378     */
01379         LOCK(thr->lock);
01380         thr->busy = BUCKETDOINGTERM;
01381         UNLOCK(thr->lock);
01382     /*
01383     }
01384     else {
01385         thr->busy = BUCKETDOINGTERM;
01386     }
01387     */
01388     AN.RepPoint = AT.RepCount + 1;
01389
01390     if ( ( e->vflags & ISFACTORIZED ) != 0 && term[1] == HAAKJE ) {
01391         StoreTerm(BHEAD term);
01392     }
01393     else {
01394         if ( AR.DeferFlag ) {
01395             AR.CurDum = AN.IndDum = Expressions[AR.CurExpr].numdummies + AM.IndDum;
01396         }
01397         else {
01398             AN.IndDum = AM.IndDum;
01399             AR.CurDum = ReNumber(BHEAD term);
01400         }
01401         if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
01402         if ( AN.ncmod ) {
01403             if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
01404             else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
01405         }
01406         else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
01407         if ( ( AP.PreDebug & THREADSDEBUG ) != 0 ) {
01408             MLOCK(ErrorMessageLock);
01409             MesPrint("Thread %w executing term:");
01410             PrintTerm(term,"LLG");
01411             MUNLOCK(ErrorMessageLock);
01412         }
01413         if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
01414             && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
01415             PolyFunClean(BHEAD term);
01416         }
01417         if ( Generator(BHEAD term,0) ) {
01418             LowerSortLevel();
01419             MLOCK(ErrorMessageLock);
01420             MesPrint("Error in processing one term in thread %d in module
01421 %d",identity,AC.CModule);
01421             MUNLOCK(ErrorMessageLock);
01422             Terminate(-1);
01423         }
01424         AN.ninterms++;
01425     }
01426     /*
01427     if ( AT.LoadBalancing ) { */
01428         LOCK(thr->lock);
01429         thr->busy = BUCKETPREPARINGTERM;
01430         UNLOCK(thr->lock);
01431     /*
01432     }
01433     else {
01434         thr->busy = BUCKETPREPARINGTERM;
01435     }
01436     */
01437     if ( thr->free == BUCKETTERMINATED ) {
01438         if ( thr->usenum == thr->totnum ) {
01439             thr->free = BUCKETCOMINGFREE;
01440         }
01441         else {
01442             thr->free = BUCKETRELEASED;
01443             tobereleased = 1;
01444         }
01445     }
01446     if ( tobereleased ) goto bucketstolen;
01447     } while ( *ttin );
01448     thr->free = BUCKETCOMINGFREE;
01449     bucketstolen;;
01450     /*
01451     if ( AT.LoadBalancing ) { */
01452         LOCK(thr->lock);
01453         thr->busy = BUCKETTOBERELEASED;
01454         UNLOCK(thr->lock);
01455     }
01456     /*
01457     else {

```



```

01455         thr->busy = BUCKETTOBERELEASED;
01456     }
01457 */
01458     AT.WorkPointer = term;
01459     break;
01460 /*
01461     #] LOWESTLEVELGENERATION :
01462     #[ FINISHEXPRESSION :
01463 */
01464 #ifdef WITHSORTBOTS
01465     case CLAIMOUTPUT:
01466         LOCK(AT.SB.MasterBlockLock[1]);
01467         break;
01468 #endif
01469     case FINISHEXPRESSION:
01470 /*
01471         Finish the sort
01472
01473         Start with claiming the first block
01474         Once we have claimed it we can let the master know that
01475         everything is all right.
01476 */
01477         LOCK(AT.SB.MasterBlockLock[1]);
01478         ThreadClaimedBlock(identity);
01479 /*
01480         Entry for when we work with sortbots
01481 */
01482 #ifdef WITHSORTBOTS
01483         /* fall through */
01484         case FINISHEXPRESSION2:
01485 #endif
01486 /*
01487         Now we may need here an fsync on the sort file
01488 */
01489         if ( AC.ThreadSortFileSynch ) {
01490             if ( AT.S0->file.handle >= 0 ) {
01491                 SynchFile(AT.S0->file.handle);
01492             }
01493         }
01494         AT.SB.FillBlock = 1;
01495         AT.SB.MasterFill[1] = AT.SB.MasterStart[1];
01496         errorcode = EndSort(BHEAD AT.S0->sBuffer,0);
01497         UNLOCK(AT.SB.MasterBlockLock[AT.SB.FillBlock]);
01498         UpdateMaxSize();
01499         if ( errorcode ) {
01500             MLOCK(ErrorMessageLock);
01501             MesPrint("Error terminating sort in thread %d in module %d",identity,AC.CModule);
01502             MUNLOCK(ErrorMessageLock);
01503             Terminate(-1);
01504         }
01505         break;
01506 /*
01507     #] FINISHEXPRESSION :
01508     #[ CLEANUPEXPRESSION :
01509 */
01510     case CLEANUPEXPRESSION:
01511 /*
01512         Cleanup everything and wait for the next expression
01513 */
01514         if ( AR.outfile->handle >= 0 ) {
01515             CloseFile(AR.outfile->handle);
01516             AR.outfile->handle = -1;
01517             remove(AR.outfile->name);
01518             AR.outfile->POfill = AR.outfile->POfull = AR.outfile->PObuffer;
01519             PUTZERO(AR.outfile->POposition);
01520             PUTZERO(AR.outfile->filesize);
01521         }
01522         else {
01523             AR.outfile->POfill = AR.outfile->POfull = AR.outfile->PObuffer;
01524             PUTZERO(AR.outfile->POposition);
01525             PUTZERO(AR.outfile->filesize);
01526         }
01527     {
01528         CBUF *C = cbuf+AT.ebufnum;
01529         WORD **w, ii;
01530         if ( C->numrhs > 0 || C->numlhs > 0 ) {
01531             if ( C->rhs ) {
01532                 w = C->rhs; ii = C->numrhs;
01533                 do { *w++ = 0; } while ( --ii > 0 );
01534             }
01535             if ( C->lhs ) {
01536                 w = C->lhs; ii = C->numlhs;
01537                 do { *w++ = 0; } while ( --ii > 0 );
01538             }
01539             C->numlhs = C->numrhs = 0;
01540             ClearTree(AT.ebufnum);
01541             C->Pointer = C->Buffer;

```

```

01542         }
01543     }
01544     break;
01545 /*
01546     #] CLEANUPEXPRESSION :
01547     #[ HIGHERLEVELGENERATION :
01548 */
01549     case HIGHERLEVELGENERATION:
01550 /*
01551         When foliating halfway the tree.
01552         This should only be needed in a second level load balancing
01553 */
01554         term = AT.WorkSpace; AT.WorkPointer = term + *term;
01555         if ( Generator(BHEAD term,AR.level) ) {
01556             LowerSortLevel();
01557             MLOCK(ErrorMessageLock);
01558             MesPrint("Error in load balancing one term at level %d in thread %d in module
01559 %d",AR.level,AT.identity,AC.CModule);
01559             MUNLOCK(ErrorMessageLock);
01560             Terminate(-1);
01561         }
01562         AT.WorkPointer = term;
01563         break;
01564 /*
01565     #] HIGHERLEVELGENERATION :
01566     #[ STARTNEWMODULE :
01567 */
01568     case STARTNEWMODULE:
01569 /*
01570         For resetting variables.
01571 */
01572         SpecialCleanup(B);
01573         break;
01574 /*
01575     #] STARTNEWMODULE :
01576     #[ TERMINATETHREAD :
01577 */
01578     case TERMINATETHREAD:
01579         goto EndOfThread;
01580 /*
01581     #] TERMINATETHREAD :
01582     #[ DOONEEXPRESSION :
01583
01584         When a thread has to do a complete (not too big) expression.
01585         The number of the expression to be done is in AR.exprtodo.
01586         The code is mostly taken from Processor. The only difference
01587         is with what to do with the output.
01588         The output should go to the scratch buffer of the worker
01589         (which is free at the right moment). If this buffer is too
01590         small we have a problem. We could write to file or give the
01591         master what we have and from now on the master has to collect
01592         pieces until things are complete.
01593         Note: this assumes that the expressions don't keep their order.
01594         If they have to keep their order, don't use this feature.
01595 */
01596     case DOONEEXPRESSION: {
01597
01598         POSITION position, outposition;
01599         FILEHANDLE *fi, *fout, *oldoutfile;
01600         LONG dd = 0;
01601         WORD oldBracketOn = AR.BracketOn;
01602         WORD *oldBrackBuf = AT.BrackBuf;
01603         WORD oldbracketindexflag = AT.bracketindexflag;
01604         WORD fromspectator = 0;
01605         e = Expressions + AR.exprtodo;
01606         i = AR.exprtodo;
01607         AR.CurExpr = i;
01608         AR.SortType = AC.SortType;
01609         AR.expchanged = 0;
01610         if ( ( e->vflags & ISFACTORIZED ) != 0 ) {
01611             AR.BracketOn = 1;
01612             AT.BrackBuf = AM.BracketFactors;
01613             AT.bracketindexflag = 1;
01614         }
01615         position = AS.OldOnFile[i];
01616         if ( e->status == HIDDENLEXPRESSION || e->status == HIDDENGEEXPRESSION
01617             || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEEXPRESSION ) {
01618             AR.GetFile = 2; fi = AR.hidefile;
01619         }
01620         else {
01621             AR.GetFile = 0; fi = AR.infile;
01622         }
01623
01624 /*
01625         PUTZERO(fi->PPosition);
01626         if ( fi->handle >= 0 ) {
01627             fi->POfill = fi->POfull = fi->PObuffer;

```

```

01628     }
01629     */
01630     SetScratch(fi,&position);
01631     term = oldwork = AT.WorkPointer;
01632     AR.CompressPointer = AR.CompressBuffer;
01633     AR.CompressPointer[0] = 0;
01634     AR.KeptInHold = 0;
01635     if ( GetTerm(BHEAD term) <= 0 ) {
01636         MLOCK(ErrorMessageLock);
01637         MesPrint("Expression %d has problems in scratchfile (t)",i);
01638         MUNLOCK(ErrorMessageLock);
01639         Terminate(-1);
01640     }
01641     if ( AT.bracketindexflag > 0 ) OpenBracketIndex(i);
01642     term[3] = i;
01643     if ( term[5] < 0 ) {
01644         fromspectator = -term[5];
01645         PUTZERO(AM.SpectatorFiles[fromspectator-1].readpos);
01646         term[5] = AC.cbufnum;
01647     }
01648     PUTZERO(outposition);
01649     fout = AR.outfile;
01650     fout->POfill = fout->POfull = fout->PObuffer;
01651     fout->POposition = outposition;
01652     if ( fout->handle >= 0 ) {
01653         fout->POposition = outposition;
01654     }
01655     /*
01656     The next statement is needed because we need the system
01657     to believe that the expression is at position zero for
01658     the moment. In this worker, with no memory of other expressions,
01659     it is. This is needed for when a bracket index is made
01660     because there e->onfile is an offset. Afterwards, when the
01661     expression is written to its final location in the masters
01662     output e->onfile will get its real value.
01663     */
01664     PUTZERO(e->onfile);
01665     if ( PutOut(BHEAD term,&outposition,fout,0) < 0 ) goto ProcErr;
01666
01667     AR.DeferFlag = AC.ComDefer;
01668
01669     AR.sLevel = AB[0]->R.sLevel;
01670     term = AT.WorkPointer;
01671     NewSort(BHEAD0);
01672     AR.MaxDum = AM.IndDum;
01673     AN.ninterms = 0;
01674     if ( fromspectator ) {
01675         while ( GetFromSpectator(term,fromspectator-1) ) {
01676             AT.WorkPointer = term + *term;
01677             AN.RepPoint = AT.RepCount + 1;
01678             AN.IndDum = AM.IndDum;
01679             AR.CurDum = ReNumber(BHEAD term);
01680             if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
01681             if ( AN.ncmod ) {
01682                 if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
01683                 else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
01684             }
01685             else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
01686             if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
01687                 && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
01688                 PolyFunClean(BHEAD term);
01689             }
01690             if ( Generator(BHEAD term,0) ) {
01691                 LowerSortLevel(); goto ProcErr;
01692             }
01693         }
01694     }
01695     else {
01696         while ( GetTerm(BHEAD term) ) {
01697             SeekScratch(fi,&position);
01698             AN.ninterms++; dd = AN.deferskipped;
01699             if ( ( e->vflags & ISFACTORIZED ) != 0 && term[1] == HAAKJE ) {
01700                 StoreTerm(BHEAD term);
01701             }
01702             else {
01703                 if ( AC.CollectFun && *term <= (AM.MaxTer/(2*(LONG)sizeof(WORD))) ) {
01704                     if ( GetMoreTerms(term) < 0 ) {
01705                         LowerSortLevel(); goto ProcErr;
01706                     }
01707                     SeekScratch(fi,&position);
01708                 }
01709                 AT.WorkPointer = term + *term;
01710                 AN.RepPoint = AT.RepCount + 1;
01711                 if ( AR.DeferFlag ) {
01712                     AR.CurDum = AN.IndDum = Expressions[AR.exprtodo].numdummies;
01713                 }
01714             }
01715         }
01716     }

```

```

01715         AN.IndDum = AM.IndDum;
01716         AR.CurDum = ReNumber(BHEAD term);
01717     }
01718     if ( AC.SymChangeFlag ) MarkDirty(term,DIRTYSYMFLAG);
01719     if ( AN.ncmod ) {
01720         if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term,DIRTYFLAG);
01721         else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
01722     }
01723     else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
01724     if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
01725         && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
01726         PolyFunClean(BHEAD term);
01727     }
01728     if ( Generator(BHEAD term,0) ) {
01729         LowerSortLevel(); goto ProcErr;
01730     }
01731     AN.ninterms += dd;
01732 }
01733 SetScratch(fi,&position);
01734 if ( fi == AR.hidefile ) {
01735     AR.InHiBuf = (fi->POfull-fi->PObuffer)
01736         -DIFBASE(position,fi->POposition)/sizeof(WORD);
01737 }
01738 else {
01739     AR.InInBuf = (fi->POfull-fi->PObuffer)
01740         -DIFBASE(position,fi->POposition)/sizeof(WORD);
01741 }
01742 }
01743 }
01744 AN.ninterms += dd;
01745 if ( EndSort(BHEAD AT.S0->sBuffer,0) < 0 ) goto ProcErr;
01746 e->numdummies = AR.MaxDum - AM.IndDum;
01747 AR.BraceOn = oldBraceOn;
01748 AT.BrackBuf = oldBrackBuf;
01749 if ( ( e->vflags & TOBEFACTORED ) != 0 )
01750     poly_factorize_expression(e);
01751 else if ( ( ( e->vflags & TOBEUNFACTORED ) != 0 )
01752     && ( ( e->vflags & ISFACTORIZED ) != 0 ) )
01753     poly_unfactorize_expression(e);
01754 if ( AT.S0->TermsLeft ) e->vflags &= ~ISZERO;
01755 else e->vflags |= ISZERO;
01756 if ( AR.expchanged == 0 ) e->vflags |= ISUNMODIFIED;
01757 if ( AT.S0->TermsLeft ) AR.expflags |= ISZERO;
01758 if ( AR.expchanged ) AR.expflags |= ISUNMODIFIED;
01759 AR.GetFile = 0;
01760 AT.bracketindexflag = oldbracketindexflag;
01761 /*
01762 Now copy the whole thing from fout to AR0.outfile
01763 Do this in one go to keep the lock occupied as short as possible
01764 */
01765 SeekScratch(fout,&outposition);
01766 LOCK(AS.outputslock);
01767 oldoutfile = AB[0]->R.outfile;
01768 if ( e->status == INTOHIDELEXPRESSION || e->status == INTOHIDEGEXPRESSION ) {
01769     AB[0]->R.outfile = AB[0]->R.hidefile;
01770 }
01771 SeekScratch(AB[0]->R.outfile,&position);
01772 e->onfile = position;
01773 if ( CopyExpression(fout,AB[0]->R.outfile) < 0 ) {
01774     AB[0]->R.outfile = oldoutfile;
01775     UNLOCK(AS.outputslock);
01776     MLOCK(ErrorMessageLock);
01777     MesPrint("Error copying output of 'InParallel' expression to master. Thread:
01778 %d",identity);
01779     MUNLOCK(ErrorMessageLock);
01780     goto ProcErr;
01781 }
01782 AB[0]->R.outfile = oldoutfile;
01783 AB[0]->R.hidefile->POfull = AB[0]->R.hidefile->POfill;
01784 AB[0]->R.expflags = AR.expflags;
01785 UNLOCK(AS.outputslock);
01786 if ( fout->handle >= 0 ) { /* Now get rid of the file */
01787     CloseFile(fout->handle);
01788     fout->handle = -1;
01789     remove(fout->name);
01790     PUTZERO(fout->POposition);
01791     PUTZERO(fout->filesize);
01792     fout->POfill = fout->POfull = fout->PObuffer;
01793 }
01794 UpdateMaxSize();
01795
01796 AT.WorkPointer = oldwork;
01797 } break;
01798 /*
01799 #] DOONEEXPRESSION :
01800

```

```

01801      #] DOBRACKETS :
01802
01803      In case we have a bracket index we can have the worker treat
01804      one or more of the entries in the bracket index.
01805      The advantage is that identical terms will meet each other
01806      sooner in the sorting and hence fewer compares will be needed.
01807      Also this way the master doesn't need to fill the buckets.
01808      The main problem is the load balancing which can become very
01809      bad when there is a long tail without things outside the bracket.
01810
01811      We get sent:
01812      1: The number of the first bracket to be done
01813      2: The number of the last bracket to be done
01814  */
01815      case DOBRACKETS: {
01816          BRACKETINFO *binfo;
01817          BRACKETINDEX *bi;
01818          FILEHANDLE *fi;
01819          POSITION stoppos, where;
01820          e = Expressions + AR.CurExpr;
01821          binfo = e->bracketinfo;
01822          thr = AN.threadbuck;
01823          bi = &(binfo->indexbuffer[thr->firstbracket]);
01824          if ( AR.GetFile == 2 ) fi = AR.hidefile;
01825          else fi = AR.infile;
01826          where = bi->start;
01827          ADD2POS (where, AS.OldOnFile[AR.CurExpr]);
01828          SetScratch(fi, &(where));
01829          stoppos = binfo->indexbuffer[thr->lastbracket].next;
01830          ADD2POS (stoppos, AS.OldOnFile[AR.CurExpr]);
01831          AN.ninterms = thr->firstterm;
01832  /*
01833      Now we have to put the 'value' of the bracket in the
01834      Compress buffer.
01835  */
01836          ttco = AR.CompressBuffer;
01837          tt = binfo->bracketbuffer + bi->bracket;
01838          i = tt;
01839          NCOPY(ttco, tt, i)
01840          AR.CompressPointer = ttco;
01841          term = AT.WorkPointer;
01842          while ( GetTerm(BHEAD term) ) {
01843              SeekScratch(fi, &where);
01844              AT.WorkPointer = term + *term;
01845              AN.IndDum = AM.IndDum;
01846              AR.CurDum = ReNumber(BHEAD term);
01847              if ( AC.SymChangeFlag ) MarkDirty(term, DIRTYSYMFLAG);
01848              if ( AN.ncmod ) {
01849                  if ( ( AC.modmode & ALSOFUNARGS ) != 0 ) MarkDirty(term, DIRTYFLAG);
01850                  else if ( AR.PolyFun ) PolyFunDirty(BHEAD term);
01851              }
01852              else if ( AC.PolyRatFunChanged ) PolyFunDirty(BHEAD term);
01853              if ( ( AR.PolyFunType == 2 ) && ( AC.PolyRatFunChanged == 0 )
01854                  && ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION ) ) {
01855                  PolyFunClean(BHEAD term);
01856              }
01857              if ( ( AP.PreDebug & THREADSDEBUG ) != 0 ) {
01858                  MLOCK(ErrorMessageLock);
01859                  MesPrint("Thread %w executing term:");
01860                  PrintTerm(term, "DoBrackets");
01861                  MUNLOCK(ErrorMessageLock);
01862              }
01863              AT.WorkPointer = term + *term;
01864              if ( Generator(BHEAD term, 0) ) {
01865                  LowerSortLevel();
01866                  MLOCK(ErrorMessageLock);
01867                  MesPrint("Error in processing one term in thread %d in module
01868                      %d", identity, AC.CModule);
01869                  MUNLOCK(ErrorMessageLock);
01870                  Terminate(-1);
01871              }
01872              AN.ninterms++;
01873              SetScratch(fi, &(where));
01874              if ( ISGEPOS(where, stoppos) ) break;
01875          }
01876          AT.WorkPointer = term;
01877          thr->free = BUCKETCOMINGFREE;
01878          break;
01879  /*
01880      #] DOBRACKETS :
01881      #] CLEARCLOCK :
01882
01883      The program only comes here after a .clear
01884  */
01885      case CLEARCLOCK:
01886  /*          LOCK(clearclocklock); */

```

```

01887         sumtimerinfo[identity] += TimeCPU(1);
01888         timerinfo[identity] = TimeCPU(0);
01889     /*         UNLOCK(clearclocklock); */
01890         break;
01891     /*
01892     #] CLEARCLOCK :
01893     #[ MCTSEXPANDTREE :
01894     */
01895     case MCTSEXPANDTREE:
01896         AT.optimtimes = AB[0]->T.optimtimes;
01897         find_Horner_MCTS_expand_tree();
01898         break;
01899     /*
01900     #] MCTSEXPANDTREE :
01901     #[ OPTIMIZEEXPRESSION :
01902     */
01903     case OPTIMIZEEXPRESSION:
01904         optimize_expression_given_Horner();
01905         break;
01906     /*
01907     #] OPTIMIZEEXPRESSION :
01908     */
01909     default:
01910         MLOCK(ErrorMessageLock);
01911         MesPrint("Illegal wakeup signal %d for thread %d",wakeupsignal,identity);
01912         MUNLOCK(ErrorMessageLock);
01913         Terminate(-1);
01914         break;
01915     }
01916     /* we need the following update in case we are using checkpoints. then we
01917        need to readjust the clocks when recovering using this information */
01918     timerinfo[identity] = TimeCPU(1);
01919 }
01920 EndOfThread;;
01921 /*
01922     This is the end of the thread. We cleanup and exit.
01923     If we are using flint, call the per-thread cleanup function. This keep valgrind happy.
01924     */
01925 #ifdef WITHFLINT
01926     flint_final_cleanup_thread();
01927 #endif
01928     FinalizeOneThread(identity);
01929     return(0);
01930 ProcErr:
01931     Terminate(-1);
01932     return(0);
01933 }
01934
01935 /*
01936     #] RunThread :
01937     #[ RunSortBot :
01938     */
01939 #ifdef WITHSORTBOTS
01940 void *RunSortBot(void *dummy)
01941 {
01942     int identity, wakeupsignal, identityretv;
01943     ALLPRIVATES *B, *BB;
01944     DUMMYUSE(dummy);
01945     identity = SetIdentity(&identityretv);
01946     threadpointers[identity] = pthread_self();
01947     B = InitializeOneThread(identity);
01948     while ( ( wakeupsignal = SortBotWait(identity) ) > 0 ) {
01949         switch ( wakeupsignal ) {
01950             /*
01951             #] INISORTBOT :
01952             */
01953             case INISORTBOT:
01954                 AR.CompressBuffer = AB[0]->R.CompressBuffer;
01955                 AR.ComprTop = AB[0]->R.ComprTop;
01956                 AR.CompressPointer = AB[0]->R.CompressPointer;
01957                 AR.CurExpr = AB[0]->R.CurExpr;
01958                 AR.PolyFun = AB[0]->R.PolyFun;
01959                 AR.PolyFunInv = AB[0]->R.PolyFunInv;
01960                 AR.PolyFunType = AB[0]->R.PolyFunType;
01961                 AR.PolyFunExp = AB[0]->R.PolyFunExp;
01962                 AR.PolyFunVar = AB[0]->R.PolyFunVar;
01963                 AR.PolyFunPow = AB[0]->R.PolyFunPow;
01964                 AR.SortType = AC.SortType;
01965                 if ( AR.PolyFun == 0 ) { AT.SS->PolyFlag = 0; }
01966                 else if ( AR.PolyFunType == 1 ) { AT.SS->PolyFlag = 1; }
01967                 else if ( AR.PolyFunType == 2 ) {
01968                     if ( AR.PolyFunExp == 2
01969                         || AR.PolyFunExp == 3 ) AT.SS->PolyFlag = 1;
01970                     else AT.SS->PolyFlag = 2;
01971                 }
01972                 AT.SS->PolyWise = 0;

```

```

01981         AN.ncmod = AC.ncmod;
01982         LOCK(AT.SB.MasterBlockLock[1]);
01983         BB = AB[AT.SortBotIn1];
01984         LOCK(BB->T.SB.MasterBlockLock[BB->T.SB.MasterNumBlocks]);
01985         BB = AB[AT.SortBotIn2];
01986         LOCK(BB->T.SB.MasterBlockLock[BB->T.SB.MasterNumBlocks]);
01987         AT.SB.FillBlock = 1;
01988         AT.SB.MasterFill[1] = AT.SB.MasterStart[1];
01989         SETBASEPOSITION(AN.theposition,0);
01990         break;
01991     /*
01992     #] INISORTBOT :
01993     #[ RUNSORTBOT :
01994 */
01995     case RUNSORTBOT:
01996         SortBotMerge(B);
01997         break;
01998 /*
01999     #] RUNSORTBOT :
02000     #[ TERMINATETHREAD :
02001 */
02002     case TERMINATETHREAD:
02003         goto EndOfThread;
02004 /*
02005     #] TERMINATETHREAD :
02006     #[ CLEARCLOCK :
02007
02008     The program only comes here after a .clear
02009 */
02010     case CLEARCLOCK:
02011         /* LOCK(clearclocklock); */
02012         sumtimerinfo[identity] += TimeCPU(1);
02013         timerinfo[identity] = TimeCPU(0);
02014         /* UNLOCK(clearclocklock); */
02015         break;
02016 /*
02017     #] CLEARCLOCK :
02018 */
02019     default:
02020         MLOCK(ErrorMessageLock);
02021         MesPrint("Illegal wakeup signal %d for thread %d",wakeupsignal,identity);
02022         MUNLOCK(ErrorMessageLock);
02023         Terminate(-1);
02024         break;
02025     }
02026 }
02027 EndOfThread;;
02028 /*
02029     This is the end of the thread. We cleanup and exit.
02030     If we are using flint, call the per-thread cleanup function. This keep valgrind happy.
02031 */
02032 #ifdef WITHFLINT
02033     flint_final_cleanup_thread();
02034 #endif
02035     FinalizeOneThread(identity);
02036     return(0);
02037 }
02038 #endif
02039
02040
02041 /*
02042     #] RunSortBot :
02043     #[ IAmAvailable :
02044 */
02045 void IAmAvailable(int identity)
02046 {
02047     int top;
02048     LOCK(availabilitylock);
02049     top = topofavailables;
02050     listofavailables[topofavailables++] = identity;
02051     if ( top == 0 ) {
02052         UNLOCK(availabilitylock);
02053         LOCK(wakeupmasterlock);
02054         wakeupmaster = identity;
02055         pthread_cond_signal(&wakeupmasterconditions);
02056         UNLOCK(wakeupmasterlock);
02057     }
02058     else {
02059         UNLOCK(availabilitylock);
02060     }
02061 }
02062
02063 /*
02064     #] IAmAvailable :
02065     #[ GetAvailableThread :
02066 */
02067 int GetAvailableThread(void)

```

```

02087 {
02088     int retval = -1;
02089     LOCK(availabilitylock);
02090     if ( topofavailables > 0 ) retval = listofavailables[--topofavailables];
02091     UNLOCK(availabilitylock);
02092     if ( retval >= 0 ) {
02093 /*
02094         Make sure the thread is indeed waiting and not between
02095         saying that it is available and starting to wait.
02096 */
02097         LOCK(wakeuplocks[retval]);
02098         UNLOCK(wakeuplocks[retval]);
02099     }
02100     return(retval);
02101 }
02102
02103 /*
02104 #] GetAvailableThread :
02105 #[ ConditionalGetAvailableThread :
02106 */
02114 int ConditionalGetAvailableThread(void)
02115 {
02116     int retval = -1;
02117     if ( topofavailables > 0 ) {
02118         LOCK(availabilitylock);
02119         if ( topofavailables > 0 ) {
02120             retval = listofavailables[--topofavailables];
02121         }
02122         UNLOCK(availabilitylock);
02123         if ( retval >= 0 ) {
02124 /*
02125             Make sure the thread is indeed waiting and not between
02126             saying that it is available and starting to wait.
02127 */
02128             LOCK(wakeuplocks[retval]);
02129             UNLOCK(wakeuplocks[retval]);
02130         }
02131     }
02132     return(retval);
02133 }
02134
02135 /*
02136 #] ConditionalGetAvailableThread :
02137 #[ GetThread :
02138 */
02148 int GetThread(int identity)
02149 {
02150     int retval = -1, j;
02151     LOCK(availabilitylock);
02152     for ( j = 0; j < topofavailables; j++ ) {
02153         if ( identity == listofavailables[j] ) break;
02154     }
02155     if ( j < topofavailables ) {
02156         --topofavailables;
02157         for ( ; j < topofavailables; j++ ) {
02158             listofavailables[j] = listofavailables[j+1];
02159         }
02160         retval = identity;
02161     }
02162     UNLOCK(availabilitylock);
02163     return(retval);
02164 }
02165
02166 /*
02167 #] GetThread :
02168 #[ ThreadWait :
02169 */
02179 int ThreadWait(int identity)
02180 {
02181     int retval, top, j;
02182     LOCK(wakeuplocks[identity]);
02183     LOCK(availabilitylock);
02184     top = topofavailables;
02185     for ( j = topofavailables; j > 0; j-- )
02186         listofavailables[j] = listofavailables[j-1];
02187     listofavailables[0] = identity;
02188     topofavailables++;
02189     if ( top == 0 || topofavailables == numberofworkers ) {
02190         UNLOCK(availabilitylock);
02191         LOCK(wakeupmasterlock);
02192         wakeupmaster = identity;
02193         pthread_cond_signal(&wakeupmasterconditions);
02194         UNLOCK(wakeupmasterlock);
02195     }
02196     else {
02197         UNLOCK(availabilitylock);
02198     }

```



```

02199     while ( wakeup[identity] == 0 ) {
02200         pthread_cond_wait (&(wakeupconditions[identity]), &(wakeuplocks[identity]));
02201     }
02202     retval = wakeup[identity];
02203     wakeup[identity] = 0;
02204     UNLOCK(wakeuplocks[identity]);
02205     return(retval);
02206 }
02207
02208 /*
02209  #] ThreadWait :
02210  #[ SortBotWait :
02211  */
02212
02213 #ifdef WITHSORTBOTS
02214 int SortBotWait(int identity)
02215 {
02216     int retval;
02217     LOCK(wakeuplocks[identity]);
02218     LOCK(availabilitylock);
02219     topsortbotavailables++;
02220     if ( topsortbotavailables >= numberofsortbots ) {
02221         UNLOCK(availabilitylock);
02222         LOCK(wakeupsortbotlock);
02223         wakeupmaster = identity;
02224         pthread_cond_signal (&wakeupsortbotconditions);
02225         UNLOCK(wakeupsortbotlock);
02226     }
02227     else {
02228         UNLOCK(availabilitylock);
02229     }
02230     while ( wakeup[identity] == 0 ) {
02231         pthread_cond_wait (&(wakeupconditions[identity]), &(wakeuplocks[identity]));
02232     }
02233     retval = wakeup[identity];
02234     wakeup[identity] = 0;
02235     UNLOCK(wakeuplocks[identity]);
02236     return(retval);
02237 }
02238 #endif
02239
02240 /*
02241  #] SortBotWait :
02242  #[ ThreadClaimedBlock :
02243  */
02244 int ThreadClaimedBlock(int identity)
02245 {
02246     LOCK(availabilitylock);
02247     numberclaimed++;
02248     if ( numberclaimed >= numberofworkers ) {
02249         UNLOCK(availabilitylock);
02250         LOCK(wakeupmasterlock);
02251         wakeupmaster = identity;
02252         pthread_cond_signal (&wakeupmasterconditions);
02253         UNLOCK(wakeupmasterlock);
02254     }
02255     else {
02256         UNLOCK(availabilitylock);
02257     }
02258     return(0);
02259 }
02260
02261 /*
02262  #] ThreadClaimedBlock :
02263  #[ MasterWait :
02264  */
02265 int MasterWait(void)
02266 {
02267     int retval;
02268     LOCK(wakeupmasterlock);
02269     while ( wakeupmaster == 0 ) {
02270         pthread_cond_wait (&wakeupmasterconditions, &wakeupmasterlock);
02271     }
02272     retval = wakeupmaster;
02273     wakeupmaster = 0;
02274     UNLOCK(wakeupmasterlock);
02275     return(retval);
02276 }
02277
02278 /*
02279  #] MasterWait :
02280  #[ MasterWaitThread :
02281  */
02282 int MasterWaitThread(int identity)
02283 {
02284     int retval;

```

```

02318     LOCK(wakeupmasterthreadlocks[identity]);
02319     while ( wakeupmasterthread[identity] == 0 ) {
02320         pthread_cond_wait(&(wakeupmasterthreadconditions[identity])
02321             , &(wakeupmasterthreadlocks[identity]));
02322     }
02323     retval = wakeupmasterthread[identity];
02324     wakeupmasterthread[identity] = 0;
02325     UNLOCK(wakeupmasterthreadlocks[identity]);
02326     return(retval);
02327 }
02328
02329 /*
02330  #] MasterWaitThread :
02331  #[ MasterWaitAll :
02332  */
02333 void MasterWaitAll(void)
02334 {
02335     LOCK(wakeupmasterlock);
02336     while ( topofavailables < numberofworkers ) {
02337         pthread_cond_wait(&(wakeupmasterconditions), &(wakeupmasterlock));
02338     }
02339     UNLOCK(wakeupmasterlock);
02340     return;
02341 }
02342
02343 /*
02344  #] MasterWaitAll :
02345  #[ MasterWaitAllSortBots :
02346  */
02347 void MasterWaitAllSortBots(void)
02348 {
02349     LOCK(wakeupsortbotlock);
02350     while ( topsortbotavailables < numberofsortbots ) {
02351         pthread_cond_wait(&(wakeupsortbotconditions), &(wakeupsortbotlock));
02352     }
02353     UNLOCK(wakeupsortbotlock);
02354     return;
02355 }
02356
02357 #endif
02358
02359 /*
02360  #] MasterWaitAllSortBots :
02361  #[ MasterWaitAllBlocks :
02362  */
02363 void MasterWaitAllBlocks(void)
02364 {
02365     LOCK(wakeupmasterlock);
02366     while ( numberclaimed < numberofworkers ) {
02367         pthread_cond_wait(&(wakeupmasterconditions), &(wakeupmasterlock));
02368     }
02369     UNLOCK(wakeupmasterlock);
02370     return;
02371 }
02372
02373 /*
02374  #] MasterWaitAllBlocks :
02375  #[ WakeupThread :
02376  */
02377 void WakeupThread(int identity, int signalnumber)
02378 {
02379     if ( signalnumber == 0 ) {
02380         MLOCK(ErrorMessageLock);
02381         MesPrint("Illegal wakeup signal for thread %d", identity);
02382         MUNLOCK(ErrorMessageLock);
02383         Terminate(-1);
02384     }
02385     LOCK(wakeuplocks[identity]);
02386     wakeup[identity] = signalnumber;
02387     pthread_cond_signal(&(wakeupconditions[identity]));
02388     UNLOCK(wakeuplocks[identity]);
02389 }
02390
02391 /*
02392  #] WakeupThread :
02393  #[ WakeupMasterFromThread :
02394  */
02395 void WakeupMasterFromThread(int identity, int signalnumber)
02396 {
02397     if ( signalnumber == 0 ) {
02398         MLOCK(ErrorMessageLock);
02399         MesPrint("Illegal wakeup signal for master %d", identity);
02400         MUNLOCK(ErrorMessageLock);
02401         Terminate(-1);
02402     }

```

```

02438     }
02439     LOCK(wakeupmasterthreadlocks[identity]);
02440     wakeupmasterthread[identity] = signalnumber;
02441     pthread_cond_signal(&(wakeupmasterthreadconditions[identity]));
02442     UNLOCK(wakeupmasterthreadlocks[identity]);
02443 }
02444
02445 /*
02446  #] WakeupMasterFromThread :
02447  #[ SendOneBucket :
02448 */
02449 int SendOneBucket(int type)
02450 {
02451     ALLPRIVATES *B0 = AB[0];
02452     THREADBUCKET *thr = 0;
02453     int j, k, id;
02454     for ( j = 0; j < numthreadbuckets; j++ ) {
02455         if ( threadbuckets[j]->free == BUCKETFILLED ) {
02456             thr = threadbuckets[j];
02457             for ( k = j+1; k < numthreadbuckets; k++ )
02458                 threadbuckets[k-1] = threadbuckets[k];
02459             threadbuckets[numthreadbuckets-1] = thr;
02460             break;
02461         }
02462     }
02463     AN0.ninterms++;
02464     while ( ( id = GetAvailableThread() ) < 0 ) { MasterWait(); }
02465 /*
02466  Prepare the thread. Give it the term and variables.
02467 */
02468     LoadOneThread(0,id,thr,0);
02469     thr->busy = BUCKETASSIGNED;
02470     thr->free = BUCKETINUSE;
02471     numberoffullbuckets--;
02472 /*
02473  And signal the thread to run.
02474  Form now on we may only interfere with this bucket
02475  1: after it has been marked BUCKETCOMINGFREE
02476  2: when thr->busy == BUCKETDOINGTERM and then only when protected by
02477     thr->lock. This would be for load balancing.
02478 */
02479     WakeupThread(id,type);
02480 /* AN0.ninterms += thr->ddterms; */
02481     return(0);
02482 }
02483
02484 /*
02485  #] SendOneBucket :
02486  #[ InParallelProcessor :
02487 */
02488 int InParallelProcessor(void)
02489 {
02490     GETIDENTITY
02491     int i, id, retval = 0, num = 0;
02492     EXPRESSIONS e;
02493     if ( numberofworkers >= 2 ) {
02494         SetWorkerFiles();
02495         for ( i = 0; i < NumExpressions; i++ ) {
02496             e = Expressions+i;
02497             if ( e->partodo <= 0 ) continue;
02498             if ( e->status == LOCALEXPRESSION || e->status == GLOBALEXPRESSION
02499                 || e->status == UNHIDELEXPRESSION || e->status == UNHIDEGEXPRESSION
02500                 || e->status == INTOHIDELEXPRESSION || e->status == INTOHIDEGEXPRESSION ) {
02501                 else {
02502                     e->partodo = 0;
02503                     continue;
02504                 }
02505             if ( e->counter == 0 ) { /* Expression with zero terms */
02506                 e->partodo = 0;
02507                 continue;
02508             }
02509             /*
02510              This expression should go to an idle worker
02511             */
02512             while ( ( id = GetAvailableThread() ) < 0 ) { MasterWait(); }
02513             LoadOneThread(0,id,0,-1);
02514             AB[id]->R.exprtudo = i;
02515             WakeupThread(id,DOONEEXPRESSION);
02516             num++;
02517         }
02518     }
02519 /*
02520  Now we have to wait for all workers to finish
02521 */
02522     if ( num > 0 ) MasterWaitAll();
02523
02524     if ( AC.CollectFun ) AR.DeferFlag = 0;

```

```

02549     }
02550     else {
02551         for ( i = 0; i < NumExpressions; i++ ) {
02552             Expressions[i].partodo = 0;
02553         }
02554     }
02555     return(retval);
02556 }
02557
02558 /*
02559  #] InParallelProcessor :
02560  #[ ThreadsProcessor :
02561 */
02562 int ThreadsProcessor(EXPRESSIONS e, WORD LastExpression, WORD fromspectator)
02563 {
02564     ALLPRIVATES *B0 = AB[0], *B = B0;
02565     int id, oldgzipCompress, endofinput = 0, j, still, k, defcount = 0, bra = 0, first = 1;
02566     LONG dd = 0, ddd, thrbufsiz, thrbufsiz0, thrbufsiz2, numbucket = 0, numpasses;
02567     LONG num, i;
02568     WORD *oldworkpointer = AT0.WorkPointer, *tt, *ttco = 0, *tl = 0, ter, *tstop = 0, *t2;
02569     THREADBUCKET *thr = 0;
02570     FILEHANDLE *oldoutfile = AR0.outfile;
02571     GETTERM GetTermP = &GetTerm;
02572     POSITION eonfile = AS.OldOnFile[e-Expressions];
02573     numberoffullbuckets = 0;
02574
02575     /*
02576      Start up all threads. The lock needs to be around the whole loop
02577      to keep processes from terminating quickly and putting themselves
02578      in the list of available threads again.
02579     */
02580     AM.tracebackflag = 1;
02581
02582     AS.sLevel = AR0.sLevel;
02583     LOCK(availabilitylock);
02584     topofavailables = 0;
02585     for ( id = 1; id <= numberofworkers; id++ ) {
02586         WakeupThread(id, STARTNEWEXPRESSION);
02587     }
02588     UNLOCK(availabilitylock);
02589     NewSort(BHEAD0);
02590     AN0.ninterms = 1;
02591
02592     /*
02593      Now for redefine
02594     */
02595     if ( AC.numpfirstnum > 0 ) {
02596         for ( j = 0; j < AC.numpfirstnum; j++ ) {
02597             AC.inputnumbers[j] = -1;
02598         }
02599     }
02600     MasterWaitAll();
02601
02602     /*
02603      Determine a reasonable bucket size.
02604      This is based on the value of AC.ThreadBucketSize and the number
02605      of terms. We want at least 5 buckets per worker at the moment.
02606      Some research should show whether this is reasonable.
02607
02608      The number of terms in the expression is in e->counter
02609     */
02610     thrbufsiz2 = thrbufsiz = AC.ThreadBucketSize-1;
02611     if ( ( e->counter / ( numberofworkers * 5 ) ) < thrbufsiz ) {
02612         thrbufsiz = e->counter / ( numberofworkers * 5 ) - 1;
02613         if ( thrbufsiz < 0 ) thrbufsiz = 0;
02614     }
02615     thrbufsiz0 = thrbufsiz;
02616     numpasses = 5; /* this is just for trying */
02617     thrbufsiz = thrbufsiz0 / ( 2 << numpasses );
02618
02619     /*
02620      Mark all buckets as free and take the first.
02621     */
02622     for ( j = 0; j < numthreadbuckets; j++ )
02623         threadbuckets[j]->free = BUCKETFREE;
02624     thr = threadbuckets[0];
02625
02626     /*
02627      #[ Whole brackets :
02628
02629      First we look whether we have to work with entire brackets
02630      This is the case when there is a non-NULL address in e->bracketinfo.
02631      Of course we shouldn't have interference from a collect or keep statement.
02632     */
02633     #ifdef WHOLEBRACKETS
02634     if ( e->bracketinfo && AC.CollectFun == 0 && AR0.DeferFlag == 0 ) {
02635         FILEHANDLE *curfile;
02636         int didone = 0;
02637         LONG num, n;
02638         AN0.expr = e;
02639         for ( n = 0; n < e->bracketinfo->indexfill; n++ ) {
02640             num = TreatIndexEntry(B0,n);

```

```

02660         if ( num > 0 ) {
02661             didone = 1;
02662         /*
02663             This bracket can be sent off.
02664             1: Look for an empty bucket
02665         */
02666         ReTry;;
02667         for ( j = 0; j < numthreadbuckets; j++ ) {
02668             switch ( threadbuckets[j]->free ) {
02669                 case BUCKETFREE:
02670                     thr = threadbuckets[j];
02671                     goto Found1;
02672                 case BUCKETCOMINGFREE:
02673                     thr = threadbuckets[j];
02674                     thr->free = BUCKETFREE;
02675                     for ( k = j+1; k < numthreadbuckets; k++ )
02676                         threadbuckets[k-1] = threadbuckets[k];
02677                     threadbuckets[numthreadbuckets-1] = thr;
02678                     j--;
02679                     break;
02680                 default:
02681                     break;
02682             }
02683         }
02684         Found1;;
02685         if ( j < numthreadbuckets ) {
02686             /*
02687                 Found an empty bucket. Fill it.
02688             */
02689             thr->firstbracket = n;
02690             thr->lastbracket = n + num - 1;
02691             thr->type = BUCKETDOINGBRACKET;
02692             thr->free = BUCKETFILLED;
02693             thr->firstterm = AN0.ninterms;
02694             for ( j = n; j < n+num; j++ ) {
02695                 AN0.ninterms += e->bracketinfo->indexbuffer[j].termsinbracket;
02696             }
02697             n += num-1;
02698             numberoffullbuckets++;
02699             if ( topofavailables > 0 ) {
02700                 SendOneBucket(DOBRACKETS);
02701             }
02702         }
02703         /*
02704             All buckets are in use.
02705             Look/wait for an idle worker. Give it a bucket.
02706             After that, retry for a bucket
02707         */
02708         else {
02709             while ( topofavailables <= 0 ) {
02710                 MasterWait();
02711             }
02712             SendOneBucket(DOBRACKETS);
02713             goto ReTry;
02714         }
02715     }
02716 }
02717 if ( didone ) {
02718     /*
02719         And now put the input back in the original position.
02720     */
02721     switch ( e->status ) {
02722         case UNHIDELEXPRESSION:
02723         case UNHIDEGEXPRESSION:
02724         case DROPHLEXPRESSION:
02725         case DROPHGEXPRESSION:
02726         case HIDDENLEXPRESSION:
02727         case HIDDENGEXPRESSION:
02728             curfile = AR0.hidefile;
02729             break;
02730         default:
02731             curfile = AR0.infile;
02732             break;
02733     }
02734     SetScratch(curfile,&eonfile);
02735     GetTerm(B0,AT0.WorkPointer);
02736     /*
02737         Now we point the GetTerm that is used to the one that is selective
02738     */
02739     GetTermP = &GetTerm2;
02740     /*
02741         Next wait till there is a bucket available and initialize thr to it.
02742     */
02743     for(;;) {
02744         for ( j = 0; j < numthreadbuckets; j++ ) {
02745             switch ( threadbuckets[j]->free ) {
02746                 case BUCKETFREE:

```

```

02747         thr = threadbuckets[j];
02748         goto Found2;
02749     case BUCKETCOMINGFREE:
02750         thr = threadbuckets[j];
02751         thr->free = BUCKETFREE;
02752         for ( k = j+1; k < numthreadbuckets; k++ )
02753             threadbuckets[k-1] = threadbuckets[k];
02754         threadbuckets[numthreadbuckets-1] = thr;
02755         j--;
02756         break;
02757     default:
02758         break;
02759     }
02760     while ( topofavailables <= 0 ) {
02761         MasterWait();
02762     }
02763     while ( topofavailables > 0 && numberoffullbuckets > 0 ) {
02764         SendOneBucket (DOBRACKETS);
02765     }
02766 }
02767 Found2:;
02768 while ( numberoffullbuckets > 0 ) {
02769     while ( topofavailables <= 0 ) {
02770         MasterWait();
02771     }
02772     while ( topofavailables > 0 && numberoffullbuckets > 0 ) {
02773         SendOneBucket (DOBRACKETS);
02774     }
02775 }
02776
02777 /*
02778     Disable the 'warming up' with smaller buckets.
02779
02780     numpasses = 0;
02781     thrbufsiz = thrbufsiz0;
02782 */
02783     AN0.lastinindex = -1;
02784 }
02785 MasterWaitAll();
02786 }
02787 #endif
02788 /*
02789     #] Whole brackets :
02790
02791     Now the loop to start a bucket
02792 */
02793     for(;;) {
02794         if ( fromspectator ) {
02795             ter = GetFromSpectator(thr->threadbuffer,fromspectator-1);
02796             if ( ter == 0 ) fromspectator = 0;
02797         }
02798         else {
02799             ter = GetTermP(B0,thr->threadbuffer);
02800         }
02801         /*
02802             At this point we could check whether the input term is
02803             just an expression that resides in a scratch file.
02804             If this is the case we should store the current input info
02805             (file and buffer content) and redirect the input.
02806             At the end we can go back to where we were.
02807             There are two possibilities: we are in the same scratchfile
02808             as the main input and the other expression is still in the
02809             input buffer, or we have to do some reading. The reading is
02810             of course done in GetTerm.
02811
02812             How to set this up can also be studied in TestSub (in file proces.c)
02813             where it checks for EXPRESSION.
02814         */
02815         if ( ter < 0 ) break;
02816         if ( ter == 0 ) { endofinput = 1; goto Finalize; }
02817         dd = AN0.deferskipped;
02818         if ( AR0.DeferFlag ) {
02819             defcount = 0;
02820             thr->deferbuffer[defcount++] = AR0.DefPosition;
02821             ttco = thr->compressbuffer; t1 = AR0.CompressBuffer; j = *t1;
02822             NCOPY(ttco,t1,j);
02823         }
02824         else if ( first && ( AC.CollectFun == 0 ) ) { /* Brackets ? */
02825             first = 0;
02826             t1 = tstop = thr->threadbuffer;
02827             tstop += *tstop; tstop -= ABS(tstop[-1]);
02828             t1++;
02829             while ( t1 < tstop ) {
02830                 if ( t1[0] == HAAKJE ) { bra = 1; break; }
02831                 t1 += t1[1];
02832             }
02833             t1 = thr->threadbuffer;

```

```

02834     }
02835     /*
02836     Check whether we have a collect,function going. If so execute it.
02837     */
02838     if ( AC.CollectFun && *(thr->threadbuffer) < (AM.MaxTer/((LONG)sizeof(WORD))-10) ) {
02839         if ( ( dd = GetMoreTerms(thr->threadbuffer) ) < 0 ) {
02840             LowerSortLevel(); goto ProcErr;
02841         }
02842     }
02843     /*
02844     Check whether we have a priority task:
02845     */
02846     if ( topofavailables > 0 && numberoffullbuckets > 0 ) SendOneBucket(LOWESTLEVELGENERATION);
02847     /*
02848     Now put more terms in the bucket. Position tt after the first term
02849     */
02850     tt = thr->threadbuffer; tt += *tt;
02851     thr->totnum = 1;
02852     thr->usenum = 0;
02853     /*
02854     Next we worry about the 'slow startup' in which we make the initial
02855     buckets smaller, so that we get all threads busy as soon as possible.
02856     */
02857     if ( numpasses > 0 ) {
02858         numbucket++;
02859         if ( numbucket >= numberofworkers ) {
02860             numbucket = 0;
02861             numpasses--;
02862             if ( numpasses == 0 ) thrbufsiz = thrbufsiz0;
02863             else thrbufsiz = thrbufsiz0 / (2 « numpasses);
02864         }
02865         thrbufsiz2 = thrbufsiz + thrbufsiz/5; /* for completing brackets */
02866     }
02867     /*
02868     we have already 1+dd terms
02869     */
02870     while ( ( dd < thrbufsiz ) &&
02871         ( tt - thr->threadbuffer ) < ( thr->threadbuffersize - AM.MaxTer/((LONG)sizeof(WORD)) - 2
02872     ) ) {
02873         /*
02874         First check:
02875         */
02876         if ( topofavailables > 0 && numberoffullbuckets > 0 )
02877             SendOneBucket(LOWESTLEVELGENERATION);
02878         /*
02879         There is room in the bucket. Fill yet another term.
02880         */
02881         if ( GetTermP(B0,tt) == 0 ) { endofinput = 1; break; }
02882         dd++;
02883         thr->totnum++;
02884         dd += AN0.deferskipped;
02885         if ( AR0.DeferFlag ) {
02886             thr->deferbuffer[defcount++] = AR0.DefPosition;
02887             t1 = AR0.CompressBuffer; j = *t1;
02888             NCOPY(ttco,t1,j);
02889         }
02890         if ( AC.CollectFun && *tt < (AM.MaxTer/((LONG)sizeof(WORD))-10) ) {
02891             if ( ( ddd = GetMoreTerms(tt) ) < 0 ) {
02892                 LowerSortLevel(); goto ProcErr;
02893             }
02894             dd += ddd;
02895             t1 = tt;
02896             tt += *tt;
02897         }
02898         /*
02899         Check whether there are regular brackets and if we have no DeferFlag
02900         and no collect, we try to add more terms till we finish the current
02901         bracket. We should however not overdo it. Let us say: up to 20%
02902         more terms are allowed.
02903         */
02904         if ( bra ) {
02905             tstop = t1 + *t1; tstop -= ABS(tstop[-1]);
02906             t2 = t1+1;
02907             while ( t2 < tstop ) {
02908                 if ( t2[0] == HAAKJE ) { break; }
02909                 t2 += t2[1];
02910             }
02911             if ( t2[0] == HAAKJE ) {
02912                 t2 += t2[1]; num = t2 - t1;
02913                 while ( ( dd < thrbufsiz2 ) &&
02914                     ( tt - thr->threadbuffer ) < ( thr->threadbuffersize - AM.MaxTer - 2 ) ) {
02915                     /*
02916                     First check:
02917                     */
02918                     if ( topofavailables > 0 && numberoffullbuckets > 0 )
02919                         SendOneBucket(LOWESTLEVELGENERATION);

```

```

02918 /*
02919         There is room in the bucket. Fill yet another term.
02920 */
02921         if ( GetTermP(B0,tt) == 0 ) { endofinput = 1; break; }
02922 /*
02923         Same bracket?
02924 */
02925         tstop = tt + *tt; tstop -= ABS(tstop[-1]);
02926         if ( tstop-tt < num ) { /* Different: abort */
02927             AR0.KeptInHold = 1;
02928             break;
02929         }
02930         for ( i = 1; i < num; i++ ) {
02931             if ( t1[i] != tt[i] ) break;
02932         }
02933         if ( i < num ) { /* Different: abort */
02934             AR0.KeptInHold = 1;
02935             break;
02936         }
02937 /*
02938         Same bracket. We need this term.
02939 */
02940         dd++;
02941         thr->totnum++;
02942         tt += *tt;
02943     }
02944 }
02945 }
02946 thr->ddterms = dd; /* total number of terms including keep brackets */
02947 thr->firstterm = AN0.ninterms;
02948 AN0.ninterms += dd;
02949 *tt = 0; /* mark end of bucket */
02950 thr->free = BUCKETFILLED;
02951 thr->type = BUCKETDOINGTERMS;
02952 numberoffullbuckets++;
02953 if ( topofavailables <= 0 && endofinput == 0 ) {
02954 /*
02955         Problem: topofavailables may already be > 0, but the
02956         thread has not yet gone into waiting. Can the signal get lost?
02957         How can we tell that a thread is waiting for a signal?
02958
02959         All threads are busy. Try to load up another bucket.
02960         In the future we could be more sophisticated.
02961         At the moment we load a complete bucket which could be
02962         1000 terms or even more.
02963         In principle it is better to keep a full bucket ready
02964         and check after each term we put in the next bucket. That
02965         way we don't waste time of the workers.
02966 */
02967         for ( j = 0; j < numthreadbuckets; j++ ) {
02968             switch ( threadbuckets[j]->free ) {
02969                 case BUCKETFREE:
02970                     thr = threadbuckets[j];
02971                     if ( !endofinput ) goto NextBucket;
02972 /*
02973                     If we are at the end of the input we mark
02974                     the free buckets in a special way. That way
02975                     we don't keep running into them.
02976 */
02977                     thr->free = BUCKETATEND;
02978                     break;
02979                 case BUCKETCOMINGFREE:
02980                     thr = threadbuckets[j];
02981                     thr->free = BUCKETFREE;
02982 /*
02983                     Bucket has just been finished.
02984                     Put at the end of the list. We don't want
02985                     an early bucket to wait to be treated last.
02986 */
02987                     for ( k = j+1; k < numthreadbuckets; k++ )
02988                         threadbuckets[k-1] = threadbuckets[k];
02989                     threadbuckets[numthreadbuckets-1] = thr;
02990                     j--; /* we have to redo the same number j. */
02991                     break;
02992                 default:
02993                     break;
02994             }
02995         }
02996 /*
02997         We have no free bucket or we are at the end.
02998         The only thing we can do now is wait for a worker to come free,
02999         provided there are still buckets to send.
03000 */
03001     }
03002 /*
03003         Look for the next bucket to send. There is at least one full bucket!
03004 */

```



```

03005         for ( j = 0; j < numthreadbuckets; j++ ) {
03006             if ( threadbuckets[j]->free == BUCKETFILLED ) {
03007                 thr = threadbuckets[j];
03008                 for ( k = j+1; k < numthreadbuckets; k++ )
03009                     threadbuckets[k-1] = threadbuckets[k];
03010                 threadbuckets[numthreadbuckets-1] = thr;
03011                 break;
03012             }
03013         }
03014     /*
03015         Wait for a thread to become available
03016         The bucket we are going to use is in thr.
03017     */
03018 DoBucket:;
03019     AN0.ninterms++;
03020     while ( ( id = GetAvailableThread() ) < 0 ) { MasterWait(); }
03021     /*
03022         Prepare the thread. Give it the term and variables.
03023     */
03024     LoadOneThread(0,id,thr,0);
03025     LOCK(thr->lock);
03026     thr->busy = BUCKETASSIGNED;
03027     UNLOCK(thr->lock);
03028     thr->free = BUCKETINUSE;
03029     numberoffullbuckets--;
03030     /*
03031         And signal the thread to run.
03032         From now on we may only interfere with this bucket
03033         1: after it has been marked BUCKETCOMINGFREE
03034         2: when thr->busy == BUCKETDOINGTERM and then only when protected by
03035            thr->lock. This would be for load balancing.
03036     */
03037     WakeupThread(id,LOWESTLEVELGENERATION);
03038     /* AN0.ninterms += thr->ddterms; */
03039     /*
03040         Now look whether there is another bucket filled and a worker available
03041     */
03042     if ( topofavailables > 0 ) { /* there is a worker */
03043         for ( j = 0; j < numthreadbuckets; j++ ) {
03044             if ( threadbuckets[j]->free == BUCKETFILLED ) {
03045                 thr = threadbuckets[j];
03046                 for ( k = j+1; k < numthreadbuckets; k++ )
03047                     threadbuckets[k-1] = threadbuckets[k];
03048                 threadbuckets[numthreadbuckets-1] = thr;
03049                 goto DoBucket; /* and we found a bucket */
03050             }
03051         }
03052     /*
03053         no bucket is loaded but there is a thread available
03054         find a bucket to load. If there is none (all are USED or ATEND)
03055         we jump out of the loop.
03056     */
03057         for ( j = 0; j < numthreadbuckets; j++ ) {
03058             switch ( threadbuckets[j]->free ) {
03059                 case BUCKETFREE:
03060                     thr = threadbuckets[j];
03061                     if ( !endofinput ) goto NextBucket;
03062                     thr->free = BUCKETATEND;
03063                     break;
03064                 case BUCKETCOMINGFREE:
03065                     thr = threadbuckets[j];
03066                     if ( endofinput ) {
03067                         thr->free = BUCKETATEND;
03068                     }
03069                     else {
03070                         thr->free = BUCKETFREE;
03071                         for ( k = j+1; k < numthreadbuckets; k++ )
03072                             threadbuckets[k-1] = threadbuckets[k];
03073                         threadbuckets[numthreadbuckets-1] = thr;
03074                         j--;
03075                     }
03076                     break;
03077                 default:
03078                     break;
03079             }
03080         }
03081         if ( j >= numthreadbuckets ) break;
03082     }
03083     else {
03084     /*
03085         No worker available.
03086         Look for a bucket to load.
03087         Its number will be in "still"
03088     */
03089 Finalize:;
03090         still = -1;
03091         for ( j = 0; j < numthreadbuckets; j++ ) {

```

```

03092         switch ( threadbuckets[j]->free ) {
03093             case BUCKETFREE:
03094                 thr = threadbuckets[j];
03095                 if ( !endofinput ) goto NextBucket;
03096                 thr->free = BUCKETATEND;
03097                 break;
03098             case BUCKETCOMINGFREE:
03099                 thr = threadbuckets[j];
03100                 if ( endofinput ) thr->free = BUCKETATEND;
03101                 else {
03102                     thr->free = BUCKETFREE;
03103                     for ( k = j+1; k < numthreadbuckets; k++ )
03104                         threadbuckets[k-1] = threadbuckets[k];
03105                     threadbuckets[numthreadbuckets-1] = thr;
03106                     j--;
03107                 }
03108                 break;
03109             case BUCKETFILLED:
03110                 if ( still < 0 ) still = j;
03111                 break;
03112             default:
03113                 break;
03114         }
03115     }
03116     if ( still < 0 ) {
03117         /*
03118             No buckets to be executed and no buckets FREE.
03119             We must be at the end. Break out of the loop.
03120         */
03121         break;
03122     }
03123     thr = threadbuckets[still];
03124     for ( k = still+1; k < numthreadbuckets; k++ )
03125         threadbuckets[k-1] = threadbuckets[k];
03126     threadbuckets[numthreadbuckets-1] = thr;
03127     goto DoBucket;
03128 }
03129 NextBucket:;
03130 }
03131 /*
03132     Now the stage one load balancing.
03133     If the load has been readjusted we have again filled buckets.
03134     In that case we jump back in the loop.
03135
03136     Tricky point: when do the workers see the new value of AT.LoadBalancing?
03137     It should activate the locks on thr->busy
03138 */
03139 if ( AC.ThreadBalancing ) {
03140     for ( id = 1; id <= numberofworkers; id++ ) {
03141         AB[id]->T.LoadBalancing = 1;
03142     }
03143     if ( LoadReadjusted() ) goto Finalize;
03144     for ( id = 1; id <= numberofworkers; id++ ) {
03145         AB[id]->T.LoadBalancing = 0;
03146     }
03147 }
03148 if ( AC.ThreadBalancing ) {
03149     /*
03150         The AS.Balancing flag should have Generator look for
03151         free workers and apply the "buro" method.
03152
03153         There is still a serious problem.
03154         When for instance a sum_, there may be space created in a local
03155         compiler buffer for a wildcard substitution or whatever.
03156         Compiler buffer execution scribble space.....
03157         This isn't copied along?
03158         Look up ebufnum. There are 12 places with AddRHS!
03159         Problem: one process allocates in ebuf. Then term is given to
03160         other process. It would like to use from this ebuf, but the sender
03161         finishes first and removes the ebuf (and/or overwrites it).
03162
03163         Other problem: local $ variables aren't copied along.
03164     */
03165     AS.Balancing = 0;
03166 }
03167 MasterWaitAll();
03168 AS.Balancing = 0;
03169 /*
03170     When we deal with the last expression we can now remove the input
03171     scratch file. This saves potentially much disk space (up to 1/3)
03172 */
03173 if ( LastExpression ) {
03174     UpdateMaxSize();
03175     if ( AR0.infile->handle >= 0 ) {
03176         CloseFile(AR0.infile->handle);
03177         AR0.infile->handle = -1;
03178         remove(AR0.infile->name);

```

```

03179         PUTZERO(AR0.infile->PPosition);
03180         AR0.infile->POfill = AR0.infile->POfull = AR0.infile->PObuffer;
03181     }
03182 }
03183 /*
03184 We order the threads to finish in the MasterMerge routine
03185 It will start with waiting for all threads to finish.
03186 One could make an administration in which threads that have
03187 finished can start already with the final sort but
03188 1: The load balancing should not make this super urgent
03189 2: It would definitely not be very compatible with the second
03190 stage load balancing.
03191 */
03192 oldgzipCompress = AR0.gzipCompress;
03193 AR0.gzipCompress = 0;
03194 if ( AR0.outtohide ) AR0.outfile = AR0.hidefile;
03195 if ( MasterMerge() < 0 ) {
03196     if ( AR0.outtohide ) AR0.outfile = oldoutfile;
03197     AR0.gzipCompress = oldgzipCompress;
03198     goto ProcErr;
03199 }
03200 if ( AR0.outtohide ) AR0.outfile = oldoutfile;
03201 AR0.gzipCompress = oldgzipCompress;
03202 /*
03203 Now wait for all threads to be ready to give them the cleaning up signal.
03204 With the new MasterMerge routine we can do the cleanup already automatically
03205 avoiding having to send these signals.
03206 */
03207 MasterWaitAll();
03208 AR0.sLevel--;
03209 for ( id = 1; id < AM.totalnumberofthreads; id++ ) {
03210     if ( GetThread(id) > 0 ) WakeupThread(id,CLEANUPEXPRESSION);
03211 }
03212 e->numdummies = 0;
03213 for ( id = 1; id < AM.totalnumberofthreads; id++ ) {
03214     if ( AB[id]->R.MaxDum - AM.IndDum > e->numdummies )
03215         e->numdummies = AB[id]->R.MaxDum - AM.IndDum;
03216     AR0.expchanged |= AB[id]->R.expchanged;
03217 }
03218 /*
03219 And wait for all to be clean.
03220 */
03221 MasterWaitAll();
03222 AT0.WorkPointer = oldworkpointer;
03223 return(0);
03224 ProcErr:;
03225 return(-1);
03226 }
03227
03228 /*
03229 #] ThreadsProcessor :
03230 #[ LoadReadjusted :
03231 */
03250 int LoadReadjusted(void)
03251 {
03252     ALLPRIVATES *B0 = AB[0];
03253     THREADBUCKET *thr = 0, *thrtogo = 0;
03254     int numtogo, numfree, numbusy, n, nperbucket, extra, i, j, u, bus;
03255     LONG numinput;
03256     WORD *t1, *c1, *t2, *c2, *t3;
03257 /*
03258 Start with waiting for at least one free processor.
03259 We don't want the master competing for time when all are busy.
03260 */
03261 while ( topofavailables <= 0 ) MasterWait();
03262 /*
03263 Now look for the fullest bucket and make a list of free buckets
03264 The bad part is that most numbers can change at any moment.
03265 */
03266 restart:;
03267 numtogo = 0;
03268 numfree = 0;
03269 numbusy = 0;
03270 for ( j = 0; j < numthreadbuckets; j++ ) {
03271     thr = threadbuckets[j];
03272     if ( thr->free == BUCKETFREE || thr->free == BUCKETATEND
03273         || thr->free == BUCKETCOMINGFREE ) {
03274         freebuckets[numfree++] = thr;
03275     }
03276     else if ( thr->type != BUCKETDOINGTERMS ) {}
03277     else if ( thr->totnum > 1 ) { /* never steal from a bucket with one term */
03278         LOCK(thr->lock);
03279         bus = thr->busy;
03280         UNLOCK(thr->lock);
03281         if ( thr->free == BUCKETINUSE ) {
03282             n = thr->totnum-thr->usenum;
03283             if ( bus == BUCKETASSIGNED ) numbusy++;

```

```

03284         else if ( ( bus != BUCKETASSIGNED )
03285             && ( n > numtogo ) ) {
03286             numtogo = n;
03287             thrtogo = thr;
03288         }
03289     }
03290     else if ( bus == BUCKETTOBERELEASED
03291         && thr->free == BUCKETRELEASED ) {
03292         freebuckets[numfree++] = thr;
03293         thr->free = BUCKETATEND;
03294         LOCK(thr->lock);
03295         thr->busy = BUCKETPREPARINGTERM;
03296         UNLOCK(thr->lock);
03297     }
03298 }
03299 }
03300 if ( numfree == 0 ) return(0); /* serious problem */
03301 if ( numtogo > 0 ) { /* provisionally there is something to be stolen */
03302     thr = thrtogo;
03303 /*
03304     If the number has changed there is good progress.
03305     Maybe there is another thread that needs assistance.
03306     We start all over.
03307 */
03308     if ( thr->totnum-thr->usenum < numtogo ) goto restart;
03309 /*
03310     If the thread is in the term loading phase
03311     (thr->busy == BUCKETPREPARINGTERM) we better stay away from it.
03312     We wait now for the thread to be busy, and don't allow it
03313     now to drop out of this state till we are done here.
03314     This all depends on whether AT.LoadBalancing == 1 is seen by
03315     the thread.
03316 */
03317     LOCK(thr->lock);
03318     if ( thr->busy != BUCKETDOINGTERM ) {
03319         UNLOCK(thr->lock);
03320         goto restart;
03321     }
03322     if ( thr->totnum-thr->usenum < numtogo ) {
03323         UNLOCK(thr->lock);
03324         goto restart;
03325     }
03326     thr->free = BUCKETTERMINATED;
03327 /*
03328     The above will signal the thread we want to terminate.
03329     Next all effort goes into making sure the landing is soft.
03330     Unfortunately we don't want to wait for a signal, because the thread
03331     may be working for a long time on a single term.
03332 */
03333     if ( thr->usenum == thr->totnum ) {
03334 /*
03335         Terminated in the mean time or by now working on the
03336         last term. Try again.
03337 */
03338         thr->free = BUCKETATEND;
03339         UNLOCK(thr->lock);
03340         goto restart;
03341     }
03342     goto intercepted;
03343 }
03344 /* This has always been commented. Indeed no lock is held here. */
03345 /* UNLOCK(thr->lock); */
03346 if ( numbusy > 0 ) {
03347     /* JD: this avoids large runtimes for tform tests under valgrind.
03348        What seems to happen is we return from here, goto Finalize, and
03349        end up in LoadReadjusted again without the threads having a
03350        chance to update their busy status. Then we end up here again.
03351        Sleep the thread for, say, lus to allow threads to aquire the lock. */
03352     struct timespec sleeptime;
03353     sleeptime.tv_sec = 0;
03354     sleeptime.tv_nsec = 1000L;
03355     nanosleep(&sleeptime, NULL);
03356     return(1); /* Wait a bit.... */
03357 }
03358 return(0);
03359 intercepted;;
03360 /*
03361     We intercepted one successfully. Now it becomes interesting. Action:
03362     1: determine how many terms per free bucket.
03363     2: find the first untreated term.
03364     3: put the terms in the free buckets.
03365
03366     Remember: we still have the lock to avoid interference from the thread
03367     that is being robbed. We were holding it and then jumped here with
03368     goto intercepted.
03369 */
03370     numinput = thr->firstterm + thr->usenum;

```

```

03371     nperbucket = numtogo / numfree;
03372     extra = numtogo - nperbucket*numfree;
03373     if ( AR0.DeferFlag ) {
03374         t1 = thr->threadbuffer; c1 = thr->compressbuffer; u = thr->usenum;
03375         for ( n = 0; n < thr->usenum; n++ ) { t1 += *t1; c1 += *c1; }
03376         t3 = t1;
03377         if ( extra > 0 ) {
03378             for ( i = 0; i < extra; i++ ) {
03379                 thrtogo = freebuckets[i];
03380                 t2 = thrtogo->threadbuffer;
03381                 c2 = thrtogo->compressbuffer;
03382                 thrtogo->free = BUCKETFILLED;
03383                 thrtogo->type = BUCKETDOINGTERMS;
03384                 thrtogo->totnum = nperbucket+1;
03385                 thrtogo->ddterms = 0;
03386                 thrtogo->usenum = 0;
03387                 thrtogo->busy = BUCKETASSIGNED;
03388                 thrtogo->firstterm = numinput;
03389                 numinput += nperbucket+1;
03390                 for ( n = 0; n <= nperbucket; n++ ) {
03391                     j = *t1; NCOPY(t2,t1,j);
03392                     j = *c1; NCOPY(c2,c1,j);
03393                     thrtogo->deferbuffer[n] = thr->deferbuffer[u++];
03394                 }
03395                 *t2 = *c2 = 0;
03396             }
03397         }
03398         if ( nperbucket > 0 ) {
03399             for ( i = extra; i < numfree; i++ ) {
03400                 thrtogo = freebuckets[i];
03401                 t2 = thrtogo->threadbuffer;
03402                 c2 = thrtogo->compressbuffer;
03403                 thrtogo->free = BUCKETFILLED;
03404                 thrtogo->type = BUCKETDOINGTERMS;
03405                 thrtogo->totnum = nperbucket;
03406                 thrtogo->ddterms = 0;
03407                 thrtogo->usenum = 0;
03408                 thrtogo->busy = BUCKETASSIGNED;
03409                 thrtogo->firstterm = numinput;
03410                 numinput += nperbucket;
03411                 for ( n = 0; n < nperbucket; n++ ) {
03412                     j = *t1; NCOPY(t2,t1,j);
03413                     j = *c1; NCOPY(c2,c1,j);
03414                     thrtogo->deferbuffer[n] = thr->deferbuffer[u++];
03415                 }
03416                 *t2 = *c2 = 0;
03417             }
03418         }
03419     }
03420     else {
03421         t1 = thr->threadbuffer;
03422         for ( n = 0; n < thr->usenum; n++ ) { t1 += *t1; }
03423         t3 = t1;
03424         if ( extra > 0 ) {
03425             for ( i = 0; i < extra; i++ ) {
03426                 thrtogo = freebuckets[i];
03427                 t2 = thrtogo->threadbuffer;
03428                 thrtogo->free = BUCKETFILLED;
03429                 thrtogo->type = BUCKETDOINGTERMS;
03430                 thrtogo->totnum = nperbucket+1;
03431                 thrtogo->ddterms = 0;
03432                 thrtogo->usenum = 0;
03433                 thrtogo->busy = BUCKETASSIGNED;
03434                 thrtogo->firstterm = numinput;
03435                 numinput += nperbucket+1;
03436                 for ( n = 0; n <= nperbucket; n++ ) {
03437                     j = *t1; NCOPY(t2,t1,j);
03438                 }
03439                 *t2 = 0;
03440             }
03441         }
03442         if ( nperbucket > 0 ) {
03443             for ( i = extra; i < numfree; i++ ) {
03444                 thrtogo = freebuckets[i];
03445                 t2 = thrtogo->threadbuffer;
03446                 thrtogo->free = BUCKETFILLED;
03447                 thrtogo->type = BUCKETDOINGTERMS;
03448                 thrtogo->totnum = nperbucket;
03449                 thrtogo->ddterms = 0;
03450                 thrtogo->usenum = 0;
03451                 thrtogo->busy = BUCKETASSIGNED;
03452                 thrtogo->firstterm = numinput;
03453                 numinput += nperbucket;
03454                 for ( n = 0; n < nperbucket; n++ ) {
03455                     j = *t1; NCOPY(t2,t1,j);
03456                 }
03457                 *t2 = 0;

```

```

03458     }
03459     }
03460 }
03461 *t3 = 0; /* This is some form of extra insurance */
03462 if ( thr->free == BUCKETRELEASED && thr->busy == BUCKETTOBERELEASED ) {
03463     thr->free = BUCKETATEND; thr->busy = BUCKETPREPARINGTERM;
03464 }
03465 UNLOCK(thr->lock);
03466 return(1);
03467 }
03468
03469 /*
03470 #] LoadReadjusted :
03471 #[ SortStrategy :
03472 */
03505 /*
03506 #] SortStrategy :
03507 #[ PutToMaster :
03508 */
03531 int PutToMaster(PHEAD WORD *term)
03532 {
03533     int i,j,nexti,ret = 0;
03534     WORD *t, *fill, *top, zero = 0;
03535     if ( term == 0 ) { /* Mark the end of the expression */
03536         t = &zero; j = 1;
03537     }
03538     else {
03539         t = term; ret = j = *term;
03540         if ( j == 0 ) { j = 1; } /* Just in case there is a spurious end */
03541     }
03542     i = AT.SB.FillBlock; /* The block we are working at */
03543     fill = AT.SB.MasterFill[i]; /* Where we are filling */
03544     top = AT.SB.MasterStop[i]; /* End of the block */
03545     while ( j > 0 ) {
03546         while ( j > 0 && fill < top ) {
03547             *fill++ = *t++; j--;
03548         }
03549         if ( j > 0 ) {
03550             /*
03551              We reached the end of the block.
03552              Get the next block and release this block.
03553              The order is important. This is why there should be at least
03554              4 blocks or deadlocks can occur.
03555              */
03556             nexti = i+1;
03557             if ( nexti > AT.SB.MasterNumBlocks ) {
03558                 nexti = 1;
03559             }
03560             LOCK(AT.SB.MasterBlockLock[nexti]);
03561             UNLOCK(AT.SB.MasterBlockLock[i]);
03562             AT.SB.MasterFill[i] = AT.SB.MasterStart[i];
03563             AT.SB.FillBlock = i = nexti;
03564             fill = AT.SB.MasterStart[i];
03565             top = AT.SB.MasterStop[i];
03566         }
03567     }
03568     AT.SB.MasterFill[i] = fill;
03569     return(ret);
03570 }
03571
03572 /*
03573 #] PutToMaster :
03574 #[ SortBotOut :
03575 */
03576
03577 #ifdef WITHSORTBOTS
03578
03587 int
03588 SortBotOut(PHEAD WORD *term)
03589 {
03590     WORD im;
03591
03592     if ( AT.identity != 0 ) return(PutToMaster(BHEAD term));
03593
03594     if ( term == 0 ) {
03595         if ( FlushOut(&SortBotPosition,AR.outfile,1) ) return(-1);
03596         ADDPOS(AT.SS->SizeInFile[0],1);
03597         return(0);
03598     }
03599     else {
03600         numberofterms++;
03601         if ( ( im = PutOut(BHEAD term,&SortBotPosition,AR.outfile,1) ) < 0 ) {
03602             MLOCK(ErrorMessageLock);
03603             MesPrint("Called from MasterMerge/SortBotOut");
03604             MUNLOCK(ErrorMessageLock);
03605             return(-1);
03606         }
03607     }

```

```

03607         ADDPOS(AT.SS->SizeInFile[0],im);
03608         return(im);
03609     }
03610 }
03611
03612 #endif
03613
03614 /*
03615     #] SortBotOut :
03616     #[ MasterMerge :
03617 */
03635 int MasterMerge(void)
03636 {
03637     ALLPRIVATES *B0 = AB[0], *B = 0;
03638     SORTING *S = AT0.SS;
03639     WORD **poin, **poin2, ul, k, i, im, *m1, j;
03640     WORD lpat, mpat, level, l1, l2, r1, r2, r3, c;
03641     WORD *m2, *m3, r31, r33, ki, *rr;
03642     UWORD *coef;
03643     POSITION position;
03644     FILEHANDLE *fin, *fout;
03645 #ifdef WITHSORTBOTS
03646     if ( numberofworkers > 2 ) return(SortBotMasterMerge());
03647 #endif
03648     fin = &S->file;
03649     if ( AR0.PolyFun == 0 ) { S->PolyFlag = 0; }
03650     else if ( AR0.PolyFunType == 1 ) { S->PolyFlag = 1; }
03651     else if ( AR0.PolyFunType == 2 ) {
03652         if ( AR0.PolyFunExp == 2
03653             || AR0.PolyFunExp == 3 ) S->PolyFlag = 1;
03654         else S->PolyFlag = 2;
03655     }
03656     S->TermsLeft = 0;
03657     coef = AN0.SoScratC;
03658     poin = S->poina; poin2 = S->poin2a;
03659     rr = AR0.CompressPointer;
03660     *rr = 0;
03661 /*
03662     #[ Setup :
03663 */
03664     S->inNum = numberofthreads;
03665     fout = AR0.outfile;
03666 /*
03667     Load the patches. The threads have to finish their sort first.
03668 */
03669     S->lPatch = S->inNum - 1;
03670 /*
03671     Claim all zero blocks. We need them anyway.
03672     In principle the workers should never get into these.
03673     We also claim all last blocks. This is a safety procedure that
03674     should prevent the workers from working their way around the clock
03675     before the master gets started again.
03676 */
03677     AS.MasterSort = 1;
03678     numberclaimed = 0;
03679     for ( i = 1; i <= S->lPatch; i++ ) {
03680         B = AB[i];
03681         LOCK(AT.SB.MasterBlockLock[0]);
03682         LOCK(AT.SB.MasterBlockLock[AT.SB.MasterNumBlocks]);
03683     }
03684 /*
03685     Now wake up the threads and have them start their final sorting.
03686     They should start with claiming their block and the master is
03687     not allowed to continue until that has been done.
03688     This waiting of the master will be done below in MasterWaitAllBlocks
03689 */
03690     for ( i = 0; i < S->lPatch; i++ ) {
03691         GetThread(i+1);
03692         WakeupThread(i+1,FINISHEXPRESSION);
03693     }
03694 /*
03695     Prepare the output file.
03696 */
03697     if ( fout->handle >= 0 ) {
03698         PUTZERO(position);
03699         SeekFile(fout->handle,&position,SEEK_END);
03700         ADDPOS(position,((fout->POfill-fout->PObuffer)*sizeof(WORD)));
03701     }
03702     else {
03703         SETBASEPOSITION(position,(fout->POfill-fout->PObuffer)*sizeof(WORD));
03704     }
03705 /*
03706     Wait for all threads to finish loading their first block.
03707 */
03708     MasterWaitAllBlocks();
03709 /*
03710     Claim all first blocks.

```

```

03711      We don't release the last blocks.
03712      The strategy is that we always keep the previous block.
03713      In principle it looks like it isn't needed for the last block but
03714      actually it is to keep the front from overrunning the tail when writing.
03715  */
03716      for ( i = 1; i <= S->lPatch; i++ ) {
03717          B = AB[i];
03718          LOCK(AT.SB.MasterBlockLock[1]);
03719          AT.SB.MasterBlock = 1;
03720      }
03721  /*
03722      #] Setup :
03723
03724      Now construct the tree:
03725  */
03726      lpat = 1;
03727      do { lpat <= 1; } while ( lpat < S->lPatch );
03728      mpat = ( lpat » 1 ) - 1;
03729      k = lpat - S->lPatch;
03730  /*
03731      k is the number of empty places in the tree. they will
03732      be at the even positions from 2 to 2*k
03733  */
03734      for ( i = 1; i < lpat; i++ ) { S->tree[i] = -1; }
03735      for ( i = 1; i <= k; i++ ) {
03736          im = ( i * 2 ) - 1;
03737          poin[im] = AB[i]->T.SB.MasterStart[AB[i]->T.SB.MasterBlock];
03738          poin2[im] = poin[im] + *(poin[im]);
03739          S->used[i] = im;
03740          S->ktoi[im] = i-1;
03741          S->tree[mpat+i] = 0;
03742          poin[im-1] = poin2[im-1] = 0;
03743      }
03744      for ( i = (k*2)+1; i <= lpat; i++ ) {
03745          S->used[i-k] = i;
03746          S->ktoi[i] = i-k-1;
03747          poin[i] = AB[i-k]->T.SB.MasterStart[AB[i-k]->T.SB.MasterBlock];
03748          poin2[i] = poin[i] + *(poin[i]);
03749      }
03750  /*
03751      the array poin tells the position of the i-th element of the S->tree
03752      'S->used' is a stack with the S->tree elements that need to be entered
03753      into the S->tree. at the beginning this is S->lPatch. during the
03754      sort there will be only very few elements.
03755      poin2 is the next value of poin. it has to be determined
03756      before the comparisons as the position or the size of the
03757      term indicated by poin may change.
03758      S->ktoi translates a S->tree element back to its stream number.
03759
03760      start the sort
03761  */
03762      level = S->lPatch;
03763  /*
03764      introduce one term
03765  */
03766  OneTerm:
03767      k = S->used[level];
03768      i = k + lpat - 1;
03769      if ( !(poin[k]) ) {
03770          do { if ( !( i »= 1 ) ) goto EndOfMerge; } while ( !S->tree[i] );
03771          if ( S->tree[i] == -1 ) {
03772              S->tree[i] = 0;
03773              level--;
03774              goto OneTerm;
03775          }
03776          k = S->tree[i];
03777          S->used[level] = k;
03778          S->tree[i] = 0;
03779      }
03780  /*
03781      move terms down the tree
03782  */
03783      while ( i »= 1 ) {
03784          if ( S->tree[i] > 0 ) {
03785              if ( ( c = CompareTerms(B0, poin[S->tree[i]],poin[k],(WORD)0) ) > 0 ) {
03786  /*
03787                  S->tree[i] is the smaller. Exchange and go on.
03788  */
03789                  S->used[level] = S->tree[i];
03790                  S->tree[i] = k;
03791                  k = S->used[level];
03792              }
03793              else if ( !c ) { /* Terms are equal */
03794  /*
03795                  S->TermsLeft--;
03796                  Here the terms are equal and their coefficients
03797                  have to be added.

```



```

03798 */
03799     l1 = *( m1 = poin[S->tree[i]] );
03800     l2 = *( m2 = poin[k] );
03801     if ( S->PolyWise ) { /* Here we work with PolyFun */
03802         WORD *ttl, *w;
03803         ttl = m1;
03804         m1 += S->PolyWise;
03805         m2 += S->PolyWise;
03806         if ( S->PolyFlag == 2 ) {
03807             w = poly_ratfun_add(B0,m1,m2);
03808             if ( *ttl + w[1] - m1[1] > AM.MaxTer/((LONG)sizeof(WORD)) ) {
03809                 MLOCK(ErrorMessageLock);
03810                 MesPrint("Term too complex in PolyRatFun addition. MaxTermSize of %10l is
too small",AM.MaxTer);
03811                 MUNLOCK(ErrorMessageLock);
03812                 Terminate(-1);
03813             }
03814             AT0.WorkPointer = w;
03815             if ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 && w[1] > FUNHEAD ) {
03816                 goto cancelled;
03817             }
03818         }
03819         else {
03820             w = AT0.WorkPointer;
03821             if ( w + m1[1] + m2[1] > AT0.WorkTop ) {
03822                 MLOCK(ErrorMessageLock);
03823                 MesPrint("MasterMerge: A Workspace of %10l is too small",AM.WorkSize);
03824                 MUNLOCK(ErrorMessageLock);
03825                 Terminate(-1);
03826             }
03827             AddArgs(B0,m1,m2,w);
03828         }
03829         r1 = w[1];
03830         if ( r1 <= FUNHEAD
03831             || ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 ) )
03832             { goto cancelled; }
03833         if ( r1 == m1[1] ) {
03834             NCOPY(m1,w,r1);
03835         }
03836         else if ( r1 < m1[1] ) {
03837             r2 = m1[1] - r1;
03838             m2 = w + r1;
03839             m1 += m1[1];
03840             while ( --r1 >= 0 ) *--m1 = *--m2;
03841             m2 = m1 - r2;
03842             r1 = S->PolyWise;
03843             while ( --r1 >= 0 ) *--m1 = *--m2;
03844             *m1 -= r2;
03845             poin[S->tree[i]] = m1;
03846         }
03847         else {
03848             r2 = r1 - m1[1];
03849             m2 = ttl - r2;
03850             r1 = S->PolyWise;
03851             m1 = ttl;
03852             *m1 += r2;
03853             poin[S->tree[i]] = m2;
03854             NCOPY(m2,m1,r1);
03855             r1 = w[1];
03856             NCOPY(m2,w,r1);
03857         }
03858     }
03859     #ifdef WITHFLOAT
03860     else if ( AT.SortFloatMode ) {
03861         WORD *term1, *term2;
03862         term1 = poin[S->tree[i]];
03863         term2 = poin[k];
03864         if ( MergeWithFloat(B0, &term1,&term2) == 0 )
03865             goto cancelled;
03866         poin[S->tree[i]] = term1;
03867     }
03868     #endif
03869     else {
03870         r1 = *( m1 += l1 - 1 );
03871         m1 -= ABS(r1) - 1;
03872         r1 = ( ( r1 > 0 ) ? (r1-1) : (r1+1) ) >> 1;
03873         r2 = *( m2 += l2 - 1 );
03874         m2 -= ABS(r2) - 1;
03875         r2 = ( ( r2 > 0 ) ? (r2-1) : (r2+1) ) >> 1;
03876
03877         if ( AddRat(B0,(UWORD *)m1,r1,(UWORD *)m2,r2,coef,&r3) ) {
03878             MLOCK(ErrorMessageLock);
03879             MesCall("MasterMerge");
03880             MUNLOCK(ErrorMessageLock);
03881             SETERROR(-1)
03882         }
03883     }

```

```

03884         if ( AN.ncmod != 0 ) {
03885             if ( ( AC.modmode & POSNEG ) != 0 ) {
03886                 NormalModulus(coef,&r3);
03887             }
03888             else if ( BigLong(coef,r3,(UWORD *)AC.cmod,ABS(AN.ncmod)) >= 0 ) {
03889                 WORD ii;
03890                 SubPLon(coef,r3,(UWORD *)AC.cmod,ABS(AN.ncmod),coef,&r3);
03891                 coef[r3] = 1;
03892                 for ( ii = 1; ii < r3; ii++ ) coef[r3+ii] = 0;
03893             }
03894         }
03895         r3 *= 2;
03896         r33 = ( r3 > 0 ) ? ( r3 + 1 ) : ( r3 - 1 );
03897         if ( r3 < 0 ) r3 = -r3;
03898         if ( r1 < 0 ) r1 = -r1;
03899         r1 *= 2;
03900         r31 = r3 - r1;
03901         if ( !r3 ) { /* Terms cancel */
03902             cancelled:
03903                 ul = S->used[level] = S->tree[i];
03904                 S->tree[i] = -1;
03905             /*
03906              *
03907              * We skip to the next term in stream ul
03908              *
03909              im = *poin2[ul];
03910              poin[ul] = poin2[ul];
03911              ki = S->ktoi[ul];
03912              if ( (poin[ul] + im + COMPINC) >=
03913              AB[ki+1]->T.SB.MasterStop[AB[ki+1]->T.SB.MasterBlock]
03914              && im > 0 ) {
03915                  /*
03916                   * We made it to the end of the block. We have to
03917                   * release the previous block and claim the next.
03918                   *
03919                   B = AB[ki+1];
03920                   i = AT.SB.MasterBlock;
03921                   if ( i == 1 ) {
03922                       UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterNumBlocks]);
03923                   }
03924                   else {
03925                       UNLOCK(AT.SB.MasterBlockLock[i-1]);
03926                   }
03927                   if ( i == AT.SB.MasterNumBlocks ) {
03928                       /*
03929                        * Move the remainder down into block 0
03930                        *
03931                        WORD *from, *to;
03932                        to = AT.SB.MasterStart[1];
03933                        from = AT.SB.MasterStop[i];
03934                        while ( from > poin[ul] ) *--to = *--from;
03935                        poin[ul] = to;
03936                        i = 1;
03937                    }
03938                    else { i++; }
03939                    LOCK(AT.SB.MasterBlockLock[i]);
03940                    AT.SB.MasterBlock = i;
03941                    poin2[ul] = poin[ul] + im;
03942                }
03943                else {
03944                    poin2[ul] += im;
03945                }
03946                S->used[++level] = k;
03947            }
03948            else if ( !r31 ) { /* copy coef into term1 */
03949                goto CopCof2;
03950            }
03951            else if ( r31 < 0 ) { /* copy coef into term1
03952                and adjust the length of term1 */
03953                goto CopCoef;
03954            }
03955            /*
03956             * this is the dreaded calamity.
03957             * is there enough space?
03958             *
03959             if( (poin[S->tree[i]]+11+r31) >= poin2[S->tree[i]] ) {
03960                 /*
03961                  * no space! now the special trick for which
03962                  * we left 2*maxlng spaces open at the beginning
03963                  * of each patch.
03964                  *
03965                  if ( (11 + r31)*((LONG)sizeof(WORD)) >= AM.MaxTer ) {
03966                      MLOCK(ErrorMessageLock);
03967                      MesPrint("MasterMerge: Coefficient overflow during sort");
03968                      MUNLOCK(ErrorMessageLock);
03969                      goto ReturnError;
03970                  }

```

```

03971             m2 = poin[S->tree[i]];
03972             m3 = ( poin[S->tree[i]] -= r31 );
03973             do { *m3++ = *m2++; } while ( m2 < m1 );
03974             m1 = m3;
03975         }
03976 CopCoef:
03977             *(poin[S->tree[i]]) += r31;
03978 CopCof2:
03979             m2 = (WORD *)coef; im = r3;
03980             NCOPY(m1,m2,im);
03981             *m1 = r33;
03982         }
03983     }
03984 /*
03985         Now skip to the next term in stream k.
03986 */
03987 NextTerm:
03988     im = poin2[k][0];
03989     poin[k] = poin2[k];
03990     ki = S->ktoi[k];
03991     if ( (poin[k] + im + COMPINC) >=
03992         AB[ki+1]->T.SB.MasterStop[AB[ki+1]->T.SB.MasterBlock]
03993         && im > 0 ) {
03994 /*
03995         We made it to the end of the block. We have to
03996         release the previous block and claim the next.
03997 */
03998         B = AB[ki+1];
03999         i = AT.SB.MasterBlock;
04000         if ( i == 1 ) {
04001             UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterNumBlocks]);
04002         }
04003         else {
04004             UNLOCK(AT.SB.MasterBlockLock[i-1]);
04005         }
04006         if ( i == AT.SB.MasterNumBlocks ) {
04007 /*
04008             Move the remainder down into block 0
04009 */
04010             WORD *from, *to;
04011             to = AT.SB.MasterStart[1];
04012             from = AT.SB.MasterStop[i];
04013             while ( from > poin[k] ) *--to = *--from;
04014             poin[k] = to;
04015             i = 1;
04016         }
04017         else { i++; }
04018         LOCK(AT.SB.MasterBlockLock[i]);
04019         AT.SB.MasterBlock = i;
04020         poin2[k] = poin[k] + im;
04021     }
04022     else {
04023         poin2[k] += im;
04024     }
04025     goto OneTerm;
04026 }
04027 }
04028 else if ( S->tree[i] < 0 ) {
04029     S->tree[i] = k;
04030     level--;
04031     goto OneTerm;
04032 }
04033 }
04034 /*
04035     found the smallest in the set. indicated by k.
04036     write to its destination.
04037 */
04038 S->TermsLeft++;
04039 if ( ( im = PutOut(B0,poin[k],&position,fout,1) ) < 0 ) {
04040     MLOCK(ErrorMessageLock);
04041     MesPrint("Called from MasterMerge with k = %d (stream %d)",k,S->ktoi[k]);
04042     MUNLOCK(ErrorMessageLock);
04043     goto ReturnError;
04044 }
04045 ADDPOS(S->SizeInFile[0],im);
04046 goto NextTerm;
04047 EndOfMerge:
04048 if ( FlushOut(&position,fout,1) ) goto ReturnError;
04049 ADDPOS(S->SizeInFile[0],1);
04050 CloseFile(fin->handle);
04051 remove(fin->name);
04052 fin->handle = -1;
04053 position = S->SizeInFile[0];
04054 MULPOS(position,sizeof(WORD));
04055 S->GenTerms = 0;
04056 for ( j = 1; j <= numberofworkers; j++ ) {
04057     S->GenTerms += AB[j]->T.SS->GenTerms;

```

```

04058     }
04059     WriteStats(&position,STATSPOSTSORT,NOCHECKLOGTYPE);
04060     Expressions[ARO.CurExpr].counter = S->TermsLeft;
04061     Expressions[ARO.CurExpr].size = position;
04062 /*
04063     Release all locks
04064 */
04065     for ( i = 1; i <= S->lPatch; i++ ) {
04066         B = AB[i];
04067         UNLOCK(AT.SB.MasterBlockLock[0]);
04068         if ( AT.SB.MasterBlock == 1 ) {
04069             UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterNumBlocks]);
04070         }
04071         else {
04072             UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterBlock-1]);
04073         }
04074         UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterBlock]);
04075     }
04076     AS.MasterSort = 0;
04077     return(0);
04078 ReturnError:
04079     for ( i = 1; i <= S->lPatch; i++ ) {
04080         B = AB[i];
04081         UNLOCK(AT.SB.MasterBlockLock[0]);
04082         if ( AT.SB.MasterBlock == 1 ) {
04083             UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterNumBlocks]);
04084         }
04085         else {
04086             UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterBlock-1]);
04087         }
04088         UNLOCK(AT.SB.MasterBlockLock[AT.SB.MasterBlock]);
04089     }
04090     AS.MasterSort = 0;
04091     return(-1);
04092 }
04093
04094 /*
04095     #] MasterMerge :
04096     #[ SortBotMasterMerge :
04097 */
04098
04099 #ifdef WITHSORTBOTS
04100
04116 int SortBotMasterMerge(void)
04117 {
04118     FILEHANDLE *fin, *fout;
04119     ALLPRIVATES *B = AB[0], *BB;
04120     POSITION position;
04121     SORTING *S = AT.SS;
04122     int i, j;
04123 /*
04124     Get the sortbots get to claim their writing blocks.
04125     We have to wait till all have been claimed because they also have to
04126     claim the last writing blocks of the workers to prevent the head of
04127     the circular buffer to overrun the tail.
04128
04129     Before waiting we can do some needed initializations.
04130     Also the master has to claim the last writing blocks of its input.
04131 */
04132     topsortbotavailables = 0;
04133     for ( i = numberofworkers+1; i <= numberofworkers+numberofsortbots; i++ ) {
04134         WakeupThread(i,INISORTBOT);
04135     }
04136
04137     AS.MasterSort = 1;
04138     fout = AR.outfile;
04139     numberofterms = 0;
04140     AR.CompressPointer[0] = 0;
04141     numberclaimed = 0;
04142     BB = AB[AT.SortBotIn1];
04143     LOCK(BB->T.SB.MasterBlockLock[BB->T.SB.MasterNumBlocks]);
04144     BB = AB[AT.SortBotIn2];
04145     LOCK(BB->T.SB.MasterBlockLock[BB->T.SB.MasterNumBlocks]);
04146
04147     MasterWaitAllSortBots();
04148 /*
04149     Now we can start up the workers. They will claim their writing blocks.
04150     Here the master will wait till all writing blocks have been claimed.
04151 */
04152     for ( i = 1; i <= numberofworkers; i++ ) {
04153         j = GetThread(i);
04154         WakeupThread(i,FINISHEXPRESSION);
04155     }
04156 /*
04157     Prepare the output file in the mean time.
04158 */
04159     if ( fout->handle >= 0 ) {

```

```

04160         PUTZERO(SortBotPosition);
04161         SeekFile(fout->handle,&SortBotPosition,SEEK_END);
04162         ADDPOS(SortBotPosition,((fout->Pofill-fout->PObuffer)*sizeof(WORD)));
04163     }
04164     else {
04165         SETBASEPOSITION(SortBotPosition,(fout->Pofill-fout->PObuffer)*sizeof(WORD));
04166     }
04167     MasterWaitAllBlocks();
04168 /*
04169     Now we can start the sortbots after which the master goes in
04170     sortbot mode to do its part of the job (the very final merge and
04171     the writing to output file).
04172 */
04173     topsortbotavailables = 0;
04174     for ( i = numberofworkers+1; i <= numberofworkers+numberofsortbots; i++ ) {
04175         WakeupThread(i,RUNSORTBOT);
04176     }
04177     if ( SortBotMerge(BHEAD0) ) {
04178         MLOCK(ErrorMessageLock);
04179         MesPrint("Called from SortBotMasterMerge");
04180         MUNLOCK(ErrorMessageLock);
04181         AS.MasterSort = 0;
04182         return(-1);
04183     }
04184 /*
04185     And next the cleanup
04186 */
04187     if ( S->file.handle >= 0 )
04188     {
04189         fin = &S->file;
04190         CloseFile(fin->handle);
04191         remove(fin->name);
04192         fin->handle = -1;
04193     }
04194     position = S->SizeInFile[0];
04195     MULPOS(position,sizeof(WORD));
04196     S->GenTerms = 0;
04197     for ( j = 1; j <= numberofworkers; j++ ) {
04198         S->GenTerms += AB[j]->T.SS->GenTerms;
04199     }
04200     S->TermsLeft = numberofterms;
04201     WriteStats(&position,STATSPOSTSORT,NOCHECKLOGTYPE);
04202     Expressions[AR.CurExpr].counter = S->TermsLeft;
04203     Expressions[AR.CurExpr].size = position;
04204     AS.MasterSort = 0;
04205 /*
04206     The next statement is to prevent one of the sortbots not having
04207     completely cleaned up before the next module starts.
04208     If this statement is omitted every once in a while one of the sortbots
04209     is still running when the next expression starts and misses its
04210     initialization. The result is usually disastrous.
04211 */
04212     MasterWaitAllSortBots();
04213
04214     return(0);
04215 }
04216
04217 #endif
04218
04219 /*
04220     #] SortBotMasterMerge :
04221     #[ SortBotMerge :
04222 */
04223
04224 #ifdef WITHSORTBOTS
04225
04231 int SortBotMerge(PHEAD0)
04232 {
04233     GETBIDENTITY
04234     ALLPRIVATES *Bin1 = AB[AT.SortBotIn1],*Bin2 = AB[AT.SortBotIn2];
04235     WORD *term1, *term2, *next, *wp;
04236     int blin1, blin2; /* Current block numbers */
04237     int error = 0;
04238     WORD l1, l2, *m1, *m2, *w, r1, r2, r3, r33, r31, *ttl1, ii;
04239     WORD *to, *from, im, c;
04240     UWORD *coef;
04241     SORTING *S = AT.SS;
04242 #ifdef WITHFLOAT
04243     WORD *fun1, *fun2, *fun3, *tstop1, *tstop2, size1, size2, size3, l3, jj, *m3;
04244 #endif
04245 /*
04246     Set the pointers to the input terms and the output space
04247 */
04248     coef = AN.SoScratC;
04249     blin1 = 1;
04250     blin2 = 1;
04251     if ( AT.identity == 0 ) {

```

```

04252         wp = AT.WorkPointer;
04253     }
04254     else {
04255         wp = AT.WorkPointer = AT.WorkSpace;
04256     }
04257 /*
04258     Get the locks for reading the input
04259     This means that we can start once these locks have been cleared
04260     which means that there will be input.
04261 */
04262     LOCK(Bin1->T.SB.MasterBlockLock[blin1]);
04263     LOCK(Bin2->T.SB.MasterBlockLock[blin2]);
04264
04265     term1 = Bin1->T.SB.MasterStart[blin1];
04266     term2 = Bin2->T.SB.MasterStart[blin2];
04267     AT.SB.FillBlock = 1;
04268 /*
04269     Now the main loop. Keep going until one of the two hits the end.
04270 */
04271     while ( *term1 && *term2 ) {
04272         if ( ( c = CompareTerms(BHEAD term1,term2,(WORD)0) ) > 0 ) {
04273 /*
04274             #[ One is smallest :
04275 */
04276             if ( SortBotOut(BHEAD term1) < 0 ) {
04277                 MLOCK(ErrorMessageLock);
04278                 MesPrint("Called from SortBotMerge with thread = %d",AT.identity);
04279                 MUNLOCK(ErrorMessageLock);
04280                 error = -1;
04281                 goto ReturnError;
04282             }
04283             im = *term1;
04284             next = term1 + im;
04285             if ( next >= Bin1->T.SB.MasterStop[blin1] || ( *next &&
04286                 next+*next+COMPINC > Bin1->T.SB.MasterStop[blin1] ) ) {
04287                 if ( blin1 == 1 ) {
04288                     UNLOCK(Bin1->T.SB.MasterBlockLock[Bin1->T.SB.MasterNumBlocks]);
04289                 }
04290                 else {
04291                     UNLOCK(Bin1->T.SB.MasterBlockLock[blin1-1]);
04292                 }
04293                 if ( blin1 == Bin1->T.SB.MasterNumBlocks ) {
04294 /*
04295                     Move the remainder down into block 0
04296 */
04297                     to = Bin1->T.SB.MasterStart[1];
04298                     from = Bin1->T.SB.MasterStop[Bin1->T.SB.MasterNumBlocks];
04299                     while ( from > next ) *--to = *--from;
04300                     next = to;
04301                     blin1 = 1;
04302                 }
04303                 else {
04304                     blin1++;
04305                 }
04306                 LOCK(Bin1->T.SB.MasterBlockLock[blin1]);
04307                 Bin1->T.SB.MasterBlock = blin1;
04308             }
04309             term1 = next;
04310 /*
04311             #[ One is smallest :
04312 */
04313         }
04314         else if ( c < 0 ) {
04315 /*
04316             #[ Two is smallest :
04317 */
04318             if ( SortBotOut(BHEAD term2) < 0 ) {
04319                 MLOCK(ErrorMessageLock);
04320                 MesPrint("Called from SortBotMerge with thread = %d",AT.identity);
04321                 MUNLOCK(ErrorMessageLock);
04322                 error = -1;
04323                 goto ReturnError;
04324             }
04325             next2:
04326             im = *term2;
04327             next = term2 + im;
04328             if ( next >= Bin2->T.SB.MasterStop[blin2] || ( *next
04329                 && next+*next+COMPINC > Bin2->T.SB.MasterStop[blin2] ) ) {
04330                 if ( blin2 == 1 ) {
04331                     UNLOCK(Bin2->T.SB.MasterBlockLock[Bin2->T.SB.MasterNumBlocks]);
04332                 }
04333                 else {
04334                     UNLOCK(Bin2->T.SB.MasterBlockLock[blin2-1]);
04335                 }
04336                 if ( blin2 == Bin2->T.SB.MasterNumBlocks ) {
04337 /*
04338                     Move the remainder down into block 0

```

```

04339         to = Bin2->T.SB.MasterStart[1];
04340         from = Bin2->T.SB.MasterStop[Bin2->T.SB.MasterNumBlocks];
04341         while ( from > next ) *--to = *--from;
04342         next = to;
04343         blin2 = 1;
04344     }
04345     else {
04346         blin2++;
04347     }
04348     LOCK(Bin2->T.SB.MasterBlockLock[blin2]);
04349     Bin2->T.SB.MasterBlock = blin2;
04350 }
04351 term2 = next;
04352 /*
04353     #] Two is smallest :
04354 */
04355 }
04356 else {
04357 /*
04358     #[ Equal :
04359 */
04360     l1 = *( m1 = term1 );
04361     l2 = *( m2 = term2 );
04362     if ( S->PolyWise ) { /* Here we work with PolyFun */
04363         ttl = m1;
04364         m1 += S->PolyWise;
04365         m2 += S->PolyWise;
04366         if ( S->PolyFlag == 2 ) {
04367             AT.WorkPointer = wp;
04368             w = poly_ratfun_add(BHEAD m1,m2);
04369             if ( *ttl + w[1] - m1[1] > AM.MaxTer/((LONG)sizeof(WORD)) ) {
04370                 MLOCK(ErrorMessageLock);
04371                 MesPrint("Term too complex in PolyRatFun addition. MaxTermSize of %10l is too
small",AM.MaxTer);
04372                 MUNLOCK(ErrorMessageLock);
04373                 Terminate(-1);
04374             }
04375             AT.WorkPointer = wp;
04376             if ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 && w[1] > FUNHEAD ) {
04377                 goto cancelled;
04378             }
04379         }
04380         else {
04381             w = wp;
04382             if ( w + m1[1] + m2[1] > AT.WorkTop ) {
04383                 MLOCK(ErrorMessageLock);
04384                 MesPrint("SortBotMerge(%d): A MaxtermSize of %10l is too small",
AT.identity,AM.MaxTer/sizeof(WORD));
04385                 MesPrint("m1[1] = %d, m2[1] = %d, Space =
%1",m1[1],m2[1],(LONG)(AT.WorkTop-wp));
04387                 PrintTerm(term1,"term1");
04388                 PrintTerm(term2,"term2");
04389                 MesPrint("PolyWise = %d",S->PolyWise);
04390                 MUNLOCK(ErrorMessageLock);
04391                 Terminate(-1);
04392             }
04393             AddArgs(BHEAD m1,m2,w);
04394         }
04395         r1 = w[1];
04396         if ( r1 <= FUNHEAD
|| ( w[FUNHEAD] == -SNUMBER && w[FUNHEAD+1] == 0 ) )
04397             { goto cancelled; }
04398         if ( r1 == m1[1] ) {
04399             NCOPY(m1,w,r1);
04400         }
04401         else if ( r1 < m1[1] ) {
04402             r2 = m1[1] - r1;
04403             m2 = w + r1;
04404             m1 += m1[1];
04405             while ( --r1 >= 0 ) *--m1 = *--m2;
04406             m2 = m1 - r2;
04407             r1 = S->PolyWise;
04408             while ( --r1 >= 0 ) *--m1 = *--m2;
04409             *m1 -= r2;
04410             term1 = m1;
04411         }
04412         else {
04413             r2 = r1 - m1[1];
04414             m2 = ttl - r2;
04415             r1 = S->PolyWise;
04416             m1 = ttl;
04417             *m1 += r2;
04418             term1 = m2;
04419             NCOPY(m2,m1,r1);
04420             r1 = w[1];
04421             NCOPY(m2,w,r1);
04422         }
04423     }

```

```

04424     }
04425 #ifdef WITHFLOAT
04426     else if ( AT.SortFloatMode ) {
04427 /*
04428         The terms are in m1/term1 and m2/term2 and their length in l1 and l2.
04429         We have to locate the floats which have already been
04430         verified in the compare routine. Once we have the new
04431         term we can jump to the code that writes away the
04432         result after adding the rationals.
04433 */
04434         tstop1 = m1+1; size1 = tstop1[-1]; tstop1 -= ABS(size1);
04435         tstop2 = m2+1; size2 = tstop2[-1]; tstop2 -= ABS(size2);
04436         if ( AT.SortFloatMode == 3 ) {
04437             fun1 = m1+1;
04438             while ( fun1[0] != FLOATFUN && fun1+fun1[1] < tstop1 ) {
04439                 fun1 += fun1[1];
04440             }
04441             if ( size1 < 0 ) fun1[FUNHEAD+3] = -fun1[FUNHEAD+3];
04442             UnpackFloat(aux1,fun1);
04443             fun2 = m2+1;
04444             while ( fun2[0] != FLOATFUN && fun2+fun2[1] < tstop2 ) {
04445                 fun2 += fun2[1];
04446             }
04447             if ( size2 < 0 ) fun2[FUNHEAD+3] = -fun2[FUNHEAD+3];
04448             UnpackFloat(aux2,fun2);
04449         }
04450         else if ( AT.SortFloatMode == 1 ) {
04451             fun1 = m1+1;
04452             while ( fun1[0] != FLOATFUN && fun1+fun1[1] < tstop1 ) {
04453                 fun1 += fun1[1];
04454             }
04455             if ( size1 < 0 ) fun1[FUNHEAD+3] = -fun1[FUNHEAD+3];
04456             UnpackFloat(aux1,fun1);
04457             fun2 = tstop2;
04458             RatToFloat(aux2,(UWORD *)fun2,size2);
04459         }
04460         else if ( AT.SortFloatMode == 2 ) {
04461             fun1 = tstop1;
04462             RatToFloat(aux1,(UWORD *)fun1,size1);
04463             fun2 = m2+1;
04464             while ( fun2[0] != FLOATFUN && fun2+fun2[1] < tstop2 ) {
04465                 fun2 += fun2[1];
04466             }
04467             if ( size2 < 0 ) fun2[FUNHEAD+3] = -fun2[FUNHEAD+3];
04468             UnpackFloat(aux2,fun2);
04469         }
04470         else {
04471             MLOCK(ErrorMessageLock);
04472             MesPrint("Illegal value %d for AT.SortFloatMode in
SortBotMerge.",AT.SortFloatMode);
04473             MUNLOCK(ErrorMessageLock);
04474             Terminate(-1);
04475             return(0);
04476         }
04477         mpf_add(aux3,aux1,aux2);
04478         size3 = mpf_sgn(aux3);
04479         if ( size3 == 0 ) { /* Cancelling! Rare! */
04480             goto cancelled;
04481         }
04482         else if ( size3 < 0 ) mpf_neg(aux3,aux3);
04483         fun3 = TermMalloc("MasterMerge");
04484         PackFloat(fun3,aux3);
04485         l3 = fun3[1]+(fun1-m1)+3; /* new size */
04486
04487         if ( l3 != l1 ) {
04488 /*
04489             Copy it all to wp
04490 */
04491             if ( (l3)*((LONG)sizeof(WORD)) >= AM.MaxTer ) {
04492                 MLOCK(ErrorMessageLock);
04493                 MesPrint("MasterMerge: Coefficient overflow during sort");
04494                 MUNLOCK(ErrorMessageLock);
04495                 goto ReturnError;
04496             }
04497             m3 = wp; m2 = term1;
04498             while ( m2 < fun1 ) *m3++ = *m2++;
04499             for ( jj = 0; jj < fun3[1]; jj++ ) *m3++ = fun3[jj];
04500             *m3++ = 1; *m3++ = 1;
04501             *m3++ = size3 < 0 ? -3: 3;
04502             *wp = m3-wp;
04503             TermFree(fun3,"MasterMerge");
04504             goto PutOutwp;
04505         }
04506         for ( jj = 0; jj < fun3[1]; jj++ ) fun1[jj] = fun3[jj];
04507         fun1 += fun3[1];
04508         *fun1++ = 1; *fun1++ = 1;
04509         *fun1++ = size3 < 0 ? -3: 3;

```



```

04510         *term1 = fun1-term1;
04511         TermFree(fun3,"MasterMerge");
04512     }
04513 #endif
04514     else {
04515         r1 = *( m1 += 11 - 1 );
04516         m1 -= ABS(r1) - 1;
04517         r1 = ( ( r1 > 0 ) ? (r1-1) : (r1+1) ) » 1;
04518         r2 = *( m2 += 12 - 1 );
04519         m2 -= ABS(r2) - 1;
04520         r2 = ( ( r2 > 0 ) ? (r2-1) : (r2+1) ) » 1;
04521
04522         if ( AddRat(BHEAD (UWORD *)m1,r1,(UWORD *)m2,r2,coef,&r3) ) {
04523             MLOCK(ErrorMessageLock);
04524             MesCall("SortBotMerge");
04525             MUNLOCK(ErrorMessageLock);
04526             SETERROR(-1)
04527         }
04528
04529         if ( AN.ncmod != 0 ) {
04530             if ( ( AC.modmode & POSNEG ) != 0 ) {
04531                 NormalModulus(coef,&r3);
04532             }
04533             else if ( BigLong(coef,r3,(UWORD *)AC.cmod,ABS(AN.ncmod)) >= 0 ) {
04534                 SubPLon(coef,r3,(UWORD *)AC.cmod,ABS(AN.ncmod),coef,&r3);
04535                 coef[r3] = 1;
04536                 for ( ii = 1; ii < r3; ii++ ) coef[r3+ii] = 0;
04537             }
04538         }
04539         if ( !r3 ) { goto cancelled; }
04540         r3 *= 2;
04541         r33 = ( r3 > 0 ) ? ( r3 + 1 ) : ( r3 - 1 );
04542         if ( r3 < 0 ) r3 = -r3;
04543         if ( r1 < 0 ) r1 = -r1;
04544         r1 *= 2;
04545         r31 = r3 - r1;
04546         if ( !r31 ) { /* copy coef into term1 */
04547             m2 = (WORD *)coef; im = r3;
04548             NCOPY(m1,m2,im);
04549             *m1 = r33;
04550         }
04551         else {
04552             to = wp; from = term1;
04553             while ( from < m1 ) *to++ = *from++;
04554             from = (WORD *)coef; im = r3;
04555             NCOPY(to,from,im);
04556             *to++ = r33;
04557             wp[0] = to - wp;
04558 PutOutwp:
04559             if ( SortBotOut(BHEAD wp) < 0 ) {
04560                 MLOCK(ErrorMessageLock);
04561                 MesPrint("Called from SortBotMerge with thread = %d",AT.identity);
04562                 MUNLOCK(ErrorMessageLock);
04563                 error = -1;
04564                 goto ReturnError;
04565             }
04566             goto cancelled;
04567         }
04568     }
04569     if ( SortBotOut(BHEAD term1) < 0 ) {
04570         MLOCK(ErrorMessageLock);
04571         MesPrint("Called from SortBotMerge with thread = %d",AT.identity);
04572         MUNLOCK(ErrorMessageLock);
04573         error = -1;
04574         goto ReturnError;
04575     }
04576 cancelled:; /* Now we need two new terms */
04577     im = *term1;
04578     next = term1 + im;
04579     if ( next >= Bin1->T.SB.MasterStop[blin1] || ( *next &&
04580 next+*next+COMPINC > Bin1->T.SB.MasterStop[blin1] ) ) {
04581         if ( blin1 == 1 ) {
04582             UNLOCK(Bin1->T.SB.MasterBlockLock[Bin1->T.SB.MasterNumBlocks]);
04583         }
04584         else {
04585             UNLOCK(Bin1->T.SB.MasterBlockLock[blin1-1]);
04586         }
04587         if ( blin1 == Bin1->T.SB.MasterNumBlocks ) {
04588             /*
04589             Move the remainder down into block 0
04590             */
04591             to = Bin1->T.SB.MasterStart[1];
04592             from = Bin1->T.SB.MasterStop[Bin1->T.SB.MasterNumBlocks];
04593             while ( from > next ) *--to = *--from;
04594             next = to;
04595             blin1 = 1;
04596         }

```

```

04597         else {
04598             blin1++;
04599         }
04600         LOCK(Bin1->T.SB.MasterBlockLock[blin1]);
04601         Bin1->T.SB.MasterBlock = blin1;
04602     }
04603     term1 = next;
04604     goto next2;
04605 /*
04606     #] Equal :
04607 */
04608 }
04609 }
04610 /*
04611     Copy the tail
04612 */
04613     if ( *term1 ) {
04614 /*
04615         #] Tail in one :
04616 */
04617         while ( *term1 ) {
04618             if ( SortBotOut(BHEAD term1) < 0 ) {
04619                 MLOCK(ErrorMessageLock);
04620                 MesPrint("Called from SortBotMerge with thread = %d",AT.identity);
04621                 MUNLOCK(ErrorMessageLock);
04622                 error = -1;
04623                 goto ReturnError;
04624             }
04625             im = *term1;
04626             next = term1 + im;
04627             if ( next >= Bin1->T.SB.MasterStop[blin1] || ( *next &&
04628                 next+*next+COMPINC > Bin1->T.SB.MasterStop[blin1] ) ) {
04629                 if ( blin1 == 1 ) {
04630                     UNLOCK(Bin1->T.SB.MasterBlockLock[Bin1->T.SB.MasterNumBlocks]);
04631                 }
04632                 else {
04633                     UNLOCK(Bin1->T.SB.MasterBlockLock[blin1-1]);
04634                 }
04635                 if ( blin1 == Bin1->T.SB.MasterNumBlocks ) {
04636 /*
04637                     Move the remainder down into block 0
04638 */
04639                     to = Bin1->T.SB.MasterStart[1];
04640                     from = Bin1->T.SB.MasterStop[Bin1->T.SB.MasterNumBlocks];
04641                     while ( from > next ) *--to = *--from;
04642                     next = to;
04643                     blin1 = 1;
04644                 }
04645                 else {
04646                     blin1++;
04647                 }
04648                 LOCK(Bin1->T.SB.MasterBlockLock[blin1]);
04649                 Bin1->T.SB.MasterBlock = blin1;
04650             }
04651             term1 = next;
04652         }
04653 /*
04654         #] Tail in one :
04655 */
04656     }
04657     else if ( *term2 ) {
04658 /*
04659         #] Tail in two :
04660 */
04661         while ( *term2 ) {
04662             if ( SortBotOut(BHEAD term2) < 0 ) {
04663                 MLOCK(ErrorMessageLock);
04664                 MesPrint("Called from SortBotMerge with thread = %d",AT.identity);
04665                 MUNLOCK(ErrorMessageLock);
04666                 error = -1;
04667                 goto ReturnError;
04668             }
04669             im = *term2;
04670             next = term2 + im;
04671             if ( next >= Bin2->T.SB.MasterStop[blin2] || ( *next
04672                 && next+*next+COMPINC > Bin2->T.SB.MasterStop[blin2] ) ) {
04673                 if ( blin2 == 1 ) {
04674                     UNLOCK(Bin2->T.SB.MasterBlockLock[Bin2->T.SB.MasterNumBlocks]);
04675                 }
04676                 else {
04677                     UNLOCK(Bin2->T.SB.MasterBlockLock[blin2-1]);
04678                 }
04679                 if ( blin2 == Bin2->T.SB.MasterNumBlocks ) {
04680 /*
04681                     Move the remainder down into block 0
04682 */
04683                     to = Bin2->T.SB.MasterStart[1];

```

```

04684         from = Bin2->T.SB.MasterStop[Bin2->T.SB.MasterNumBlocks];
04685         while ( from > next ) --to = ---from;
04686         next = to;
04687         blin2 = 1;
04688     }
04689     else {
04690         blin2++;
04691     }
04692     LOCK(Bin2->T.SB.MasterBlockLock[blin2]);
04693     Bin2->T.SB.MasterBlock = blin2;
04694 }
04695     term2 = next;
04696 }
04697 /*
04698     #] Tail in two :
04699 */
04700 }
04701     SortBotOut(BHEAD 0);
04702 ReturnError;;
04703 /*
04704     Release all locks
04705 */
04706     UNLOCK(Bin1->T.SB.MasterBlockLock[blin1]);
04707     if ( blin1 > 1 ) {
04708         UNLOCK(Bin1->T.SB.MasterBlockLock[blin1-1]);
04709     }
04710     else {
04711         UNLOCK(Bin1->T.SB.MasterBlockLock[Bin1->T.SB.MasterNumBlocks]);
04712     }
04713     UNLOCK(Bin2->T.SB.MasterBlockLock[blin2]);
04714     if ( blin2 > 1 ) {
04715         UNLOCK(Bin2->T.SB.MasterBlockLock[blin2-1]);
04716     }
04717     else {
04718         UNLOCK(Bin2->T.SB.MasterBlockLock[Bin2->T.SB.MasterNumBlocks]);
04719     }
04720     if ( AT.identity > 0 ) {
04721         UNLOCK(AT.SB.MasterBlockLock[AT.SB.FillBlock]);
04722     }
04723 /*
04724     And that was all folks
04725 */
04726     return(error);
04727 }
04728
04729 #endif
04730
04731 /*
04732     #] SortBotMerge :
04733     #[ IniSortBlocks :
04734 */
04735
04736 static int SortBlocksInitialized = 0;
04737
04744 int IniSortBlocks(int numworkers)
04745 {
04746     ALLPRIVATES *B;
04747     SORTING *S;
04748     LONG totalsize, workersize, blocksize, numberofterms;
04749     int maxter, id, j;
04750     int numberofblocks = NUMBEROFBLOCKSINSERT, numparts;
04751     WORD *w;
04752
04753     if ( SortBlocksInitialized ) return(0);
04754     SortBlocksInitialized = 1;
04755     if ( numworkers == 0 ) return(0);
04756
04757 #ifdef WITHSORTBOTS
04758     if ( numworkers > 2 ) {
04759         numparts = 2*numworkers - 2;
04760         numberofblocks = numberofblocks/2;
04761     }
04762     else {
04763         numparts = numworkers;
04764     }
04765 #else
04766     numparts = numworkers;
04767 #endif
04768     S = AM.S0;
04769     totalsize = S->LargeSize + S->SmallEsize;
04770     workersize = totalsize / numparts;
04771     maxter = AM.MaxTer/sizeof(WORD);
04772     blocksize = ( workersize - maxter )/numberofblocks;
04773     numberofterms = blocksize / maxter;
04774     if ( numberofterms < MINIMUMNUMBEROFTERMS ) {
04775 /*
04776         This should have been taken care of in RecalcSetups.

```

```

04777 */
04778     MesPrint("We have a problem with the size of the blocks in IniSortBlocks");
04779     Terminate(-1);
04780 }
04781 /*
04782     Layout:  For each worker
04783               block 0: size is maxter WORDS
04784               numberofblocks blocks of size blocksize WORDS
04785 */
04786 w = S->lBuffer;
04787 if ( w == 0 ) w = S->sBuffer;
04788 for ( id = 1; id <= numparts; id++ ) {
04789     B = AB[id];
04790     AT.SB.MasterBlockLock = (pthread_mutex_t *)Malloc1(
04791         sizeof(pthread_mutex_t)*(numberofblocks+1),"MasterBlockLock");
04792     AT.SB.MasterStart = (WORD **)Malloc1(sizeof(WORD *)*(numberofblocks+1)*3,"MasterBlock");
04793     AT.SB.MasterFill = AT.SB.MasterStart + (numberofblocks+1);
04794     AT.SB.MasterStop = AT.SB.MasterFill + (numberofblocks+1);
04795     AT.SB.MasterNumBlocks = numberofblocks;
04796     AT.SB.MasterBlock = 0;
04797     AT.SB.FillBlock = 0;
04798     AT.SB.MasterFill[0] = AT.SB.MasterStart[0] = w;
04799     w += maxter;
04800     AT.SB.MasterStop[0] = w;
04801     AT.SB.MasterBlockLock[0] = dummylock;
04802     for ( j = 1; j <= numberofblocks; j++ ) {
04803         AT.SB.MasterFill[j] = AT.SB.MasterStart[j] = w;
04804         w += blocksize;
04805         AT.SB.MasterStop[j] = w;
04806         AT.SB.MasterBlockLock[j] = dummylock;
04807     }
04808 }
04809 if ( w > S->sTop2 ) {
04810     MesPrint("Counting problem in IniSortBlocks");
04811     Terminate(-1);
04812 }
04813 return(0);
04814 }
04815
04816 /*
04817     #] IniSortBlocks :
04818     #[ UpdateSortBlocks :
04819 */
04820
04825 int UpdateSortBlocks(int numworkers)
04826 {
04827     ALLPRIVATES *B;
04828     SORTING *S;
04829     LONG totalsize, workersize, blocksize, numberofterms;
04830     int maxter, id, j;
04831     int numberofblocks = NUMBEROFBLOCKSINSORT, numparts;
04832     WORD *w;
04833
04834     if ( numworkers == 0 ) return(0);
04835
04836 #ifndef WITHSORTBOTS
04837     if ( numworkers > 2 ) {
04838         numparts = 2*numworkers - 2;
04839         numberofblocks = numberofblocks/2;
04840     }
04841     else {
04842         numparts = numworkers;
04843     }
04844 #else
04845     numparts = numworkers;
04846 #endif
04847     S = AM.S0;
04848     totalsize = S->LargeSize + S->SmallEsize;
04849     workersize = totalsize / numparts;
04850     maxter = AM.MaxTer/sizeof(WORD);
04851     blocksize = ( workersize - maxter )/numberofblocks;
04852     numberofterms = blocksize / maxter;
04853     if ( numberofterms < MINIMUMNUMBEROFTERMS ) {
04854         /*
04855             This should have been taken care of in RecalcSetups.
04856         */
04857         MesPrint("We have a problem with the size of the blocks in UpdateSortBlocks");
04858         Terminate(-1);
04859     }
04860     /*
04861         Layout:  For each worker
04862                   block 0: size is maxter WORDS
04863                   numberofblocks blocks of size blocksize WORDS
04864     */
04865     w = S->lBuffer;
04866     if ( w == 0 ) w = S->sBuffer;
04867     for ( id = 1; id <= numparts; id++ ) {

```

```

04868     B = AB[id];
04869     AT.SB.MasterFill[0] = AT.SB.MasterStart[0] = w;
04870     w += maxter;
04871     AT.SB.MasterStop[0] = w;
04872     for ( j = 1; j <= numberofblocks; j++ ) {
04873         AT.SB.MasterFill[j] = AT.SB.MasterStart[j] = w;
04874         w += blocksize;
04875         AT.SB.MasterStop[j] = w;
04876     }
04877 }
04878 if ( w > S->sTop2 ) {
04879     MesPrint("Counting problem in UpdateSortBlocks");
04880     Terminate(-1);
04881 }
04882 return(0);
04883 }
04884
04885 /*
04886 #] UpdateSortBlocks :
04887 #[ DefineSortBotTree :
04888 */
04889 #ifdef WITHSORTBOTS
04890 void DefineSortBotTree(void)
04891 {
04892     ALLPRIVATES *B;
04893     int n, i, from;
04894     if ( numberofworkers <= 2 ) return;
04895     n = numberofworkers*2-2;
04896     for ( i = numberofworkers+1, from = 1; i <= n; i++ ) {
04897         B = AB[i];
04898         AT.SortBotIn1 = from++;
04899         AT.SortBotIn2 = from++;
04900     }
04901     B = AB[0];
04902     AT.SortBotIn1 = from++;
04903     AT.SortBotIn2 = from++;
04904 }
04905 #endif
04906
04907 /*
04908 #] DefineSortBotTree :
04909 #[ GetTerm2 :
04910
04911 Routine does a GetTerm but only when a bracket index is involved and
04912 only from brackets that have been judged not suitable for treatment
04913 as complete brackets by a single worker. Whether or not a bracket should
04914 be treated by a single worker is decided by TreatIndexEntry
04915 */
04916 WORD GetTerm2(PHEAD WORD *term)
04917 {
04918     WORD *ttco, *ttt, retval;
04919     LONG n,i;
04920     FILEHANDLE *fi;
04921     EXPRESSIONS e = AN.expr;
04922     BRACKETINFO *b = e->bracketinfo;
04923     BRACKETINDEX *bi = b->indexbuffer;
04924     POSITION where, eonfile = AS.OldOnFile[e-Expressions], bstart, bnext;
04925
04926     /*
04927     1: Get the current position.
04928     */
04929     switch ( e->status ) {
04930     case UNHIDELEXPRESSION:
04931     case UNHIDEGEXPRESSION:
04932     case DROPHLEXPRESSION:
04933     case DROPHGEXPRESSION:
04934     case HIDDENLEXPRESSION:
04935     case HIDDENGEXPRESSION:
04936         fi = AR.hidefile;
04937         break;
04938     default:
04939         fi = AR.infile;
04940         break;
04941     }
04942     if ( AR.KeptInHold ) {
04943         retval = GetTerm(BHEAD term);
04944         return(retval);
04945     }
04946     SeekScratch(fi,&where);
04947     if ( AN.lastinindex < 0 ) {
04948         /*
04949         We have to test whether we have to do the first bracket
04950         */
04951         if ( ( n = TreatIndexEntry(BHEAD 0) ) <= 0 ) {

```

```

04960         AN.lastindex = n;
04961         where = bi[n].start;
04962         ADD2POS(where,eonfile);
04963         SetScratch(fi,&where);
04964     /*
04965         Put the bracket in the Compress buffer.
04966     */
04967         ttco = AR.CompressBuffer;
04968         tt = b->bracketbuffer + bi[0].bracket;
04969         i = *tt;
04970         NCOPY(ttco,tt,i)
04971         AR.CompressPointer = ttco;
04972         retval = GetTerm(BHEAD term);
04973         return(retval);
04974     }
04975     else AN.lastindex = n-1;
04976 }
04977 /*
04978 2: Find the corresponding index number
04979 a: test whether it is in the current bracket
04980 */
04981 n = AN.lastindex;
04982 bstart = bi[n].start;
04983 ADD2POS(bstart,eonfile);
04984 bnext = bi[n].next;
04985 ADD2POS(bnext,eonfile);
04986 if ( ISLESSPOS(bstart,where) && ISLESSPOS(where,bnext) ) {
04987     retval = GetTerm(BHEAD term);
04988     return(retval);
04989 }
04990 for ( n++; n < b->indexfill; n++ ) {
04991     i = TreatIndexEntry(BHEAD n);
04992     if ( i <= 0 ) {
04993     /*
04994         Put the bracket in the Compress buffer.
04995     */
04996         ttco = AR.CompressBuffer;
04997         tt = b->bracketbuffer + bi[n].bracket;
04998         i = *tt;
04999         NCOPY(ttco,tt,i)
05000         AR.CompressPointer = ttco;
05001         AN.lastindex = n;
05002         where = bi[n].start;
05003         ADD2POS(where,eonfile);
05004         SetScratch(fi,&(where));
05005         retval = GetTerm(BHEAD term);
05006         return(retval);
05007     }
05008     else n += i - 1;
05009 }
05010 return(0);
05011 }
05012 /*
05013 #] GetTerm2 :
05014 #[ TreatIndexEntry :
05015 */
05016 int TreatIndexEntry(PHEAD LONG n)
05017 {
05018     BRACKETINFO *b = AN.expr->bracketinfo;
05019     LONG numbra = b->indexfill - 1, i;
05020     LONG totterms;
05021     BRACKETINDEX *bi;
05022     POSITION pos1, average;
05023     /*
05024     1: number of the bracket should be such that there is one bucket
05025     for each worker remaining.
05026     */
05027     if ( ( numbra - n ) <= numberofworkers ) return(0);
05028     /*
05029     2: size of the bracket contents should be less than what remains in
05030     the expression divided by the number of workers.
05031     */
05032     bi = b->indexbuffer;
05033     DIFPOS(pos1,bi[numbra].next,bi[n].next); /* Size of what remains */
05034     BASEPOSITION(average) = DIVPOS(pos1,(3*numberofworkers));
05035     DIFPOS(pos1,bi[n].next,bi[n].start); /* Size of the current bracket */
05036     if ( ISLESSPOS(average,pos1) ) return(0);
05037     /*
05038     It passes.
05039     Now check whether we can do more brackets
05040     */
05041     totterms = bi->termsinbracket;
05042     if ( totterms > 2*AC.ThreadBucketSize ) return(1);
05043     for ( i = 1; i < numbra-n; i++ ) {
05044         DIFPOS(pos1,bi[n+i].next,bi[n].start); /* Size of the combined brackets */
05045         if ( ISLESSPOS(average,pos1) ) return(i);
05046     }

```

```

05055         totterms += bi->termsinbracket;
05056         if ( totterms > 2*AC.ThreadBucketSize ) return(i+1);
05057     }
05058     /*
05059     We have a problem at the end of the system. Just do one.
05060     */
05061     return(1);
05062 }
05063
05064 /*
05065     #] TreatIndexEntry :
05066     #[ SetHideFiles :
05067     */
05068
05069 void SetHideFiles(void) {
05070     int i;
05071     ALLPRIVATES *B, *B0 = AB[0];
05072     for ( i = 1; i <= numberofworkers; i++ ) {
05073         B = AB[i];
05074         AR.hidefile->handle = AR0.hidefile->handle;
05075         if ( AR.hidefile->handle < 0 ) {
05076             AR.hidefile->PObuffer = AR0.hidefile->PObuffer;
05077             AR.hidefile->PStop = AR0.hidefile->PStop;
05078             AR.hidefile->POfill = AR0.hidefile->POfill;
05079             AR.hidefile->POfull = AR0.hidefile->POfull;
05080             AR.hidefile->POsize = AR0.hidefile->POsize;
05081             AR.hidefile->POposition = AR0.hidefile->POposition;
05082             AR.hidefile->filesize = AR0.hidefile->filesize;
05083         }
05084         else {
05085             AR.hidefile->PObuffer = AR.hidefile->wPObuffer;
05086             AR.hidefile->PStop = AR.hidefile->wPStop;
05087             AR.hidefile->POfill = AR.hidefile->wPOfill;
05088             AR.hidefile->POfull = AR.hidefile->wPOfull;
05089             AR.hidefile->POsize = AR.hidefile->wPOsize;
05090             PUTZERO(AR.hidefile->POposition);
05091         }
05092     }
05093 }
05094
05095 /*
05096     #] SetHideFiles :
05097     #[ IniFbufs :
05098     */
05099
05100 void IniFbufs(void)
05101 {
05102     int i;
05103     for ( i = 0; i < AM.totalnumberofthreads; i++ ) {
05104         IniFbuffer(AB[i]->T.fbufnum);
05105     }
05106 }
05107
05108 /*
05109     #] IniFbufs :
05110     #[ SetMods :
05111     */
05112
05113 void SetMods(void)
05114 {
05115     ALLPRIVATES *B;
05116     int i, n, j;
05117     for ( j = 0; j < AM.totalnumberofthreads; j++ ) {
05118         B = AB[j];
05119         AN.ncmod = AC.ncmod;
05120         if ( AN.cmod != 0 ) M_free(AN.cmod, "AN.cmod");
05121         n = ABS(AN.ncmod);
05122         AN.cmod = (UWORD *)Malloc1(sizeof(WORD)*n, "AN.cmod");
05123         for ( i = 0; i < n; i++ ) AN.cmod[i] = AC.cmod[i];
05124     }
05125 }
05126
05127 /*
05128     #] SetMods :
05129     #[ UnSetMods :
05130     */
05131
05132 void UnSetMods(void)
05133 {
05134     ALLPRIVATES *B;
05135     int j;
05136     for ( j = 0; j < AM.totalnumberofthreads; j++ ) {
05137         B = AB[j];
05138         if ( AN.cmod != 0 ) M_free(AN.cmod, "AN.cmod");
05139         AN.cmod = 0;
05140     }
05141 }

```

```

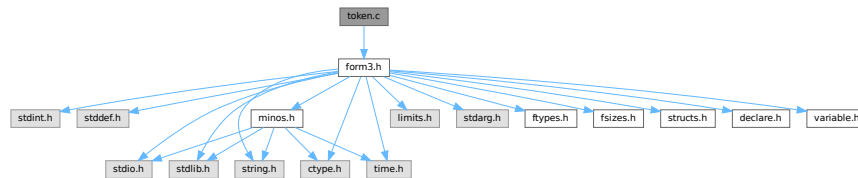
05142
05143 /*
05144  #] UnSetMods :
05145  #[ find_Horner_MCTS_expand_tree_threaded :
05146 */
05147
05148 void find_Horner_MCTS_expand_tree_threaded(void) {
05149     int id;
05150     while (( id = GetAvailableThread() ) < 0)
05151         MasterWait();
05152     WakeupThread(id,MCTSEXPANDTREE);
05153 }
05154
05155 /*
05156  #] find_Horner_MCTS_expand_tree_threaded :
05157  #[ optimize_expression_given_Horner_threaded :
05158 */
05159
05160 extern void optimize_expression_given_Horner_threaded(void) {
05161     int id;
05162     while (( id = GetAvailableThread() ) < 0)
05163         MasterWait();
05164     WakeupThread(id,OPTIMIZEEXPRESSION);
05165 }
05166
05167 /*
05168  #] optimize_expression_given_Horner_threaded :
05169 */
05170
05171 #endif

```

4.145 token.c File Reference

```
#include "form3.h"
```

Include dependency graph for token.c:



Macros

- #define [CHECKPOLY](#) {if(polyflag)MesPrint("&Illegal use of polynomial function"); polyflag = 0; }

Functions

- int [tokenize](#) (UBYTE *in, WORD leftright)
- void [WriteTokens](#) (SBYTE *in)
- int [simp1token](#) (SBYTE *s)
- int [simpwtoken](#) (SBYTE *s)
- int [simp2token](#) (SBYTE *s)
- int [simp3atoken](#) (SBYTE *s, int mode)
- int [simp3btoken](#) (SBYTE *s, int mode)
- int [simp4token](#) (SBYTE *s)
- int [simp5token](#) (SBYTE *s, int mode)
- int [simp6token](#) (SBYTE *tokens, int mode)

Variables

- `char * ttypes []`

4.145.1 Detailed Description

The tokenizer. This is a part of the compiler that does an intermediate type of translation. It does look up the names etc and can do a number of optimizations. The resulting output is a stream of bytes which can be processed by the code generator (in the file [compiler.c](#))

Definition in file [token.c](#).

4.145.2 Macro Definition Documentation

4.145.2.1 CHECKPOLY

```
#define CHECKPOLY {if(polyflag)MesPrint("&Illegal use of polynomial function"); polyflag = 0;
}
```

Definition at line 56 of file [token.c](#).

4.145.3 Function Documentation

4.145.3.1 tokenize()

```
int tokenize (
    UBYTE * in,
    WORD leftright )
```

Definition at line 58 of file [token.c](#).

4.145.3.2 WriteTokens()

```
void WriteTokens (
    SBYTE * in )
```

Definition at line 652 of file [token.c](#).

4.145.3.3 simp1token()

```
int simp1token (
    SBYTE * s )
```

Definition at line 700 of file [token.c](#).

4.145.3.4 simpwtoken()

```
int simpwtoken (
    SBYTE * s )
```

Definition at line 789 of file [token.c](#).

4.145.3.5 simp2token()

```
int simp2token (
    SBYTE * s )
```

Definition at line 943 of file [token.c](#).

4.145.3.6 simp3atoken()

```
int simp3atoken (
    SBYTE * s,
    int mode )
```

Definition at line 1191 of file [token.c](#).

4.145.3.7 simp3btoken()

```
int simp3btoken (
    SBYTE * s,
    int mode )
```

Definition at line 1432 of file [token.c](#).

4.145.3.8 simp4token()

```
int simp4token (
    SBYTE * s )
```

Definition at line 1843 of file [token.c](#).

4.145.3.9 simp5token()

```
int simp5token (
    SBYTE * s,
    int mode )
```

Definition at line 2003 of file [token.c](#).

4.145.3.10 simp6token()

```
int simp6token (
    SBYTE * tokens,
    int mode )
```

Definition at line 2049 of file [token.c](#).

4.145.4 Variable Documentation

4.145.4.1 ttypes

```
char* ttypes[]
```

Initial value:

```
= { "\n", "S", "I", "V", "F", "set", "E", "dotp", "#",
    "sub", "d_", "$", "dub", "(", ")", "?", "??", ".", "[", "]",
    ",", "(", ")", "*", "/", "^", "+", "-", "!", "end", "{", "}",
    "N_?", "conj", "()", "#d", "^d", "_", "snum" }
```

Definition at line 647 of file [token.c](#).

4.146 token.c

[Go to the documentation of this file.](#)

```
00001
00008 /* # [ License : */
00009 /*
00010 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00011 * When using this file you are requested to refer to the publication
00012 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00013 * This is considered a matter of courtesy as the development was paid
00014 * for by FOM the Dutch physics granting agency and we would like to
00015 * be able to track its scientific use to convince FOM of its value
00016 * for the community.
00017 *
00018 * This file is part of FORM.
00019 *
00020 * FORM is free software: you can redistribute it and/or modify it under the
00021 * terms of the GNU General Public License as published by the Free Software
00022 * Foundation, either version 3 of the License, or (at your option) any later
00023 * version.
00024 *
00025 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00026 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00027 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00028 * details.
00029 *
00030 * You should have received a copy of the GNU General Public License along
00031 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00032 */
00033 /* # [ License : */
00034 /*
00035 * # [ Includes :
00036 */
00037
00038 #include "form3.h"
00039
00040 /*
00041 * # [ Includes :
00042 * # [ Compiler :
00043 * # [ tokenize :
00044
00045 * Takes the input in 'in' and translates it into tokens.
00046 * The tokens are put in the token buffer which starts at 'AC.tokens'
00047 * and runs till 'AC.toptokens'
00048 * We may assume that the various types of brackets match properly.
00049 * object = -1: after , or (
00050 * object = 0: name/variable/number etc is allowed
```

```

00051         object = 1: variable.
00052         object = 2: number
00053         object = 3: ) after subexpression
00054 */
00055
00056 #define CHECKPOLY {if(polyflag)MesPrint("&Illegal use of polynomial function"); polyflag = 0; }
00057
00058 int tokenize(UBYTE *in, WORD leftright)
00059 {
00060     int error = 0, object, funlevel = 0, bracelevel = 0, explevel = 0, numexp;
00061     int polyflag = 0;
00062     WORD number, type;
00063     UBYTE *s = in, c;
00064     SBYTE *out, *outtop, num[MAXNUMSIZE], *t;
00065     LONG i;
00066     if ( AC.tokens == 0 ) {
00067         SBYTE **ppp = &(AC.tokens); /* to avoid a compiler warning */
00068         SBYTE **pppp = &(AC.toptokens);
00069         DoubleBuffer((void **)ppp, (void **)pppp, sizeof(SBYTE), "start tokens");
00070     }
00071     out = AC.tokens;
00072     outtop = AC.toptokens - MAXNUMSIZE;
00073     AC.dumnumflag = 0;
00074     object = 0;
00075     while ( *in ) {
00076         if ( out > outtop ) {
00077             LONG oldsize = (LONG)(out - AC.tokens);
00078             SBYTE **ppp = &(AC.tokens); /* to avoid a compiler warning */
00079             SBYTE **pppp = &(AC.toptokens);
00080             DoubleBuffer((void **)ppp, (void **)pppp, sizeof(SBYTE), "expand tokens");
00081             out = AC.tokens + oldsize;
00082             outtop = AC.toptokens - MAXNUMSIZE;
00083         }
00084         switch ( FG.cTable[*in] ) {
00085             case 0: /* a-zA-Z */
00086                 CHECKPOLY
00087                 s = in++;
00088                 while ( FG.cTable[*in] == 0 || FG.cTable[*in] == 1
00089                     || *in == '_' ) in++;
00090             dovariable: c = *in; *in = 0;
00091                 if ( object > 0 ) {
00092                     MesPrint("&Illegal position for %s",s);
00093                     if ( !error ) error = 1;
00094                 }
00095                 if ( out > AC.tokens && ( out[-1] == TWILDCARD || out[-1] == TNOT ) ) {
00096                     type = GetName(AC.varnames,s,&number,NOAUTO);
00097                 }
00098                 else {
00099                     type = GetName(AC.varnames,s,&number,WITHAUTO);
00100                 }
00101                 if ( type < 0 )
00102                     type = GetName(AC.exprnames,s,&number,NOAUTO);
00103                 switch ( type ) {
00104                     case CSYMBOL:          *out++ = TSYMBOL;          break;
00105                     case CINDEX:
00106                         if ( number >= (AM.IndDum-AM.OffsetIndex) ) {
00107                             if ( c != '?' ) {
00108                                 MesPrint("&Generated indices should be of the type Nnumber_?");
00109                                 error = 1;
00110                             }
00111                             else {
00112                                 *in++ = c; c = *in; *in = 0;
00113                                 AC.dumnumflag = 1;
00114                             }
00115                         }
00116                         *out++ = TINDEX;
00117                         break;
00118                     case CVECTOR:          *out++ = TVECTOR;          break;
00119                     case CFUNCTION:
00120 #ifdef WITHMPI
00121                         /*
00122                          * In the preprocessor, random functions in #svar=... and #inside
00123                          * may cause troubles, because the program flow on a slave may be
00124                          * different from those on others. We set AC.RhsExprInModuleFlag in order
00125                          * to make the change of $-variable be done on the master and thus keep the
00126                          * consistency among the master and all slave processes. The previous value
00127                          * of AC.RhsExprInModuleFlag will be restored after #svar=... and #inside.
00128                          */
00129                         if ( AP.PreAssignFlag || AP.PreInsideLevel ) {
00130                             switch ( number + FUNCTION ) {
00131                                 case RANDOMFUNCTION:
00132                                 case RANPERM:
00133                                     AC.RhsExprInModuleFlag = 1;
00134                             }
00135                         }
00136 #endif
00137                         *out++ = TFUNCTION;

```

```

00138             break;
00139         case CSET: *out++ = TSET; break;
00140         case CEXPRESSION: *out++ = TEXPRESSION;
00141             if ( leftright == LHSIDE ) {
00142                 if ( !error ) error = 1;
00143                 MesPrint("&Expression not allowed in LH-side of
substitution: %s",s);
00144             }
00145         /*[06nov2003 mt]:*/
00146         #ifdef WITHMPI
00147             else { /*RHSide*/
00148                 /* NOTE: We always set AC.RhsExprInModuleFlag regardless
of
00149                 * AP.PreAssignFlag or AP.PreInsideLevel because we
have to detect
00150                 * RHS expressions even in those cases. */
00151                 AC.RhsExprInModuleFlag = 1;
00152             }
00153             if ( !AP.PreAssignFlag && !AP.PreInsideLevel )
00154                 Expressions[number].vflags |= ISINRHS;
00155         #endif
00156         /*:[06nov2003 mt]*/
00157             if ( AC.exprfillwarning == 0 ) {
00158                 AC.exprfillwarning = 1;
00159             }
00160             break;
00161             case CDELTA: *out++ = TDELTA; *in = c;
00162                 object = 1; continue;
00163             case CDUBIOUS: *out++ = TDUBIOUS; break;
00164             default: *out++ = TDUBIOUS;
00165                 if ( !error ) error = 1;
00166                 MesPrint("&Undeclared variable %s",s);
00167                 number = AddDubious(s);
00168                 break;
00169             }
00170             object = 1;
00171         donumber: i = 0;
00172             do { num[i++] = (SBYTE)(number & 0x7F); number >>= 7; } while ( number );
00173             while ( --i >= 0 ) *out++ = num[i];
00174             *in = c;
00175             break;
00176             case 1: /* 0-9 */
00177             {
00178                 #ifdef WITHFLOAT
00179                     int spec;
00180                     UBYTE *in2;
00181                 #endif
00182                 CHECKPOLY
00183                 s = in;
00184                 while ( *s == '0' && FG.cTable[s[1]] == 1 ) s++;
00185                 in = s+1; i = 1;
00186                 while ( FG.cTable[*in] == 1 ) { in++; i++; }
00187                 if ( object > 0 ) {
00188                     c = *in; *in = 0;
00189                     MesPrint("&Illegal position for %s",s);
00190                     *in = c;
00191                     if ( !error ) error = 1;
00192                 }
00193                 if ( i == 1 && *in == '_' && ( *s == '5' || *s == '6'
|| *s == '7' ) ) {
00194                     in++; *out++ = TSGAMMA; *out++ = (SBYTE)(*s - '4');
00195                     object = 1;
00196                     break;
00197                 }
00198             }
00199         #ifdef WITHFLOAT
00200             in2 = CheckFloat(in,&spec);
00201             if ( in2 > in ) {
00202                 if ( spec == -1 ) {
00203                     MesPrint("&The floating point system has not been started: %s",in);
00204                     if ( !error ) error = 1;
00205                 }
00206                 else {
00207                     in = in2;
00208                 }
00209             }
00210             dofloat:
00211                 while ( out + (in-s) >= AC.toptokens ) {
00212                     LONG oldsize = (LONG)(out - AC.tokens);
00213                     SBYTE **ppp = &(AC.tokens); /* to avoid a compiler warning */
00214                     SBYTE **pppp = &(AC.toptokens);
00215                     DoubleBuffer((void **)ppp,(void **)pppp,sizeof(SBYTE),"more tokens");
00216                     out = AC.tokens + oldsize;
00217                     outtop = AC.toptokens - MAXNUMSIZE;
00218                 }
00219                 *out++ = TFLOAT;
00220                 while ( s < in ) *out++ = *s++;
00221             }
00222             else

```

```

00222 #endif
00223 {
00224     *out++ = TNUMBER;
00225     if ( ( i & 1 ) != 0 ) *out++ = (SBYTE) (*s++ - '0');
00226     while ( out + (in-s)/2 >= AC.toptokens ) {
00227         LONG oldsize = (LONG) (out - AC.tokens);
00228         SBYTE **ppp = &(AC.tokens); /* to avoid a compiler warning */
00229         SBYTE **pppp = &(AC.toptokens);
00230         DoubleBuffer((void **)ppp, (void **)pppp, sizeof(SBYTE), "more tokens");
00231         out = AC.tokens + oldsize;
00232         outtop = AC.toptokens - MAXNUMSIZE;
00233     }
00234     while ( s < in ) { /* We store in base 100 */
00235         *out++ = (SBYTE) (( *s - '0' ) * 10 + ( s[1] - '0' ));
00236         s += 2;
00237     }
00238 }
00239 object = 2;
00240 }
00241 break;
00242 case 2: /* . $ _ ? # ' */
00243     CHECKPOLY
00244     if ( *in == '?' ) {
00245         if ( leftright == LHSIDE ) {
00246             if ( object == 1 ) { /* follows a name */
00247                 in++; *out++ = TWILDCARD;
00248                 if ( FG.cTable[in[0]] == 0 || in[0] == '[' || in[0] == '{' ) object = 0;
00249             }
00250             else if ( object == -1 ) { /* follows comma or ( */
00251                 in++; s = in;
00252                 while ( FG.cTable[*in] == 0 || FG.cTable[*in] == 1 ) in++;
00253                 c = *in; *in = 0;
00254                 if ( FG.cTable[*s] != 0 ) {
00255                     MesPrint("&Illegal name for argument list variable %s",s);
00256                     error = 1;
00257                 }
00258                 else {
00259                     i = AddWildcardName((UBYTE *)s);
00260                     *in = c;
00261                     *out++ = TWILDARG;
00262                     *out++ = (SBYTE)i;
00263                 }
00264                 object = 1;
00265             }
00266             else {
00267                 MesPrint("&Illegal position for ?");
00268                 error = 1;
00269                 in++;
00270             }
00271         }
00272         else {
00273             if ( object != -1 ) goto IllPos;
00274             in++;
00275             if ( FG.cTable[*in] == 0 || FG.cTable[*in] == 1 ) {
00276                 s = in;
00277                 while ( FG.cTable[*in] == 0 || FG.cTable[*in] == 1 ) in++;
00278                 c = *in; *in = 0;
00279                 i = GetWildcardName((UBYTE *)s);
00280                 if ( i <= 0 ) {
00281                     MesPrint("&Undefined argument list variable %s",s);
00282                     error = 1;
00283                 }
00284                 *in = c;
00285                 *out++ = TWILDARG;
00286                 *out++ = (SBYTE)i;
00287             }
00288             else {
00289                 if ( AC.vectorlikeLHS == 0 ) {
00290                     MesPrint("&Generated index ? only allowed in vector substitution",s);
00291                     error = 1;
00292                 }
00293                 *out++ = TGENINDEX;
00294             }
00295             object = 1;
00296         }
00297     }
00298     else if ( *in == '.' ) {
00299         if ( object == 1 ) { /* follows a name */
00300             *out++ = TDOT;
00301             object = 0;
00302             in++;
00303         }
00304     }
00305     #ifndef WITHFLOAT
00306     else if ( object == 0 || object == -1 ) {
00307         /*
00308         Test for floating point number
00309         */

```

```

00309         int spec;
00310         s = CheckFloat(in,&spec);
00311         if ( s > in ) {
00312             if ( spec == -1 ) {
00313                 MesPrint("&The floating point system has not been started: %s",in);
00314                 if ( !error ) error = 1;
00315                 in++;
00316             }
00317             else {
00318                 UBYTE *a = s; s = in; in = a;
00319                 goto dofloat;
00320             }
00321         }
00322         else goto IllPos;
00323     }
00324 #endif
00325     else goto IllPos;
00326 }
00327 else if ( *in == '$' ) { /* $ variable */
00328     in++;
00329     s = in;
00330     if ( FG.cTable[*in] == 0 ) {
00331         while ( FG.cTable[*in] == 0 || FG.cTable[*in] == 1 ) in++;
00332         if ( *in == '_' && AP.PreAssignFlag == 2 ) in++;
00333         c = *in; *in = 0;
00334         if ( object > 0 ) {
00335             if ( object != 1 || leftright == RHSIDE ) {
00336                 MesPrint("&Illegal position for $%s",s);
00337                 if ( !error ) error = 1;
00338                 /* else can be assignment in wildcard */
00339             }
00340             else {
00341                 if ( ( number = GetDollar(s) ) < 0 ) {
00342                     number = AddDollar(s,0,0,0);
00343                 }
00344             }
00345             else if ( ( number = GetDollar(s) ) < 0 ) {
00346                 MesPrint("&Undefined variable $%s",s);
00347                 if ( !error ) error = 1;
00348                 number = AddDollar(s,0,0,0);
00349             }
00350             *out++ = TDOLLAR;
00351             object = 1;
00352             if ( ( AC.exprfillwarning == 0 ) &&
00353                 ( out > AC.tokens+1 ) && ( out[-2] != TWILDCARD ) ) {
00354                 AC.exprfillwarning = 1;
00355             }
00356             goto donumber;
00357         }
00358         else {
00359             MesPrint("&Illegal name for $ variable after %s",in);
00360             if ( !error ) error = 1;
00361         }
00362     }
00363     else if ( *in == '#' ) {
00364         in++;
00365         if ( object == 1 ) { /* follows a name */
00366             *out++ = TCONJUGATE;
00367         }
00368         else {
00369             MesPrint("&Illegal position for %#",in);
00370             error = 1;
00371         }
00372     }
00373     else goto IllPos;
00374     break;
00375 case 3: /* [ ] */
00376     CHECKPOLY
00377     if ( *in == '[' ) {
00378         if ( object == 1 ) { /* after name */
00379             t = out-1;
00380             if ( *t == RPARENTHESIS ) {
00381                 *out++ = LBACE; *out++ = LPARENTHESIS;
00382                 bracelevel++; explevel = bracelevel;
00383             }
00384             else {
00385                 while ( *t >= 0 && t > AC.tokens ) t--;
00386                 if ( *t == TEXPRESSION ) {
00387                     *out++ = LBACE; *out++ = LPARENTHESIS;
00388                     bracelevel++; explevel = bracelevel;
00389                 }
00390                 else { *out++ = LBACE; bracelevel++; }
00391             }
00392             object = 0;
00393         }
00394         else { /* name. find matching ] */
00395             s = in;

```

```

00396         in = SkipAName(in);
00397         goto dovariable;
00398     }
00399 }
00400 else {
00401     if ( explevel > 0 && explevel == bracelevel ) {
00402         *out++ = RPARENTHESIS; explevel = 0;
00403     }
00404     *out++ = RBACE; object = 1; bracelevel--;
00405 }
00406 in++;
00407 break;
00408 case 4: /* ( ) = ; , */
00409     if ( *in == '(' ) {
00410         if ( funlevel >= AM.MaxParLevel ) {
00411             MesPrint("&More than %d levels of parentheses",AM.MaxParLevel);
00412             return(-1);
00413         }
00414         if ( object == 1 ) { /* After name -> function,vector */
00415             AC.tokenarglevel[funlevel++] = TYPEISFUN;
00416             *out++ = TFUNOPEN;
00417             if ( polyflag ) {
00418                 if ( in[1] != ')' && in[1] != ',' ) {
00419                     *out++ = TNUMBER; *out++ = (SBYTE)(polyflag);
00420                     *out++ = TCOMMA;
00421                     *out++ = LPARENTHESIS;
00422                 }
00423                 else {
00424                     *out++ = LPARENTHESIS;
00425                     *out++ = TNUMBER; *out++ = (SBYTE)(polyflag);
00426                 }
00427                 polyflag = 0;
00428             }
00429             else if ( in[1] != ')' && in[1] != ',' ) {
00430                 *out++ = LPARENTHESIS;
00431             }
00432         }
00433         else if ( object <= 0 ) {
00434             CHECKPOLY
00435             AC.tokenarglevel[funlevel++] = TYPEISSUB;
00436             *out++ = LPARENTHESIS;
00437         }
00438         else {
00439             polyflag = 0;
00440             AC.tokenarglevel[funlevel++] = TYPEISMYSTERY;
00441             MesPrint("&Illegal position for (: %s",in);
00442             if ( error >= 0 ) error = -1;
00443         }
00444         object = -1;
00445     }
00446     else if ( *in == ')' ) {
00447         funlevel--;
00448         if ( funlevel < 0 ) {
00449             /* if ( funflag == 0 ) { */
00450                 MesPrint("&There is an unmatched parenthesis");
00451                 if ( error >= 0 ) error = -1;
00452             /* } */
00453         }
00454         else if ( object <= 0
00455             && ( AC.tokenarglevel[funlevel] != TYPEISFUN
00456             || out[-1] != TFUNOPEN ) ) {
00457             MesPrint("&Illegal position for closing parenthesis.");
00458             if ( error >= 0 ) error = -1;
00459             if ( AC.tokenarglevel[funlevel] == TYPEISFUN ) object = 1;
00460             else object = 3;
00461         }
00462         else {
00463             if ( AC.tokenarglevel[funlevel] == TYPEISFUN ) {
00464                 if ( out[-1] == TFUNOPEN ) out--;
00465                 else {
00466                     if ( out[-1] != TCOMMA ) *out++ = RPARENTHESIS;
00467                     *out++ = TFUNCLOSE;
00468                 }
00469                 object = 1;
00470             }
00471             else if ( AC.tokenarglevel[funlevel] == TYPEISSUB ) {
00472                 *out++ = RPARENTHESIS;
00473                 object = 3;
00474             }
00475         }
00476     }
00477     else if ( *in == ',' ) {
00478         if ( /* object > 0 && */ funlevel > 0 &&
00479             AC.tokenarglevel[funlevel-1] == TYPEISFUN ) {
00480             if ( out[-1] != TFUNOPEN && out[-1] != TCOMMA )
00481                 *out++ = RPARENTHESIS;
00482             else { *out++ = TNUMBER; *out++ = 0; }

```



```

00483         *out++ = TCOMMA;
00484         if ( in[1] != ',' && in[1] != ')' )
00485             *out++ = LPARENTHESIS;
00486         else if ( in[1] == ')' ) {
00487             *out++ = TNUMBER; *out++ = 0;
00488         }
00489     }
00490     /*
00491     else if ( object > 0 ) {
00492     }
00493     */
00494     else {
00495         MesPrint("&Illegal position for comma: %s",in);
00496         MesPrint("&Forgotten ; ?");
00497         if ( error >= 0 ) error = -1;
00498     }
00499     object = -1;
00500 }
00501 else goto IllPos;
00502 in++;
00503 break;
00504 case 5: /* + - * % / ^ : */
00505     CHECKPOLY
00506     if ( *in == ':' || *in == '%' ) goto IllPos;
00507     if ( *in == '*' || *in == '/' || *in == '^' ) {
00508         if ( object <= 0 ) {
00509             MesPrint("&Illegal position for operator: %s",in);
00510             if ( error >= 0 ) error = -1;
00511         }
00512         else if ( *in == '*' ) *out++ = TMULTIPLY;
00513         else if ( *in == '/' ) *out++ = TDIVIDE;
00514         else *out++ = TPOWER;
00515         in++;
00516     }
00517     else {
00518         i = 1;
00519         while ( *in == '+' || *in == '-' ) {
00520             if ( *in == '-' ) i = -i;
00521             in++;
00522         }
00523         if ( i == 1 ) {
00524             if ( out > AC.tokens && out[-1] != TFUNOPEN &&
00525                 out[-1] != LPARENTHESIS && out[-1] != TCOMMA
00526                 && out[-1] != LBRACE )
00527                 *out++ = TPLUS;
00528         }
00529         else *out++ = TMINUS;
00530     }
00531     object = 0;
00532     break;
00533 case 6: /* Whitespace */
00534     in++; break;
00535 case 7: /* { | } */
00536     CHECKPOLY
00537     if ( *in == '{' ) {
00538         if ( object > 0 ) {
00539             MesPrint("&Illegal position for %s",in);
00540             if ( !error ) error = 1;
00541         }
00542         s = in+1;
00543         SKIPBRA2(in)
00544         number = DoTempSet(s,in);
00545         in++;
00546         if ( number >= 0 ) {
00547             *out++ = TSET;
00548             i = 0;
00549             do { num[i++] = (SBYTE)(number & 0x7F); number >>= 7; } while ( number );
00550             while ( --i >= 0 ) *out++ = num[i];
00551         }
00552         else if ( error == 0 ) error = 1;
00553         object = 1;
00554     }
00555     else goto IllPos;
00556     break;
00557 case 8: /* ! & < > */
00558     CHECKPOLY
00559     if ( *in != '!' || leftright == RHSIDE
00560         || object != 1 || out[-1] != TWILDCARD ) goto IllPos;
00561     *out++ = TNOT;
00562     if ( FG.cTable[in[1]] == 0 || in[1] == '[' || in[1] == '{' ) object = 0;
00563     in++;
00564     break;
00565 default:
00566     IllPos: MesPrint("&Illegal character at this position: %s",in);
00567     if ( error >= 0 ) error = -1;
00568     in++;
00569     polyflag = 0;

```

```

00570             break;
00571         }
00572     }
00573     *out++ = TENDOFIT;
00574     AC.endoftokens = out;
00575     if ( funlevel > 0 || bracelevel != 0 ) {
00576         if ( funlevel > 0 ) MesPrint("&Unmatched parentheses");
00577         if ( bracelevel != 0 ) MesPrint("&Unmatched braces");
00578         return(-1);
00579     }
00580     if ( AC.TokensWriteFlag ) WriteTokens(AC.tokens);
00581 /*
00582     Simplify fixed set elements
00583 */
00584     if ( error == 0 && simpltoken(AC.tokens) ) error = 1;
00585 /*
00586     Collect wildcards for the prototype. Simplify the leftover wildcards
00587 */
00588     if ( error == 0 && leftright == LHSIDE && simpwtoken(AC.tokens) )
00589         error = 1;
00590 /*
00591     Now prepare the set[n] objects in the RHS.
00592 */
00593     if ( error == 0 && leftright == RHSIDE && simp4token(AC.tokens) )
00594         error = 1;
00595 /*
00596     Simplify simple function arguments (and 1/fac_ and 1/invfac_)
00597 */
00598     if ( error == 0 && simp2token(AC.tokens) ) error = 1;
00599 /*
00600     Next we try to remove composite denominators or exponents and
00601     replace them by their internal functions. This may involve expanding
00602     the buffer. The return code of 3a is negative if there is an error
00603     and positive if indeed we need to do some work.
00604     simp3btoken does the work
00605 */
00606     numexp = 0;
00607     if ( error == 0 && ( numexp = simp3atoken(AC.tokens,leftright) ) < 0 )
00608         error = 1;
00609     if ( numexp > 0 ) {
00610         // Make some space at the beginning of AC.tokens, filled with EMPTY.
00611         SBYTE *tt;
00612         out = AC.tokens;
00613         while ( *out != TENDOFIT ) out++;
00614         while ( out+numexp*9 > outtop ) {
00615             LONG oldsize = (LONG)(out - AC.tokens);
00616             SBYTE **ppp = &(AC.tokens); /* to avoid a compiler warning */
00617             SBYTE **pppp = &(AC.toptokens);
00618             DoubleBuffer((void **)ppp, (void **)pppp, sizeof(SBYTE), "out tokens");
00619             out = AC.tokens + oldsize;
00620             outtop = AC.toptokens - MAXNUMSIZE;
00621         }
00622         tt = out + numexp*9;
00623         while ( out >= AC.tokens ) { *tt-- = *out--; }
00624         while ( tt >= AC.tokens ) { *tt-- = EMPTY; }
00625         if ( error == 0 && simp3btoken(AC.tokens,leftright) ) error = 1;
00626         if ( error == 0 && simp2token(AC.tokens) ) error = 1;
00627     }
00628 /*
00629     In simp5token we test for special cases like sumvariables that are
00630     already wildcards, etc.
00631 */
00632     if ( error == 0 && simp5token(AC.tokens,leftright) ) error = 1;
00633 /*
00634     In simp6token we test for special cases like factorized expressions
00635     that occur in the RHS in an improper way.
00636 */
00637     if ( error == 0 && simp6token(AC.tokens,leftright) ) error = 1;
00638     return(error);
00639 }
00640 }
00641 /*
00642     #] tokenize :
00643     #[ WriteTokens :
00644 */
00645 /*
00646 char *ttypes[] = { "\n", "S", "I", "V", "F", "set", "E", "dotp", "#",
00647     "sub", "d_", "$", "dub", "(", ")", "?", "??", ".", "[", "]",
00648     ",", "((, " ))", "*", "/", "^", "+", "-", "!", "end", "{", "}",
00649     "N_?", "conj", "()", "#d", "^d", "_", "snum" };
00650
00651 void WriteTokens(SBYTE *in)
00652 {
00653     int numinline = 0, x, n = sizeof(ttypes)/sizeof(char *);
00654     char outbuf[81], *s, *out, c;
00655     out = outbuf;

```

```

00657     while ( *in != TENDOFIT ) {
00658         if ( *in < 0 ) {
00659             if ( *in >= -n ) {
00660                 s = ttypes[-*in];
00661                 while ( *s ) { *out++ = *s++; numinline++; }
00662             }
00663             else {
00664                 *out++ = '-'; x = -*in; numinline++;
00665                 goto writenumber;
00666             }
00667         }
00668         else {
00669             x = *in;
00670 writenumber:
00671             s = out;
00672             do {
00673                 *out++ = (char)(( x % 10 ) + '0');
00674                 numinline++;
00675                 x = x / 10;
00676             } while ( x );
00677             c = out[-1]; out[-1] = *s; *s = c;
00678         }
00679         if ( numinline > 70 ) {
00680             *out = 0;
00681             MesPrint("%s",outbuf);
00682             out = outbuf; numinline = 0;
00683         }
00684         else {
00685             *out++ = ' '; numinline++;
00686         }
00687         in++;
00688     }
00689     if ( numinline > 0 ) { *out = 0; MesPrint("%s",outbuf); }
00690 }
00691
00692 /*
00693     #] WriteTokens :
00694     #[ simpltoken :
00695
00696     Routine substitutes set elements if possible.
00697     This means sets with a fixed argument like setname[3].
00698 */
00699
00700 int simpltoken(SBYTE *s)
00701 {
00702     int error = 0, n, i, base;
00703     WORD numsub;
00704     SBYTE *fill = s, *start, *t, numtab[10];
00705     SETS set;
00706     while ( *s != TENDOFIT ) {
00707         if ( *s == RBRACE ) {
00708             start = fill-1;
00709             while ( *start != LBRACE ) start--;
00710             t = start - 1;
00711             while ( *t >= 0 ) t--;
00712             if ( *t == TSET && ( start[1] == TNUMBER || start[1] == TNUMBER1 ) ) {
00713                 base = start[1] == TNUMBER ? 100: 128;
00714                 start += 2;
00715                 numsub = *start++;
00716                 while ( *start >= 0 && start < fill )
00717                     { numsub = base*numsub + *start++; }
00718                 if ( start == fill ) {
00719                     start = t;
00720                     t++; n = *t++; while ( *t >= 0 ) { n = 128*n + *t++; }
00721                     set = Sets+n;
00722                     if ( ( set->type != CRANGE )
00723                         && ( numsub > 0 && numsub <= set->last-set->first ) ) {
00724                         fill = start;
00725                         n = SetElements[set->first+numsub-1];
00726                         switch (set->type) {
00727                             case CSYMBOL:
00728                                 if ( n > MAXPOWER ) {
00729                                     n -= 2*MAXPOWER;
00730                                     if ( n < 0 ) { n = -n; *fill++ = TMINUS; }
00731                                     *fill++ = TNUMBER1;
00732                                 }
00733                                 else *fill++ = TSYMBOL;
00734                                 break;
00735                             case CINDEX:
00736                                 if ( n < AM.OffsetIndex ) *fill++ = TNUMBER1;
00737                                 else {
00738                                     *fill++ = TINDEX;
00739                                     n -= AM.OffsetIndex;
00740                                 }
00741                                 break;
00742                             case CVECTOR: *fill++ = TVECTOR;
00743                                     n -= AM.OffsetVector; break;

```

```

00744             case CFUNCTION: *fill++ = TFUNCTION;
00745                 n -= FUNCTION; break;
00746             case CNUMBER: *fill++ = TNUMBER1; break;
00747             case CDUBIOUS: *fill++ = TDUBIOUS; n = 1; break;
00748         }
00749         i = 0;
00750     if ( n < 0 ) {
00751         MesPrint("Value of n = %d",n);
00752     }
00753         do { numtab[i++] = (SBYTE)(n & 0x7F); n >>= 7; } while ( n );
00754         while ( --i >= 0 ) *fill++ = numtab[i];
00755     }
00756     else {
00757         MesPrint("&Illegal element %d in set",numsub);
00758         error++;
00759     }
00760     s++; continue;
00761 }
00762 }
00763 *fill++ = *s++;
00764 }
00765 else *fill++ = *s++;
00766 }
00767 *fill++ = TENDOFIT;
00768 return(error);
00769 }
00770
00771 /*
00772     #] simpptoken :
00773     #[ simpwtoken :
00774
00775     Only to be called in the LHS.
00776     Hunts down the wildcards and writes them to the wildcardbuffer.
00777     Next it causes the ProtoType to be constructed.
00778     All wildcards are simplified into the trailing TWILDCARD,
00779     because the specifics are stored in the prototype.
00780     These specifics also include the transfer of wildcard values
00781     to $variables.
00782
00783     Types of wildcards:
00784     a?, a?set, a?!set, a?set[i], A?set1?set2, ?a
00785     After this we can strip the set information.
00786     We still need the ? because of the wildcarding offset in code generation
00787 */
00788
00789 int simpwtoken(SBYTE *s)
00790 {
00791     int error = 0, first = 1, notflag;
00792     WORD num, numto, numdollar, *w = AC.WildC, *wstart, *wtop;
00793     SBYTE *fill = s, *t, *v, *s0 = s;
00794     while ( *s != TENDOFIT ) {
00795         if ( *s == TWILDCARD ) {
00796             notflag = 0; t = fill;
00797             while ( t > s0 && t[-1] >= 0 ) t--;
00798             v = t; num = 0; *fill++ = *s++;
00799             while ( *v >= 0 ) num = 128*num + *v++;
00800             if ( t > s0 ) t--;
00801             AC.NwildC += 4;
00802             if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
00803             switch ( *t ) {
00804                 case TSYMBOL:
00805                 case TDUBIOUS:
00806                     *w++ = SYMTOSYM; *w++ = 4; *w++ = num; *w++ = num; break;
00807                 case TINDEX:
00808                     num += AM.OffsetIndex;
00809                     *w++ = INDTOIND; *w++ = 4; *w++ = num; *w++ = num; break;
00810                 case TVECTOR:
00811                     num += AM.OffsetVector;
00812                     *w++ = VECTOVEC; *w++ = 4; *w++ = num; *w++ = num; break;
00813                 case TFUNCTION:
00814                     num += FUNCTION;
00815                     *w++ = FUNTOFUN; *w++ = 4; *w++ = num; *w++ = num; break;
00816                 default:
00817                     MesPrint("&Illegal type of wildcard in LHS");
00818                     error = -1;
00819                     *w++ = SYMTOSYM; *w++ = 4; *w++ = num; *w++ = num; break;
00820             }
00821         }
00822     }
00823     /*
00824     Now the sets. The s pointer sits after the ?
00825     */
00826     wstart = w;
00827     if ( *s == TNOT && s[1] == TSET ) { notflag = 1; s++; }
00828     if ( *s == TSET ) {
00829         s++; num = 0; while ( *s >= 0 ) num = 128*num + *s++;
00830         if ( notflag == 0 && *s == TWILDCARD && s[1] == TSET ) {
00831             s += 2; numto = 0; while ( *s >= 0 ) numto = 128*numto + *s++;

```

```

00831         if ( num < AM.NumFixedSets || numto < AM.NumFixedSets
00832         || Sets[num].type == CRANGE || Sets[numto].type == CRANGE ) {
00833             MesPrint("&This type of set not allowed in this wildcard construction");
00834             error = 1;
00835         }
00836         else {
00837             AC.NwildC += 4;
00838             if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
00839             *w++ = FROMSET; *w++ = 4; *w++ = num; *w++ = numto;
00840             wstart = w;
00841         }
00842     }
00843     else if ( notflag == 0 && *s == LBRACE && s[1] == TSYMBOL ) {
00844         if ( num < AM.NumFixedSets || Sets[num].type == CRANGE ) {
00845             MesPrint("&This type of set not allowed in this wildcard construction");
00846             error = 1;
00847         }
00848         v = s; s += 2;
00849         numto = 0; while ( *s >= 0 ) numto = 128*numto + *s++;
00850         if ( *s == TWILDCARD ) s++; /* most common mistake */
00851         if ( *s == RBRACE ) {
00852             s++;
00853             AC.NwildC += 8;
00854             if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
00855             *w++ = SETTONUM; *w++ = 4; *w++ = num; *w++ = numto;
00856             wstart = w;
00857             *w++ = SYMTOSYM; *w++ = 4; *w++ = numto; *w++ = 0;
00858         }
00859         else if ( *s == TDOLLAR ) {
00860             s++; numdollar = 0;
00861             while ( *s >= 0 ) numdollar = 128*numdollar + *s++;
00862             if ( *s == RBRACE ) {
00863                 s++;
00864                 AC.NwildC += 12;
00865                 if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
00866                 *w++ = SETTONUM; *w++ = 4; *w++ = num; *w++ = numto;
00867                 wstart = w;
00868                 *w++ = SYMTOSYM; *w++ = 4; *w++ = numto; *w++ = 0;
00869                 *w++ = LOADDOLLAR; *w++ = 4; *w++ = numdollar;
00870                 *w++ = numdollar;
00871             }
00872             else { s = v; goto singlewild; }
00873         }
00874         else { s = v; goto singlewild; }
00875     }
00876     else {
00877 singlewild:
00878         num += notflag * 2*WILDOFFSET;
00879         AC.NwildC += 4;
00880         if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
00881         *w++ = FROMSET; *w++ = 4; *w++ = num; *w++ = -WILDOFFSET;
00882         wstart = w;
00883     }
00884     else if ( *s != TDOLLAR && *s != TENDOFIT && *s != RPARENTHESIS
00885     && *s != RBRACE && *s != TCOMMA && *s != TFUNCLOSE && *s != TMULTIPLY
00886     && *s != TPOWER && *s != TDIVIDE && *s != TPLUS && *s != TMINUS
00887     && *s != TPOWER1 && *s != TEMPTY && *s != TFUNOPEN && *s != TDOT ) {
00888         MesPrint("&Illegal type of wildcard in LHS");
00889         error = -1;
00890     }
00891     if ( *s == TDOLLAR ) {
00892         s++; numdollar = 0;
00893         while ( *s >= 0 ) numdollar = 128*numdollar + *s++;
00894         AC.NwildC += 4;
00895         if ( AC.NwildC > 4*AM.MaxWildcards ) goto firsterr;
00896         wtop = w + 4;
00897         if ( wstart < w ) {
00898             while ( w > wstart ) { w[4] = w[0]; w--; }
00899         }
00900         *w++ = LOADDOLLAR; *w++ = 4; *w++ = numdollar; *w++ = numdollar;
00901         w = wtop;
00902     }
00903 }
00904 else if ( *s == TWILDARG ) {
00905     *fill++ = *s++;
00906     num = 0;
00907     while ( *s >= 0 ) { num = 128*num + *s; *fill++ = *s++; }
00908     AC.NwildC += 4;
00909     if ( AC.NwildC > 4*AM.MaxWildcards ) {
00910 firsterr:
00911         if ( first ) {
00912             MesPrint("&More than %d wildcards",AM.MaxWildcards);
00913             error = -1;
00914             first = 0;
00915         }
00916     }
00917     else { *w++ = ARGTOARG; *w++ = 4; *w++ = num; *w++ = -1; }
00918     if ( *s == TDOLLAR ) {

```

```

00918         s++; num = 0; while ( *s >= 0 ) num = 128*num + *s++;
00919         AC.NwildC += 4;
00920         if ( AC.NwildC > 4*AM.MaxWildcards ) goto firstterr;
00921         *w++ = LOADDOLLAR; *w++ = 4; *w++ = num; *w++ = num;
00922     }
00923 }
00924     else *fill++ = *s++;
00925 }
00926 *fill++ = TENDOFIT;
00927 AC.WildC = w;
00928 return(error);
00929 }
00930
00931 /*
00932     #] simpwtoken :
00933     #[ simp2token :
00934
00935     Deals with function arguments.
00936     The tokenizer has given function arguments extra parentheses.
00937     We remove the double parentheses.
00938     Next we remove the parentheses around the simple arguments.
00939
00940     It also replaces /fac_() by *invfac_() and /invfac_() by *fac_()
00941 */
00942
00943 int simp2token(SBYTE *s)
00944 {
00945     SBYTE *to, *fill, *t, *v, *w, *s0 = s, *vv;
00946     int error = 0, n;
00947     /*
00948     Set substitutions
00949     */
00950     fill = to = s;
00951     while ( *s != TENDOFIT ) {
00952         if ( *s == LPARENTHESIS && s[1] == LPARENTHESIS ) {
00953             t = s+1; n = 0;
00954             while ( n >= 0 ) {
00955                 t++;
00956                 if ( *t == LPARENTHESIS ) n++;
00957                 else if ( *t == RPARENTHESIS ) n--;
00958             }
00959             if ( t[1] == RPARENTHESIS ) {
00960                 *t = EMPTY; s++;
00961             }
00962             *fill++ = *s++;
00963         }
00964         else if ( *s == EMPTY ) s++;
00965         else if ( *s == AM.facnum && ( fill > (s0+1) ) && fill[-2] == TDIVIDE
00966             && fill[-1] == TFUNCTION ) {
00967             fill[-2] = TMULTIPLY; *fill++ = (SBYTE)(AM.invfacnum); s++;
00968         }
00969         else if ( *s == AM.invfacnum && ( fill > (s0+1) ) && fill[-2] == TDIVIDE
00970             && fill[-1] == TFUNCTION ) {
00971             fill[-2] = TMULTIPLY; *fill++ = (SBYTE)(AM.facnum); s++;
00972         }
00973         else *fill++ = *s++;
00974     }
00975     *fill++ = TENDOFIT;
00976     /*
00977     Second round: try to locate 'simple' arguments and strip their brackets
00978
00979     We add (9-feb-2010) to the simple arguments integers of any size
00980     */
00981     fill = s = to;
00982     while ( *s != TENDOFIT ) {
00983         if ( *s == LPARENTHESIS ) {
00984             t = s; n = 0;
00985             while ( n >= 0 ) {
00986                 t++;
00987                 if ( *t == LPARENTHESIS ) n++;
00988                 else if ( *t == RPARENTHESIS ) n--;
00989             }
00990             if ( t[1] == TFUNCLOSE && s[1] != TWILDARG ) { /* Check for last argument in sum */
00991                 v = fill - 1; n = 0;
00992                 while ( n >= 0 && v >= to ) {
00993                     if ( *v == TFUNOPEN ) n--;
00994                     else if ( *v == TFUNCLOSE ) n++;
00995                     v--;
00996                 }
00997                 if ( v > to ) {
00998                     while ( *v >= 0 ) v--;
00999                     if ( *v == TFUNCTION ) { v++;
01000                         n = 0; while ( *v >= 0 && v < fill ) n = 128*n + *v++;
01001                         if ( n == AM.sumnum || n == AM.sumpnum ) {
01002                             *fill++ = *s++; continue;
01003                         }
01004                     }
01005                     else if ( ( n == (FIRSTBRACKET-FUNCTION)

```

```

01005         || n == (TERMSINEXPR-FUNCTION)
01006         || n == (SIZEOFFUNCTION-FUNCTION)
01007         || n == (NUMFACTORS-FUNCTION)
01008         || n == (GCDFUNCTION-FUNCTION)
01009         || n == (DIVFUNCTION-FUNCTION)
01010         || n == (REMFUNCTION-FUNCTION)
01011         || n == (INVERSEFUNCTION-FUNCTION)
01012         || n == (MULFUNCTION-FUNCTION)
01013         || n == (FACTORIN-FUNCTION)
01014         || n == (FIRSTTERM-FUNCTION)
01015         || n == (CONTENTTERM-FUNCTION) )
01016     && fill[-1] == TFUNOPEN ) {
01017         v = s+1;
01018         if ( *v == TEXPRESSON ) {
01019             v++;
01020             n = 0; while ( *v >= 0 ) n = 128*n + *v++;
01021             if ( v == t ) {
01022                 *t = TEMPT; s++;
01023             }
01024         }
01025     }
01026 }
01027 }
01028 }
01029 if ( ( fill > to )
01030 && ( ( fill[-1] == TFUNOPEN || fill[-1] == TCOMMA )
01031 && ( t[1] == TFUNCLOSE || t[1] == TCOMMA ) ) ) {
01032     v = s + 1;
01033     switch ( *v ) {
01034     case TMINUS:
01035         v++;
01036         if ( *v == TVECTOR ) {
01037             w = v+1; while ( *w >= 0 ) w++;
01038             if ( w == t ) {
01039                 *t = TEMPT; s++;
01040             }
01041         }
01042     else {
01043         if ( *v == TNUMBER || *v == TNUMBER1 ) {
01044             if ( BITSINWORD == 16 ) { ULONG x; WORD base;
01045                 base = ( *v == TNUMBER ) ? 100: 128;
01046                 vv = v+1; x = 0; while ( *vv >= 0 ) { x = x*base + *vv++; }
01047                 if ( ( vv != t ) || ( ( vv - v ) > 4 ) || ( x > (MAXPOSITIVE+1) ) )
01048                     *fill++ = *s++;
01049                 else { *t = TEMPT; s++; break; }
01050             }
01051             else if ( BITSINWORD == 32 ) { ULONG x; WORD base;
01052                 base = ( *v == TNUMBER ) ? 100: 128;
01053                 vv = v+1; x = 0; while ( *vv >= 0 ) { x = x*base + *vv++; }
01054                 if ( ( vv != t ) || ( ( vv - v ) > 6 ) || ( x > (MAXPOSITIVE+1) ) )
01055                     *fill++ = *s++;
01056                 else { *t = TEMPT; s++; break; }
01057             }
01058             else {
01059                 if ( ( v+2 == t ) || ( v+3 == t && v[2] >= 0 ) )
01060                     { *t = TEMPT; s++; break; }
01061                 else *fill++ = *s++;
01062             }
01063         }
01064     else if ( *v == LPARENTHESIS && t[-1] == RPARENTHESIS ) {
01065         w = v; n = 0;
01066         while ( n >= 0 ) {
01067             w++;
01068             if ( *w == LPARENTHESIS ) n++;
01069             else if ( *w == RPARENTHESIS ) n--;
01070         }
01071         if ( w == ( t-1 ) ) { *t = TEMPT; s++; }
01072         else *fill++ = *s++;
01073     }
01074     else *fill++ = *s++;
01075     break;
01076 }
01077 /* fall through */
01078 case TSETNUM:
01079     v++; while ( *v >= 0 ) v++;
01080     goto tcommon;
01081 case TSYMBOL:
01082     if ( ( v[1] == COEFFSYMBOL || v[1] == NUMERATORSYMBOL
01083         || v[1] == DENOMINATORSYMBOL ) && v[2] < 0 ) {
01084         *fill++ = *s++; break;
01085     }
01086     /* fall through */
01087 case TSET:
01088 case TVECTOR:
01089 case TINDEX:
01090 case TFUNCTION:
01091 case TDOLLAR:

```

```

01092         case TDUBIOUS:
01093         case TSGAMMA:
01094             tcommon:
01095                 v++; while ( *v >= 0 ) v++;
01096                 if ( v == t || ( v[0] == TWILDCARD && v+1 == t ) )
01097                     { *t = TEMPTY; s++; }
01098                 else *fill++ = *s++;
01099                 break;
01100         case TGENINDEX:
01101             v++;
01102             if ( v == t ) { *t = TEMPTY; s++; }
01103             else *fill++ = *s++;
01104             break;
01105         case TNUMBER:
01106         case TNUMBER1:
01107             if ( BITSINWORD == 16 ) { ULONG x; WORD base;
01108                 base = ( *v == TNUMBER ) ? 100: 128;
01109                 vv = v+1; x = 0; while ( *vv >= 0 ) { x = x*base + *vv++; }
01110                 if ( ( vv != t ) || ( ( vv - v ) > 4 ) || ( x > MAXPOSITIVE ) )
01111                     *fill++ = *s++;
01112                 else { *t = TEMPTY; s++; break; }
01113             }
01114             else if ( BITSINWORD == 32 ) { ULONG x; WORD base;
01115                 base = ( *v == TNUMBER ) ? 100: 128;
01116                 vv = v+1; x = 0; while ( *vv >= 0 ) { x = x*base + *vv++; }
01117                 if ( ( vv != t ) || ( ( vv - v ) > 6 ) || ( x > MAXPOSITIVE ) )
01118                     *fill++ = *s++;
01119                 else { *t = TEMPTY; s++; break; }
01120             }
01121             else {
01122                 if ( ( v+2 == t ) || ( v+3 == t && v[2] >= 0 ) )
01123                     { *t = TEMPTY; s++; break; }
01124                 else *fill++ = *s++;
01125             }
01126             break;
01127         case TWILDARG:
01128             v++; while ( *v >= 0 ) v++;
01129             if ( v == t ) { *t = TEMPTY; s++; }
01130             else *fill++ = *s++;
01131             break;
01132         case TEXPRESSON:
01133             /*
01134             First establish that there is only the expression
01135             in this argument.
01136             */
01137             vv = s+1;
01138             while ( vv < t ) {
01139                 if ( *vv != TEXPRESSON ) break;
01140                 vv++; while ( *vv >= 0 ) vv++;
01141             }
01142             if ( vv < t ) { *fill++ = *s++; break; }
01143             /*
01144             Find the function
01145             */
01146             w = fill-1; n = 0;
01147             while ( n >= 0 && w >= to ) {
01148                 if ( *w == TFUNOPEN ) n--;
01149                 else if ( *w == TFUNCLOSE ) n++;
01150                 w--;
01151             }
01152             w--; while ( w > to && *w >= 0 ) w--;
01153             if ( *w != TFUNCTION ) { *fill++ = *s++; break; }
01154             w++; n = 0;
01155             while ( *w >= 0 ) { n = 128*n + *w++; }
01156             if ( n == GCDFUNCTION-FUNCTION
01157                 || n == DIVFUNCTION-FUNCTION
01158                 || n == REMFUNCTION-FUNCTION
01159                 || n == INVERSEFUNCTION-FUNCTION
01160                 || n == MULFUNCTION-FUNCTION ) {
01161                 *t = TEMPTY; s++;
01162             }
01163             else *fill++ = *s++;
01164             break;
01165         default: *fill++ = *s++; break;
01166     }
01167     }
01168     else *fill++ = *s++;
01169 }
01170 else if ( *s == TEMPTY ) s++;
01171 else *fill++ = *s++;
01172 }
01173 *fill++ = TENDOFIT;
01174 return(error);
01175 }
01176 /*
01177 #] simp2token :
01178 #[ simp3atoken :

```



```

01179
01180     We hunt for denominators and exponents that seem hidden.
01181     For the denominators we have to recognize:
01182         /fun /fun() /fun^power /fun()^power
01183         /set[n] /set[n]() /set[n]^power /set[n]()^power
01184         /symbol^power (power no number or symbol wildcard)
01185         /dotpr^power (id)
01186         /#^power (id)
01187         /() /()^power
01188         /vect /index /vect(anything) /vect(anything)^power
01189 */
01190
01191 int simp3atoken(SBYTE *s, int mode)
01192 {
01193     int error = 0, n, numexp = 0, denom, base, numprot, i;
01194     SBYTE *t, c;
01195     LONG num;
01196     WORD *prot;
01197     if ( mode == RHSIDE ) {
01198         prot = AC.ProtoType;
01199         numprot = prot[1] - SUBEXPSIZE;
01200         prot += SUBEXPSIZE;
01201     }
01202     else { prot = 0; numprot = 0; }
01203     while ( *s != TENDOFIT ) {
01204         denom = 1;
01205         if ( *s == TDIVIDE ) { denom = -1; s++; }
01206         c = *s;
01207         switch(c) {
01208             case TSYMBOL:
01209             case TNUMBER:
01210             case TNUMBER1:
01211                 s++; while ( *s >= 0 ) s++; /* skip the object */
01212                 if ( *s == TWILDCARD ) s++; /* and the possible wildcard */
01213             dosymbol:
01214                 if ( *s != TPOWER ) continue; /* No power -> done */
01215                 s++; /* Skip the power */
01216                 if ( *s == TMINUS ) s++; /* negative: no difference here */
01217                 if ( *s == TNUMBER || *s == TNUMBER1 ) {
01218                     base = *s == TNUMBER ? 100: 128; /* NUMBER = base 100 */
01219                     s++; /* Now we compose the power */
01220                     num = s++; /* If the number is way too large */
01221                     while ( *s >= 0 ) { /* it may look like not too big */
01222                         if ( num > MAXPOWER ) break; /* Hence... */
01223                         num = base*num + *s++;
01224                     }
01225                     while ( *s >= 0 ) s++; /* Finish the number if needed */
01226                     if ( *s == TPOWER ) goto doublepower;
01227                     if ( num <= MAXPOWER ) continue; /* Simple case */
01228                 }
01229                 else if ( *s == TSYMBOL && c != TNUMBER && c != TNUMBER1 ) {
01230                     s++; n = 0; while ( *s >= 0 ) { n = 128*n + *s++; }
01231                     if ( *s == TWILDCARD ) { s++;
01232                         if ( *s == TPOWER ) goto doublepower;
01233                         continue; }
01234                 /*
01235                 Now we have to test whether n happens to be a wildcard
01236                 */
01237                 if ( mode == RHSIDE ) {
01238                     n += 2*MAXPOWER;
01239                     for ( i = 0; i < numprot; i += 4 ) {
01240                         if ( prot[i+2] == n && prot[i] == SYMTOSYM ) break;
01241                     }
01242                     if ( i < numprot ) break;
01243                 }
01244                 if ( *s == TPOWER ) goto doublepower;
01245             }
01246             numexp++;
01247             break;
01248         case TINDEX:
01249             s++; while ( *s >= 0 ) s++;
01250             if ( *s == TWILDCARD ) s++;
01251             doindex:
01252                 if ( denom < 0 || *s == TPOWER ) {
01253                     MesPrint("&Index to a power or in denominator is illegal");
01254                     error = 1;
01255                 }
01256                 break;
01257         case TVECTOR:
01258             s++; while ( *s >= 0 ) s++;
01259             if ( *s == TWILDCARD ) s++;
01260             dovector:
01261                 if ( *s == TFUNOPEN ) {
01262                     s++; n = 1;
01263                     for (;;) {
01264                         if ( *s == TFUNOPEN ) {
01265                             n++;

```

```

01266             MesPrint("&Illegal vector index");
01267             error = 1;
01268         }
01269         else if ( *s == TFUNCLOSE ) {
01270             n--;
01271             if ( n <= 0 ) break;
01272         }
01273         s++;
01274     }
01275     s++;
01276 }
01277 else if ( *s == TDOT ) goto dodot;
01278 if ( denom < 0 || *s == TPOWER || *s == TPOWER1 ) numexp++;
01279 break;
01280 case TFUNCTION:
01281     s++; while ( *s >= 0 ) s++;
01282     if ( *s == TWILDCARD ) s++;
01283 dofunction:
01284     t = s;
01285     if ( *t == TFUNOPEN ) {
01286         t++; n = 1;
01287         for(;;) {
01288             if ( *t == TFUNOPEN ) n++;
01289             else if ( *t == TFUNCLOSE ) { if ( --n <= 0 ) break; }
01290             t++;
01291         }
01292         t++; s++;
01293     }
01294     if ( denom < 0 || *t == TPOWER || *t == TPOWER1 ) numexp++;
01295     break;
01296 #ifdef WITHFLOAT
01297 case TFLOAT:
01298     s++; while ( *s >= 0 ) s++;
01299     if ( denom < 0 || *s == TPOWER || *s == TPOWER1 ) numexp++;
01300     break;
01301 #endif
01302 case TEXPRESSION:
01303     s++; while ( *s >= 0 ) s++;
01304     t = s;
01305     if ( *t == TFUNOPEN ) {
01306         t++; n = 1;
01307         for(;;) {
01308             if ( *t == TFUNOPEN ) n++;
01309             else if ( *t == TFUNCLOSE ) { if ( --n <= 0 ) break; }
01310             t++;
01311         }
01312         t++;
01313     }
01314     if ( *t == LBRACE ) {
01315         t++; n = 1;
01316         for(;;) {
01317             if ( *t == LBRACE ) n++;
01318             else if ( *t == RBRACE ) { if ( --n <= 0 ) break; }
01319             t++;
01320         }
01321         t++;
01322     }
01323     if ( denom < 0 || ( ( *t == TPOWER || *t == TPOWER1 )
01324         && t[1] == TMINUS ) ) numexp++;
01325     break;
01326 case TDOLLAR:
01327     s++; while ( *s >= 0 ) s++;
01328     if ( denom < 0 || ( ( *s == TPOWER || *s == TPOWER1 )
01329         && s[1] == TMINUS ) ) numexp++;
01330     break;
01331 case LPARENTHESIS:
01332     s++; n = 1; t = s;
01333     for(;;) {
01334         if ( *t == LPARENTHESIS ) n++;
01335         else if ( *t == RPARENTHESIS ) { if ( --n <= 0 ) break; }
01336         t++;
01337     }
01338     t++;
01339     if ( denom > 0 && ( *t == TPOWER || *t == TPOWER1 ) ) {
01340         if ( ( t[1] == TNUMBER || t[1] == TNUMBER1 ) && t[2] >= 0
01341             && t[3] < 0 ) break;
01342         numexp++;
01343     }
01344     else if ( denom < 0 && ( *t == TPOWER || *t == TPOWER1 ) ) {
01345         if ( t[1] == TMINUS && ( t[2] == TNUMBER
01346             || t[2] == TNUMBER1 ) && t[3] >= 0
01347             && t[4] < 0 ) break;
01348         numexp++;
01349     }
01350     else if ( denom < 0 || ( ( *t == TPOWER || *t == TPOWER1 )
01351         && ( t[1] == TMINUS || t[1] == LPARENTHESIS ) ) ) numexp++;
01352     break;

```

```

01353         case TSET:
01354             s++; n = *s++; while ( *s >= 0 ) { n = 128*n + *s++; }
01355             n = Sets[n].type;
01356             switch ( n ) {
01357                 case CSYMBOL: goto dosymbol;
01358                 case CINDEX: goto doindex;
01359                 case CVECTOR: goto dovector;
01360                 case CFUNCTION: goto dofunction;
01361                 case CNUMBER: goto dosymbol;
01362                 case CMODEL:
01363                     if ( denom < 0 || *s == TPOWER ) {
01364                         MesPrint("%A model to a power or in denominator is illegal");
01365                         error = 1;
01366                     }
01367                     break;
01368                 case ANYTYPE:
01369                     if ( denom < 0 || *s == TPOWER ) {
01370                         MesPrint("%A set without type to a power or in denominator is illegal");
01371                         error = 1;
01372                     }
01373                     break;
01374                 default: error = 1; break;
01375             }
01376             break;
01377         case TDOT:
01378             dodot:
01379             s++;
01380             if ( *s == TVECTOR ) { s++; while ( *s >= 0 ) s++; }
01381             else if ( *s == TSET ) {
01382                 s++; n = *s++; while ( *s >= 0 ) { n = 128*n + *s++; }
01383                 if ( Sets[n].type != CVECTOR ) {
01384                     MesPrint("&Set in dotproduct is not a set of vectors");
01385                     error = 1;
01386                 }
01387                 if ( *s == LBRACE ) {
01388                     s++; n = 1;
01389                     for(;;) {
01390                         if ( *s == LBRACE ) n++;
01391                         else if ( *s == RBRACE ) { if ( --n <= 0 ) break; }
01392                         s++;
01393                     }
01394                 }
01395                 else {
01396                     MesPrint("&Set without argument in dotproduct");
01397                     error = 1;
01398                 }
01399             }
01400             else if ( *s == TSETNUM ) {
01401                 s++; n = *s++; while ( *s >= 0 ) { n = 128*n + *s++; }
01402                 if ( *s != TVECTOR ) goto nodot;
01403                 s++; n = *s++; while ( *s >= 0 ) { n = 128*n + *s++; }
01404                 if ( Sets[n].type != CVECTOR ) {
01405                     MesPrint("&Set in dotproduct is not a set of vectors");
01406                     error = 1;
01407                 }
01408             }
01409             else {
01410                 nodot:
01411                 MesPrint("&Illegal second element in dotproduct");
01412                 error = 1;
01413                 s++; while ( *s >= 0 ) s++;
01414             }
01415             goto dosymbol;
01416         default:
01417             s++; while ( *s >= 0 ) s++;
01418             break;
01419     }
01420     if ( error ) return(-1);
01421     return(numexp);
01422 doublepower:
01423     MesPrint("&Dubious notation with object^power1^power2");
01424     return(-1);
01425 }
01426
01427 /*
01428     #] simp3atoken :
01429     #[ simp3btoken :
01430 */
01431
01432 int simp3btoken(SBYTE *s, int mode)
01433 {
01434     int error = 0, i, numprot, n, denom, base, inset = 0, dotp, sube = 0;
01435     SBYTE *t, c, *fill, *ff, *ss;
01436     LONG num;
01437     WORD *prot;
01438
01439     // Work in a temporary buffer, to avoid clashes between input and output

```

```

01440     SBYTE *tmptokens = (SBYTE*)Malloc1((AC.toptokens-AC.tokens)*sizeof(SBYTE), "simp3btoken scratch");
01441     SBYTE* tmptoptokens = tmptokens + (AC.toptokens-AC.tokens);
01442     fill = tmptokens;
01443
01444     if ( mode == RHSIDE ) {
01445         prot = AC.ProtoType;
01446         numprot = prot[1] - SUBEXPSIZE;
01447         prot += SUBEXPSIZE;
01448     }
01449     else { prot = 0; numprot = 0; }
01450     while ( *s == EMPTY ) s++;
01451     while ( *s != TENDOFIT ) {
01452         // If we are near the end of the output buffer, we'd better reallocate
01453         // both tmptokens and AC.tokens (which must fit the final result).
01454         // Experimentally, fill can move by at most 5 per loop iteration, but
01455         // this buffer of 20 of might not be enough for all cases!
01456         while ( tmptoptokens - fill < 20 ) {
01457             LONG tmppos;
01458             tmppos = s - AC.tokens;
01459             DoubleBuffer((void*)&(AC.tokens), (void*)&(AC.toptokens), sizeof(SBYTE), "simp3btoken
double");
01460             s = AC.tokens + tmppos;
01461             tmppos = fill - tmptokens;
01462             DoubleBuffer((void*)&tmptokens, (void*)&tmptoptokens, sizeof(SBYTE), "simp3btoken
scratch double");
01463             fill = tmptokens + tmppos;
01464         }
01465         if ( *s == EMPTY ) { s++; continue; }
01466         denom = 1;
01467         if ( *s == TDIVIDE ) { denom = -1; *fill++ = *s++; }
01468         ff = fill; ss = s; c = *s;
01469         if ( c == TSETNUM ) {
01470             *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01471             c = *s;
01472         }
01473         dotp = 0;
01474         switch(c) {
01475             case TSYMBOL:
01476             case TNUMBER:
01477             case TNUMBER1:
01478                 *fill++ = *s++;
01479                 while ( *s >= 0 ) *fill++ = *s++;
01480                 if ( *s == TWILDCARD ) *fill++ = *s++;
01481             dosymbol:
01482                 t = s;
01483                 if ( *s != TPOWER ) continue;
01484                 *fill++ = *s++;
01485                 if ( *s == TMINUS ) *fill++ = *s++;
01486                 if ( *s == TPLUS ) s++;
01487                 if ( *s == TSETNUM ) {
01488                     *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01489                     inset = 1;
01490                 }
01491                 else inset = 0;
01492                 if ( *s == TNUMBER || *s == TNUMBER1 ) {
01493                     base = *s == TNUMBER ? 100: 128;
01494                     *fill++ = *s++;
01495                     num = *s++; *fill++ = num;
01496                     while ( *s >= 0 ) {
01497                         if ( num > MAXPOWER ) break;
01498                         *fill++ = *s;
01499                         num = base*num + *s++;
01500                     }
01501                     while ( *s >= 0 ) *fill++ = *s++;
01502                     if ( num <= MAXPOWER ) continue;
01503                     goto putexpl;
01504                 }
01505                 else if ( *s == TSYMBOL && c != TNUMBER && c != TNUMBER1 ) {
01506                     *fill++ = *s++;
01507                     n = 0; while ( *s >= 0 ) { n = 128*n + *s; *fill++ = *s++; }
01508                     if ( *s == TWILDCARD ) { *fill++ = *s++;
01509                         if ( *s == TPOWER ) goto doublepower;
01510                     break; }
01511             /*
01512             Now we have to test whether n happens to be a wildcard
01513             */
01514             if ( mode == RHSIDE && inset == 0 ) {
01515                 n += WILDOFFSET; /*
01516                 for ( i = 0; i < numprot; i += 4 ) {
01517                     if ( prot[i+2] == n && prot[i] == SYMTOSYM ) break;
01518                 }
01519                 if ( i < numprot ) break;
01520             }
01521
01522             putexpl:
01523             fill = ff;
01524             if ( denom < 0 ) fill[-1] = TMULTIPLY;
01525             *fill++ = TFUNCTION; *fill++ = (SBYTE) (AM.expnum); *fill++ = TFUNOPEN;

```

```

01525         if ( dotp ) *fill++ = LPARENTHESIS;
01526         while ( ss < t ) *fill++ = *ss++;
01527         if ( dotp ) *fill++ = RPARENTHESIS;
01528         *fill++ = TCOMMA;
01529         ss++; /* Skip TPOWER */
01530         if ( *ss == TMINUS ) { denom = -denom; ss++; }
01531         if ( denom < 0 ) {
01532             *fill++ = LPARENTHESIS;
01533             *fill++ = TMINUS;
01534             while ( ss < s ) *fill++ = *ss++;
01535             *fill++ = RPARENTHESIS;
01536         }
01537         else {
01538             while ( ss < s ) *fill++ = *ss++;
01539         }
01540         *fill++ = TFUNCLOSE;
01541         if ( *ss == TPOWER ) goto doublepower;
01542     }
01543     else { /* other objects can be composite */
01544         goto dofunpower;
01545     }
01546     break;
01547 case TINDEX:
01548     *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01549     if ( *s == TWILDCARD ) *fill++ = *s++;
01550     break;
01551 case TVECTOR:
01552     *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01553     if ( *s == TWILDCARD ) *fill++ = *s++;
01554 dovector:
01555     if ( *s == TFUNOPEN ) {
01556         while ( *s != TFUNCLOSE ) *fill++ = *s++;
01557         *fill++ = *s++;
01558     }
01559     else if ( *s == TDOT ) goto dodot;
01560     t = s;
01561     goto dofunpower;
01562 #ifdef WITHFLOAT
01563 case TFLOAT:
01564     *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01565     t = s;
01566     sube = 0;
01567     goto dofunpower;
01568 #endif
01569 case TFUNCTION:
01570     *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01571     if ( *s == TWILDCARD ) *fill++ = *s++;
01572 dofunction:
01573     t = s;
01574     if ( *t == TFUNOPEN ) {
01575         t++; n = 1;
01576         for(;;) {
01577             if ( *t == TFUNOPEN ) n++;
01578             else if ( *t == TFUNCLOSE ) { if ( --n <= 0 ) break; }
01579             t++;
01580         }
01581         t++; *fill++ = *s++;
01582     }
01583     sube = 0;
01584 dofunpower:
01585     if ( *t == TPOWER || *t == TPOWER1 ) {
01586         if ( sube ) {
01587             if ( ( t[1] == TNUMBER || t[1] == TNUMBER1 )
01588                 && denom > 0 ) {
01589                 if ( t[2] >= 0 && t[3] < 0 ) { sube = 0; break; }
01590             }
01591             else if ( t[1] == TMINUS && denom < 0 &&
01592                 ( t[2] == TNUMBER || t[2] == TNUMBER1 ) ) {
01593                 if ( t[2] >= 0 && t[3] < 0 ) { sube = 0; break; }
01594             }
01595             sube = 0;
01596         }
01597         fill = ff;
01598         *fill++ = TFUNCTION; *fill++ = (SBYTE)(AM.expnum); *fill++ = TFUNOPEN;
01599         *fill++ = LPARENTHESIS;
01600         while ( ss < t ) *fill++ = *ss++;
01601         t++;
01602         *fill++ = RPARENTHESIS; *fill++ = TCOMMA;
01603         if ( *t == TMINUS ) { t++; denom = -denom; }
01604         *fill++ = LPARENTHESIS;
01605         if ( denom < 0 ) *fill++ = TMINUS;
01606         if ( *t == LPARENTHESIS ) {
01607             *fill++ = *t++; n = 0;
01608             while ( n >= 0 ) {
01609                 if ( *t == LPARENTHESIS ) n++;
01610                 else if ( *t == RPARENTHESIS ) n--;
01611                 *fill++ = *t++;

```

```

01612     }
01613 }
01614 else if ( *t == TFUNCTION || *t == TDUBIOUS ) {
01615     *fill++ = *t++; while ( *t >= 0 ) *fill++ = *t++;
01616     if ( *t == TWILDCARD ) *fill++ = *t++;
01617     if ( *t == TFUNOPEN ) {
01618         *fill++ = *t++; n = 0;
01619         while ( n >= 0 ) {
01620             if ( *t == TFUNOPEN ) n++;
01621             else if ( *t == TFUNCLOSE ) n--;
01622             *fill++ = *t++;
01623         }
01624     }
01625 }
01626 else if ( *t == TSET ) {
01627     *fill++ = *t++; n = 0;
01628     while ( *t >= 0 ) { n = 128*n + *t; *fill++ = *t++; }
01629     if ( *t == LBRACE ) {
01630         if ( n < AM.NumFixedSets || Sets[n].type == CRANGE ) {
01631             MesPrint("This type of usage of sets is not allowed");
01632             error = 1;
01633         }
01634         *fill++ = *t++; n = 0;
01635         while ( n >= 0 ) {
01636             if ( *t == LBRACE ) n++;
01637             else if ( *t == RBRACE ) n--;
01638             *fill++ = *t++;
01639         }
01640     }
01641 }
01642 else if ( *t == TEXPRESSION ) {
01643     *fill++ = *t++; while ( *t >= 0 ) *fill++ = *t++;
01644     if ( *t == TFUNOPEN ) {
01645         *fill++ = *t++; n = 0;
01646         while ( n >= 0 ) {
01647             if ( *t == TFUNOPEN ) n++;
01648             else if ( *t == TFUNCLOSE ) n--;
01649             *fill++ = *t++;
01650         }
01651     }
01652     if ( *t == LBRACE ) {
01653         *fill++ = *t++; n = 0;
01654         while ( n >= 0 ) {
01655             if ( *t == LBRACE ) n++;
01656             else if ( *t == RBRACE ) n--;
01657             *fill++ = *t++;
01658         }
01659     }
01660 }
01661 else if ( *t == TVECTOR ) {
01662     *fill++ = *t++; while ( *t >= 0 ) *fill++ = *t++;
01663     if ( *t == TFUNOPEN ) {
01664         *fill++ = *t++; n = 0;
01665         while ( n >= 0 ) {
01666             if ( *t == TFUNOPEN ) n++;
01667             else if ( *t == TFUNCLOSE ) n--;
01668             *fill++ = *t++;
01669         }
01670     }
01671     else if ( *t == TDOT ) {
01672         *fill++ = *t++;
01673         if ( *t == TVECTOR || *t == TDUBIOUS ) {
01674             *fill++ = *t++; while ( *t >= 0 ) *fill++ = *t++;
01675         }
01676         else if ( *t == TSET ) {
01677             *fill++ = *t++; num = 0;
01678             while ( *t >= 0 ) { num = 128*num + *t; *fill++ = *t++; }
01679             if ( Sets[num].type != CVECTOR ) {
01680                 MesPrint("&Illegal set type in dotproduct");
01681                 error = 1;
01682             }
01683             if ( *t == LBRACE ) {
01684                 *fill++ = *t++; n = 0;
01685                 while ( n >= 0 ) {
01686                     if ( *t == LBRACE ) n++;
01687                     else if ( *t == RBRACE ) n--;
01688                     *fill++ = *t++;
01689                 }
01690             }
01691         }
01692         else if ( *t == TSETNUM ) {
01693             *fill++ = *t++;
01694             while ( *t >= 0 ) { *fill++ = *t++; }
01695             *fill++ = *t++;
01696             while ( *t >= 0 ) { *fill++ = *t++; }
01697         }
01698     }

```

```

01699         else {
01700             MesPrint("&Illegal second element in dotproduct");
01701             error = 1;
01702         }
01703     }
01704     else {
01705         *fill++ = *t++; while ( *t >= 0 ) *fill++ = *t++;
01706         if ( *t == TWILDCARD ) *fill++ = *t++;
01707     }
01708     *fill++ = RPARENTHESIS; *fill++ = TFUNCLOSE;
01709     if ( *t == TPOWER ) goto doublepower;
01710     while ( fill > ff ) *--t = *--fill;
01711     s = t;
01712 }
01713 else if ( denom < 0 ) {
01714     fill = ff; ff[-1] = TMULTIPLY;
01715     *fill++ = TFUNCTION; *fill++ = (SBYTE) (AM.denomnum);
01716     *fill++ = TFUNOPEN; *fill++ = LPARENTHESIS;
01717     while ( ss < t ) *fill++ = *ss++;
01718     *fill++ = RPARENTHESIS; *fill++ = TFUNCLOSE;
01719     while ( fill > ff ) *--t = *--fill;
01720     s = t; denom = 1; sube = 0;
01721     break;
01722 }
01723 sube = 0;
01724 break;
01725 case TEXPRESSION:
01726     *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01727     t = s;
01728     if ( *t == TFUNOPEN ) {
01729         t++; n = 1;
01730         for(;;) {
01731             if ( *t == TFUNOPEN ) n++;
01732             else if ( *t == TFUNCLOSE ) { if ( --n <= 0 ) break; }
01733             t++;
01734         }
01735         t++;
01736     }
01737     if ( *t == LBRACE ) {
01738         t++; n = 1;
01739         for(;;) {
01740             if ( *t == LBRACE ) n++;
01741             else if ( *t == RBRACE ) { if ( --n <= 0 ) break; }
01742             t++;
01743         }
01744         t++;
01745     }
01746     if ( t > s || denom < 0 || ( ( *t == TPOWER || *t == TPOWER1 )
01747         && t[1] == TMINUS ) ) goto dofunpower;
01748     else goto dosymbol;
01749 case TDOLLAR:
01750     *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01751     goto dosymbol;
01752 case LPARENTHESIS:
01753     *fill++ = *s++; n = 1; t = s;
01754     for(;;) {
01755         if ( *t == LPARENTHESIS ) n++;
01756         else if ( *t == RPARENTHESIS ) { if ( --n <= 0 ) break; }
01757         t++;
01758     }
01759     t++; sube = 1;
01760     goto dofunpower;
01761 case TSET:
01762     *fill++ = *s++; n = *s++; *fill++ = (SBYTE)n;
01763     while ( *s >= 0 ) { *fill++ = *s; n = 128*n + *s++; }
01764     n = Sets[n].type;
01765     switch ( n ) {
01766         case CSYMBOL: goto dosymbol;
01767         case CINDEX: break;
01768         case CVECTOR: goto dovector;
01769         case CFUNCTION: goto dofunction;
01770         case CNUMBER: goto dosymbol;
01771         case CMODEL: break;
01772         case ANYTYPE: break;
01773         default: error = 1; break;
01774     }
01775     break;
01776 case TDOT:
01777     *fill++ = *s++;
01778     if ( *s == TVECTOR ) {
01779         *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01780     }
01781     else if ( *s == TSET ) {
01782         *fill++ = *s++; n = *s++; *fill++ = (SBYTE)n;
01783         while ( *s >= 0 ) { *fill++ = *s; n = 128*n + *s++; }
01784         if ( *s == LBRACE ) {
01785             if ( n < AM.NumFixedSets || Sets[n].type == CRANGE ) {

```

```

01786             MesPrint("&This type of usage of sets is not allowed");
01787             error = 1;
01788         }
01789         *fill++ = *s++; n = 1;
01790         for(;;) {
01791             if ( *s == LBRACE ) n++;
01792             else if ( *s == RBRACE ) { if ( --n <= 0 ) break; }
01793             *fill++ = *s++;
01794         }
01795         *fill++ = *s++;
01796     }
01797     else {
01798         MesPrint("&Set without argument in dotproduct");
01799         error = 1;
01800     }
01801 }
01802 else if ( *s == TSETNUM ) {
01803     *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01804     if ( *s != TVECTOR ) goto nodot;
01805     *fill++ = *s++; while ( *s >= 0 ) *fill++ = *s++;
01806 }
01807 else {
01808     MesPrint("&Illegal second element in dotproduct");
01809     error = 1;
01810     *fill++ = *s++;
01811     while ( *s >= 0 ) *fill++ = *s++;
01812 }
01813 dotp = 1;
01814 goto dosymbol;
01815 default:
01816     *fill++ = *s++;
01817     while ( *s >= 0 ) *fill++ = *s++;
01818     break;
01819 }
01820 }
01821 *fill = TENDOFIT;
01822 // Now copy the modified tokens back to the original buffer
01823 fill = tmptokens;
01824 s = AC.tokens;
01825 do {
01826     *s++ = *fill;
01827 } while ( *fill++ != TENDOFIT );
01828 M_free(tmptokens, "simp3btoken scratch");
01829 return(error);
01830 doublepower:;
01831 MesPrint("&Dubious notation with power of power");
01832 M_free(tmptokens, "simp3btoken scratch");
01833 return(-1);
01834 }
01835
01836 /*
01837     #] simp3btoken :
01838     #[ simp4token :
01839
01840     Deal with the set[n] objects in the RHS.
01841 */
01842
01843 int simp4token(SBYTE *s)
01844 {
01845     int error = 0, n, nsym, settype;
01846     WORD i, *w, *wstop, level;
01847     SBYTE *const s0 = s;
01848     SBYTE *fill = s, *s1, *s2, *s3, type, s1buf[10];
01849     SBYTE *tbuf = s, *t, *t1;
01850
01851     while ( *s != TENDOFIT ) {
01852         if ( *s != TSET ) {
01853             if ( *s == EMPTY ) s++;
01854             else *fill++ = *s++;
01855             continue;
01856         }
01857         if ( fill >= (s0+1) && fill[-1] == TWILDCARD ) { *fill++ = *s++; continue; }
01858         if ( fill >= (s0+2) && fill[-1] == TNOT && fill[-2] == TWILDCARD ) { *fill++ = *s++; continue; }
01859     }
01860     s1 = s++; n = 0; while ( *s >= 0 ) { n = 128*n + *s++; }
01861     i = Sets[n].type;
01862     if ( *s != LBRACE ) { while ( s1 < s ) *fill++ = *s1++; continue; }
01863     if ( n < AM.NumFixedSets || i == CRANGE ) {
01864         MesPrint("&It is not allowed to refer to individual elements of built in or ranged sets");
01865         error = 1;
01866     }
01867     s++;
01868     if ( *s != TSYMBOL && *s != TDOLLAR ) {
01869         MesPrint("&Set index in RHS is not a wildcard symbol or $-variable");
01870         error = 1;
01871         while ( s1 < s ) *fill++ = *s1++;
01872         continue;
01873     }

```



```

01872     }
01873     settype = ( *s == TDOLLAR );
01874     s++; nsym = 0; s2 = s;
01875     while ( *s >= 0 ) nsym = 128*nsym + *s++;
01876     if ( *s != RBRACE ) {
01877         MesPrint("&Improper set argument in RHS");
01878         error = 1;
01879         while ( s1 < s ) *fill++ = *s1++;
01880         continue;
01881     }
01882     s++;
01883     /*
01884     Verify that nsym is a wildcard
01885     */
01886     if ( !settype ) {
01887         w = AC.ProtoType; wstop = w + w[1]; w += SUBEXPSIZE;
01888         while ( w < wstop ) {
01889             if ( *w == SYMTOSYM && w[2] == nsym ) break;
01890             w += w[1];
01891         }
01892         if ( w >= wstop ) {
01893             /*
01894             It could still be a summation parameter!
01895             */
01896             t = fill - 1;
01897             while ( t >= tbuf ) {
01898                 if ( *t == TFUNCLOSE ) {
01899                     level = 1; t--;
01900                     while ( t >= tbuf ) {
01901                         if ( *t == TFUNCLOSE ) level++;
01902                         else if ( *t == TFUNOPEN ) {
01903                             level--;
01904                             if ( level == 0 ) break;
01905                         }
01906                         t--;
01907                     }
01908                 }
01909                 else if ( *t == RBRACE ) {
01910                     level = 1; t--;
01911                     while ( t >= tbuf ) {
01912                         if ( *t == RBRACE ) level++;
01913                         else if ( *t == LBRACE ) {
01914                             level--;
01915                             if ( level == 0 ) break;
01916                         }
01917                         t--;
01918                     }
01919                 }
01920                 else if ( *t == RPARENTHESIS ) {
01921                     level = 1; t--;
01922                     while ( t >= tbuf ) {
01923                         if ( *t == RPARENTHESIS ) level++;
01924                         else if ( *t == LPARENTHESIS ) {
01925                             level--;
01926                             if ( level == 0 ) break;
01927                         }
01928                         t--;
01929                     }
01930                 }
01931                 else if ( *t == TFUNOPEN ) {
01932                     t1 = t-1;
01933                     while ( *t1 > 0 && t1 > tbuf ) t1--;
01934                     if ( *t1 == TFUNCTION ) {
01935                         t1++; level = 0;
01936                         while ( *t1 > 0 ) level = level*128+*t1++;
01937                         if ( level == (SUMF1-FUNCTION)
01938                             || level == (SUMF2-FUNCTION) ) {
01939                             t1 = t + 1;
01940                             if ( *t1 == LPARENTHESIS ) t1++;
01941                             if ( *t1 == TSYMBOL ) {
01942                                 if ( ( t1[1] == COEFFSYMBOL
01943                                     || t1[1] == NUMERATORSYMBOL
01944                                     || t1[1] == DENOMINATORSYMBOL )
01945                                     && t1[2] < 0 ) {}
01946                                 else {
01947                                     t1++; level = 0;
01948                                     while ( *t1 >= 0 && t1 < fill ) level = 128*level + *t1++;
01949                                     if ( level == nsym && t1 < fill ) {
01950                                         if ( t[1] == LPARENTHESIS
01951                                             && *t1 == RPARENTHESIS && t1[1] == TCOMMA ) break;
01952                                         if ( t[1] != LPARENTHESIS && *t1 == TCOMMA ) break;
01953                                     }
01954                                 }
01955                             }
01956                         }
01957                     }
01958                 }

```

```

01959         t--;
01960     }
01961     if ( t < tbuf ) {
01962         fill--;
01963         MesPrint("&Set index in RHS is not a wildcard symbol");
01964         error = 1;
01965         while ( s1 < s ) *fill++ = *s1++;
01966         continue;
01967     }
01968 }
01969 }
01970 /*
01971     Now replace by a set marker: TSETNUM,nsym,TYPE,setnumber
01972 */
01973     switch ( i ) {
01974         case CSYMBOL: type = TSYMBOL; break;
01975         case CINDEX: type = TINDEX; break;
01976         case CVECTOR: type = TVECTOR; break;
01977         case CFUNCTION: type = TFUNCTION; break;
01978         case CNUMBER: type = TNUMBER1; break;
01979         case CDUBIOUS: type = TDUBIOUS; break;
01980         default:
01981             MesPrint("&Unknown set type in simp4token");
01982             error = 1; type = CDUBIOUS; break;
01983     }
01984     s3 = slbuf; s1++;
01985     while ( *s1 >= 0 ) *s3++ = *s1++;
01986     *s3 = -1; s1 = slbuf;
01987     if ( settype ) *fill++ = TSETDOL;
01988     else *fill++ = TSETNUM;
01989     while ( *s2 >= 0 ) *fill++ = *s2++;
01990     *fill++ = type; while ( *s1 >= 0 ) *fill++ = *s1++;
01991 }
01992 *fill++ = TENDOFIT;
01993 return(error);
01994 }
01995
01996 /*
01997     #] simp4token :
01998     #[ simp5token :
01999
02000     Making sure that first argument of sumfunction is not a wildcard already
02001 */
02002
02003 int simp5token(SBYTE *s, int mode)
02004 {
02005     int error = 0, n, type;
02006     WORD *w, *wstop;
02007     if ( mode == RHSIDE ) {
02008         while ( *s != TENDOFIT ) {
02009             if ( *s == TFUNCTION ) {
02010                 s++; n = 0; while ( *s >= 0 ) n = 128*n + *s++;
02011                 if ( n == AM.sumnum || n == AM.sumpnum ) {
02012                     if ( *s != TFUNOPEN ) continue;
02013                     s++;
02014                     if ( *s != TSYMBOL && *s != TINDEX ) continue;
02015                     type = *s++;
02016                     n = 0; while ( *s >= 0 ) n = 128*n + *s++;
02017                     if ( type == TINDEX ) n += AM.OffsetIndex;
02018                     if ( *s != TCOMMA ) continue;
02019                     w = AC.ProtoType;
02020                     wstop = w + w[1];
02021                     w += SUBEXPSIZE;
02022                     while ( w < wstop ) {
02023                         if ( w[2] == n ) {
02024                             if ( ( type == TSYMBOL && ( w[0] == SYMTOSYM
02025                                 || w[0] == SYMTONUM || w[0] == SYMTOSUB ) ) || (
02026                                 type == TINDEX && ( w[0] == INDTOIND
02027                                 || w[0] == INDTOSUB ) ) ) {
02028                                 error = 1;
02029                                 MesPrint("&Parameter of sum function is already a wildcard");
02030                             }
02031                         }
02032                         w += w[1];
02033                     }
02034                 }
02035             }
02036             else s++;
02037         }
02038     }
02039     return(error);
02040 }
02041
02042 /*
02043     #] simp5token :
02044     #[ simp6token :
02045

```

```

02046     Making sure that factorized expressions are used properly
02047 */
02048
02049 int simp6token(SBYTE *tokens, int mode)
02050 {
02051     /* EXPRESSIONS e = Expressions; */
02052     int error = 0, n;
02053     int level = 0, haveone = 0;
02054     SBYTE *s = tokens, *ss;
02055     LONG numterms;
02056     WORD funnum = 0;
02057     GETIDENTITY
02058     if ( mode == RHSIDE ) {
02059         while ( *s == TPLUS || *s == TMINUS ) s++;
02060         numterms = 1;
02061         while ( *s != TENDOFIT ) {
02062             if ( *s == LPARENTHESIS ) level++;
02063             else if ( *s == RPARENTHESIS ) level--;
02064             else if ( *s == TFUNOPEN ) level++;
02065             else if ( *s == TFUNCLOSE ) level--;
02066             else if ( ( *s == TPLUS || *s == TMINUS ) && level == 0 ) {
02067                 /*
02068                     Special exception: x-1 etc.
02069                 */
02070                 if ( s[-1] != TPOWER && s[-1] != TPLUS && s[-1] != TMINUS ) {
02071                     numterms++;
02072                 }
02073             }
02074             else if ( *s == TEXPRESSION ) {
02075                 ss = s;
02076                 s++; n = 0; while ( *s >= 0 ) n = 128*n + *s++;
02077             }
02078             if ( Expressions[n].status == STOREDEXPRESSION ) {
02079                 POSITION position;
02080                 /*
02081                 #ifdef WITHPTHREADS
02082                     RENUMBER renumber;
02083                 #endif
02084                 */
02085                 RENUMBER renumber;
02086
02087                 WORD Tmproto[SUBEXPSIZE];
02088                 Tmproto[0] = EXPRESSION;
02089                 Tmproto[1] = SUBEXPSIZE;
02090                 Tmproto[2] = n;
02091                 Tmproto[3] = 1;
02092                 { int ie; for ( ie = 4; ie < SUBEXPSIZE; ie++ ) Tmproto[ie] = 0; }
02093                 AT.TMaddr = Tmproto;
02094                 PUTZERO(position);
02095                 /*
02096                 #ifdef WITHPTHREADS
02097                     if ( (
02098                         renumber =
02099                     #endif
02100                         GetTable(n,&position,0) ) == 0 )
02101                 */
02102                 if ( ( renumber = GetTable(n,&position,0) ) == 0 )
02103                 {
02104                     error = 1;
02105                     MesPrint("&Problems getting information about stored expression %s(4) "
02106                         ,EXPRNAME(n));
02107                 }
02108                 /*
02109                 #ifdef WITHPTHREADS
02110                 */
02111                 if ( renumber->symb.lo != AN.dummyrenumlist )
02112                     M_free(renumber->symb.lo,"VarSpace");
02113                 M_free(renumber,"Renumber");
02114                 /*
02115                 #endif
02116                 */
02117             }
02118
02119             if ( ( ( AS.Oldvflags[n] & ISFACTORIZED ) != 0 ) && *s != LBRACE ) {
02120                 if ( level == 0 ) {
02121                     haveone = 1;
02122                 }
02123                 else if ( error == 0 ) {
02124                     if ( ss[-1] != TFUNOPEN || funnum != NUMFACTORS-FUNCTION ) {
02125                         MesPrint("&Illegal use of factorized expression(s) in RHS");
02126                         error = 1;
02127                     }
02128                 }
02129             }
02130             continue;
02131         }
02132         else if ( *s == TFUNCTION ) {

```

```

02133         s++; funnum = 0; while ( *s >= 0 ) funnum = 128*fnum + *s++;
02134         continue;
02135     }
02136     s++;
02137 }
02138 if ( haveone ) {
02139     if ( numterms > 1 ) {
02140         MesPrint("&Factorized expression in RHS in an expression of more than one term.");
02141         error = 1;
02142     }
02143     else if ( AC.ToBeInFactors == 0 ) {
02144         MesPrint("&Attempt to put a factorized expression inside an unfactorized
expression.");
02145         error = 1;
02146     }
02147 }
02148 }
02149 return(error);
02150 }
02151
02152 /*
02153     #] simp6token :
02154     #] Compiler :
02155 */

```

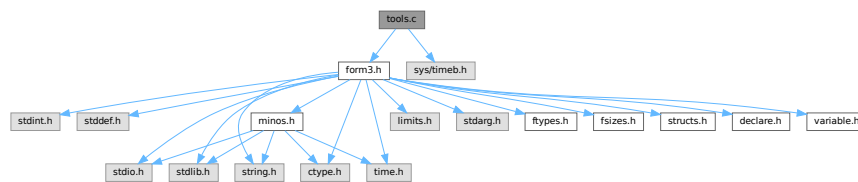
4.147 tools.c File Reference

```

#include "form3.h"
#include <sys/timeb.h>

```

Include dependency graph for tools.c:



Macros

- #define [FILLVALUE](#) 0
- #define [COPYFILEBUFSIZE](#) 40960L
- #define [TERMMEMSTARTNUM](#) 16
- #define [TERMEXTRAWORDS](#) 10
- #define [NUMBERMEMSTARTNUM](#) 16
- #define [NUMBEREXTRAWORDS](#) 10L
- #define [DODOUBLE](#)(x)
- #define [DOEXPAND](#)(x)

Functions

- UBYTE * [LoadInputFile](#) (UBYTE *filename, int type)
- UBYTE [ReadFromStream](#) (STREAM *stream)
- UBYTE [GetFromStream](#) (STREAM *stream)
- UBYTE [LookInStream](#) (STREAM *stream)
- STREAM * [OpenStream](#) (UBYTE *name, int type, int prevarmode, int raiselow)
- int [LocateFile](#) (UBYTE **name, int type)

- [STREAM](#) * [CloseStream](#) ([STREAM](#) *stream)
- [STREAM](#) * [CreateStream](#) (UBYTE *where)
- LONG [GetStreamPosition](#) ([STREAM](#) *stream)
- void [PositionStream](#) ([STREAM](#) *stream, LONG position)
- int [ReverseStatements](#) ([STREAM](#) *stream)
- void [StartFiles](#) (void)
- int [OpenFile](#) (char *name)
- int [OpenAddFile](#) (char *name)
- int [ReOpenFile](#) (char *name)
- int [CreateFile](#) (char *name)
- int [CreateLogFile](#) (char *name)
- void [CloseFile](#) (int handle)
- int [CopyFile](#) (char *source, char *dest)
- int [CreateHandle](#) (void)
- LONG [ReadFile](#) (int handle, UBYTE *buffer, LONG size)
- LONG [ReadPosFile](#) (PHEAD [FILEHANDLE](#) *fi, UBYTE *buffer, LONG size, [POSITION](#) *pos)
- LONG [WriteFileToFile](#) (int handle, UBYTE *buffer, LONG size)
- void [SeekFile](#) (int handle, [POSITION](#) *offset, int origin)
- LONG [TellFile](#) (int handle)
- void [TELLFILE](#) (int handle, [POSITION](#) *position)
- void [FlushFile](#) (int handle)
- int [GetPosFile](#) (int handle, fpos_t *pospointer)
- int [SetPosFile](#) (int handle, fpos_t *pospointer)
- void [SynchFile](#) (int handle)
- void [TruncateFile](#) (int handle)
- int [GetChannel](#) (char *name, int mode)
- int [GetAppendChannel](#) (char *name)
- int [CloseChannel](#) (char *name)
- void [UpdateMaxSize](#) (void)
- int [StrCmp](#) (UBYTE *s1, UBYTE *s2)
- int [StrlCmp](#) (UBYTE *s1, UBYTE *s2)
- int [StrHlCmp](#) (UBYTE *s1, UBYTE *s2)
- int [StrlCont](#) (UBYTE *s1, UBYTE *s2)
- int [CmpArray](#) (WORD *t1, WORD *t2, WORD n)
- int [ConWord](#) (UBYTE *s1, UBYTE *s2)
- int [StrLen](#) (UBYTE *s)
- void [NumToStr](#) (UBYTE *s, LONG x)
- void [WriteString](#) (int type, UBYTE *str, int num)
- void [WriteUnfinString](#) (int type, UBYTE *str, int num)
- UBYTE * [AddToString](#) (UBYTE *outstring, UBYTE *extrastring, int par)
- UBYTE * [strDup1](#) (UBYTE *instring, char *ifwrong)
- UBYTE * [EndOfToken](#) (UBYTE *s)
- UBYTE * [ToToken](#) (UBYTE *s)
- UBYTE * [SkipField](#) (UBYTE *s, int level)
- WORD [ReadSnum](#) (UBYTE **p)
- UBYTE * [NumCopy](#) (WORD y, UBYTE *to)
- char * [LongCopy](#) (LONG y, char *to)
- char * [LongLongCopy](#) (off_t *y, char *to)
- UBYTE * [MakeDate](#) (void)
- int [set_in](#) (UBYTE ch, [set_of_char](#) set)
- [one_byte](#) [set_set](#) (UBYTE ch, [set_of_char](#) set)
- [one_byte](#) [set_del](#) (UBYTE ch, [set_of_char](#) set)
- [one_byte](#) [set_sub](#) ([set_of_char](#) set, [set_of_char](#) set1, [set_of_char](#) set2)
- void [iniTools](#) (void)
- void * [Malloc1](#) (LONG size, const char *messageifwrong)

- void [M_free](#) (void *x, const char *where)
- void [M_check1](#) (void)
- void [M_print](#) (void)
- void [TermMallocAddMemory](#) (PHEAD0)
- void [NumberMallocAddMemory](#) (PHEAD0)
- void [CacheNumberMallocAddMemory](#) (PHEAD0)
- void * [FromList](#) (LIST *L)
- void * [From0List](#) (LIST *L)
- void * [FromVarList](#) (LIST *L)
- int [DoubleList](#) (void ***lijst, int *oldsize, int objectsize, char *nameoftype)
- int [DoubleLList](#) (void ***lijst, LONG *oldsize, int objectsize, char *nameoftype)
- void [DoubleBuffer](#) (void **start, void **stop, int size, char *text)
- void [ExpandBuffer](#) (void **buffer, LONG *oldsize, int type)
- LONG [iexp](#) (LONG x, int p)
- void [ToGeneral](#) (WORD *r, WORD *m, WORD par)
- int [ToFast](#) (WORD *r, WORD *m)
- WORD [ToPolyFunGeneral](#) (PHEAD WORD *term)
- int [IsLikeVector](#) (WORD *arg)
- int [AreArgsEqual](#) (WORD *arg1, WORD *arg2)
- int [CompareArgs](#) (WORD *arg1, WORD *arg2)
- int [CompArg](#) (WORD *s1, WORD *s2)
- LONG [TimeWallClock](#) (WORD par)
- LONG [TimeChildren](#) (WORD par)
- LONG [TimeCPU](#) (WORD par)
- int [Crash](#) (void)
- int [TestTerm](#) (WORD *term)
- int [DistrN](#) (int n, int *cpl, int ncpl, int *scratch)

Variables

- FILES ** [filelist](#)
- int [numinfilelist](#) = 0
- int [filelistsize](#) = 0
- WRITEFILE [WriteFile](#) = &WriteFileToFile

4.147.1 Detailed Description

Low level routines for many types of task. There are routines for manipulating the input system (streams and files) routines for string manipulation, the memory allocation interface, and the clock. The last is the most sensitive to ports. In the past nearly every port to another OS or computer gave trouble. Nowadays it is slightly better but the poor POSIX compliance of LINUX again gave problems for the multithreaded version.

Definition in file [tools.c](#).

4.147.2 Macro Definition Documentation

4.147.2.1 FILLVALUE

```
#define FILLVALUE 0
```

Definition at line 49 of file [tools.c](#).

4.147.2.2 TERMMEMSTARTNUM

```
#define TERMMEMSTARTNUM 16
```

Provides memory for one term (or one small polynomial) This means that the memory is limited to a buffer of size AM.MaxTer plus a few extra words. In parallel versions, each worker has its own memory pool.

The way we use the memory is by: term = TermMalloc(BHEAD0); and later we free it by TermFree(BHEAD term);

Layout: We have a list of available pointers to buffers: AT.TermMemHeap Its size is AT.TermMemMax We take from the top (indicated by AT.TermMemTop). When we run out of buffers we assign new ones (doubling the amount) and we have to extend the AT.TermMemHeap array. Important: There is no checking that the returned memory is legal, ie is memory that was handed out earlier.

Definition at line 2483 of file [tools.c](#).

4.147.2.3 TERMEXTRAWORDS

```
#define TERMEXTRAWORDS 10
```

Definition at line 2484 of file [tools.c](#).

4.147.2.4 NUMBERMEMSTARTNUM

```
#define NUMBERMEMSTARTNUM 16
```

Provides memory for one Long number This means that the memory is limited to a buffer of size AM.MaxTal In parallel versions, each worker has its own memory pool.

The way we use the memory is by: num = NumberMalloc(BHEAD0); Number = AT.NumberMemHeap[num]; and later we free it by NumberFree(BHEAD num);

Layout: We have a list of available pointers to buffers: AT.NumberMemHeap Its size is AT.NumberMemMax We take from the top (indicated by AT.NumberMemTop). When we run out of buffers we assign new ones (doubling the amount) and we have to extend the AT.NumberMemHeap array. Important: There is no checking on the returned memory!!!!

Definition at line 2583 of file [tools.c](#).

4.147.2.5 NUMBEREXTRAWORDS

```
#define NUMBEREXTRAWORDS 10L
```

Definition at line 2584 of file [tools.c](#).

4.147.2.6 DODOUBLE

```
#define DODOUBLE(  
    x )
```

Value:

```
{ x *s, *t, *u; if ( *start ) { \
  oldsize = *(x **)stop - *(x **)start; newsize = 2*oldsize; \
  t = u = (x *)Malloca1(newsize*sizeof(x),text); s = *(x **)start; \
  for ( i = 0; i < oldsize; i++ ) { *t++ = *s++; } M_free(*start,"double"); } \
else { newsize = 100; u = (x *)Malloca1(newsize*sizeof(x),text); } \
*start = (void *)u; *stop = (void *) (u+newsize); }
```

Definition at line 2906 of file [tools.c](#).

4.147.2.7 DOEXPAND

```
#define DOEXPAND(
    x )
```

Value:

```
{ x *newbuffer, *t, *m;
  t = newbuffer = (x *)Malloc1((newsize+2)*type,"ExpandBuffer");
  if ( *buffer ) { m = (x *)*buffer; i = *oldsize;
    while ( --i >= 0 ) {t++ = *m++;} M_free(*buffer,"ExpandBuffer");
  } *buffer = newbuffer; *oldsize = newsize; }
```

Definition at line 2932 of file [tools.c](#).

4.147.3 Function Documentation

4.147.3.1 LoadInputFile()

```
UBYTE * LoadInputFile (
    UBYTE * filename,
    int type )
```

Definition at line 112 of file [tools.c](#).

4.147.3.2 ReadFromStream()

```
UBYTE ReadFromStream (
    STREAM * stream )
```

Definition at line 151 of file [tools.c](#).

4.147.3.3 GetFromStream()

```
UBYTE GetFromStream (
    STREAM * stream )
```

Definition at line 286 of file [tools.c](#).

4.147.3.4 LookInStream()

```
UBYTE LookInStream (
    STREAM * stream )
```

Definition at line 309 of file [tools.c](#).

4.147.3.5 OpenStream()

```
STREAM * OpenStream (
    UBYTE * name,
    int type,
    int prevarmode,
    int raiselow )
```

Definition at line 321 of file [tools.c](#).

4.147.3.6 LocateFile()

```
int LocateFile (
    UBYTE ** name,
    int type )
```

Definition at line 576 of file [tools.c](#).

4.147.3.7 CloseStream()

```
STREAM * CloseStream (
    STREAM * stream )
```

Definition at line 663 of file [tools.c](#).

4.147.3.8 CreateStream()

```
STREAM * CreateStream (
    UBYTE * where )
```

Definition at line 784 of file [tools.c](#).

4.147.3.9 GetStreamPosition()

```
LONG GetStreamPosition (
    STREAM * stream )
```

Definition at line 815 of file [tools.c](#).

4.147.3.10 PositionStream()

```
void PositionStream (
    STREAM * stream,
    LONG position )
```

Definition at line 825 of file [tools.c](#).

4.147.3.11 ReverseStatements()

```
int ReverseStatements (
    STREAM * stream )
```

Definition at line 857 of file [tools.c](#).

4.147.3.12 StartFiles()

```
void StartFiles (
    void )
```

Definition at line 958 of file [tools.c](#).

4.147.3.13 OpenFile()

```
int OpenFile (  
    char * name )
```

Definition at line [985](#) of file [tools.c](#).

4.147.3.14 OpenAddFile()

```
int OpenAddFile (  
    char * name )
```

Definition at line [1004](#) of file [tools.c](#).

4.147.3.15 ReOpenFile()

```
int ReOpenFile (  
    char * name )
```

Definition at line [1025](#) of file [tools.c](#).

4.147.3.16 CreateFile()

```
int CreateFile (  
    char * name )
```

Definition at line [1045](#) of file [tools.c](#).

4.147.3.17 CreateLogFile()

```
int CreateLogFile (  
    char * name )
```

Definition at line [1062](#) of file [tools.c](#).

4.147.3.18 CloseFile()

```
void CloseFile (  
    int handle )
```

Definition at line [1080](#) of file [tools.c](#).

4.147.3.19 CopyFile()

```
int CopyFile (
    char * source,
    char * dest )
```

Copies a file with name **source* to a file named **dest*. The involved files must not be open. Returns non-zero if an error occurred. Uses if possible the combined large and small sorting buffers as cache.

Definition at line 1103 of file [tools.c](#).

Referenced by [DoRecovery\(\)](#).

4.147.3.20 CreateHandle()

```
int CreateHandle (
    void )
```

Definition at line 1164 of file [tools.c](#).

4.147.3.21 ReadFile()

```
LONG ReadFile (
    int handle,
    UBYTE * buffer,
    LONG size )
```

Definition at line 1219 of file [tools.c](#).

4.147.3.22 ReadPosFile()

```
LONG ReadPosFile (
    PHEAD FILEHANDLE * fi,
    UBYTE * buffer,
    LONG size,
    POSITION * pos )
```

Definition at line 1269 of file [tools.c](#).

4.147.3.23 WriteFileToFile()

```
LONG WriteFileToFile (
    int handle,
    UBYTE * buffer,
    LONG size )
```

Definition at line 1335 of file [tools.c](#).

4.147.3.24 SeekFile()

```
void SeekFile (
    int handle,
    POSITION * offset,
    int origin )
```

Definition at line 1373 of file [tools.c](#).

4.147.3.25 TellFile()

```
LONG TellFile (
    int handle )
```

Definition at line 1398 of file [tools.c](#).

4.147.3.26 TELLFILE()

```
void TELLFILE (
    int handle,
    POSITION * position )
```

Definition at line 1408 of file [tools.c](#).

4.147.3.27 FlushFile()

```
void FlushFile (
    int handle )
```

Definition at line 1422 of file [tools.c](#).

4.147.3.28 GetPosFile()

```
int GetPosFile (
    int handle,
    fpos_t * pospointer )
```

Definition at line 1436 of file [tools.c](#).

4.147.3.29 SetPosFile()

```
int SetPosFile (
    int handle,
    fpos_t * pospointer )
```

Definition at line 1450 of file [tools.c](#).

4.147.3.30 SynchFile()

```
void SynchFile (  
    int handle )
```

Definition at line 1470 of file [tools.c](#).

4.147.3.31 TruncateFile()

```
void TruncateFile (  
    int handle )
```

Definition at line 1492 of file [tools.c](#).

4.147.3.32 GetChannel()

```
int GetChannel (  
    char * name,  
    int mode )
```

Definition at line 1512 of file [tools.c](#).

4.147.3.33 GetAppendChannel()

```
int GetAppendChannel (  
    char * name )
```

Definition at line 1548 of file [tools.c](#).

4.147.3.34 CloseChannel()

```
int CloseChannel (  
    char * name )
```

Definition at line 1578 of file [tools.c](#).

4.147.3.35 UpdateMaxSize()

```
void UpdateMaxSize (  
    void )
```

Definition at line 1612 of file [tools.c](#).

4.147.3.36 StrCmp()

```
int StrCmp (  
    UBYTE * s1,  
    UBYTE * s2 )
```

Definition at line 1702 of file [tools.c](#).

4.147.3.37 StrICmp()

```
int StrICmp (
    UBYTE * s1,
    UBYTE * s2 )
```

Definition at line 1713 of file [tools.c](#).

4.147.3.38 StrHICmp()

```
int StrHICmp (
    UBYTE * s1,
    UBYTE * s2 )
```

Definition at line 1724 of file [tools.c](#).

4.147.3.39 StrICont()

```
int StrICont (
    UBYTE * s1,
    UBYTE * s2 )
```

Definition at line 1735 of file [tools.c](#).

4.147.3.40 CmpArray()

```
int CmpArray (
    WORD * t1,
    WORD * t2,
    WORD n )
```

Definition at line 1747 of file [tools.c](#).

4.147.3.41 ConWord()

```
int ConWord (
    UBYTE * s1,
    UBYTE * s2 )
```

Definition at line 1761 of file [tools.c](#).

4.147.3.42 StrLen()

```
int StrLen (
    UBYTE * s )
```

Definition at line 1773 of file [tools.c](#).

4.147.3.43 NumToStr()

```
void NumToStr (
    UBYTE * s,
    LONG x )
```

Definition at line 1785 of file [tools.c](#).

4.147.3.44 WriteString()

```
void WriteString (
    int type,
    UBYTE * str,
    int num )
```

Definition at line 1809 of file [tools.c](#).

4.147.3.45 WriteUnfinString()

```
void WriteUnfinString (
    int type,
    UBYTE * str,
    int num )
```

Definition at line 1843 of file [tools.c](#).

4.147.3.46 AddToString()

```
UBYTE * AddToString (
    UBYTE * outstring,
    UBYTE * extrastring,
    int par )
```

Definition at line 1868 of file [tools.c](#).

4.147.3.47 strDup1()

```
UBYTE * strDup1 (
    UBYTE * instring,
    char * ifwrong )
```

Definition at line 1906 of file [tools.c](#).

4.147.3.48 EndOfToken()

```
UBYTE * EndOfToken (
    UBYTE * s )
```

Skips over alphanumeric characters to find the end of the token.

Note

This function does not handle formal names (e.g., [x+a]). To handle them, use `SkipAName` instead.

Parameters

in	<i>s</i>	Pointer to a null-terminated input buffer.

Returns

Pointer to the first non-alphanumeric character (i.e., the character immediately after the token), or the null terminator if the token reaches the end of the string.

Definition at line [1932](#) of file [tools.c](#).

4.147.3.49 ToToken()

```
UBYTE * ToToken (
    UBYTE * s )
```

Skips over non-alphanumeric characters to find the start of a token.

Note

This function does not handle formal names (e.g., `[x+a]`). To handle them, consider simply skipping whitespace, e.g., by using `SkipSpaces`.

Parameters

in	<i>s</i>	Pointer to a null-terminated input buffer.

Returns

Pointer to the first alphanumeric character (i.e., the start of the token), or the null terminator if none is found.

Definition at line [1955](#) of file [tools.c](#).

4.147.3.50 SkipField()

```
UBYTE * SkipField (
    UBYTE * s,
    int level )
```

Skips from *s* to the end of a declaration field (e.g., `x, 1+2*[x+a]`, or `f ({x, [x+a] }, 1)`).

Parameters

in	<i>s</i>	Pointer to a null-terminated input buffer.
in	<i>level</i>	Number of parentheses that still have to be closed.

Returns

Pointer to the character after the field, which is either a comma at level 0 or the null terminator.

Definition at line 1976 of file [tools.c](#).

4.147.3.51 ReadSnum()

```
WORD ReadSnum (
    UBYTE ** p )
```

Definition at line 2003 of file [tools.c](#).

4.147.3.52 NumCopy()

```
UBYTE * NumCopy (
    WORD y,
    UBYTE * to )
```

Definition at line 2027 of file [tools.c](#).

4.147.3.53 LongCopy()

```
char * LongCopy (
    LONG y,
    char * to )
```

Definition at line 2054 of file [tools.c](#).

4.147.3.54 LongLongCopy()

```
char * LongLongCopy (
    off_t * y,
    char * to )
```

Definition at line 2081 of file [tools.c](#).

4.147.3.55 MakeDate()

```
UBYTE * MakeDate (
    void )
```

Definition at line 2117 of file [tools.c](#).

4.147.3.56 set_in()

```
int set_in (
    UBYTE ch,
    set_of_char set )
```

Definition at line 2133 of file [tools.c](#).

4.147.3.57 set_set()

```
one_byte set_set (
    UBYTE ch,
    set_of_char set )
```

Definition at line 2153 of file [tools.c](#).

4.147.3.58 set_del()

```
one_byte set_del (
    UBYTE ch,
    set_of_char set )
```

Definition at line 2174 of file [tools.c](#).

4.147.3.59 set_sub()

```
one_byte set_sub (
    set_of_char set,
    set_of_char set1,
    set_of_char set2 )
```

Definition at line 2196 of file [tools.c](#).

4.147.3.60 iniTools()

```
void iniTools (
    void )
```

Definition at line 2222 of file [tools.c](#).

4.147.3.61 Malloc1()

```
void * Malloc1 (
    LONG size,
    const char * messageifwrong )
```

Definition at line 2243 of file [tools.c](#).

4.147.3.62 M_free()

```
void M_free (
    void * x,
    const char * where )
```

Definition at line 2319 of file [tools.c](#).

4.147.3.63 M_check1()

```
void M_check1 (
    void )
```

Definition at line [2452](#) of file [tools.c](#).

4.147.3.64 M_print()

```
void M_print (
    void )
```

Definition at line [2453](#) of file [tools.c](#).

4.147.3.65 TermMallocAddMemory()

```
void TermMallocAddMemory (
    PHEAD0 )
```

Definition at line [2486](#) of file [tools.c](#).

4.147.3.66 NumberMallocAddMemory()

```
void NumberMallocAddMemory (
    PHEAD0 )
```

Definition at line [2590](#) of file [tools.c](#).

4.147.3.67 CacheNumberMallocAddMemory()

```
void CacheNumberMallocAddMemory (
    PHEAD0 )
```

Definition at line [2664](#) of file [tools.c](#).

4.147.3.68 FromList()

```
void * FromList (
    LIST * L )
```

Definition at line [2716](#) of file [tools.c](#).

4.147.3.69 From0List()

```
void * From0List (
    LIST * L )
```

Definition at line [2742](#) of file [tools.c](#).

4.147.3.70 FromVarList()

```
void * FromVarList (
    LIST * L )
```

Definition at line 2771 of file [tools.c](#).

4.147.3.71 DoubleList()

```
int DoubleList (
    void *** lijst,
    int * oldsize,
    int objectsize,
    char * nameoftype )
```

Definition at line 2813 of file [tools.c](#).

4.147.3.72 DoubleLList()

```
int DoubleLList (
    void *** lijst,
    LONG * oldsize,
    int objectsize,
    char * nameoftype )
```

Definition at line 2865 of file [tools.c](#).

4.147.3.73 DoubleBuffer()

```
void DoubleBuffer (
    void ** start,
    void ** stop,
    int size,
    char * text )
```

Definition at line 2913 of file [tools.c](#).

4.147.3.74 ExpandBuffer()

```
void ExpandBuffer (
    void ** buffer,
    LONG * oldsize,
    int type )
```

Definition at line 2938 of file [tools.c](#).

4.147.3.75 iexp()

```
LONG iexp (
    LONG x,
    int p )
```

Definition at line [2962](#) of file [tools.c](#).

4.147.3.76 ToGeneral()

```
void ToGeneral (
    WORD * r,
    WORD * m,
    WORD par )
```

Definition at line [2993](#) of file [tools.c](#).

4.147.3.77 ToFast()

```
int ToFast (
    WORD * r,
    WORD * m )
```

Definition at line [3037](#) of file [tools.c](#).

4.147.3.78 ToPolyFunGeneral()

```
WORD ToPolyFunGeneral (
    PHEAD WORD * term )
```

Definition at line [3090](#) of file [tools.c](#).

4.147.3.79 IsLikeVector()

```
int IsLikeVector (
    WORD * arg )
```

Definition at line [3165](#) of file [tools.c](#).

4.147.3.80 AreArgsEqual()

```
int AreArgsEqual (
    WORD * arg1,
    WORD * arg2 )
```

Definition at line [3193](#) of file [tools.c](#).

4.147.3.81 CompareArgs()

```
int CompareArgs (
    WORD * arg1,
    WORD * arg2 )
```

Definition at line 3212 of file [tools.c](#).

4.147.3.82 CompArg()

```
int CompArg (
    WORD * s1,
    WORD * s2 )
```

Definition at line 3243 of file [tools.c](#).

4.147.3.83 TimeWallClock()

```
LONG TimeWallClock (
    WORD par )
```

Returns the wall-clock time.

Parameters

<i>par</i>	If zero, the wall-clock time will be reset to 0.

Returns

The wall-clock time in centiseconds.

Definition at line 3413 of file [tools.c](#).

Referenced by [DoCheckpoint\(\)](#), [DoRecovery\(\)](#), [find_Horner_MCTS\(\)](#), [optimize_expression_given_Horner\(\)](#), [optimize_greedy\(\)](#), and [WriteStats\(\)](#).

4.147.3.84 TimeChildren()

```
LONG TimeChildren (
    WORD par )
```

Definition at line 3469 of file [tools.c](#).

4.147.3.85 TimeCPU()

```
LONG TimeCPU (
    WORD par )
```

Returns the CPU time.

Parameters

<i>par</i>	If zero, the CPU time will be reset to 0.

Returns

The CPU time in milliseconds.

Definition at line 3487 of file [tools.c](#).

Referenced by [PF_GetSlaveTimes\(\)](#), [PF_Processor\(\)](#), and [WriteStats\(\)](#).

4.147.3.86 Crash()

```
int Crash (
    void )
```

Definition at line 3770 of file [tools.c](#).

4.147.3.87 TestTerm()

```
int TestTerm (
    WORD * term )
```

Tests the consistency of the term. Returns 0 when the term is OK. Any nonzero value is trouble. In the current version the testing isn't 100% complete. For instance, we don't check the validity of the symbols nor do we check the range of their powers. Etc. This should be extended when the need is there.

Parameters

<i>term</i>	the term to be tested

Definition at line 3798 of file [tools.c](#).

References [TestTerm\(\)](#).

Referenced by [TestTerm\(\)](#).

Here is the call graph for this function:



4.147.3.88 DistrN()

```
int DistrN (
    int n,
    int * cpl,
    int ncpl,
    int * scratch )
```

Definition at line [3984](#) of file [tools.c](#).

4.147.4 Variable Documentation

4.147.4.1 filelist

```
FILES** filelist
```

Definition at line [65](#) of file [tools.c](#).

4.147.4.2 numinfilelist

```
int numinfilelist = 0
```

Definition at line [66](#) of file [tools.c](#).

4.147.4.3 filelistsize

```
int filelistsize = 0
```

Definition at line [67](#) of file [tools.c](#).

4.147.4.4 WriteFile

```
WRITEFILE WriteFile = &WriteFileToFile
```

Definition at line [1359](#) of file [tools.c](#).

4.148 tools.c

[Go to the documentation of this file.](#)

```

00001
00011 /* #[ License : */
00012 /*
00013 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00014 *   When using this file you are requested to refer to the publication
00015 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00016 *   This is considered a matter of courtesy as the development was paid
00017 *   for by FOM the Dutch physics granting agency and we would like to
00018 *   be able to track its scientific use to convince FOM of its value
00019 *   for the community.
00020 *
00021 *   This file is part of FORM.
00022 *
00023 *   FORM is free software: you can redistribute it and/or modify it under the
00024 *   terms of the GNU General Public License as published by the Free Software
00025 *   Foundation, either version 3 of the License, or (at your option) any later
00026 *   version.
00027 *
00028 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00029 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00030 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00031 *   details.
00032 *
00033 *   You should have received a copy of the GNU General Public License along
00034 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00035 */
00036 /* #] License : */
00037 /*
00038 *   #[ Includes :
00039 *   Note: TERMMALLOCDEBUG tests part of the TermMalloc and NumberMalloc
00040 *   system. To work properly it needs MEMORYMACROS in declare.h
00041 *   not to be defined to make sure that all calls will be diverted
00042 *   to the routines here.
00043 #define TERMMALLOCDEBUG
00044 #define FILLVALUE 126
00045 #define MALLOCDEBUGOUTPUT
00046 #define MALLOCDEBUG 1
00047 */
00048 #ifndef FILLVALUE
00049     #define FILLVALUE 0
00050 #endif
00051
00052 /*
00053 *   The enhanced malloc debugger, see comments in the beginning of the
00054 *   file mallocprotect.h
00055 *   MALLOCPROTECT == -1 -- protect left side, used block is left-aligned.
00056 *   MALLOCPROTECT == 0 -- protect both sides, used block is left-aligned;
00057 *   MALLOCPROTECT == 1 -- protect both sides, used block is right-aligned;
00058 *   ATTENTION! The macro MALLOCPROTECT must be defined
00059 *   BEFORE #include mallocprotect.h
00060 #define MALLOCPROTECT 1
00061 */
00062
00063 #include "form3.h"
00064
00065 FILES **filelist;
00066 int numinfilelist = 0;
00067 int filelistsize = 0;
00068 #ifdef MALLOCDEBUG
00069 #define BANNER (4*sizeof(LONG))
00070 void *malloclist[60000];
00071 LONG mallocsizes[60000];
00072 char *mallocstrings[60000];
00073 int nummalloclist = 0;
00074 #endif
00075
00076 #ifdef GPP
00077 extern "C" getdtablesize();
00078 #endif
00079
00080 #ifdef WITHSTATS
00081 LONG numwrites = 0;
00082 LONG numreads = 0;
00083 LONG numseeks = 0;
00084 LONG nummallocs = 0;
00085 LONG numfrees = 0;
00086 #endif
00087
00088 #ifdef MALLOCPROTECT
00089 #ifdef TRAP_SIGNALS
00090 #error "MALLOCPROTECT":  undefine "TRAP_SIGNALS" in unix.h first!
00091 #endif

```

```

00092 #include "mallocprotect.h"
00093
00094 #ifdef M_alloc
00095 #undef M_alloc
00096 #endif
00097
00098 #define M_alloc mprotectMalloc
00099
00100 #endif
00101
00102 #ifdef TERMMALLOCDEBUG
00103 WORD **DebugHeap1, **DebugHeap2;
00104 #endif
00105
00106 /*
00107     #] Includes :
00108     #[ Streams :
00109         #[ LoadInputFile :
00110 */
00111
00112 UBYTE *LoadInputFile(UBYTE *filename, int type)
00113 {
00114     int handle;
00115     LONG filesize;
00116     UBYTE *buffer, *name = filename;
00117     POSITION scrpos;
00118     handle = LocateFile(&name,type);
00119     if ( handle < 0 ) return(0);
00120     PUTZERO(scrpos);
00121     SeekFile(handle,&scrpos,SEEK_END);
00122     TELLFILE(handle,&scrpos);
00123     filesize = BASEPOSITION(scrpos);
00124     PUTZERO(scrpos);
00125     SeekFile(handle,&scrpos,SEEK_SET);
00126     buffer = (UBYTE *)Malloca(filesize+2,"LoadInputFile");
00127     if ( ReadFile(handle,buffer,filesize) != filesize ) {
00128         Error1("Read error for file ",name);
00129         M_free(buffer,"LoadInputFile");
00130         if ( name != filename ) M_free(name,"FromLoadInputFile");
00131         CloseFile(handle);
00132         return(0);
00133     }
00134     CloseFile(handle);
00135     if ( type == PROCEDUREFILE || type == SETUPFILE ) {
00136         buffer[filesize] = '\n';
00137         buffer[filesize+1] = 0;
00138     }
00139     else {
00140         buffer[filesize] = 0;
00141     }
00142     if ( name != filename ) M_free(name,"FromLoadInputFile");
00143     return(buffer);
00144 }
00145
00146 /*
00147     #] LoadInputFile :
00148     #[ ReadFromStream :
00149 */
00150
00151 UBYTE ReadFromStream(STREAM *stream)
00152 {
00153     UBYTE c;
00154     POSITION scrpos;
00155 #ifdef WITHPIPE
00156     if ( stream->type == PIPESTREAM ) {
00157 #ifndef WITHMPI
00158         FILE *f;
00159         int cc;
00160         RWLOCKR(AM.handlelock);
00161         f = (FILE *) (filelist[stream->handle]);
00162         UNRWLOCK(AM.handlelock);
00163         cc = getc(f);
00164         if ( cc == EOF ) return(ENDOFSTREAM);
00165         c = (UBYTE)cc;
00166 #else
00167         if ( stream->pointer >= stream->top ) {
00168             /* The master reads the pipe and broadcasts it to the slaves. */
00169             LONG len;
00170             if ( PF.me == MASTER ) {
00171                 FILE *f;
00172                 UBYTE *p, *end;
00173                 RWLOCKR(AM.handlelock);
00174                 f = (FILE *) filelist[stream->handle];
00175                 UNRWLOCK(AM.handlelock);
00176                 p = stream->buffer;
00177                 end = stream->buffer + stream->buffersize;
00178                 while ( p < end ) {

```

```

00179         int cc = getc(f);
00180         if ( cc == EOF ) {
00181             break;
00182         }
00183         *p++ = (UBYTE)cc;
00184     }
00185     len = p - stream->buffer;
00186     PF_BroadcastNumber(len);
00187 }
00188 else {
00189     len = PF_BroadcastNumber(0);
00190 }
00191 if ( len > 0 ) {
00192     PF_Bcast(stream->buffer, len);
00193 }
00194 stream->pointer = stream->buffer;
00195 stream->inbuffer = len;
00196 stream->top = stream->buffer + stream->inbuffer;
00197 if ( stream->pointer == stream->top ) return ENDOFSTREAM;
00198 }
00199 c = (UBYTE)*stream->pointer++;
00200 #endif
00201 if ( stream->eqnum == 1 ) { stream->eqnum = 0; stream->linenumber++; }
00202 if ( c == LINEFEED ) stream->eqnum = 1;
00203 return(c);
00204 }
00205 #endif
00206 /*[14apr2004 mt]:*/
00207 #ifdef WITHEXTERNALCHANNEL
00208     if ( stream->type == EXTERNALCHANNELSTREAM ) {
00209         int cc;
00210         cc = getcFromExtChannel();
00211         /*[18may20006 mt]:*/
00212         /*if ( cc == EOF ) return(ENDOFSTREAM);*/
00213         if ( cc < 0 ) {
00214             if( cc == EOF )
00215                 return(ENDOFSTREAM);
00216             else{
00217                 Error0("No current external channel");
00218                 Terminate(-1);
00219             }
00220         }/*if ( cc < 0 )*/
00221         /*:[18may20006 mt]:*/
00222         c = (UBYTE)cc;
00223         if ( stream->eqnum == 1 ) { stream->eqnum = 0; stream->linenumber++; }
00224         if ( c == LINEFEED ) stream->eqnum = 1;
00225         return(c);
00226     }
00227 #endif /*ifdef WITHEXTERNALCHANNEL*/
00228 /*:[14apr2004 mt]:*/
00229     if ( stream->type == INPUTSTREAM ) {
00230         if ( stream->pointer < stream->top ) {
00231             c = *stream->pointer++;
00232         }
00233         else {
00234             if ( ReadFile(stream->handle,&c,1) != 1 ) {
00235                 return(ENDOFSTREAM);
00236             }
00237             if ( stream->fileposition == 0 ) {
00238                 if ( !stream->buffer ) {
00239                     stream->buffer = (UBYTE *)Malloc1(stream->buffer,"input stream buffer");
00240                     stream->pointer = stream->top = stream->buffer;
00241                 }
00242             }
00243             else {
00244                 if ( stream->top - stream->buffer >= stream->buffer ) {
00245                     LONG oldsize = stream->buffer;
00246                     DoubleBuffer((void**)&stream->buffer, (void**)&stream->top, sizeof(UBYTE), "double input stream buffer");
00247                     stream->buffer = stream->buffer;
00248                     stream->pointer = stream->top = stream->buffer + oldsize;
00249                 }
00250             }
00251             *stream->pointer = c;
00252             stream->pointer = ++stream->top;
00253             stream->inbuffer++;
00254         }
00255     }
00256     if ( stream->eqnum == 1 ) { stream->eqnum = 0; stream->linenumber++; }
00257     if ( c == LINEFEED ) stream->eqnum = 1;
00258     return(c);
00259 }
00260 if ( stream->pointer >= stream->top ) {
00261     if ( stream->type != FILESTREAM ) return(ENDOFSTREAM);
00262     if ( stream->fileposition != stream->bufferposition+stream->inbuffer ) {
00263         stream->fileposition = stream->bufferposition+stream->inbuffer;
00264         SETBASEPOSITION(scrpos, stream->fileposition);

```

```

00265         SeekFile(stream->handle, &scrpos, SEEK_SET);
00266     }
00267     stream->bufferposition = stream->fileposition;
00268     stream->inbuffer = ReadFile(stream->handle,
00269         stream->buffer, stream->buffersize);
00270     if ( stream->inbuffer <= 0 ) return(EOFSTREAM);
00271     stream->top = stream->buffer + stream->inbuffer;
00272     stream->pointer = stream->buffer;
00273     stream->fileposition = stream->bufferposition + stream->inbuffer;
00274 }
00275 if ( stream->eqnum == 1 ) { stream->eqnum = 0; stream->linenumber++; }
00276 c = *(stream->pointer)++;
00277 if ( c == LINEFEED ) stream->eqnum = 1;
00278 return(c);
00279 }
00280
00281 /*
00282     #] ReadFromStream :
00283     #[ GetFromStream :
00284 */
00285
00286 UBYTE GetFromStream(STREAM *stream)
00287 {
00288     UBYTE c1, c2;
00289     if ( stream->isnextchar > 0 ) {
00290         return(stream->nextchar[--stream->isnextchar]);
00291     }
00292     c1 = ReadFromStream(stream);
00293     if ( c1 == LINEFEED || c1 == CARRIAGERETURN ) {
00294         c2 = ReadFromStream(stream);
00295         if ( c2 == c1 || ( c2 != LINEFEED && c2 != CARRIAGERETURN ) ) {
00296             stream->isnextchar = 1;
00297             stream->nextchar[0] = c2;
00298         }
00299         return(LINEFEED);
00300     }
00301     else return(c1);
00302 }
00303
00304 /*
00305     #] GetFromStream :
00306     #[ LookInStream :
00307 */
00308
00309 UBYTE LookInStream(STREAM *stream)
00310 {
00311     UBYTE c = GetFromStream(stream);
00312     UngetFromStream(stream, c);
00313     return(c);
00314 }
00315
00316 /*
00317     #] LookInStream :
00318     #[ OpenStream :
00319 */
00320
00321 STREAM *OpenStream(UBYTE *name, int type, int prevarmode, int raiselow)
00322 {
00323     STREAM *stream;
00324     UBYTE *rhsofvariable, *s, *newname, c;
00325     POSITION scrpos;
00326     int handle, num;
00327     LONG filesize;
00328     switch ( type ) {
00329         case REVERSEFILESTREAM:
00330         case FILESTREAM:
00331     }
00332     /*
00333         Notice that FILESTREAM is only used for text files:
00334         The #include files and the main input file (.frm)
00335         Hence we do not worry about files longer than 2 Gbytes.
00336     */
00337     newname = name;
00338     handle = LocateFile(&newname, -1);
00339     if ( handle < 0 ) return(0);
00340     PUTZERO(scrpos);
00341     SeekFile(handle, &scrpos, SEEK_END);
00342     TELLFILE(handle, &scrpos);
00343     filesize = BASEPOSITION(scrpos);
00344     PUTZERO(scrpos);
00345     SeekFile(handle, &scrpos, SEEK_SET);
00346     if ( filesize > AM.MaxStreamSize && type == FILESTREAM )
00347         filesize = AM.MaxStreamSize;
00348     stream = CreateStream((UBYTE *) "filestream");
00349     /*
00350         The extra +1 in the Malloc1 is potentially needed in ReverseStatements!
00351     */
00352     stream->buffer = (UBYTE *) Malloc1(filesize+1, "name of input stream");

```

```

00352     stream->inbuffer = ReadFile(handle,stream->buffer,filesize);
00353     if ( type == REVERSEFILESTREAM ) {
00354         if ( ReverseStatements(stream) ) {
00355             M_free(stream->buffer,"name of input stream");
00356             return(0);
00357         }
00358     }
00359     stream->top = stream->buffer + stream->inbuffer;
00360     stream->pointer = stream->buffer;
00361     stream->handle = handle;
00362     stream->bufferSize = fileSize;
00363     stream->fileposition = stream->inbuffer;
00364     if ( newname != name ) stream->name = newname;
00365     else if ( name ) stream->name = strdup(name,"name of input stream");
00366     else
00367         stream->name = 0;
00368     stream->prevline = stream->linenumber = 1;
00369     stream->eqnum = 0;
00370     break;
00371 case PREVARSTREAM:
00372     if ( ( rhsOfVariable = GetPreVar(name,WITHERROR) ) == 0 ) return(0);
00373     stream = CreateStream((UBYTE *)"var-stream");
00374     stream->buffer = stream->pointer = s = rhsOfVariable;
00375     while ( *s ) s++;
00376     stream->top = s;
00377     stream->inbuffer = s - stream->buffer;
00378     stream->name = AC.CurrentStream->name;
00379     stream->linenumber = AC.CurrentStream->linenumber;
00380     stream->prevline = AC.CurrentStream->prevline;
00381     stream->eqnum = AC.CurrentStream->eqnum;
00382     stream->pname = strdup(name,"stream->pname");
00383     stream->olddelay = AP.AllowDelay;
00384     s = stream->pname; while ( *s ) s++;
00385     while ( s[-1] == '+' || s[-1] == '-' ) s--;
00386     *s = 0;
00387     UnsetAllowDelay();
00388     break;
00389 case DOLLARSTREAM:
00390     if ( ( num = GetDollar(name) ) < 0 ) {
00391         WORD numfac = 0;
00392         /*
00393          Here we have to test first whether we have $x[1], $x[0]
00394          or just an undefined $x.
00395          */
00396         s = name; while ( *s && *s != '[' ) s++;
00397         if ( *s == 0 ) return(0);
00398         c = *s; *s = 0;
00399         if ( ( num = GetDollar(name) ) < 0 ) return(0);
00400         *s = c;
00401         s++;
00402         if ( *s == 0 || FG.cTable[*s] != 1 || *s == ']' ) {
00403             MesPrint("@Illegal factor number for dollar variable");
00404             return(0);
00405         }
00406         while ( *s && FG.cTable[*s] == 1 ) {
00407             numfac = 10*numfac+*s---'0';
00408         }
00409         if ( *s != '[' || s[1] != 0 ) {
00410             MesPrint("@Illegal factor number for $ variable");
00411             return(0);
00412         }
00413         stream = CreateStream((UBYTE *)"dollar-stream");
00414         stream->buffer = stream->pointer = s = WriteDollarFactorToBuffer(num,numfac,1);
00415     }
00416     else {
00417         stream = CreateStream((UBYTE *)"dollar-stream");
00418         stream->buffer = stream->pointer = s = WriteDollarToBuffer(num,1);
00419     }
00420     while ( *s ) s++;
00421     stream->top = s;
00422     stream->inbuffer = s - stream->buffer;
00423     stream->name = AC.CurrentStream->name;
00424     stream->linenumber = AC.CurrentStream->linenumber;
00425     stream->prevline = AC.CurrentStream->prevline;
00426     stream->eqnum = AC.CurrentStream->eqnum;
00427     stream->pname = strdup(name,"stream->pname");
00428     s = stream->pname; while ( *s ) s++;
00429     while ( s[-1] == '+' || s[-1] == '-' ) s--;
00430     *s = 0;
00431     /* We 'stole' the buffer. Later we can free it. */
00432     AO.DollarOutSizeBuffer = 0;
00433     AO.DollarOutBuffer = 0;
00434     AO.DollarInOutBuffer = 0;
00435     break;
00436 case PREREADSTREAM:
00437 case PREREADSTREAM2:
00438 case PREREADSTREAM3:

```

```

00439         case PRECALCSTREAM:
00440             stream = CreateStream((UBYTE *)"calculator");
00441             stream->buffer = stream->pointer = s = name;
00442             while ( *s ) s++;
00443             stream->top = s;
00444             stream->inbuffer = s - stream->buffer;
00445             stream->name = AC.CurrentStream->name;
00446             stream->linenumber = AC.CurrentStream->linenumber;
00447             stream->prevline = AC.CurrentStream->prevline;
00448             stream->eqnum = 0;
00449             break;
00450 #ifndef WITHPIPE
00451         case PIPESTREAM:
00452             stream = CreateStream((UBYTE *)"pipe");
00453 #ifndef WITHMPI
00454             {
00455                 FILE *f;
00456                 if ( ( f = popen((char *)name,"r") ) == 0 ) {
00457                     Error0("@Cannot create pipe");
00458                 }
00459                 stream->handle = CreateHandle();
00460                 RWLOCKW(AM.handlelock);
00461                 filelist[stream->handle] = (FILES *)f;
00462                 UNRWLOCK(AM.handlelock);
00463             }
00464             stream->buffer = stream->top = 0;
00465             stream->inbuffer = 0;
00466 #else
00467             {
00468                 /* Only the master opens the pipe. */
00469                 FILE *f;
00470                 if ( PF.me == MASTER ) {
00471                     f = popen((char *)name, "r");
00472                     PF_BroadcastNumber(f == 0);
00473                     if ( f == 0 ) Error0("@Cannot create pipe");
00474                 }
00475                 else {
00476                     if ( PF_BroadcastNumber(0) ) Error0("@Cannot create pipe");
00477                     f = (FILE *)123; /* dummy */
00478                 }
00479                 stream->handle = CreateHandle();
00480                 RWLOCKW(AM.handlelock);
00481                 filelist[stream->handle] = (FILES *)f;
00482                 UNRWLOCK(AM.handlelock);
00483             }
00484             /* stream->buffer as a send/receive buffer. */
00485             stream->buffer = (UBYTE *)Malloc1(stream->buffer, "pipe buffer");
00486             stream->inbuffer = 0;
00487             stream->top = stream->buffer;
00488             stream->pointer = stream->buffer;
00489 #endif
00490             stream->name = strDup1((UBYTE *)"pipe","pipe");
00491             stream->prevline = stream->linenumber = 1;
00492             stream->eqnum = 0;
00493             break;
00494 #endif
00495 #endif
00496 /*[14apr2004 mt]:*/
00497 #ifndef WITHEXTERNALCHANNEL
00498         case EXTERNALCHANNELSTREAM:
00499             /*Block*/
00500             int n, *tmpn;
00501             if( (n=getCurrentExternalChannel()) == 0 )
00502                 Error0("@No current external channel");
00503             stream = CreateStream((UBYTE *)"externalchannel");
00504             stream->handle = CreateHandle();
00505             tmpn = (int *)Malloc1(sizeof(int),"external channel handle");
00506             *tmpn = n;
00507             RWLOCKW(AM.handlelock);
00508             filelist[stream->handle] = (FILES *)tmpn;
00509             UNRWLOCK(AM.handlelock);
00510             /*Block*/
00511             stream->buffer = stream->top = 0;
00512             stream->inbuffer = 0;
00513             stream->name = strDup1((UBYTE *)"externalchannel","externalchannel");
00514             stream->prevline = stream->linenumber = 1;
00515             stream->eqnum = 0;
00516             break;
00517 #endif /*[14apr2004 mt]:*/
00518 /*:*/
00519         case INPUTSTREAM:
00520             /*
00521              * Assume that "name" stores a file descriptor (UNIX) or a FILE
00522              * pointer (Windows). We don't close it automatically on closing
00523              * the INPUTSTREAM stream (e.g., for stdin).
00524              */
00525             stream = CreateStream((UBYTE *)"input stream");

```

```

00526         stream->handle = CreateHandle();
00527     {
00528         FILES *f = (FILES *)Malloca(sizeof(int),"input stream handle");
00529         /* NOTE: in both cases name=0 indicates stdin. */
00530 #ifdef UNIX
00531         f->descriptor = (int)(ssize_t)name;
00532 #else
00533         f = name ? (FILES *)name : stdin;
00534 #endif
00535         RWLOCKW(AM.handlelock);
00536         filelist[stream->handle] = f;
00537         UNRWLOCK(AM.handlelock);
00538     }
00539     stream->buffer = stream->pointer = stream->top = 0;
00540     stream->inbuffer = 0;
00541     stream->name = strdup((UBYTE *) (name ? "INPUT" : "STDIN"),"input stream name");
00542     stream->prevline = stream->linenumber = 1;
00543     stream->eqnum = 0;
00544     /*
00545      * fileposition == -1: default
00546      * fileposition == 0: cache the input
00547      * See also: ReadFromStream, TryFileSetups
00548      */
00549     stream->fileposition = -1;
00550     break;
00551 default:
00552     return(0);
00553 }
00554 stream->bufferposition = 0;
00555 stream->isnextchar = 0;
00556 stream->type = type;
00557 stream->previousNoShowInput = AC.NoShowInput;
00558 stream->afterwards = raiselow;
00559 if ( AC.CurrentStream ) stream->previous = AC.CurrentStream - AC.Streams;
00560 else stream->previous = -1;
00561 stream->FoldName = 0;
00562 if ( prevarmode == 0 ) stream->prevvars = -1;
00563 else if ( prevarmode > 0 ) stream->prevvars = NumPre;
00564 else if ( prevarmode < 0 ) stream->prevvars = -prevarmode-1;
00565 AC.CurrentStream = stream;
00566 if ( type == PREREADSTREAM || type == PREREADSTREAM3 || type == PRECALCSTREAM
00567     || type == DOLLARSTREAM ) AC.NoShowInput = 1;
00568 return(stream);
00569 }
00570
00571 /*
00572     #] OpenStream :
00573     #[ LocateFile :
00574 */
00575
00576 int LocateFile(UBYTE **name, int type)
00577 {
00578     int handle, namesize, i;
00579     UBYTE *s, *to, *u1, *u2, *newname, *indir;
00580     handle = OpenFile((char *) (*name));
00581     if ( handle >= 0 ) return(handle);
00582     if ( type == SETUPFILE && AM.SetupFile ) {
00583         handle = OpenFile((char *) (AM.SetupFile));
00584         if ( handle >= 0 ) return(handle);
00585         MesPrint("Could not open setup file %s", (char *) (AM.SetupFile));
00586     }
00587     namesize = 4; s = *name;
00588     while ( *s ) { s++; namesize++; }
00589     if ( type == SETUPFILE ) indir = AM.SetupDir;
00590     else indir = AM.IncDir;
00591     if ( indir ) {
00592
00593         s = indir; i = 0;
00594         while ( *s ) { s++; i++; }
00595         newname = (UBYTE *)Malloca(namesize+i,"LocateFile");
00596         s = indir; to = newname;
00597         while ( *s ) *to++ = *s++;
00598         if ( to > newname && to[-1] != SEPARATOR ) *to++ = SEPARATOR;
00599         s = *name;
00600         while ( *s ) *to++ = *s++;
00601         *to = 0;
00602         handle = OpenFile((char *) newname);
00603         if ( handle >= 0 ) {
00604             *name = newname;
00605             return(handle);
00606         }
00607         M_free(newname,"LocateFile, incdir/file");
00608     }
00609     if ( type == SETUPFILE ) {
00610         handle = OpenFile(setupfilename);
00611         if ( handle >= 0 ) return(handle);
00612         s = (UBYTE *)getenv("FORMSETUP");

```

```

00613         if ( s ) {
00614             handle = OpenFile((char *)s);
00615             if ( handle >= 0 ) return(handle);
00616             MesPrint("Could not open setup file %s",s);
00617         }
00618     }
00619     if ( type != SETUPFILE && AM.Path ) {
00620         ul = AM.Path;
00621         while ( *ul ) {
00622             u2 = ul; i = 0;
00623 #ifndef WINDOWS
00624             while ( *ul && *ul != ';' ) {
00625                 ul++; i++;
00626             }
00627 #else
00628             while ( *ul && *ul != ':' ) {
00629                 if ( *ul == '\\\' ) ul++;
00630                 ul++; i++;
00631             }
00632 #endif
00633             newname = (UBYTE *)Malloc1(namesize+i,"LocateFile");
00634             s = u2; to = newname;
00635             while ( s < ul ) {
00636 #ifndef WINDOWS
00637                 if ( *s == '\\\' ) s++;
00638 #endif
00639                 *to++ = *s++;
00640             }
00641             if ( to > newname && to[-1] != SEPARATOR ) *to++ = SEPARATOR;
00642             s = *name;
00643             while ( *s ) *to++ = *s++;
00644             *to = 0;
00645             handle = OpenFile((char *)newname);
00646             if ( handle >= 0 ) {
00647                 *name = newname;
00648                 return(handle);
00649             }
00650             M_free(newname,"LocateFile Path/file");
00651             if ( *ul ) ul++;
00652         }
00653     }
00654     if ( type != SETUPFILE && type >= -1 ) Error1("LocateFile: Cannot find file",*name);
00655     return(-1);
00656 }
00657
00658 /*
00659     #] LocateFile :
00660     #[ CloseStream :
00661 */
00662
00663 STREAM *CloseStream(STREAM *stream)
00664 {
00665     int newstr = stream->previous, sgn;
00666     UBYTE *t, numbuf[24];
00667     LONG x;
00668     if ( stream->FoldName ) {
00669         M_free(stream->FoldName,"stream->FoldName");
00670         stream->FoldName = 0;
00671     }
00672     if ( stream->type == FILESTREAM || stream->type == REVERSEFILESTREAM ) {
00673         CloseFile(stream->handle);
00674         if ( stream->buffer != 0 ) M_free(stream->buffer,"name of input stream");
00675         stream->buffer = 0;
00676     }
00677 #ifndef WITHPIPE
00678     else if ( stream->type == PIPESTREAM ) {
00679         RWLOCKW(AM.handlelock);
00680 #endif
00681     if ( PF.me == MASTER )
00682 #endif
00683         pclose((FILE *) (filelist[stream->handle]));
00684         filelist[stream->handle] = 0;
00685         numinfilelist--;
00686         UNRWLOCK(AM.handlelock);
00687 #ifndef WITHMPI
00688         if ( stream->buffer != 0 ) {
00689             M_free(stream->buffer, "pipe buffer");
00690             stream->buffer = 0;
00691         }
00692 #endif
00693     }
00694 #endif
00695 /*[14apr2004 mt]:*/
00696 #ifndef WITHEXTERNALCHANNEL
00697     else if ( stream->type == EXTERNALCHANNELSTREAM ) {
00698         int *tmpn;
00699         RWLOCKW(AM.handlelock);

```



```

00700         tmpn = (int *) (filelist[stream->handle]);
00701         filelist[stream->handle] = 0;
00702         numinfilelist--;
00703         UNRWLOCK(AM.handlelock);
00704         M_free(tmpn, "external channel handle");
00705     }
00706 #endif /*ifdef WITHEXTERNALCHANNEL*/
00707 /*:[14apr2004 mt]*/
00708     else if ( stream->type == INPUTSTREAM ) {
00709         FILES *f;
00710         RWLOCKW(AM.handlelock);
00711         f = filelist[stream->handle];
00712         filelist[stream->handle] = 0;
00713         numinfilelist--;
00714         UNRWLOCK(AM.handlelock);
00715         M_free(f, "input stream handle");
00716     }
00717     else if ( stream->type == PREVARSTREAM && (
00718 stream->afterwards == PRERAISEAFTER || stream->afterwards == PRELOWERAFTER ) ) {
00719         t = stream->buffer; x = 0; sgn = 1;
00720         while ( *t == '-' || *t == '+' ) {
00721             if ( *t == '-' ) sgn = -sgn;
00722             t++;
00723         }
00724         if ( FG.cTable[*t] == 1 ) {
00725             while ( *t && FG.cTable[*t] == 1 ) x = 10*x + *t++ - '0';
00726             if ( *t == 0 ) {
00727                 if ( stream->afterwards == PRERAISEAFTER ) x = sgn*x + 1;
00728                 else x = sgn*x - 1;
00729                 NumToStr(numbuf, x);
00730                 PutPreVar(stream->pname, numbuf, 0, 1);
00731             }
00732         }
00733     }
00734     else if ( stream->type == DOLLARSTREAM && (
00735 stream->afterwards == PRERAISEAFTER || stream->afterwards == PRELOWERAFTER ) ) {
00736         if ( stream->afterwards == PRERAISEAFTER ) x = 1;
00737         else x = -1;
00738         DollarRaiseLow(stream->pname, x);
00739         if ( stream->buffer ) M_free(stream->buffer, "stream->buffer");
00740         stream->buffer = 0;
00741     }
00742     else if ( stream->type == PRECALCSTREAM || stream->type == DOLLARSTREAM ) {
00743         if ( stream->buffer ) M_free(stream->buffer, "stream->buffer");
00744         stream->buffer = 0;
00745     }
00746     if ( stream->name && stream->type != PREVARSTREAM
00747 && stream->type != PREREADSTREAM && stream->type != PREREADSTREAM2 && stream->type !=
PREREADSTREAM3
&& stream->type != PRECALCSTREAM && stream->type != DOLLARSTREAM ) {
00748         M_free(stream->name, "stream->name");
00749     }
00750     stream->name = 0;
00751 /* if ( stream->type != FILESTREAM ) */
00752 AC.NoShowInput = stream->previousNoShowInput;
00753 stream->buffer = 0; /* To make sure we will not reuse it */
00754 stream->pointer = 0;
00755 /*
00756 Look whether we have to pop preprocessor variables.
00757 */
00758 if ( stream->prevars >= 0 ) {
00759     while ( NumPre > stream->prevars ) {
00760         NumPre--;
00761         M_free(PreVar[NumPre].name, "PreVar[NumPre].name");
00762         PreVar[NumPre].name = PreVar[NumPre].value = 0;
00763     }
00764 }
00765 if ( stream->type == PREVARSTREAM ) {
00766     AP.AllowDelay = stream->olddelay;
00767     ClearMacro(stream->pname);
00768     M_free(stream->pname, "stream->pname");
00769 }
00770 else if ( stream->type == DOLLARSTREAM ) {
00771     M_free(stream->pname, "stream->pname");
00772 }
00773 AC.NumStreams--;
00774 if ( newstr >= 0 ) return(AC.Streams + newstr);
00775 else return(0);
00776 }
00777 }
00778
00779 /*
00780 #] CloseStream :
00781 #[ CreateStream :
00782 */
00783
00784 STREAM *CreateStream(UBYTE *where)
00785 {

```

```

00786     STREAM *newstreams;
00787     int numnewstreams,i;
00788     int offset;
00789     if ( AC.NumStreams >= AC.MaxNumStreams ) {
00790         if ( AC.MaxNumStreams == 0 ) numnewstreams = 10;
00791         else numnewstreams = 2*AC.MaxNumStreams;
00792         newstreams = (STREAM *)Malloc1(sizeof(STREAM)*(numnewstreams+1),"CreateStream");
00793         if ( AC.MaxNumStreams > 0 ) {
00794             offset = AC.CurrentStream - AC.Streams;
00795             for ( i = 0; i < AC.MaxNumStreams; i++ ) {
00796                 newstreams[i] = AC.Streams[i];
00797             }
00798             AC.CurrentStream = newstreams + offset;
00799         }
00800         else newstreams[0].previous = -1;
00801         AC.MaxNumStreams = numnewstreams;
00802         if ( AC.Streams ) M_free(AC.Streams, (char *)where);
00803         AC.Streams = newstreams;
00804     }
00805     newstreams = AC.Streams+AC.NumStreams++;
00806     newstreams->name = 0;
00807     return(newstreams);
00808 }
00809
00810 /*
00811     #] CreateStream :
00812     #[ GetStreamPosition :
00813 */
00814
00815 LONG GetStreamPosition(STREAM *stream)
00816 {
00817     return(stream->bufferposition + ((LONG)stream->pointer-(LONG)stream->buffer));
00818 }
00819
00820 /*
00821     #] GetStreamPosition :
00822     #[ PositionStream :
00823 */
00824
00825 void PositionStream(STREAM *stream, LONG position)
00826 {
00827     POSITION scrpos;
00828     if ( position >= stream->bufferposition
00829         && position < stream->bufferposition + stream->inbuffer ) {
00830         stream->pointer = stream->buffer + (position-stream->bufferposition);
00831     }
00832     else if ( stream->type == FILESTREAM ) {
00833         SETBASEPOSITION(scrpos,position);
00834         SeekFile(stream->handle,&scrpos,SEEK_SET);
00835         stream->inbuffer = ReadFile(stream->handle,stream->buffer,stream->buffersize);
00836         stream->pointer = stream->buffer;
00837         stream->top = stream->buffer + stream->inbuffer;
00838         stream->bufferposition = position;
00839         stream->fileposition = position + stream->inbuffer;
00840         stream->isnextchar = 0;
00841     }
00842     else {
00843         Error0("Illegal position for stream");
00844         Terminate(-1);
00845     }
00846 }
00847
00848 /*
00849     #] PositionStream :
00850     #[ ReverseStatements :
00851
00852     Reverses the order of the statements in the buffer.
00853     We allocate an extra buffer and copy a bit to and from.
00854     Note that there are some nasties that cannot be resolved.
00855 */
00856
00857 int ReverseStatements(STREAM *stream)
00858 {
00859     UBYTE *spare = (UBYTE *)Malloc1((stream->inbuffer+1)*sizeof(UBYTE),"Reverse copy");
00860     UBYTE *top = stream->buffer + stream->inbuffer, *in, *s, *ss, *out;
00861     out = spare+stream->inbuffer+1;
00862     in = stream->buffer;
00863     while ( in < top ) {
00864         s = in;
00865         if ( *s == AP.ComChar ) {
00866             tocol:;
00867             for(;;) {
00868                 if ( s == top ) { *--out = '\n'; break; }
00869                 if ( *s == '\\\\' ) {
00870                     s++;
00871                     if ( s >= top ) { /* This is an error! */
00872                         irrend:;
00873                         MesPrint("@Irregular end of reverse include file.");

```

```

00873         return(1);
00874     }
00875 }
00876 else if ( *s == '\n' ) {
00877     s++; ss = s;
00878     while ( ss > in ) *--out = *--ss;
00879     in = s;
00880     if ( out[0] == AP.ComChar && ss+6 < s && out[3] == '#' ) {
00881 /*
00882         For folds we have to exchange begin and end
00883 */
00884         if ( out[4] == '[' ) out[4] = ']';
00885         else if ( out[4] == ']' ) out[4] = '[';
00886     }
00887     break;
00888 }
00889     s++;
00890 }
00891 continue;
00892 }
00893 while ( s < top && ( *s == ' ' || *s == '\t' ) ) s++;
00894 if ( *s == '#' ) { /* preprocessor instruction */
00895     goto toeol; /* read to end of line */
00896 }
00897 if ( *s == '.' ) { /* end-of-module instruction */
00898     goto toeol; /* read to end of line */
00899 }
00900 /*
00901 Here we have a regular statement. In principle we scan to ; and its \n
00902 but there are special cases.
00903 1: ; inside a string (in print ".....;")
00904 2: multiple statements on one line.
00905 3: ; + commentary after some blanks.
00906 4: `var' can cause problems.....
00907 */
00908 while ( s < top ) {
00909     if ( *s == ';' ) {
00910         s++;
00911         while ( s < top && ( *s == ' ' || *s == '\t' ) ) s++;
00912         while ( s < top && *s == '\n' ) s++;
00913         if ( s >= top && s[-1] != '\n' ) *s++ = '\n';
00914         ss = s;
00915         while ( ss > in ) *--out = *--ss;
00916         in = s;
00917         break;
00918     }
00919     else if ( *s == '"' ) {
00920         s++;
00921         while ( s < top ) {
00922             if ( *s == '"' ) break;
00923             if ( *s == '\\' ) { s++; }
00924             s++;
00925         }
00926         if ( s >= top ) goto irrend;
00927     }
00928     else if ( *s == '\\' ) {
00929         s++;
00930         if ( s >= top ) goto irrend;
00931     }
00932     s++;
00933 }
00934 if ( in < top ) { /* Like blank lines at the end */
00935     if ( s >= top && s[-1] != '\n' ) *s++ = '\n';
00936     ss = s;
00937     while ( ss > in ) *--out = *--ss;
00938     in = s;
00939 }
00940 }
00941 if ( out == spare ) stream->inbuffer++;
00942 if ( out > spare+1 ) {
00943     MesPrint("@Internal error in #reverseinclude instruction.");
00944     return(1);
00945 }
00946 memcpy((void *) (stream->buffer), (void *) out, (size_t) (stream->inbuffer*sizeof(UBYTE)));
00947 M_free(spare, "Reverse copy");
00948 return(0);
00949 }
00950
00951 /*
00952     #] ReverseStatements :
00953     #] Streams :
00954     #[ Files :
00955     #[ StartFiles :
00956 */
00957 void StartFiles(void)
00958 {

```

```
00960     int i = CreateHandle();
00961     filelist[i] = Ustdout;
00962     AM.Stdout = i;
00963     AC.StoreHandle = -1;
00964     AC.LogHandle = -1;
00965 #ifndef WITHPTHREADS
00966     AR.Fscr[0].handle = -1;
00967     AR.Fscr[1].handle = -1;
00968     AR.Fscr[2].handle = -1;
00969     AR.FoStage4[0].handle = -1;
00970     AR.FoStage4[1].handle = -1;
00971     AR.infile = &(AR.Fscr[0]);
00972     AR.outfile = &(AR.Fscr[1]);
00973     AR.hidefile = &(AR.Fscr[2]);
00974     AR.StoreData.Handle = -1;
00975 #endif
00976     AC.Streams = 0;
00977     AC.MaxNumStreams = 0;
00978 }
00979
00980 /*
00981     #] StartFiles :
00982     #[ OpenFile :
00983 */
00984
00985 int OpenFile(char *name)
00986 {
00987     FILES *f;
00988     int i;
00989
00990     if ( ( f = Uopen(name,"rb") ) == 0 ) return(-1);
00991     /* Usetbuf(f,0); */
00992     i = CreateHandle();
00993     RWLOCKW(AM.handlelock);
00994     filelist[i] = f;
00995     UNRWLOCK(AM.handlelock);
00996     return(i);
00997 }
00998
00999 /*
01000     #] OpenFile :
01001     #[ OpenAddFile :
01002 */
01003
01004 int OpenAddFile(char *name)
01005 {
01006     FILES *f;
01007     int i;
01008     POSITION scrpos;
01009     if ( ( f = Uopen(name,"a+b") ) == 0 ) return(-1);
01010     /* Usetbuf(f,0); */
01011     i = CreateHandle();
01012     RWLOCKW(AM.handlelock);
01013     filelist[i] = f;
01014     UNRWLOCK(AM.handlelock);
01015     TELLFILE(i,&scrpos);
01016     SeekFile(i,&scrpos,SEEK_SET);
01017     return(i);
01018 }
01019
01020 /*
01021     #] OpenAddFile :
01022     #[ ReOpenFile :
01023 */
01024
01025 int ReOpenFile(char *name)
01026 {
01027     FILES *f;
01028     int i;
01029     POSITION scrpos;
01030     if ( ( f = Uopen(name,"r+b") ) == 0 ) return(-1);
01031     i = CreateHandle();
01032     RWLOCKW(AM.handlelock);
01033     filelist[i] = f;
01034     UNRWLOCK(AM.handlelock);
01035     TELLFILE(i,&scrpos);
01036     SeekFile(i,&scrpos,SEEK_SET);
01037     return(i);
01038 }
01039
01040 /*
01041     #] ReOpenFile :
01042     #[ CreateFile :
01043 */
01044
01045 int CreateFile(char *name)
01046 {
```

```

01047     FILES *f;
01048     int i;
01049     if ( ( f = Uopen(name,"w+b") ) == 0 ) return(-1);
01050     i = CreateHandle();
01051     RWLOCKW(AM.handlelock);
01052     filelist[i] = f;
01053     UNRWLOCK(AM.handlelock);
01054     return(i);
01055 }
01056
01057 /*
01058     #] CreateFile :
01059     #[ CreateLogFile :
01060 */
01061
01062 int CreateLogFile(char *name)
01063 {
01064     FILES *f;
01065     int i;
01066     if ( ( f = Uopen(name,"w+b") ) == 0 ) return(-1);
01067     Usetbuf(f,0);
01068     i = CreateHandle();
01069     RWLOCKW(AM.handlelock);
01070     filelist[i] = f;
01071     UNRWLOCK(AM.handlelock);
01072     return(i);
01073 }
01074
01075 /*
01076     #] CreateLogFile :
01077     #[ CloseFile :
01078 */
01079
01080 void CloseFile(int handle)
01081 {
01082     if ( handle >= 0 ) {
01083         FILES *f; /* we need this variable to be thread-safe */
01084         RWLOCKW(AM.handlelock);
01085         f = filelist[handle];
01086         filelist[handle] = 0;
01087         numinfilelist--;
01088         UNRWLOCK(AM.handlelock);
01089         Uclose(f);
01090     }
01091 }
01092
01093 /*
01094     #] CloseFile :
01095     #[ CopyFile :
01096 */
01097
01103 int CopyFile(char *source, char *dest)
01104 {
01105     #define COPYFILEBUFSIZE 40960L
01106     FILE *in, *out;
01107     size_t countin, countout, sumcount;
01108     char *buffer = NULL;
01109
01110     sumcount = (AM.S0->LargeSize+AM.S0->SmallEsize)*sizeof(WORD);
01111     if ( sumcount <= COPYFILEBUFSIZE ) {
01112         sumcount = COPYFILEBUFSIZE;
01113         buffer = (char*)Malloc1(sumcount, "file copy buffer");
01114     }
01115     else {
01116         buffer = (char *) (AM.S0->lBuffer);
01117     }
01118
01119     in = fopen(source, "rb");
01120     if ( in == NULL ) {
01121         perror("CopyFile: ");
01122         return(1);
01123     }
01124     out = fopen(dest, "wb");
01125     if ( out == NULL ) {
01126         perror("CopyFile: ");
01127         return(2);
01128     }
01129
01130     while ( !feof(in) ) {
01131         countin = fread(buffer, 1, sumcount, in);
01132         if ( countin != sumcount ) {
01133             if ( ferror(in) ) {
01134                 perror("CopyFile: ");
01135                 return(3);
01136             }
01137         }
01138         countout = fwrite(buffer, 1, countin, out);

```

```

01139         if ( countin != countout ) {
01140             perror("CopyFile: ");
01141             return(4);
01142         }
01143     }
01144     fclose(in);
01145     fclose(out);
01146     if ( sumcount <= COPYFILEBUFSIZE ) {
01147         M_free(buffer, "file copy buffer");
01148     }
01149     return(0);
01150 }
01151 }
01152
01153 /*
01154     #] CopyFile :
01155     #[ CreateHandle :
01156
01157     We need a lock here.
01158     Problem: the same lock is needed inside Malloc1 and M_free which
01159     is used in DoubleList when we use MALLOCDEBUG
01160
01161     Conclusion: MALLOCDEBUG will have to be a bit unsafe
01162 */
01163
01164 int CreateHandle(void)
01165 {
01166     int i, j;
01167     #ifndef MALLOCDEBUG
01168         RWLOCKW(AM.handlelock);
01169     #endif
01170     if ( filelistsize == 0 ) {
01171         filelistsize = 10;
01172         filelist = (FILES **)Malloc1(sizeof(FILES *)*filelistsize,"file handle");
01173         for ( j = 0; j < filelistsize; j++ ) filelist[j] = 0;
01174         numinfilelist = 1;
01175         i = 0;
01176     }
01177     else if ( numinfilelist >= filelistsize ) {
01178         void **fl = (void **)filelist;
01179         i = filelistsize;
01180         if ( DoubleList((void ***)&fl,&filelistsize,(int)sizeof(FILES *),
01181             "list of open files") != 0 ) Terminate(-1);
01182         filelist = (FILES **)fl;
01183         for ( j = i; j < filelistsize; j++ ) filelist[j] = 0;
01184         numinfilelist = i + 1;
01185     }
01186     else {
01187         i = filelistsize;
01188         for ( j = 0; j < filelistsize; j++ ) {
01189             if ( filelist[j] == 0 ) { i = j; break; }
01190         }
01191         numinfilelist++;
01192     }
01193     filelist[i] = (FILES *) (filelist); /* Just for now to not get into problems */
01194 /*
01195     The next code is not needed when we use open.
01196     It may be needed when we use fopen.
01197     fopen is used in minos.c without this central administration.
01198 */
01199     if ( numinfilelist > MAX_OPEN_FILES ) {
01200         #ifndef MALLOCDEBUG
01201             UNRWLOCK(AM.handlelock);
01202         #endif
01203         MesPrint("More than %d open files",MAX_OPEN_FILES);
01204         Error0("System limit. This limit is not due to FORM!");
01205     }
01206     else {
01207         #ifndef MALLOCDEBUG
01208             UNRWLOCK(AM.handlelock);
01209         #endif
01210     }
01211     return(i);
01212 }
01213
01214 /*
01215     #] CreateHandle :
01216     #[ ReadFile :
01217 */
01218
01219 LONG ReadFile(int handle, UBYTE *buffer, LONG size)
01220 {
01221     LONG inbuf = 0, r;
01222     FILES *f;
01223     char *b;
01224     b = (char *)buffer;
01225     for(;;) { /* Gotta do difficult because of VMS! */

```

```

01226         RWLOCKR(AM.handlelock);
01227         f = filelist[handle];
01228         UNRWLOCK(AM.handlelock);
01229 #ifdef WITHSTATS
01230         numreads++;
01231 #endif
01232         r = Uread(b,1,size,f);
01233         if ( r < 0 ) return(r);
01234         if ( r == 0 ) return(inbuf);
01235         inbuf += r;
01236         if ( r == size ) return(inbuf);
01237         if ( r > size ) return(-1);
01238         size -= r;
01239         b += r;
01240     }
01241 }
01242
01243 /*
01244     #] ReadFile :
01245     #[ ReadPosFile :
01246
01247     Gets words from a file(handle).
01248     First tries to get the information from the buffers.
01249     Reads a file at a position. Updates the position.
01250     Places a lock in the case of multithreading.
01251     Exists for multiple reading from the same file.
01252     size is the number of WORDs to read!!!!
01253
01254     We may need some strategy in the caching. This routine is used from
01255     GetOneTerm only. The problem is when it reads brackets and the
01256     brackets are read backwards. This is very uneconomical because
01257     each time it may read a large buffer.
01258     On the other hand, reading piece by piece in GetOneTerm takes
01259     much overhead as well.
01260     Two strategies come to mind:
01261     1: keep things as they are but limit the size of the buffers.
01262     2: have the position of 'pos' at about 1/3 of the buffer.
01263         this is of course guess work.
01264     Currently we have implemented the first method by creating the
01265     setup parameter threadscratchsize with the default value 100K.
01266     In the test program much bigger values gave a slower program.
01267 */
01268
01269 LONG ReadPosFile(PHEAD FILEHANDLE *fi, UBYTE *buffer, LONG size, POSITION *pos)
01270 {
01271     GETBIDENTITY
01272     LONG i, retval = 0;
01273     WORD *b = (WORD *)buffer, *t;
01274
01275     if ( fi->handle < 0 ) {
01276         fi->POfill = (WORD *) ((UBYTE *) (fi->PObuffer) + BASEPOSITION(*pos));
01277         t = fi->POfill;
01278         while ( size > 0 && fi->POfill < fi->POfull ) { *b++ = *t++; size--; }
01279     }
01280     else {
01281         if ( ISLESSPOS(*pos,fi->POposition) || ISGEPOSINC(*pos,fi->POposition,
01282             ((UBYTE *) (fi->POfull)-(UBYTE *) (fi->PObuffer))) ) {
01283             /*
01284                 The start is not inside the buffer. Fill the buffer.
01285             */
01286
01287             fi->POposition = *pos;
01288             LOCK(AS.inputslock);
01289             SeekFile(fi->handle,pos,SEEK_SET);
01290             retval = ReadFile(fi->handle,(UBYTE *) (fi->PObuffer),fi->POsize);
01291             UNLOCK(AS.inputslock);
01292             fi->POfull = fi->PObuffer+retval/sizeof(WORD);
01293             fi->POfill = fi->PObuffer;
01294             if ( fi != AR.hidefile ) AR.InInBuf = retval/sizeof(WORD);
01295             else AR.InHiBuf = retval/sizeof(WORD);
01296         }
01297         else {
01298             fi->POfill = (WORD *) ((UBYTE *) (fi->PObuffer) + DIFBASE(*pos,fi->POposition));
01299         }
01300         if ( fi->POfill + size <= fi->POfull ) {
01301             t = fi->POfill;
01302             while ( size > 0 ) { *b++ = *t++; size--; }
01303         }
01304         else {
01305             for (;;) {
01306                 i = fi->POfull - fi->POfill; t = fi->POfill;
01307                 if ( i > size ) i = size;
01308                 size -= i;
01309                 while ( --i >= 0 ) *b++ = *t++;
01310                 if ( size == 0 ) break;
01311                 ADDPOS(fi->POposition,(UBYTE *) (fi->POfull)-(UBYTE *) (fi->PObuffer));
01312                 LOCK(AS.inputslock);

```

```

01313         SeekFile(fi->handle,&(fi->PPosition),SEEK_SET);
01314         retval = ReadFile(fi->handle,(UBYTE *) (fi->Pbuffer),fi->Psize);
01315         UNLOCK(AS.inputslock);
01316         fi->POfull = fi->Pbuffer+retval/sizeof(WORD);
01317         fi->POfill = fi->Pbuffer;
01318         if ( fi != AR.hidefile ) AR.InInBuf = retval/sizeof(WORD);
01319         else AR.InHiBuf = retval/sizeof(WORD);
01320         if ( retval == 0 ) { t = fi->POfill; break; }
01321     }
01322 }
01323 }
01324     retval = (UBYTE *)b - buffer;
01325     fi->POfill = t;
01326     ADDPOS(*pos,retval);
01327     return(retval);
01328 }
01329
01330 /*
01331     #] ReadPosFile :
01332     #[ WriteFile :
01333 */
01334
01335 LONG WriteFileToFile(int handle, UBYTE *buffer, LONG size)
01336 {
01337     FILES *f;
01338     LONG retval, totalwritten = 0, stilltowrite;
01339     RWLOCKR(AM.handlelock);
01340     f = filelist[handle];
01341     UNRWLOCK(AM.handlelock);
01342     while ( totalwritten < size ) {
01343         stilltowrite = size - totalwritten;
01344         #ifdef WITHSTATS
01345             numwrites++;
01346         #endif
01347         retval = Uwrite((char *)buffer+totalwritten,1,stilltowrite,f);
01348         if ( retval < 0 ) return(retval);
01349         if ( retval == 0 ) return(totalwritten);
01350         totalwritten += retval;
01351     }
01352     /*
01353     if ( handle == AC.LogHandle || handle == ERROROUT ) FlushFile(handle);
01354     */
01355     return(totalwritten);
01356 }
01357 #ifndef WITHMPI
01358 /*[17nov2005]:*/
01359 WRITEFILE WriteFile = &WriteFileToFile;
01360 /*
01361 LONG (*WriteFile)(int handle, UBYTE *buffer, LONG size) = &WriteFileToFile;
01362 */
01363 /*:[17nov2005]:*/
01364 #else
01365 WRITEFILE WriteFile = &PF_WriteFileToFile;
01366 #endif
01367
01368 /*
01369     #] WriteFile :
01370     #[ SeekFile :
01371 */
01372
01373 void SeekFile(int handle, POSITION *offset, int origin)
01374 {
01375     FILES *f;
01376     RWLOCKR(AM.handlelock);
01377     f = filelist[handle];
01378     UNRWLOCK(AM.handlelock);
01379     #ifdef WITHSTATS
01380         numseeks++;
01381     #endif
01382     if ( origin == SEEK_SET ) {
01383         Useek(f,BASEPOSITION(*offset),origin);
01384         SETBASEPOSITION(*offset,(Utell(f)));
01385         return;
01386     }
01387     else if ( origin == SEEK_END ) {
01388         Useek(f,0,origin);
01389     }
01390     SETBASEPOSITION(*offset,(Utell(f)));
01391 }
01392
01393 /*
01394     #] SeekFile :
01395     #[ TellFile :
01396 */
01397
01398 LONG TellFile(int handle)
01399 {

```



```

01400     POSITION pos;
01401     TELLFILE(handle,&pos);
01402 #ifdef WITHSTATS
01403     numseeks++;
01404 #endif
01405     return(BASEPOSITION(pos));
01406 }
01407
01408 void TELLFILE(int handle, POSITION *position)
01409 {
01410     FILES *f;
01411     RWLOCKR(AM.handlelock);
01412     f = filelist[handle];
01413     UNRWLOCK(AM.handlelock);
01414     SETBASEPOSITION(*position, (Utell(f)));
01415 }
01416
01417 /*
01418     #] TellFile :
01419     #[ FlushFile :
01420 */
01421
01422 void FlushFile(int handle)
01423 {
01424     FILES *f;
01425     RWLOCKR(AM.handlelock);
01426     f = filelist[handle];
01427     UNRWLOCK(AM.handlelock);
01428     Uflush(f);
01429 }
01430
01431 /*
01432     #] FlushFile :
01433     #[ GetPosFile :
01434 */
01435
01436 int GetPosFile(int handle, fpos_t *pospointer)
01437 {
01438     FILES *f;
01439     RWLOCKR(AM.handlelock);
01440     f = filelist[handle];
01441     UNRWLOCK(AM.handlelock);
01442     return(Ugetpos(f,pospointer));
01443 }
01444
01445 /*
01446     #] GetPosFile :
01447     #[ SetPosFile :
01448 */
01449
01450 int SetPosFile(int handle, fpos_t *pospointer)
01451 {
01452     FILES *f;
01453     RWLOCKR(AM.handlelock);
01454     f = filelist[handle];
01455     UNRWLOCK(AM.handlelock);
01456     return(Usetpos(f,(fpos_t *)pospointer));
01457 }
01458
01459 /*
01460     #] SetPosFile :
01461     #[ SynchFile :
01462
01463     It may be that when we use many sort files at the same time there
01464     is a big traffic jam in the cache. This routine is experimental,
01465     just to see whether this improves the situation.
01466     It could also be that the internal disk of the Quad opteron norma
01467     is very slow.
01468 */
01469
01470 void SynchFile(int handle)
01471 {
01472     FILES *f;
01473     if ( handle >= 0 ) {
01474         RWLOCKR(AM.handlelock);
01475         f = filelist[handle];
01476         UNRWLOCK(AM.handlelock);
01477         Usync(f);
01478     }
01479 }
01480
01481 /*
01482     #] SynchFile :
01483     #[ TruncateFile :
01484
01485     It may be that when we use many sort files at the same time there
01486     is a big traffic jam in the cache. This routine is experimental,

```

```

01487         just to see whether this improves the situation.
01488         It could also be that the internal disk of the Quad opteron norma
01489         is very slow.
01490 */
01491
01492 void TruncateFile(int handle)
01493 {
01494     FILES *f;
01495     if ( handle >= 0 ) {
01496         RWLOCKR(AM.handlelock);
01497         f = filelist[handle];
01498         UNRWLOCK(AM.handlelock);
01499         Utruncate(f);
01500     }
01501 }
01502
01503 /*
01504     #] TruncateFile :
01505     #[ GetChannel :
01506
01507     Checks whether we have this file already. If so, we return its
01508     handle. If not and mode == 0, we open the file first and add it
01509     to the buffers.
01510 */
01511
01512 int GetChannel(char *name,int mode)
01513 {
01514     CHANNEL *ch;
01515     int i;
01516     FILES *f;
01517     for ( i = 0; i < NumOutputChannels; i++ ) {
01518         if ( channels[i].name == 0 ) continue;
01519         if ( StrCmp((UBYTE *)name,(UBYTE *) (channels[i].name)) == 0 ) return(channels[i].handle);
01520     }
01521     if ( mode == 1 ) {
01522         MesPrint("&File %s in print statement is not open",name);
01523         MesPrint("    You should open it first with a #write or #append instruction");
01524         return(-1);
01525     }
01526     for ( i = 0; i < NumOutputChannels; i++ ) {
01527         if ( channels[i].name == 0 ) break;
01528     }
01529     if ( i < NumOutputChannels ) { ch = &(channels[i]); }
01530     else { ch = (CHANNEL *)FromList(&AC.Channellist); }
01531     ch->name = (char *)strDup1((UBYTE *)name,"name of channel");
01532     ch->handle = CreateFile(name);
01533     RWLOCKR(AM.handlelock);
01534     f = filelist[ch->handle];
01535     UNRWLOCK(AM.handlelock);
01536     Usetbuf(f,0); /* We turn the buffer off!!!!!!*/
01537     return(ch->handle);
01538 }
01539
01540 /*
01541     #] GetChannel :
01542     #[ GetAppendChannel :
01543
01544     Checks whether we have this file already. If so, we return its
01545     handle. If not, we open the file first and add it to the buffers.
01546 */
01547
01548 int GetAppendChannel(char *name)
01549 {
01550     CHANNEL *ch;
01551     int i;
01552     FILES *f;
01553     for ( i = 0; i < NumOutputChannels; i++ ) {
01554         if ( channels[i].name == 0 ) continue;
01555         if ( StrCmp((UBYTE *)name,(UBYTE *) (channels[i].name)) == 0 ) return(channels[i].handle);
01556     }
01557     for ( i = 0; i < NumOutputChannels; i++ ) {
01558         if ( channels[i].name == 0 ) break;
01559     }
01560     if ( i < NumOutputChannels ) { ch = &(channels[i]); }
01561     else { ch = (CHANNEL *)FromList(&AC.Channellist); }
01562     ch->name = (char *)strDup1((UBYTE *)name,"name of channel");
01563     ch->handle = OpenAddFile(name);
01564     RWLOCKR(AM.handlelock);
01565     f = filelist[ch->handle];
01566     UNRWLOCK(AM.handlelock);
01567     Usetbuf(f,0); /* We turn the buffer off!!!!!!*/
01568     return(ch->handle);
01569 }
01570
01571 /*
01572     #] GetAppendChannel :
01573     #[ CloseChannel :

```

```

01574
01575     Checks whether we have this file already. If so, we close it.
01576 */
01577
01578 int CloseChannel(char *name)
01579 {
01580     int i;
01581     for ( i = 0; i < NumOutputChannels; i++ ) {
01582         if ( channels[i].name == 0 ) continue;
01583         if ( channels[i].name[0] == 0 ) continue;
01584         if ( StrCmp((UBYTE *)name,(UBYTE *) (channels[i].name)) == 0 ) {
01585             CloseFile(channels[i].handle);
01586             M_free(channels[i].name,"CloseChannel");
01587             channels[i].name = 0;
01588             return(0);
01589         }
01590     }
01591     return(0);
01592 }
01593
01594 /*
01595     #] CloseChannel :
01596     #[ UpdateMaxSize :
01597
01598     Updates the maximum size of the combined input/output/hide scratch
01599     files, the sort files and the .str file.
01600     The result becomes only visible with either
01601         ON totalsize;
01602         #: totalsize ON;
01603     or the -T in the command tail.
01604
01605     To be called, whenever a file is closed/removed or truncated to zero.
01606
01607     We have no provisions yet for expressions that remain inside the
01608     small or large buffer during the sort. The space they use there is
01609     currently ignored.
01610 */
01611
01612 void UpdateMaxSize(void)
01613 {
01614     POSITION position, sumsize;
01615     int i;
01616     FILEHANDLE *scr;
01617 #ifdef WITHMPI
01618     /* Currently, it works only on the master. The sort files on the slaves
01619      * are ignored. (TU 11 Oct 2011) */
01620     if ( PF.me != MASTER ) return;
01621 #endif
01622     PUTZERO(sumsize);
01623     if ( AM.PrintTotalSize ) {
01624         /*
01625             First the three scratch files
01626         */
01627 #ifdef WITHPTHREADS
01628         scr = AB[0]->R.Fscr;
01629 #else
01630         scr = AR.Fscr;
01631 #endif
01632         for ( i = 0; i <=2; i++ ) {
01633             if ( scr[i].handle < 0 ) {
01634                 SETBASEPOSITION(position,(scr[i].POfull-scr[i].PObuffer)*sizeof(WORD));
01635             }
01636             else {
01637                 position = scr[i].filesize;
01638             }
01639             ADD2POS(sumsize,position);
01640         }
01641         /*
01642             Now the sort file(s)
01643         */
01644 #ifdef WITHPTHREADS
01645         {
01646             int j;
01647             ALLPRIVATES *B;
01648             for ( j = 0; j < AM.totalnumberofthreads; j++ ) {
01649                 B = AB[j];
01650                 if ( AT.SS && AT.SS->file.handle >= 0 ) {
01651                     position = AT.SS->file.filesize;
01652                 }
01653                 MLOCK(ErrorMessageLock);
01654                 MesPrint("%d: %10p",j,&(AT.SS->file.filesize));
01655                 MUNLOCK(ErrorMessageLock);
01656                 /*
01657                     ADD2POS(sumsize,position);
01658                 */
01659                 if ( AR.FoStage4[0].handle >= 0 ) {
01660                     position = AR.FoStage4[0].filesize;

```

```

01661             ADD2POS (sumsize,position);
01662         }
01663     }
01664 }
01665 #else
01666     if ( AT.SS && AT.SS->file.handle >= 0 ) {
01667         position = AT.SS->file.filesize;
01668         ADD2POS (sumsize,position);
01669     }
01670     if ( AR.FoStage4[0].handle >= 0 ) {
01671         position = AR.FoStage4[0].filesize;
01672         ADD2POS (sumsize,position);
01673     }
01674 #endif
01675 /*
01676     And of course the str file.
01677 */
01678     ADD2POS (sumsize,AC.StoreFileSize);
01679 /*
01680     Finally the test whether it is bigger
01681 */
01682     if ( ISLESSPOS (AS.MaxExprSize,sumsize) ) {
01683 #ifdef WITHPTHREADS
01684         LOCK (AS.MaxExprSizeLock);
01685         if ( ISLESSPOS (AS.MaxExprSize,sumsize) ) AS.MaxExprSize = sumsize;
01686         UNLOCK (AS.MaxExprSizeLock);
01687     #else
01688         AS.MaxExprSize = sumsize;
01689     #endif
01690     }
01691 }
01692 return;
01693 }
01694
01695 /*
01696     #] UpdateMaxSize :
01697     #] Files :
01698     #[ Strings :
01699     #[ StrCmp :
01700 */
01701
01702 int StrCmp (UBYTE *s1, UBYTE *s2)
01703 {
01704     while ( *s1 && *s1 == *s2 ) { s1++; s2++; }
01705     return ((int)*s1-(int)*s2);
01706 }
01707
01708 /*
01709     #] StrCmp :
01710     #[ StrICmp :
01711 */
01712
01713 int StrICmp (UBYTE *s1, UBYTE *s2)
01714 {
01715     while ( *s1 && tolower(*s1) == tolower(*s2) ) { s1++; s2++; }
01716     return ((int)tolower(*s1)-(int)tolower(*s2));
01717 }
01718
01719 /*
01720     #] StrICmp :
01721     #[ StrHICmp :
01722 */
01723
01724 int StrHICmp (UBYTE *s1, UBYTE *s2)
01725 {
01726     while ( *s1 && tolower(*s1) == *s2 ) { s1++; s2++; }
01727     return ((int)tolower(*s1)-(int)(*s2));
01728 }
01729
01730 /*
01731     #] StrHICmp :
01732     #[ StrICont :
01733 */
01734
01735 int StrICont (UBYTE *s1, UBYTE *s2)
01736 {
01737     while ( *s1 && tolower(*s1) == tolower(*s2) ) { s1++; s2++; }
01738     if ( *s1 == 0 ) return(0);
01739     return ((int)tolower(*s1)-(int)tolower(*s2));
01740 }
01741
01742 /*
01743     #] StrICont :
01744     #[ CmpArray :
01745 */
01746
01747 int CmpArray (WORD *t1, WORD *t2, WORD n)

```

```

01748 {
01749     int i,x;
01750     for ( i = 0; i < n; i++ ) {
01751         if ( ( x = (int)(t1[i]-t2[i]) ) != 0 ) return(x);
01752     }
01753     return(0);
01754 }
01755
01756 /*
01757     #] CmpArray :
01758     #[ ConWord :
01759 */
01760
01761 int ConWord(UBYTE *s1, UBYTE *s2)
01762 {
01763     while ( *s1 && ( tolower(*s1) == tolower(*s2) ) ) { s1++; s2++; }
01764     if ( *s1 == 0 ) return(1);
01765     return(0);
01766 }
01767
01768 /*
01769     #] ConWord :
01770     #[ StrLen :
01771 */
01772
01773 int StrLen(UBYTE *s)
01774 {
01775     int i = 0;
01776     while ( *s ) { s++; i++; }
01777     return(i);
01778 }
01779
01780 /*
01781     #] StrLen :
01782     #[ NumToStr :
01783 */
01784
01785 void NumToStr(UBYTE *s, LONG x)
01786 {
01787     UBYTE *t, str[24];
01788     ULONG xx;
01789     t = str;
01790     if ( x < 0 ) { *s++ = '-'; xx = -x; }
01791     else xx = x;
01792     do {
01793         *t++ = xx % 10 + '0';
01794         xx /= 10;
01795     } while ( xx );
01796     while ( t > str ) *s++ = *--t;
01797     *s = 0;
01798 }
01799
01800 /*
01801     #] NumToStr :
01802     #[ WriteString :
01803
01804     Writes a characterstring to the various outputs.
01805     The action may depend on the flags involved.
01806     The type of output is given by type, the string by str and the
01807     number of characters in it by num
01808 */
01809 void WriteString(int type, UBYTE *str, int num)
01810 {
01811     int error = 0;
01812
01813     if ( num > 0 && str[num-1] == 0 ) { num--; }
01814     else if ( num <= 0 || str[num-1] != LINEFEED ) {
01815         AddLineFeed(str,num);
01816     }
01817     /*[15apr2004 mt]:*/
01818     if(type == EXTERNALCHANNELOUT){
01819         if(WriteFile(0,str,num) != num) error = 1;
01820     }else
01821     /*:[15apr2004 mt]*/
01822     if ( AM.silent == 0 || type == ERROROUT ) {
01823         if ( type == INPUTOUT ) {
01824             if ( !AM.FileOnlyFlag && WriteFile(AM.Stdout,(UBYTE *)"      ",4) != 4 ) error = 1;
01825             if ( AC.LogHandle >= 0 && WriteFile(AC.LogHandle,(UBYTE *)"      ",4) != 4 ) error = 1;
01826         }
01827         if ( !AM.FileOnlyFlag && WriteFile(AM.Stdout,str,num) != num ) error = 1;
01828         if ( AC.LogHandle >= 0 && WriteFile(AC.LogHandle,str,num) != num ) error = 1;
01829     }
01830     if ( error ) Terminate(-1);
01831 }
01832
01833 /*
01834     #] WriteString :

```

```

01835         # [ WriteUnfinString :
01836
01837         Writes a characterstring to the various outputs.
01838         The action may depend on the flags involved.
01839         The type of output is given by type, the string by str and the
01840         number of characters in it by num
01841     */
01842
01843 void WriteUnfinString(int type, UBYTE *str, int num)
01844 {
01845     int error = 0;
01846
01847     /*[15apr2004 mt]:*/
01848     if (type == EXTERNALCHANNELOUT) {
01849         if (WriteFile(0, str, num) != num) error = 1;
01850     } else
01851     /*:[15apr2004 mt]:*/
01852     if ( AM.silent == 0 || type == ERROROUT ) {
01853         if ( type == INPUTOUT ) {
01854             if ( !AM.FileOnlyFlag && WriteFile(AM.StdOut, (UBYTE *) " ", 4) != 4 ) error = 1;
01855             if ( AC.LogHandle >= 0 && WriteFile(AC.LogHandle, (UBYTE *) " ", 4) != 4 ) error = 1;
01856         }
01857         if ( !AM.FileOnlyFlag && WriteFile(AM.StdOut, str, num) != num ) error = 1;
01858         if ( AC.LogHandle >= 0 && WriteFile(AC.LogHandle, str, num) != num ) error = 1;
01859     }
01860     if ( error ) Terminate(-1);
01861 }
01862
01863 /*
01864         # ] WriteUnfinString :
01865         # [ AddToString :
01866     */
01867
01868 UBYTE *AddToString(UBYTE *outstring, UBYTE *extrastring, int par)
01869 {
01870     UBYTE *s = extrastring, *t, *newstring;
01871     int n, nn;
01872     while ( *s ) { s++; }
01873     n = s - extrastring;
01874     if ( outstring == 0 ) {
01875         s = extrastring;
01876         t = outstring = (UBYTE *) Malloc1(n+1, "AddToString");
01877         NCOPY(t, s, n)
01878         *t++ = 0;
01879         return(outstring);
01880     }
01881     else {
01882         t = outstring;
01883         while ( *t ) t++;
01884         nn = t - outstring;
01885         t = newstring = (UBYTE *) Malloc1(n+nn+2, "AddToString");
01886         s = outstring;
01887         NCOPY(t, s, nn)
01888         if ( par == 1 ) *t++ = ',';
01889         s = extrastring;
01890         NCOPY(t, s, n)
01891         *t = 0;
01892         M_free(outstring, "AddToString");
01893         return(newstring);
01894     }
01895 }
01896
01897 /*
01898         # ] AddToString :
01899         # [ strDupl :
01900
01901         string duplication with message passing for Malloc1, allowing
01902         this routine to give a more detailed error message if there
01903         is not enough memory.
01904     */
01905
01906 UBYTE *strDupl(UBYTE *instring, char *ifwrong)
01907 {
01908     UBYTE *s = instring, *to;
01909     while ( *s ) s++;
01910     to = s = (UBYTE *) Malloc1((s - instring)+1, ifwrong);
01911     while ( *instring ) *to++ = *instring++;
01912     *to = 0;
01913     return(s);
01914 }
01915
01916 /*
01917         # ] strDupl :
01918         # [ EndOfToken :
01919     */
01920
01921 UBYTE *EndOfToken(UBYTE *s)

```

```

01933 {
01934     UBYTE c;
01935     while ( ( c = (UBYTE)(FG.cTable[*s]) ) == 0 || c == 1 ) s++;
01936     return(s);
01937 }
01938
01939 /*
01940     #] EndOfToken :
01941     #[ ToToken :
01942 */
01943
01955 UBYTE *ToToken(UBYTE *s)
01956 {
01957     UBYTE c;
01958     while ( *s && ( c = (UBYTE)(FG.cTable[*s]) ) != 0 && c != 1 ) s++;
01959     return(s);
01960 }
01961
01962 /*
01963     #] ToToken :
01964     #[ SkipField :
01965 */
01966
01976 UBYTE *SkipField(UBYTE *s, int level)
01977 {
01978     while ( *s ) {
01979         if ( *s == ',' && level == 0 ) return(s);
01980         if ( *s == '(' ) level++;
01981         else if ( *s == ')' ) { level--; if ( level < 0 ) level = 0; }
01982         else if ( *s == '[' ) {
01983             SKIPBRA1(s)
01984         }
01985         else if ( *s == '{' ) {
01986             SKIPBRA2(s)
01987         }
01988         s++;
01989     }
01990     return(s);
01991 }
01992
01993 /*
01994     #] SkipField :
01995     #[ ReadSnum :          WORD ReadSnum(p)
01996
01997     Reads a number that should fit in a word.
01998     The number should be unsigned and a negative return value
01999     indicates an irregularity.
02000
02001 */
02002
02003 WORD ReadSnum(UBYTE **p)
02004 {
02005     LONG x = 0;
02006     UBYTE *s;
02007     s = *p;
02008     if ( FG.cTable[*s] == 1 ) {
02009         do {
02010             x = ( x << 3 ) + ( x << 1 ) + ( *s++ - '0' );
02011             if ( x > MAXPOSITIVE ) return(-1);
02012         } while ( FG.cTable[*s] == 1 );
02013         *p = s;
02014         return((WORD)x);
02015     }
02016     else return(-1);
02017 }
02018
02019 /*
02020     #] ReadSnum :
02021     #[ NumCopy :
02022
02023     Adds the decimal representation of a number to a string.
02024
02025 */
02026
02027 UBYTE *NumCopy(WORD y, UBYTE *to)
02028 {
02029     UBYTE *s;
02030     WORD i = 0, j;
02031     UWORD x;
02032     if ( y < 0 ) {
02033         *to++ = '-';
02034     }
02035     x = WordAbs(y);
02036     s = to;
02037     do { *s++ = (UBYTE)((x % 10) + '0'); i++; } while ( ( x /= 10 ) != 0 );
02038     *s-- = '\0';
02039     j = ( i - 1 ) >> 1;

```

```

02040     while ( j >= 0 ) {
02041         i = to[j]; to[j] = s[-j]; s[-j] = (UBYTE)i; j--;
02042     }
02043     return(s+1);
02044 }
02045
02046 /*
02047     #] NumCopy :
02048     #[ LongCopy :
02049
02050     Adds the decimal representation of a number to a string.
02051
02052 */
02053
02054 char *LongCopy(LONG y, char *to)
02055 {
02056     char *s;
02057     WORD i = 0, j;
02058     ULONG x;
02059     if ( y < 0 ) {
02060         *to++ = '-';
02061     }
02062     x = LongAbs(y);
02063     s = to;
02064     do { *s++ = (x % 10) + '0'; i++; } while ( ( x /= 10 ) != 0 );
02065     *s-- = '\0';
02066     j = ( i - 1 ) >> 1;
02067     while ( j >= 0 ) {
02068         i = to[j]; to[j] = s[-j]; s[-j] = (char)i; j--;
02069     }
02070     return(s+1);
02071 }
02072
02073 /*
02074     #] LongCopy :
02075     #[ LongLongCopy :
02076
02077     Adds the decimal representation of a number to a string.
02078     Bugfix feb 2003. y was not pointer!
02079 */
02080
02081 char *LongLongCopy(off_t *y, char *to)
02082 {
02083     /*
02084     * This code fails to print the maximum negative value on systems with two's
02085     * complement. To fix this, we need the unsigned version of off_t with the
02086     * same size, but unfortunately it is undefined. On the other hand, if a
02087     * system is configured with a 64-bit off_t, in practice one never reaches
02088     * 2^63 ~ 10^18 as of 2016. If one really reach such a big number, then it
02089     * would be the time to move on a 128-bit off_t.
02090     */
02091     off_t x = *y;
02092     char *s;
02093     WORD i = 0, j;
02094     if ( x < 0 ) { x = -x; *to++ = '-'; }
02095     s = to;
02096     do { *s++ = (x % 10) + '0'; i++; } while ( ( x /= 10 ) != 0 );
02097     *s-- = '\0';
02098     j = ( i - 1 ) >> 1;
02099     while ( j >= 0 ) {
02100         i = to[j]; to[j] = s[-j]; s[-j] = (char)i; j--;
02101     }
02102     return(s+1);
02103 }
02104
02105 /*
02106     #] LongLongCopy :
02107     #[ MakeDate :
02108
02109     Routine produces a string with the date and time of the run
02110 */
02111
02112 #if defined(ANSI) || defined(mBSD)
02113 #else
02114 static char notime[] = "";
02115 #endif
02116
02117 UBYTE *MakeDate(void)
02118 {
02119     #if defined(ANSI) || defined(mBSD)
02120         time_t tp;
02121         time(&tp);
02122         return((UBYTE *)ctime(&tp));
02123     #else
02124         return((UBYTE *)notime);
02125     #endif
02126 }

```



```

02127
02128 /*
02129     #] MakeDate :
02130     #[ set_in :
02131         Returns 1 if ch is in set ; 0 if ch is not in set:
02132 */
02133 int set_in(UBYTE ch, set_of_char set)
02134 {
02135     set += ch/8;
02136     switch (ch % 8){
02137         case 0: return(set->bit_0);
02138         case 1: return(set->bit_1);
02139         case 2: return(set->bit_2);
02140         case 3: return(set->bit_3);
02141         case 4: return(set->bit_4);
02142         case 5: return(set->bit_5);
02143         case 6: return(set->bit_6);
02144         case 7: return(set->bit_7);
02145     }/*switch (ch % 8)*/
02146     return(-1);
02147 }/*set_in*/
02148 /*
02149     #] set_in :
02150     #[ set_set :
02151         sets ch into set; returns *set:
02152 */
02153 one_byte set_set(UBYTE ch, set_of_char set)
02154 {
02155     one_byte tmp=(one_byte)set;
02156     set += ch/8;
02157     switch (ch % 8){
02158         case 0: set->bit_0=1;break;
02159         case 1: set->bit_1=1;break;
02160         case 2: set->bit_2=1;break;
02161         case 3: set->bit_3=1;break;
02162         case 4: set->bit_4=1;break;
02163         case 5: set->bit_5=1;break;
02164         case 6: set->bit_6=1;break;
02165         case 7: set->bit_7=1;break;
02166     }
02167     return(tmp);
02168 }/*set_set*/
02169 /*
02170     #] set_set :
02171     #[ set_del :
02172         deletes ch from set; returns *set:
02173 */
02174 one_byte set_del(UBYTE ch, set_of_char set)
02175 {
02176     one_byte tmp=(one_byte)set;
02177     set += ch/8;
02178     switch (ch % 8){
02179         case 0: set->bit_0=0;break;
02180         case 1: set->bit_1=0;break;
02181         case 2: set->bit_2=0;break;
02182         case 3: set->bit_3=0;break;
02183         case 4: set->bit_4=0;break;
02184         case 5: set->bit_5=0;break;
02185         case 6: set->bit_6=0;break;
02186         case 7: set->bit_7=0;break;
02187     }
02188     return(tmp);
02189 }/*set_del*/
02190 /*
02191     #] set_del :
02192     #[ set_sub :
02193         returns *set = set1\set2. This function may be used for initialising,
02194         set_sub(a,a,a) => now a is empty set :
02195 */
02196 one_byte set_sub(set_of_char set, set_of_char set1, set_of_char set2)
02197 {
02198     one_byte tmp=(one_byte)set;
02199     int i=0,j=0;
02200     while(j=0,i++<32)
02201     while(j<9)
02202         switch (j++){
02203             case 0: set->bit_0=(set1->bit_0&&!set2->bit_0);break;
02204             case 1: set->bit_1=(set1->bit_1&&!set2->bit_1);break;
02205             case 2: set->bit_2=(set1->bit_2&&!set2->bit_2);break;
02206             case 3: set->bit_3=(set1->bit_3&&!set2->bit_3);break;
02207             case 4: set->bit_4=(set1->bit_4&&!set2->bit_4);break;
02208             case 5: set->bit_5=(set1->bit_5&&!set2->bit_5);break;
02209             case 6: set->bit_6=(set1->bit_6&&!set2->bit_6);break;
02210             case 7: set->bit_7=(set1->bit_7&&!set2->bit_7);break;
02211             case 8: set++;set1++;set2++;
02212         };
02213     return(tmp);

```

```

02214  }/*set_sub*/
02215  /*
02216      #] set_sub :
02217      #] Strings :
02218      #[ Mixed :
02219      #[ iniTools :
02220  */
02221
02222  void iniTools(void)
02223  {
02224      #ifdef MALLOCPROTECT
02225          if ( mprotectInit() ) exit(0);
02226      #endif
02227          return;
02228  }
02229
02230  /*
02231      #] iniTools :
02232      #[ Malloc1 :
02233
02234      Malloc routine with built in error checking,
02235      and a more detailed error message.
02236      Gives the user some idea of what is happening.
02237  */
02238
02239  #ifdef MALLOCDDEBUG
02240      INILOCK(MallocLock)
02241  #endif
02242
02243  void *Malloc1(LONG size, const char *messageifwrong)
02244  {
02245      void *mem;
02246      #ifdef MALLOCDDEBUG
02247          char *t, *u;
02248          int i;
02249          LOCK(MallocLock);
02250          if ( size == 0 ) {
02251              MesPrint("%wAsking for 0 bytes in Malloc1");
02252          }
02253      #endif
02254      #ifdef WITHSTATS
02255          nummalloccs++;
02256      #endif
02257      if ( ( size & 7 ) != 0 ) { size = size - ( size&7 ) + 8; }
02258      #ifdef MALLOCDDEBUG
02259          size += 2*BANNER;
02260      #endif
02261      mem = (void *)M_alloc(size);
02262      if ( mem == 0 ) {
02263          #ifndef MALLOCDDEBUG
02264              MLOCK(ErrorMessageLock);
02265          #endif
02266          MesPrint("Attempted to allocate %l bytes.", size);
02267          MesPrint("@No memory while allocating %s", (UBYTE *)messageifwrong);
02268          #ifndef MALLOCDDEBUG
02269              MUNLOCK(ErrorMessageLock);
02270          #else
02271              UNLOCK(MallocLock);
02272          #endif
02273          Terminate(-1);
02274      }
02275      #ifdef MALLOCDDEBUG
02276          malloccsizes[nummalloclist] = size;
02277          malloccstrings[nummalloclist] = (char *)messageifwrong;
02278          malloclist[nummalloclist++] = mem;
02279          if ( AC.MemDebugFlag && filelist ) MesPrint("%wMem1 at 0x%x: %l bytes.
02280          %s", mem, size, messageifwrong);
02281          {
02282              int i = nummalloclist-1;
02283              while ( --i >= 0 ) {
02284                  if ( (char *)mem < (((char *)malloclist[i]) + malloccsizes[i])
02285                      && (char *)malloclist[i] < ((char *)mem + size) ) {
02286                      if ( filelist ) MesPrint("This memory overlaps with the block at 0x%x"
02287                      ,malloclist[i]);
02288                  }
02289              }
02290          }
02291      #endif
02292      #ifdef MALLOCDDEBUGOUTPUT
02293          printf ("Malloc1: %s, allocated %li bytes at %.8lx\n",messageifwrong,size,(unsigned long)mem);
02294          fflush (stdout);
02295      #endif
02296      t = (char *)mem;
02297      u = t + size;
02298      for ( i = 0; i < (int)BANNER; i++ ) { *t++ = FILLVALUE; *--u = FILLVALUE; }
02299      mem = (void *)t;

```

```

02300     M_check();
02301     /* MUNLOCK(ErrorMessageLock); */
02302     UNLOCK(MallocLock);
02303 #endif
02304 /*
02305     if ( size > 500000000L ) {
02306         MLOCK(ErrorMessageLock);
02307         MesPrint("Malloc1: %s, allocated %l bytes\n",messageifwrong,size);
02308         MUNLOCK(ErrorMessageLock);
02309     }
02310 */
02311     return(mem);
02312 }
02313
02314 /*
02315     #] Malloc1 :
02316     #[ M_free :
02317 */
02318
02319 void M_free(void *x, const char *where)
02320 {
02321 #ifdef MALLOCDEBUG
02322     char *t = (char *)x;
02323     int i, j, k;
02324     LONG size = 0;
02325     x = (void *)(((char *)x)-BANNER);
02326     /* MLOCK(ErrorMessageLock); */
02327     if ( AC.MemDebugFlag ) MesPrint("%wFreeing 0x%x: %s",x,where);
02328     LOCK(MallocLock);
02329     for ( i = nummalloclist-1; i >= 0; i-- ) {
02330         if ( x == malloclist[i] ) {
02331             size = mallocsizes[i];
02332             for ( j = i+1; j < nummalloclist; j++ ) {
02333                 malloclist[j-1] = malloclist[j];
02334                 mallocsizes[j-1] = mallocsizes[j];
02335                 mallocstrings[j-1] = mallocstrings[j];
02336             }
02337             nummalloclist--;
02338             break;
02339         }
02340     }
02341     if ( i < 0 ) {
02342         unsigned int xx = ((ULONG)x);
02343         printf("Error returning non-allocated address: 0x%x from %s\n",
02344             ,xx,where);
02345         /* MUNLOCK(ErrorMessageLock); */
02346         UNLOCK(MallocLock);
02347         exit(-1);
02348     }
02349     else {
02350         for ( k = 0, j = 0; k < (int)BANNER; k++ ) {
02351             if ( *--t != FILLVALUE ) j++;
02352         }
02353         if ( j ) {
02354             LONG *tt = (LONG *)x;
02355             MesPrint("%w!!!! Banner has been written in !!!!!: %x %x %x %x",
02356                 tt[0],tt[1],tt[2],tt[3]);
02357         }
02358         t += size;
02359         for ( k = 0, j = 0; k < (int)BANNER; k++ ) {
02360             if ( *--t != FILLVALUE ) j++;
02361         }
02362         if ( j ) {
02363             LONG *tt = (LONG *)x;
02364             MesPrint("%w!!!! Tail has been written in !!!!!: %x %x %x %x",
02365                 tt[0],tt[1],tt[2],tt[3]);
02366         }
02367         M_check();
02368         /* MUNLOCK(ErrorMessageLock); */
02369         UNLOCK(MallocLock);
02370     }
02371 #else
02372     DUMMYUSE(where);
02373 #endif
02374 #ifdef WITHSTATS
02375     numfrees++;
02376 #endif
02377     if ( x ) {
02378 #ifdef MALLOCDEBUGOUTPUT
02379         printf ("M_free: %s, memory freed at %.8lx\n",where,(unsigned long)x);
02380         fflush(stdout);
02381 #endif
02382     }
02383 #ifdef MALLOCPROTECT
02384     mprotectFree((void *)x);
02385 #else
02386     free(x);

```

```

02387 #endif
02388     }
02389 }
02390
02391 /*
02392     #] M_free :
02393     #[ M_check :
02394 */
02395
02396 #ifdef MALLOCDEBUG
02397
02398 void M_check1() { MesPrint("Checking Malloc"); M_check(); }
02399
02400 void M_check()
02401 {
02402     int i,j,k,error = 0;
02403     char *t;
02404     LONG *tt;
02405     for ( i = 0; i < nummalloclist; i++ ) {
02406         t = (char *) (malloclist[i]);
02407         for ( k = 0, j = 0; k < (int)BANNER; k++ ) {
02408             if ( *t++ != FILLVALUE ) j++;
02409         }
02410         if ( j ) {
02411             tt = (LONG *) (malloclist[i]);
02412             MesPrint("%w!!!! Banner %d (%s) has been written in !!!!!: %x %x %x %x",
02413                 i,mallocstrings[i],tt[0],tt[1],tt[2],tt[3]);
02414             tt[0] = tt[1] = tt[2] = tt[3] = 0;
02415             error = 1;
02416         }
02417         t = (char *) (malloclist[i] + mallocsizes[i];
02418         for ( k = 0, j = 0; k < (int)BANNER; k++ ) {
02419             if ( *--t != FILLVALUE ) j++;
02420         }
02421         if ( j ) {
02422             tt = (LONG *)t;
02423             MesPrint("%w!!!! Tail %d (%s) has been written in !!!!!: %x %x %x %x",
02424                 i,mallocstrings[i],tt[0],tt[1],tt[2],tt[3]);
02425             tt[0] = tt[1] = tt[2] = tt[3] = 0;
02426             error = 1;
02427         }
02428         if ( ( mallocstrings[i][0] == ' ' ) || ( mallocstrings[i][0] == '#' ) ) {
02429             MesPrint("%w!!!! Funny mallocstring");
02430             error = 1;
02431         }
02432     }
02433     if ( error ) {
02434         M_print();
02435         /* MUNLOCK(ErrorMessageLock); */
02436         UNLOCK(MallocLock);
02437         Terminate(-1);
02438     }
02439 }
02440
02441 void M_print()
02442 {
02443     int i;
02444     MesPrint("We have the following memory allocations left:");
02445     for ( i = 0; i < nummalloclist; i++ ) {
02446         MesPrint("0x%x: %l bytes. number %d: '%s'",malloclist[i],mallocsizes[i],i,mallocstrings[i]);
02447     }
02448 }
02449
02450 #else
02451
02452 void M_check1(void) {}
02453 void M_print(void) {}
02454
02455 #endif
02456
02457 /*
02458     #] M_check :
02459     #[ TermMalloc :
02460 */
02483 #define TERMMEMSTARTNUM 16
02484 #define TERMEXTRAWORDS 10
02485
02486 void TermMallocAddMemory(PHEAD0)
02487 {
02488     WORD *newbufs;
02489     int i, extra;
02490     if ( AT.TermMemMax == 0 ) extra = TERMMEMSTARTNUM;
02491     else extra = AT.TermMemMax;
02492     if ( AT.TermMemHeap ) M_free(AT.TermMemHeap,"TermMalloc");
02493     newbufs = (WORD *) Malloc1(extra*(AM.MaxTer+TERMEXTRAWORDS*sizeof(WORD)),"TermMalloc");
02494     AT.TermMemHeap = (WORD **) Malloc1((extra+AT.TermMemMax)*sizeof(WORD *),"TermMalloc");
02495     for ( i = 0; i < extra; i++ ) {

```

```

02496     AT.TermMemHeap[i] = newbufs + i*(AM.MaxTer/sizeof(WORD)+TERMEXTRAWORDS);
02497 }
02498 #ifdef TERMMALLOCDEBUG
02499     DebugHeap2 = (WORD **)Malloc1((extra+AT.TermMemMax)*sizeof(WORD *),"TermMalloc");
02500     for ( i = 0; i < AT.TermMemMax; i++ ) { DebugHeap2[i] = DebugHeap1[i]; }
02501     for ( i = 0; i < extra; i++ ) {
02502         DebugHeap2[i+AT.TermMemMax] = newbufs + i*(AM.MaxTer/sizeof(WORD)+TERMEXTRAWORDS);
02503     }
02504     if ( DebugHeap1 ) M_free(DebugHeap1,"TermMalloc");
02505     DebugHeap1 = DebugHeap2;
02506 #endif
02507     AT.TermMemTop = extra;
02508     AT.TermMemMax += extra;
02509 #ifdef TERMMALLOCDEBUG
02510     MesPrint("AT.TermMemMax is now %l",AT.TermMemMax);
02511 #endif
02512 }
02513
02514 #ifndef MEMORYMACROS
02515
02516 WORD *TermMalloc2(PHEAD char *text)
02517 {
02518     if ( AT.TermMemTop <= 0 ) TermMallocAddMemory(BHEAD0);
02519
02520 #ifdef TERMMALLOCDEBUG
02521     MesPrint("TermMalloc: %s, %d",text,(AT.TermMemMax-AT.TermMemTop));
02522 #endif
02523
02524 #ifdef MALLOCDEBUGOUTPUT
02525     MesPrint("TermMalloc: %s, %l/%l
02526 (%x)",text,AT.TermMemTop,AT.TermMemMax,AT.TermMemHeap[AT.TermMemTop-1]);
02527 #endif
02528     DUMMYUSE(text);
02529     return(AT.TermMemHeap[--AT.TermMemTop]);
02530 }
02531
02532 void TermFree2(PHEAD WORD *TermMem, char *text)
02533 {
02534 #ifdef TERMMALLOCDEBUG
02535     int i;
02536
02537     for ( i = 0; i < AT.TermMemMax; i++ ) {
02538         if ( TermMem == DebugHeap1[i] ) break;
02539     }
02540     if ( i >= AT.TermMemMax ) {
02541         MesPrint(" ERROR: TermFree called with an address not given by TermMalloc.");
02542         Terminate(-1);
02543     }
02544 #endif
02545     DUMMYUSE(text);
02546     AT.TermMemHeap[AT.TermMemTop++] = TermMem;
02547
02548 #ifdef TERMMALLOCDEBUG
02549     MesPrint("TermFree: %s, %d",text,(AT.TermMemMax-AT.TermMemTop));
02550 #endif
02551 #ifdef MALLOCDEBUGOUTPUT
02552     MesPrint("TermFree: %s, %l/%l (%x)",text,AT.TermMemTop,AT.TermMemMax,TermMem);
02553 #endif
02554 }
02555 #endif
02556
02557 #endif
02558
02559 /*
02560     #] TermMalloc :
02561     #[ NumberMalloc :
02562 */
02563 #define NUMBERMEMSTARTNUM 16
02564 #define NUMBEREXTRAWORDS 10L
02565
02566 #ifdef TERMMALLOCDEBUG
02567     UWORD **DebugHeap3, **DebugHeap4;
02568 #endif
02569
02570 void NumberMallocAddMemory(PHEAD0)
02571 {
02572     UWORD *newbufs;
02573     WORD extra;
02574     int i;
02575     if ( AT.NumberMemMax == 0 ) extra = NUMBERMEMSTARTNUM;
02576     else extra = AT.NumberMemMax;
02577     if ( AT.NumberMemHeap ) M_free(AT.NumberMemHeap,"NumberMalloc");
02578     newbufs = (UWORD *)Malloc1(extra*(AM.MaxTal+NUMBEREXTRAWORDS)*sizeof(UWORD),"NumberMalloc");
02579     AT.NumberMemHeap = (UWORD **)Malloc1((extra+AT.NumberMemMax)*sizeof(UWORD *),"NumberMalloc");
02580     for ( i = 0; i < extra; i++ ) {
02581         AT.NumberMemHeap[i] = newbufs + i*(LONG)(AM.MaxTal+NUMBEREXTRAWORDS);

```

```

02602     }
02603 #ifdef TERMMALLOCDEBUG
02604     DebugHeap4 = (UWORD **)Malloc1((extra+AT.NumberMemMax)*sizeof(WORD *),"NumberMalloc");
02605     for ( i = 0; i < AT.NumberMemMax; i++ ) { DebugHeap4[i] = DebugHeap3[i]; }
02606     for ( i = 0; i < extra; i++ ) {
02607         DebugHeap4[i+AT.NumberMemMax] = newbufs + i*(LONG) (AM.MaxTal+NUMBEREXTRAWORDS);
02608     }
02609     if ( DebugHeap3 ) M_free(DebugHeap3,"NumberMalloc");
02610     DebugHeap3 = DebugHeap4;
02611 #endif
02612     AT.NumberMemTop = extra;
02613     AT.NumberMemMax += extra;
02614 /*
02615 MesPrint("AT.NumberMemMax is now %l",AT.NumberMemMax);
02616 */
02617 }
02618
02619 #ifndef MEMORYMACROS
02620
02621 UWORD *NumberMalloc2(PHEAD char *text)
02622 {
02623     if ( AT.NumberMemTop <= 0 ) NumberMallocAddMemory(BHEAD text);
02624
02625 #ifdef MALLOCDEBUGOUTPUT
02626     if ( (AT.NumberMemMax-AT.NumberMemTop) > 10 )
02627         MesPrint("NumberMalloc: %s, %l/%l (%x)",text,AT.NumberMemTop,AT.NumberMemMax,AT.NumberMemHeap[AT.NumberMemTop-1]);
02628 #endif
02629     DUMMYUSE(text);
02630     return(AT.NumberMemHeap[--AT.NumberMemTop]);
02631 }
02632
02633 void NumberFree2(PHEAD UWORD *NumberMem, char *text)
02634 {
02635 #ifdef TERMMALLOCDEBUG
02636     int i;
02637     for ( i = 0; i < AT.NumberMemMax; i++ ) {
02638         if ( NumberMem == DebugHeap3[i] ) break;
02639     }
02640     if ( i >= AT.NumberMemMax ) {
02641         MesPrint(" ERROR: NumberFree called with an address not given by NumberMalloc.");
02642         Terminate(-1);
02643     }
02644 #endif
02645     DUMMYUSE(text);
02646     AT.NumberMemHeap[AT.NumberMemTop++] = NumberMem;
02647
02648 #ifdef MALLOCDEBUGOUTPUT
02649     if ( (AT.NumberMemMax-AT.NumberMemTop) > 10 )
02650         MesPrint("NumberFree: %s, %l/%l (%x)",text,AT.NumberMemTop,AT.NumberMemMax,NumberMem);
02651 #endif
02652 }
02653
02654 #endif
02655
02656 /*
02657     #] NumberMalloc :
02658     #[ CacheNumberMalloc :
02659
02660     Similar to NumberMalloc
02661 */
02662
02663 void CacheNumberMallocAddMemory(PHEAD0)
02664 {
02665     UWORD *newbufs;
02666     WORD extra;
02667     int i;
02668     if ( AT.CacheNumberMemMax == 0 ) extra = NUMBERMEMSTARTNUM;
02669     else extra = AT.CacheNumberMemMax;
02670     if ( AT.CacheNumberMemHeap ) M_free(AT.CacheNumberMemHeap,"NumberMalloc");
02671     newbufs = (UWORD *)M_malloc1(extra*(AM.MaxTal+NUMBEREXTRAWORDS)*sizeof(UWORD),"CacheNumberMalloc");
02672     AT.CacheNumberMemHeap = (UWORD **)M_malloc1((extra+AT.NumberMemMax)*sizeof(UWORD
02673 *),"CacheNumberMalloc");
02674     for ( i = 0; i < extra; i++ ) {
02675         AT.CacheNumberMemHeap[i] = newbufs + i*(LONG) (AM.MaxTal+NUMBEREXTRAWORDS);
02676     }
02677     AT.CacheNumberMemTop = extra;
02678     AT.CacheNumberMemMax += extra;
02679 }
02680
02681 #ifndef MEMORYMACROS
02682
02683 UWORD *CacheNumberMalloc2(PHEAD char *text)
02684 {
02685     if ( AT.CacheNumberMemTop <= 0 ) CacheNumberMallocAddMemory(BHEAD0);
02686

```

```

02687 #ifdef MALLOCDEBUGOUTPUT
02688     MesPrint("NumberMalloc: %s, %l/%l
(%x)",text,AT.NumberMemTop,AT.NumberMemMax,AT.NumberMemHeap[AT.NumberMemTop-1]);
02689 #endif
02690
02691     DUMMYUSE(text);
02692     return(AT.CacheNumberMemHeap[--AT.CacheNumberMemTop]);
02693 }
02694
02695 void CacheNumberFree2(PHEAD UWORD *NumberMem, char *text)
02696 {
02697     DUMMYUSE(text);
02698     AT.CacheNumberMemHeap[AT.CacheNumberMemTop++] = NumberMem;
02699
02700 #ifdef MALLOCDEBUGOUTPUT
02701     MesPrint("NumberFree: %s, %l/%l (%x)",text,AT.NumberMemTop,AT.NumberMemMax,NumberMem);
02702 #endif
02703 }
02704
02705 #endif
02706
02707 /*
02708     #] CacheNumberMalloc :
02709     #[ FromList :
02710
02711     Returns the next object in a list.
02712     If the list has been exhausted we double it (like a realloc)
02713     If the list has not been initialized yet we start with 12 elements.
02714 */
02715
02716 void *FromList(LIST *L)
02717 {
02718     void *newlist;
02719     int i, *old, *newL;
02720     if ( L->num >= L->maxnum || L->lijst == 0 ) {
02721         if ( L->maxnum == 0 ) L->maxnum = 12;
02722         else if ( L->lijst ) L->maxnum *= 2;
02723         newlist = Malloc1(L->maxnum * L->size,L->message);
02724         if ( L->lijst ) {
02725             i = ( L->num * L->size ) / sizeof(int);
02726             old = (int *)L->lijst; newL = (int *)newlist;
02727             while ( --i >= 0 ) *newL++ = *old++;
02728             if ( L->lijst ) M_free(L->lijst,"L->lijst FromList");
02729         }
02730         L->lijst = newlist;
02731     }
02732     return( ((char *) (L->lijst)) + L->size * (L->num)++ );
02733 }
02734
02735 /*
02736     #] FromList :
02737     #[ From0List :
02738
02739     Same as FromList, but we zero excess variables.
02740 */
02741
02742 void *From0List(LIST *L)
02743 {
02744     void *newlist;
02745     int i, *old, *newL;
02746     if ( L->num >= L->maxnum || L->lijst == 0 ) {
02747         if ( L->maxnum == 0 ) L->maxnum = 12;
02748         else if ( L->lijst ) L->maxnum *= 2;
02749         newlist = Malloc1(L->maxnum * L->size,L->message);
02750         i = ( L->num * L->size ) / sizeof(int);
02751         old = (int *) (L->lijst); newL = (int *) newlist;
02752         while ( --i >= 0 ) *newL++ = *old++;
02753         i = ( L->maxnum - L->num ) / sizeof(int);
02754         while ( --i >= 0 ) *newL++ = 0;
02755         if ( L->lijst ) M_free(L->lijst,"L->lijst From0List");
02756         L->lijst = newlist;
02757     }
02758     return( ((char *) (L->lijst)) + L->size * (L->num)++ );
02759 }
02760
02761 /*
02762     #] From0List :
02763     #[ FromVarList :
02764
02765     Returns the next object in a list of variables.
02766     If the list has been exhausted we double it (like a realloc)
02767     If the list has not been initialized yet we start with 12 elements.
02768     We allow at most MAXVARIABLES elements!
02769 */
02770
02771 void *FromVarList(LIST *L)
02772 {

```

```

02773 void *newlist;
02774 int i, *old, *newL;
02775 if ( L->num >= L->maxnum || L->lijst == 0 ) {
02776     if ( L->maxnum == 0 ) L->maxnum = 12;
02777     else if ( L->lijst ) {
02778         L->maxnum *= 2;
02779         if ( L == &(AP.DollarList) ) {
02780             if ( L->maxnum > MAXDOLLARVARIABLES ) L->maxnum = MAXDOLLARVARIABLES;
02781             if ( L->num >= MAXDOLLARVARIABLES ) {
02782                 MesPrint("!!!More than %l objects in list of $-variables",
02783                     MAXDOLLARVARIABLES);
02784                 Terminate(-1);
02785             }
02786         }
02787     }
02788     else {
02789         if ( L->maxnum > MAXVARIABLES ) L->maxnum = MAXVARIABLES;
02790         if ( L->num >= MAXVARIABLES ) {
02791             MesPrint("!!!More than %l objects in list of variables",
02792                 MAXVARIABLES);
02793             Terminate(-1);
02794         }
02795     }
02796     newlist = Malloc1(L->maxnum * L->size, L->message);
02797     if ( L->lijst ) {
02798         i = ( L->num * L->size ) / sizeof(int);
02799         old = (int *) (L->lijst); newL = (int *) newlist;
02800         while ( --i >= 0 ) *newL++ = *old++;
02801         if ( L->lijst ) M_free(L->lijst, "L->lijst from VarList");
02802     }
02803     L->lijst = newlist;
02804 }
02805 return( ((char *) (L->lijst)) + L->size * ((L->num)++) );
02806 }
02807
02808 /*
02809     #] FromVarList :
02810     #[ DoubleList :
02811 */
02812
02813 int DoubleList(void ***lijst, int *oldsize, int objectsize, char *nameoftype)
02814 {
02815     void **newlist;
02816     LONG i, newsize, fullsize;
02817     void **to, **from;
02818     static LONG maxlistsize = (LONG) (MAXPOSITIVE);
02819     if ( *lijst == 0 ) {
02820         if ( *oldsize > 0 ) newsize = *oldsize;
02821         else newsize = 100;
02822     }
02823     else newsize = *oldsize * 2;
02824     if ( newsize > maxlistsize ) {
02825         if ( *oldsize == maxlistsize ) {
02826             MesPrint("No memory for extra space in %s", nameoftype);
02827             return(-1);
02828         }
02829         newsize = maxlistsize;
02830     }
02831     fullsize = ( newsize * objectsize + sizeof(void *) - 1 ) & (~sizeof(void *));
02832     newlist = (void **) Malloc1(fullsize, nameoftype);
02833     if ( *lijst ) { /* Now some punning. DANGEROUS CODE in principle */
02834         to = newlist; from = *lijst; i = (*oldsize * objectsize) / sizeof(void *);
02835     }
02836     #ifdef MALLOCDDEBUG
02837     if ( filelist ) MesPrint("    oldsize: %l, objectsize: %d, fullsize: %l"
02838         , *oldsize, objectsize, fullsize);
02839     #endif
02840     /*
02841         while ( --i >= 0 ) *to++ = *from++;
02842     */
02843     if ( *lijst ) M_free(*lijst, "DoubleLList");
02844     *lijst = newlist;
02845     *oldsize = newsize;
02846     return(0);
02847 }
02848
02849 int error;
02850 LONG lsize = *oldsize;
02851
02852 maxlistsize = (LONG) (MAXPOSITIVE);
02853 error = DoubleLList(lijst, &lsize, objectsize, nameoftype);
02854 *oldsize = lsize;
02855 maxlistsize = (LONG) (MAXLONG);
02856
02857 return(error);
02858 */
02859 }

```



```

02860 /*
02861      #] DoubleList :
02862      #[ DoubleList :
02863 */
02864
02865 int DoubleList(void ***lijst, LONG *oldsize, int objectsize, char *nameoftype)
02866 {
02867     void **newlist;
02868     LONG i, newsize, fullsize;
02869     void **to, **from;
02870     static LONG maxlistsize = (LONG) (MAXLONG);
02871     if ( *lijst == 0 ) {
02872         if ( *oldsize > 0 ) newsize = *oldsize;
02873         else newsize = 100;
02874     }
02875     else newsize = *oldsize * 2;
02876     if ( newsize > maxlistsize ) {
02877         if ( *oldsize == maxlistsize ) {
02878             MesPrint("No memory for extra space in %s",nameoftype);
02879             return(-1);
02880         }
02881         newsize = maxlistsize;
02882     }
02883     fullsize = ( newsize * objectsize + sizeof(void *)-1 ) & (~sizeof(void *));
02884     newlist = (void **)Malloca1(fullsize,nameoftype);
02885     if ( *lijst ) { /* Now some punning. DANGEROUS CODE in principle */
02886         to = newlist; from = *lijst; i = (*oldsize * objectsize)/sizeof(void *);
02887     }
02888     #ifndef MALLOCDDEBUG
02889     if ( filelist ) MesPrint("      oldsize: %l, objectsize: %d, fullsize: %l"
02890         ,*oldsize,objectsize,fullsize);
02891     #endif
02892     /*
02893         while ( --i >= 0 ) *to++ = *from++;
02894     */
02895     if ( *lijst ) M_free(*lijst,"DoubleList");
02896     *lijst = newlist;
02897     *oldsize = newsize;
02898     return(0);
02899 }
02900
02901 /*
02902      #] DoubleList :
02903      #[ DoubleBuffer :
02904 */
02905
02906 #define DODOUBLE(x) { x *s, *t, *u; if ( *start ) { \
02907     oldsize = *(x **)stop - *(x **)start; newsize = 2*oldsize; \
02908     t = u = (x *)Malloca1(newsize*sizeof(x),text); s = *(x **)start; \
02909     for ( i = 0; i < oldsize; i++ ) {t++ = *s++;} M_free(*start,"double"); } \
02910     else { newsize = 100; u = (x *)Malloca1(newsize*sizeof(x),text); } \
02911     *start = (void *)u; *stop = (void *) (u+newsize); }
02912
02913 void DoubleBuffer(void **start, void **stop, int size, char *text)
02914 {
02915     LONG oldsize, newsize, i;
02916     if ( size == sizeof(char) ) DODOUBLE(char)
02917     else if ( size == sizeof(short) ) DODOUBLE(short)
02918     else if ( size == sizeof(int) ) DODOUBLE(int)
02919     else if ( size == sizeof(LONG) ) DODOUBLE(LONG)
02920     else if ( size % sizeof(int) == 0 ) DODOUBLE(int)
02921     else {
02922         MesPrint("---Cannot handle doubling buffers of size %d",size);
02923         Terminate(-1);
02924     }
02925 }
02926
02927 /*
02928      #] DoubleBuffer :
02929      #[ ExpandBuffer :
02930 */
02931
02932 #define DOEXPAND(x) { x *newbuffer, *t, *m; \
02933     t = newbuffer = (x *)Malloca1((newsize+2)*type,"ExpandBuffer"); \
02934     if ( *buffer ) { m = (x *)*buffer; i = *oldsize; \
02935         while ( --i >= 0 ) {t++ = *m++;} M_free(*buffer,"ExpandBuffer"); \
02936     } *buffer = newbuffer; *oldsize = newsize; }
02937
02938 void ExpandBuffer(void **buffer, LONG *oldsize, int type)
02939 {
02940     LONG newsize, i;
02941     if ( *oldsize <= 0 ) { newsize = 100; }
02942     else newsize = 2*(*oldsize);
02943     if ( type == sizeof(char) ) DOEXPAND(char)
02944     else if ( type == sizeof(short) ) DOEXPAND(short)
02945     else if ( type == sizeof(int) ) DOEXPAND(int)
02946     else if ( type == sizeof(LONG) ) DOEXPAND(LONG)

```

```

02947     else if ( type == sizeof(POSITION) ) DOEXPAND(POSITION)
02948     else {
02949         MesPrint("---Cannot handle expanding buffers with objects of size %d",type);
02950         Terminate(-1);
02951     }
02952 }
02953
02954 /*
02955     #] ExpandBuffer :
02956     #[ iexp :
02957
02958     Raises the long integer y to the power p.
02959     Returnvalue is long, regardless of overflow.
02960 */
02961
02962 LONG iexp(LONG x, int p)
02963 {
02964     int sign;
02965     ULONG y;
02966     ULONG ux;
02967     if ( x == 0 ) return(0);
02968     if ( p == 0 ) return(1);
02969     sign = x < 0 ? -1 : 1;
02970     if ( sign < 0 && ( p & 1 ) == 0 ) sign = 1;
02971     ux = LongAbs(x);
02972     if ( ux == 1 ) return(sign);
02973     if ( p < 0 ) return(0);
02974     y = 1;
02975     while ( p ) {
02976         if ( ( p & 1 ) != 0 ) y *= ux;
02977         p >>= 1;
02978         ux = ux*ux;
02979     }
02980     if ( sign < 0 ) y = -y;
02981     return ULONGToLong(y);
02982 }
02983
02984 /*
02985     #] iexp :
02986     #[ ToGeneral :
02987
02988     Convert a fast argument to a general argument
02989     Input in r, output in m.
02990     If par == 0 we need the argument header also.
02991 */
02992
02993 void ToGeneral(WORD *r, WORD *m, WORD par)
02994 {
02995     WORD *mm = m, j, k;
02996     if ( par ) m++;
02997     else { m[1] = 0; m += ARGHEAD + 1; }
02998     j = -r++;
02999     k = 3;
03000     /* JV: Bugfix 1-feb-2016. Old code assumed FUNHEAD to be 2 */
03001     if ( j >= FUNCTION ) { *m++ = j; *m++ = FUNHEAD; FILLFUN(m) }
03002     else {
03003         switch ( j ) {
03004             case SYMBOL: *m++ = j; *m++ = 4; *m++ = *r++; *m++ = 1; break;
03005             case SNUMBER:
03006                 if ( *r > 0 ) { *m++ = *r; *m++ = 1; *m++ = 3; }
03007                 else if ( *r == 0 ) { m--; }
03008             else { *m++ = -*r; *m++ = 1; *m++ = -3; }
03009             goto MakeSize;
03010             case MINVECTOR:
03011                 k = -k;
03012                 /* fall through */
03013             case INDEX:
03014             case VECTOR:
03015                 *m++ = INDEX; *m++ = 3; *m++ = *r++;
03016                 break;
03017         }
03018     }
03019     *m++ = 1; *m++ = 1; *m++ = k;
03020 MakeSize:
03021     *mm = m-mm;
03022     if ( !par ) mm[ARGHEAD] = *mm-ARGHEAD;
03023 }
03024
03025 /*
03026     #] ToGeneral :
03027     #[ ToFast :
03028
03029     Checks whether an argument can be converted to fast notation
03030     If this can be done it does it.
03031     Important: m should be allowed to be equal to r!
03032     Return value is 1 if conversion took place.
03033     If there was conversion the answer is in m.

```

```

03034         If there was no conversion m hasn't been touched.
03035     */
03036
03037 int ToFast(WORD *r, WORD *m)
03038 {
03039     WORD i;
03040     if ( *r == ARGHEAD ) { *m++ = -SNUMBER; *m++ = 0; return(1); }
03041     if ( *r != r[ARGHEAD]+ARGHEAD ) return(0); /* > 1 term */
03042     r += ARGHEAD;
03043     if ( *r == 4 ) {
03044         if ( r[2] != 1 || r[1] <= 0 ) return(0);
03045         *m++ = -SNUMBER; *m = ( r[3] < 0 ) ? -r[1] : r[1]; return(1);
03046     }
03047     i = *r - 1;
03048     if ( r[i-1] != 1 || r[i-2] != 1 ) return(0);
03049     if ( r[i] != 3 ) {
03050         if ( r[i] == -3 && r[2] == *r-4 && r[2] == 3 && r[1] == INDEX
03051             && r[3] < MINSPEC ) {}
03052         else return(0);
03053     }
03054     else if ( r[2] != *r - 4 ) return(0);
03055     r++;
03056     if ( *r >= FUNCTION ) {
03057         if ( r[1] <= FUNHEAD ) { *m++ = -*r; return(1); }
03058     }
03059     else if ( *r == SYMBOL ) {
03060         if ( r[1] == 4 && r[3] == 1 )
03061             { *m++ = -SYMBOL; *m++ = r[2]; return(1); }
03062     }
03063     else if ( *r == INDEX ) {
03064         if ( r[1] == 3 ) {
03065             if ( r[2] >= MINSPEC ) {
03066                 if ( r[2] >= 0 && r[2] < AM.OffsetIndex ) *m++ = -SNUMBER;
03067                 else *m++ = -INDEX;
03068             }
03069             else {
03070                 if ( r[5] == -3 ) *m++ = -MINVECTOR;
03071                 else *m++ = -VECTOR;
03072             }
03073             *m++ = r[2];
03074             return(1);
03075         }
03076     }
03077     return(0);
03078 }
03079
03080 /*
03081     #] ToFast :
03082     #[ ToPolyFunGeneral :
03083
03084     Routine forces a polyratfun into general notation if needed.
03085     If no action was needed, the return value is zero.
03086     A positive return value indicates how many arguments were converted.
03087     The new term overwrite the old.
03088 */
03089
03090 WORD ToPolyFunGeneral(PHEAD WORD *term)
03091 {
03092     WORD *t = term+1, *tt, *to, *tol, *termout, *tstop, *tnext;
03093     WORD numarg, i, change = 0;
03094     tstop = term + *term; tstop -= ABS(tstop[-1]);
03095     termout = to = AT.WorkPointer;
03096     to++;
03097     while ( t < tstop ) { /* go through the subterms */
03098         if ( *t == AR.PolyFun ) {
03099             tt = t+FUNHEAD; tnext = t + t[1];
03100             numarg = 0;
03101             while ( tt < tnext ) { numarg++; NEXTARG(tt); }
03102             if ( numarg == 2 ) { /* this needs attention */
03103                 tt = t + FUNHEAD;
03104                 tol = to;
03105                 i = FUNHEAD; NCOPY(to,t,i);
03106                 while ( tt < tnext ) { /* Do the arguments */
03107                     if ( *tt > 0 ) {
03108                         i = *tt; NCOPY(to,tt,i);
03109                     }
03110                     else if ( *tt == -SYMBOL ) {
03111                         tol[1] += 6+ARGHEAD; tol[2] |= MUSTCLEANPRF; change++;
03112                         *to++ = 8+ARGHEAD; *to++ = 0; FILLARG(to);
03113                         *to++ = 8; *to++ = SYMBOL; *to++ = 4; *to++ = tt[1];
03114                         *to++ = 1; *to++ = 1; *to++ = 1; *to++ = 3;
03115                         tt += 2;
03116                     }
03117                     else if ( *tt == -SNUMBER ) {
03118                         if ( tt[1] > 0 ) {
03119                             tol[1] += 2+ARGHEAD; tol[2] |= MUSTCLEANPRF; change++;
03120                             *to++ = 4+ARGHEAD; *to++ = 0; FILLARG(to);

```

```

03121             *to++ = 4; *to++ = tt[1]; *to++ = 1; *to++ = 3;
03122             tt += 2;
03123         }
03124         else if ( tt[1] < 0 ) {
03125             to1[1] += 2+ARGHEAD; to1[2] |= MUSTCLEANPRF; change++;
03126             *to++ = 4+ARGHEAD; *to++ = 0; FILLARG(to);
03127             *to++ = 4; *to++ = -tt[1]; *to++ = 1; *to++ = -3;
03128             tt += 2;
03129         }
03130         else {
03131             MLOCK(ErrorMessageLock);
03132             MesPrint("Internal error: Zero in PolyRatFun");
03133             MUNLOCK(ErrorMessageLock);
03134             Terminate(-1);
03135         }
03136     }
03137 }
03138 t = tnext;
03139 continue;
03140 }
03141 }
03142 i = t[1]; NCOPY(to,t,i)
03143 }
03144 if ( change ) {
03145     tt = term + *term;
03146     while ( t < tt ) *to++ = *t++;
03147     *termout = to - termout;
03148     t = term; i = *termout; tt = termout;
03149     NCOPY(t,tt,i)
03150     AT.WorkPointer = term + *term;
03151 }
03152 return(change);
03153 }
03154
03155 /*
03156     #] ToPolyFunGeneral :
03157     #[ IsLikeVector :
03158
03159     Routine determines whether a function argument is like a vector.
03160     Returnvalue: 1: is vector or index
03161                 0: is not vector or index
03162                 -1: may be an index
03163 */
03164
03165 int IsLikeVector(WORD *arg)
03166 {
03167     WORD *sstop, *t, *tstop;
03168     if ( *arg < 0 ) {
03169         if ( *arg == -VECTOR || *arg == -INDEX ) return(1);
03170         if ( *arg == -SNUMBER && arg[1] >= 0 && arg[1] < AM.OffsetIndex )
03171             return(-1);
03172         return(0);
03173     }
03174     sstop = arg + *arg; arg += ARGHEAD;
03175     while ( arg < sstop ) {
03176         t = arg + *arg;
03177         tstop = t - ABS(t[-1]);
03178         arg++;
03179         while ( arg < tstop ) {
03180             if ( *arg == INDEX ) return(1);
03181             arg += arg[1];
03182         }
03183         arg = t;
03184     }
03185     return(0);
03186 }
03187
03188 /*
03189     #] IsLikeVector :
03190     #[ AreArgsEqual :
03191 */
03192
03193 int AreArgsEqual(WORD *arg1, WORD *arg2)
03194 {
03195     int i;
03196     if ( *arg2 != *arg1 ) return(0);
03197     if ( *arg1 > 0 ) {
03198         i = *arg1;
03199         while ( --i > 0 ) { if ( arg1[i] != arg2[i] ) return(0); }
03200         return(1);
03201     }
03202     else if ( *arg1 <= -FUNCTION ) return(1);
03203     else if ( arg1[1] == arg2[1] ) return(1);
03204     return(0);
03205 }
03206
03207 /*

```

```

03208         #] AreArgsEqual :
03209         #[ CompareArgs :
03210     */
03211
03212 int CompareArgs(WORD *arg1, WORD *arg2)
03213 {
03214     int i1,i2;
03215     if ( *arg1 > 0 ) {
03216         if ( *arg2 < 0 ) return(-1);
03217         i1 = *arg1-ARGHEAD; arg1 += ARGHEAD;
03218         i2 = *arg2-ARGHEAD; arg2 += ARGHEAD;
03219         while ( i1 > 0 && i2 > 0 ) {
03220             if ( *arg1 != *arg2 ) return((int) (*arg1)-(int) (*arg2));
03221             i1--; i2--; arg1++; arg2++;
03222         }
03223         return(i1-i2);
03224     }
03225     else if ( *arg2 > 0 ) return(1);
03226     else {
03227         if ( *arg1 != *arg2 ) {
03228             if ( *arg1 < *arg2 ) return(-1);
03229             else return(1);
03230         }
03231         if ( *arg1 <= -FUNCTION ) return(0);
03232         return((int) (arg1[1])-(int) (arg2[1]));
03233     }
03234 }
03235
03236 /*
03237     #] CompareArgs :
03238     #[ CompArg :
03239
03240     returns 1 if arg1 comes first, -1 if arg2 comes first, 0 if equal
03241 */
03242
03243 int CompArg(WORD *s1, WORD *s2)
03244 {
03245     GETIDENTITY
03246     WORD *st1, *st2, x[7];
03247     int k;
03248     if ( *s1 < 0 ) {
03249         if ( *s2 < 0 ) {
03250             if ( *s1 <= -FUNCTION && *s2 <= -FUNCTION ) {
03251                 if ( *s1 > *s2 ) return(-1);
03252                 if ( *s1 < *s2 ) return(1);
03253                 return(0);
03254             }
03255             if ( *s1 > *s2 ) return(1);
03256             if ( *s1 < *s2 ) return(-1);
03257             if ( *s1 <= -FUNCTION ) return(0);
03258             s1++; s2++;
03259             if ( *s1 > *s2 ) return(1);
03260             if ( *s1 < *s2 ) return(-1);
03261             return(0);
03262         }
03263         x[1] = AT.comsym[3];
03264         x[2] = AT.comnum[1];
03265         x[3] = AT.comnum[3];
03266         x[4] = AT.comind[3];
03267         x[5] = AT.comind[6];
03268         x[6] = AT.comfun[1];
03269         if ( *s1 == -SYMBOL ) {
03270             AT.comsym[3] = s1[1];
03271             st1 = AT.comsym+8; s1 = AT.comsym;
03272         }
03273         else if ( *s1 == -SNUMBER ) {
03274             if ( s1[1] < 0 ) {
03275                 AT.comnum[1] = -s1[1]; AT.comnum[3] = -3;
03276             }
03277             else {
03278                 AT.comnum[1] = s1[1]; AT.comnum[3] = 3;
03279             }
03280             st1 = AT.comnum+4;
03281             s1 = AT.comnum;
03282         }
03283         else if ( *s1 == -INDEX || *s1 == -VECTOR ) {
03284             AT.comind[3] = s1[1]; AT.comind[6] = 3;
03285             st1 = AT.comind+7; s1 = AT.comind;
03286         }
03287         else if ( *s1 == -MINVECTOR ) {
03288             AT.comind[3] = s1[1]; AT.comind[6] = -3;
03289             st1 = AT.comind+7; s1 = AT.comind;
03290         }
03291         else if ( *s1 <= -FUNCTION ) {
03292             AT.comfun[1] = -*s1;
03293             st1 = AT.comfun+FUNHEAD+4; s1 = AT.comfun;
03294         }

```

```

03295 /*
03296         Symmetrize during compilation of id statement when properorder
03297         needs this one. Code added 10-nov-2001
03298 */
03299     else if ( *s1 == -ARGWILD ) {
03300         return(-1);
03301     }
03302     else { goto argerror; }
03303     st2 = s2 + *s2; s2 += ARGHEAD;
03304     goto docompare;
03305 }
03306 else if ( *s2 < 0 ) {
03307     x[1] = AT.comsym[3];
03308     x[2] = AT.comnum[1];
03309     x[3] = AT.comnum[3];
03310     x[4] = AT.comind[3];
03311     x[5] = AT.comind[6];
03312     x[6] = AT.comfun[1];
03313     if ( *s2 == -SYMBOL ) {
03314         AT.comsym[3] = s2[1];
03315         st2 = AT.comsym+8; s2 = AT.comsym;
03316     }
03317     else if ( *s2 == -SNUMBER ) {
03318         if ( s2[1] < 0 ) {
03319             AT.comnum[1] = -s2[1]; AT.comnum[3] = -3;
03320             st2 = AT.comnum+4;
03321         }
03322         else if ( s2[1] == 0 ) {
03323             st2 = AT.comnum+4; s2 = st2;
03324         }
03325         else {
03326             AT.comnum[1] = s2[1]; AT.comnum[3] = 3;
03327             st2 = AT.comnum+4;
03328         }
03329         s2 = AT.comnum;
03330     }
03331     else if ( *s2 == -INDEX || *s2 == -VECTOR ) {
03332         AT.comind[3] = s2[1]; AT.comind[6] = 3;
03333         st2 = AT.comind+7; s2 = AT.comind;
03334     }
03335     else if ( *s2 == -MINVECTOR ) {
03336         AT.comind[3] = s2[1]; AT.comind[6] = -3;
03337         st2 = AT.comind+7; s2 = AT.comind;
03338     }
03339     else if ( *s2 <= -FUNCTION ) {
03340         AT.comfun[1] = -*s2;
03341         st2 = AT.comfun+FUNHEAD+4; s2 = AT.comfun;
03342     }
03343 }
03344 /*
03345         Symmetrize during compilation of id statement when properorder
03346         needs this one. Code added 10-nov-2001
03347 */
03348     else if ( *s2 == -ARGWILD ) {
03349         return(1);
03350     }
03351     else { goto argerror; }
03352     st1 = s1 + *s1; s1 += ARGHEAD;
03353     goto docompare;
03354 }
03355 else {
03356     x[1] = AT.comsym[3];
03357     x[2] = AT.comnum[1];
03358     x[3] = AT.comnum[3];
03359     x[4] = AT.comind[3];
03360     x[5] = AT.comind[6];
03361     x[6] = AT.comfun[1];
03362     st1 = s1 + *s1; st2 = s2 + *s2;
03363     s1 += ARGHEAD; s2 += ARGHEAD;
03364 docompare:
03365     while ( s1 < st1 && s2 < st2 ) {
03366         if ( ( k = CompareTerms(BHEAD s1,s2,(WORD)2) ) != 0 ) {
03367             AT.comsym[3] = x[1];
03368             AT.comnum[1] = x[2];
03369             AT.comnum[3] = x[3];
03370             AT.comind[3] = x[4];
03371             AT.comind[6] = x[5];
03372             AT.comfun[1] = x[6];
03373             return(-k);
03374         }
03375         s1 += *s1; s2 += *s2;
03376     }
03377     AT.comsym[3] = x[1];
03378     AT.comnum[1] = x[2];
03379     AT.comnum[3] = x[3];
03380     AT.comind[3] = x[4];
03381     AT.comind[6] = x[5];
03382     AT.comfun[1] = x[6];

```

```

03382         if ( s1 < st1 ) return(1);
03383         if ( s2 < st2 ) return(-1);
03384     }
03385     return(0);
03386
03387 argerror:
03388     MesPrint("Illegal type of short function argument in Normalize");
03389     Terminate(-1); return(0);
03390 }
03391
03392 /*
03393     #] CompArg :
03394     #[ TimeWallClock :
03395 */
03396
03397 #ifdef HAVE_CLOCK_GETTIME
03398 #include <time.h> /* for clock_gettime() */
03399 #else
03400 #ifdef HAVE_GETTIMEOFDAY
03401 #include <sys/time.h> /* for gettimeofday() */
03402 #else
03403 #include <sys/timeb.h> /* for ftime() */
03404 #endif
03405 #endif
03406
03413 LONG TimeWallClock(WORD par)
03414 {
03415     /*
03416      * NOTE: this function is not thread-safe. Operations on tp are not atomic.
03417      */
03418
03419     #ifdef HAVE_CLOCK_GETTIME
03420         struct timespec ts;
03421         clock_gettime(CLOCK_MONOTONIC, &ts);
03422
03423         if ( par ) {
03424             return(((LONG)(ts.tv_sec)-AM.OldSecTime)*100 +
03425                 ((LONG)(ts.tv_nsec / 1000000)-AM.OldMilliTime)/10);
03426         }
03427         else {
03428             AM.OldSecTime = (LONG)(ts.tv_sec);
03429             AM.OldMilliTime = (LONG)(ts.tv_nsec / 1000000);
03430             return(0L);
03431         }
03432     #else
03433     #ifdef HAVE_GETTIMEOFDAY
03434         struct timeval t;
03435         LONG sec, msec;
03436         gettimeofday(&t, NULL);
03437         sec = (LONG)t.tv_sec;
03438         msec = (LONG)(t.tv_usec/1000);
03439         if ( par ) {
03440             return (sec-AM.OldSecTime)*100 + (msec-AM.OldMilliTime)/10;
03441         }
03442         else {
03443             AM.OldSecTime = sec;
03444             AM.OldMilliTime = msec;
03445             return(0L);
03446         }
03447     #else
03448         struct timeb tp;
03449         ftime(&tp);
03450
03451         if ( par ) {
03452             return(((LONG)(tp.time)-AM.OldSecTime)*100 +
03453                 ((LONG)(tp.millitm)-AM.OldMilliTime)/10);
03454         }
03455         else {
03456             AM.OldSecTime = (LONG)(tp.time);
03457             AM.OldMilliTime = (LONG)(tp.millitm);
03458             return(0L);
03459         }
03460     #endif
03461 #endif
03462 }
03463
03464 /*
03465     #] TimeWallClock :
03466     #[ TimeChildren :
03467 */
03468
03469 LONG TimeChildren(WORD par)
03470 {
03471     if ( par ) return(Timer(1)-AM.OldChildTime);
03472     AM.OldChildTime = Timer(1);
03473     return(0L);
03474 }

```

```

03475
03476 /*
03477     #] TimeChildren :
03478     #[ TimeCPU :
03479 */
03480
03487 LONG TimeCPU(WORD par)
03488 {
03489     GETIDENTITY
03490     if ( par ) return(Timer(0)-AR.OldTime);
03491     AR.OldTime = Timer(0);
03492     return(0L);
03493 }
03494
03495 /*
03496     #] TimeCPU :
03497     #[ Timer :
03498 */
03499 #if defined(WINDOWS)
03500
03501 LONG Timer(int par)
03502 {
03503     #ifndef WITHPTHREADS
03504         static int initialized = 0;
03505         static HANDLE hProcess;
03506         FILETIME ftCreate, ftExit, ftKernel, ftUser;
03507         DUMMYUSE(par);
03508
03509         if ( !initialized ) {
03510             hProcess = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, GetCurrentProcessId());
03511         }
03512         if ( GetProcessTimes(hProcess, &ftCreate, &ftExit, &ftKernel, &ftUser) ) {
03513             PFILETIME pftKernel = &ftKernel; /* to avoid strict-aliasing rule warnings */
03514             PFILETIME pftUser = &ftUser;
03515             __int64 t = *(__int64 *)pftKernel + *(__int64 *)pftUser; /* in 100 nsec. */
03516             return (LONG)(t / 10000); /* in msec. */
03517         }
03518         return 0;
03519     #else
03520         LONG lResult = 0;
03521         HANDLE hThread;
03522         FILETIME ftCreate, ftExit, ftKernel, ftUser;
03523         DUMMYUSE(par);
03524
03525         hThread = OpenThread(THREAD_QUERY_INFORMATION, FALSE, GetCurrentThreadId());
03526         if ( hThread ) {
03527             if ( GetThreadTimes(hThread, &ftCreate, &ftExit, &ftKernel, &ftUser) ) {
03528                 PFILETIME pftKernel = &ftKernel; /* to avoid strict-aliasing rule warnings */
03529                 PFILETIME pftUser = &ftUser;
03530                 __int64 t = *(__int64 *)pftKernel + *(__int64 *)pftUser; /* in 100 nsec. */
03531                 lResult = (LONG)(t / 10000); /* in msec. */
03532             }
03533             CloseHandle(hThread);
03534         }
03535         return lResult;
03536     #endif
03537 }
03538
03539 #elif defined(UNIX)
03540 #include <sys/time.h>
03541 #include <sys/resource.h>
03542 #ifdef WITHPOSIXCLOCK
03543 #include <time.h>
03544 /*
03545     And include -lrt in the link statement (on blade02)
03546 */
03547 #endif
03548
03549 LONG Timer(int par)
03550 {
03551     #ifdef WITHPOSIXCLOCK
03552     /*
03553         Only to be used in combination with WITHPTHREADS
03554         This clock seems to be supported by the standard.
03555         The getusage clock returns according to the standard only the combined
03556         time of the whole process. But in older versions of Linux LinuxThreads
03557         is used which gives a separate id to each thread and individual timings.
03558         In NPTL we get, according to the standard, one combined timing.
03559         To get individual timings we need to use
03560         clock_gettime(CLOCK_THREAD_CPUTIME_ID, &timing)
03561         with timing of the time
03562         struct timespec {
03563             time_t tv_sec;           Seconds.
03564             long tv_nsec;           Nanoseconds.
03565         };
03566     */
03567     */

```



```

03568     struct timespec t;
03569     if ( par == 0 ) {
03570         if ( clock_gettime(CLOCK_THREAD_CPUTIME_ID, &t) ) {
03571             MesPrint("Error in getting timing information");
03572         }
03573         return (LONG)t.tv_sec * 1000 + (LONG)t.tv_nsec / 1000000;
03574     }
03575     return(0);
03576 #else
03577     struct rusage rusage;
03578     if ( par == 1 ) {
03579         getrusage(RUSAGE_CHILDREN, &rusage);
03580         return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
03581             + (rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03582     }
03583     else {
03584         getrusage(RUSAGE_SELF, &rusage);
03585         return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
03586             + (rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03587     }
03588 #endif
03589 }
03590
03591 #elif defined(SUN)
03592 #define _TIME_T_
03593 #include <sys/time.h>
03594 #include <sys/resource.h>
03595
03596 LONG Timer(int par)
03597 {
03598     struct rusage rusage;
03599     if ( par == 1 ) {
03600         getrusage(RUSAGE_CHILDREN, &rusage);
03601         return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
03602             + (rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03603     }
03604     else {
03605         getrusage(RUSAGE_SELF, &rusage);
03606         return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
03607             + (rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03608     }
03609 }
03610
03611 #elif defined(RS6K)
03612 #include <sys/time.h>
03613 #include <sys/resource.h>
03614
03615 LONG Timer(int par)
03616 {
03617     struct rusage rusage;
03618     if ( par == 1 ) {
03619         getrusage(RUSAGE_CHILDREN, &rusage);
03620         return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
03621             + (rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03622     }
03623     else {
03624         getrusage(RUSAGE_SELF, &rusage);
03625         return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
03626             + (rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03627     }
03628 }
03629
03630 #elif defined(ANSI)
03631 LONG Timer(int par)
03632 {
03633     #ifdef ALPHA
03634         /* clock_t t, tikken = clock(); */
03635         /* MesPrint("ALPHA-clock = %l", (LONG)tikken); */
03636         /* t = tikken % CLOCKS_PER_SEC; */
03637         /* tikken /= CLOCKS_PER_SEC; */
03638         /* tikken *= 1000; */
03639         /* tikken += (t*1000)/CLOCKS_PER_SEC; */
03640         /* return((LONG)tikken); */
03641         /* #define _TIME_T_ */
03642     #include <sys/time.h>
03643     #include <sys/resource.h>
03644     struct rusage rusage;
03645     if ( par == 1 ) {
03646         getrusage(RUSAGE_CHILDREN, &rusage);
03647         return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
03648             + (rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03649     }
03650     else {
03651         getrusage(RUSAGE_SELF, &rusage);
03652         return(((LONG)(rusage.ru_utime.tv_sec)+(LONG)(rusage.ru_stime.tv_sec))*1000
03653             + (rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03654     }

```

```

03655 #else
03656 #ifndef DEC_STATION
03657     clock_t tikken = clock();
03658     return((LONG)tikken/1000);
03659 #else
03660     clock_t t, tikken = clock();
03661     t = tikken % CLK_TCK;
03662     tikken /= CLK_TCK;
03663     tikken *= 1000;
03664     tikken += (t*1000)/CLK_TCK;
03665     return(tikken);
03666 #endif
03667 #endif
03668 }
03669 #elif defined(VMS)
03670
03671 #include <time.h>
03672 void times(tbuffer_t *buffer);
03673
03674 LONG
03675 Timer(int par)
03676 {
03677     tbuffer_t buffer;
03678     if ( par == 1 ) { return(0); }
03679     else {
03680         times(&buffer);
03681         return(buffer.proc_user_time * 10);
03682     }
03683 }
03684
03685 #elif defined(MBSD)
03686
03687 #ifndef MICROTIME
03688 /*
03689     There is only a CP time clock in microseconds here
03690     This can cause problems with AO.wrap around
03691 */
03692 #else
03693 #ifdef mBSD2
03694 #include <sys/types.h>
03695 #include <sys/times.h>
03696 #include <time.h>
03697 LONG pretime = 0;
03698 #else
03699 #define _TIME_T_
03700 #include <sys/time.h>
03701 #include <sys/resource.h>
03702 #endif
03703 #endif
03704
03705 LONG Timer(int par)
03706 {
03707     #ifdef MICROTIME
03708         LONG t;
03709         if ( par == 1 ) { return(0); }
03710         t = clock();
03711         if ( ( AO.wrapnum & 1 ) != 0 ) t ^= 0x80000000;
03712         if ( t < 0 ) {
03713             t ^= 0x80000000;
03714             wrapnum++;
03715             AO.wrap += 2147584;
03716         }
03717         return(AO.wrap+(t/1000));
03718     #else
03719     #ifdef mBSD2
03720         struct tms buffer;
03721         LONG ret;
03722         ULONG a1, a2, a3, a4;
03723         if ( par == 1 ) { return(0); }
03724         times(&buffer);
03725         a1 = (ULONG)buffer.tms_utime;
03726         a2 = a1 » 16;
03727         a3 = a1 & 0xFFFFL;
03728         a3 *= 1000;
03729         a2 = 1000*a2 + (a3 » 16);
03730         a3 &= 0xFFFFL;
03731         a4 = a2/CLK_TCK;
03732         a2 %= CLK_TCK;
03733         a3 += a2 « 16;
03734         ret = (LONG)((a4 « 16) + a3 / CLK_TCK);
03735         /* ret = ((LONG)buffer.tms_utime * 1000)/CLK_TCK; */
03736         return(ret);
03737     #else
03738     #ifdef REALTIME
03739         struct timeval tp;
03740         struct timezone tzp;
03741         if ( par == 1 ) { return(0); }

```

```

03742     gettimeofday(&tp,&tzp); */
03743     return(tp.tv_sec*1000+tp.tv_usec/1000);
03744 #else
03745     struct rusage rusage;
03746     if ( par == 1 ) {
03747         getrusage(RUSAGE_CHILDREN,&rusage);
03748         return((rusage.ru_utime.tv_sec+rusage.ru_stime.tv_sec)*1000
03749             +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03750     }
03751     else {
03752         getrusage(RUSAGE_SELF,&rusage);
03753         return((rusage.ru_utime.tv_sec+rusage.ru_stime.tv_sec)*1000
03754             +(rusage.ru_utime.tv_usec/1000+rusage.ru_stime.tv_usec/1000));
03755     }
03756 #endif
03757 #endif
03758 #endif
03759 }
03760
03761 #endif
03762
03763 /*
03764     #] Timer :
03765     #[ Crash :
03766
03767     Routine for debugging purposes
03768 */
03769
03770 int Crash(void)
03771 {
03772     int retval;
03773 #ifdef DEBUGGING
03774     int *zero = 0;
03775     retval = *zero;
03776 #else
03777     retval = 0;
03778 #endif
03779     return(retval);
03780 }
03781
03782 /*
03783     #] Crash :
03784     #[ TestTerm :
03785 */
03786
03798 int TestTerm(WORD *term)
03799 {
03800     int errorcode = 0, coeffsize;
03801     WORD *t, *tt, *tstop, *endterm, *targ, *targstop, *funstop, *argterm;
03802     endterm = term + *term;
03803     coeffsize = ABS(endterm[-1]);
03804     if ( coeffsize >= *term ) {
03805         MLOCK(ErrorMessageLock);
03806         MesPrint("TestTerm: Internal inconsistency in term. Coefficient too big.");
03807         MUNLOCK(ErrorMessageLock);
03808         errorcode = 1;
03809         goto finish;
03810     }
03811     if ( ( coeffsize < 3 ) || ( ( coeffsize & 1 ) != 1 ) ) {
03812         MLOCK(ErrorMessageLock);
03813         MesPrint("TestTerm: Internal inconsistency in term. Wrong size coefficient.");
03814         MUNLOCK(ErrorMessageLock);
03815         errorcode = 2;
03816         goto finish;
03817     }
03818     t = term+1;
03819     tstop = endterm - coeffsize;
03820     while ( t < tstop ) {
03821         switch ( *t ) {
03822             case SYMBOL:
03823             case DOTPRODUCT:
03824             case INDEX:
03825             case VECTOR:
03826             case DELTA:
03827             case HAAKJE:
03828                 break;
03829             case SNUMBER:
03830             case LNUMBER:
03831                 MLOCK(ErrorMessageLock);
03832                 MesPrint("TestTerm: Internal inconsistency in term. L or S number");
03833                 MUNLOCK(ErrorMessageLock);
03834                 errorcode = 3;
03835                 goto finish;
03836             case EXPRESSION:
03837             case SUBEXPRESSION:
03838             case DOLLAREXPRESSION:

```

```

03840 /*
03841     MLOCK(ErrorMessageLock);
03842     MesPrint("TestTerm: Internal inconsistency in term. Expression survives.");
03843     MUNLOCK(ErrorMessageLock);
03844     errorcode = 4;
03845     goto finish;
03846 */
03847     break;
03848 case SETSET:
03849 case MINVECTOR:
03850 case SETEXP:
03851 case ARGFIELD:
03852     MLOCK(ErrorMessageLock);
03853     MesPrint("TestTerm: Internal inconsistency in term. Illegal subterm.");
03854     MUNLOCK(ErrorMessageLock);
03855     errorcode = 5;
03856     goto finish;
03857     break;
03858 case ARGWILD:
03859     break;
03860 default:
03861     if ( *t <= 0 ) {
03862         MLOCK(ErrorMessageLock);
03863         MesPrint("TestTerm: Internal inconsistency in term. Illegal subterm number.");
03864         MUNLOCK(ErrorMessageLock);
03865         errorcode = 6;
03866         goto finish;
03867     }
03868 /*
03869     This is a regular function.
03870 */
03871     if ( *t-FUNCTION >= NumFunctions ) {
03872         MLOCK(ErrorMessageLock);
03873         MesPrint("TestTerm: Internal inconsistency in term. Illegal function number");
03874         MUNLOCK(ErrorMessageLock);
03875         errorcode = 7;
03876         goto finish;
03877     }
03878     funstop = t + t[1];
03879     if ( funstop > tstop ) goto subtermsize;
03880     if ( t[2] != 0 ) {
03881         MLOCK(ErrorMessageLock);
03882         MesPrint("TestTerm: Internal inconsistency in term. Dirty flag nonzero.");
03883         MUNLOCK(ErrorMessageLock);
03884         errorcode = 8;
03885         goto finish;
03886     }
03887     targ = t + FUNHEAD;
03888     if ( targ > funstop ) {
03889         MLOCK(ErrorMessageLock);
03890         MesPrint("TestTerm: Internal inconsistency in term. Illegal function size.");
03891         MUNLOCK(ErrorMessageLock);
03892         errorcode = 9;
03893         goto finish;
03894     }
03895     if ( functions[*t-FUNCTION].spec >= TENSORFUNCTION ) {
03896     }
03897     else {
03898         while ( targ < funstop ) {
03899             if ( *targ < 0 ) {
03900                 if ( *targ <= -(FUNCTION+NumFunctions) ) {
03901                     MLOCK(ErrorMessageLock);
03902                     MesPrint("TestTerm: Internal inconsistency in term. Illegal function
number in argument.");
03903                     MUNLOCK(ErrorMessageLock);
03904                     errorcode = 10;
03905                     goto finish;
03906                 }
03907                 if ( *targ <= -FUNCTION ) { targ++; }
03908                 else {
03909                     if ( ( *targ != -SYMBOL ) && ( *targ != -VECTOR )
03910                     && ( *targ != -MINVECTOR )
03911                     && ( *targ != -SNUMBER )
03912                     && ( *targ != -ARGWILD )
03913                     && ( *targ != -INDEX ) ) {
03914                         MLOCK(ErrorMessageLock);
03915                         MesPrint("TestTerm: Internal inconsistency in term. Illegal object in
argument.");
03916                         MUNLOCK(ErrorMessageLock);
03917                         errorcode = 11;
03918                         goto finish;
03919                     }
03920                     targ += 2;
03921                 }
03922             }
03923             else if ( ( *targ < ARGHEAD ) || ( targ+*targ > funstop ) ) {
03924                 MLOCK(ErrorMessageLock);

```

```

03925         MesPrint("TestTerm: Internal inconsistency in term. Illegal size of
argument.");
03926         MUNLOCK(ErrorMessageLock);
03927         errorcode = 12;
03928         goto finish;
03929     }
03930     else if ( targ[1] != 0 ) {
03931         MLOCK(ErrorMessageLock);
03932         MesPrint("TestTerm: Internal inconsistency in term. Dirty flag in argument.");
03933         MUNLOCK(ErrorMessageLock);
03934         errorcode = 13;
03935         goto finish;
03936     }
03937     else {
03938         targstop = targ + *targ;
03939         argterm = targ + ARGHEAD;
03940         while ( argterm < targstop ) {
03941             if ( ( *argterm < 4 ) || ( argterm + *argterm > targstop ) ) {
03942                 MLOCK(ErrorMessageLock);
03943                 MesPrint("TestTerm: Internal inconsistency in term. Illegal termsize
in argument.");
03944                 MUNLOCK(ErrorMessageLock);
03945                 errorcode = 14;
03946                 goto finish;
03947             }
03948             if ( TestTerm(argterm) != 0 ) {
03949                 MLOCK(ErrorMessageLock);
03950                 MesPrint("TestTerm: Internal inconsistency in term. Called from
TestTerm.");
03951                 MUNLOCK(ErrorMessageLock);
03952                 errorcode = 15;
03953                 goto finish;
03954             }
03955             argterm += *argterm;
03956         }
03957         targ = targstop;
03958     }
03959 }
03960 }
03961 break;
03962 }
03963 tt = t + t[1];
03964 if ( tt > tstop ) {
03965     subtermsize:
03966         MLOCK(ErrorMessageLock);
03967         MesPrint("TestTerm: Internal inconsistency in term. Illegal subterm size.");
03968         MUNLOCK(ErrorMessageLock);
03969         errorcode = 100;
03970         goto finish;
03971     }
03972     t = tt;
03973 }
03974 return(errorcode);
03975 finish:
03976 return(errorcode);
03977 }
03978
03979 /*
03980     #] TestTerm :
03981     #[ DistrN :
03982 */
03983
03984 int DistrN(int n, int *cpl, int ncpl, int *scratch)
03985 {
03986     /*
03987     Divides n objects over ncpl bins (cpl), each time returning one
03988     of those distributions until there are no more after which the
03989     routine returns the value zero (otherwise one).
03990     The array scratch (size n) is kept for the intermediate information.
03991     The whole starts with scratch[0] == -2;
03992     */
03993     int i, j;
03994     if ( ncpl == 0 ) {
03995         if ( scratch[0] == -2 ) { scratch[0] = 0; return(1); }
03996         else return(0);
03997     }
03998     if ( scratch[0] == ncpl-1 ) {
03999         return(0);
04000     }
04001     else if ( scratch[0] == -2 ) {
04002         for ( i = 0; i < n; i++ ) scratch[i] = 0;
04003     }
04004     else {
04005         j = n-1;
04006         while ( j >= 0 ) {
04007             scratch[j]++;
04008             if ( scratch[j] < ncpl ) break;

```

```

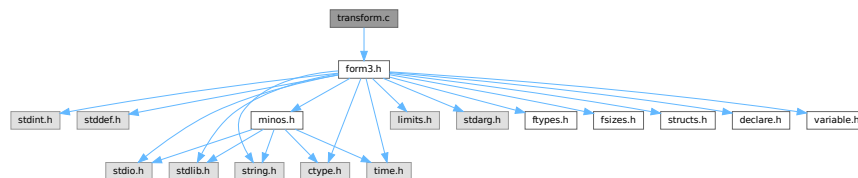
04009         j--;
04010     }
04011     j++;
04012     while ( j < n ) { scratch[j] = scratch[j-1]; j++; }
04013 }
04014 for ( i = 0; i < ncpl; i++ ) cpl[i] = 0;
04015 for ( i = 0; i < n; i++ ) { cpl[scratch[i]]++; }
04016 return(1);
04017 }
04018
04019 /*
04020     #] DistrN :
04021     #] Mixed :
04022 */

```

4.149 transform.c File Reference

```
#include "form3.h"
```

Include dependency graph for transform.c:



Functions

- int [CoTransform](#) (UBYTE *in)
- int [RunTransform](#) (PHEAD WORD *term, WORD *params)
- int [RunEncode](#) (PHEAD WORD *fun, WORD *args, WORD *info)
- int [RunDecode](#) (PHEAD WORD *fun, WORD *args, WORD *info)
- int [RunReplace](#) (PHEAD WORD *fun, WORD *args, WORD *info)
- int [RunImplode](#) (WORD *fun, WORD *args)
- int [RunExplode](#) (PHEAD WORD *fun, WORD *args)
- int [RunPermute](#) (PHEAD WORD *fun, WORD *args, WORD *info)
- int [RunReverse](#) (PHEAD WORD *fun, WORD *args)
- int [RunDedup](#) (PHEAD WORD *fun, WORD *args)
- int [RunCycle](#) (PHEAD WORD *fun, WORD *args, WORD *info)
- int [RunAddArg](#) (PHEAD WORD *fun, WORD *args)
- int [RunMulArg](#) (PHEAD WORD *fun, WORD *args)
- int [RunIsLyndon](#) (PHEAD WORD *fun, WORD *args, int par)
- WORD [RunToLyndon](#) (PHEAD WORD *fun, WORD *args, int par)
- int [RunDropArg](#) (PHEAD WORD *fun, WORD *args)
- int [RunSelectArg](#) (PHEAD WORD *fun, WORD *args)
- int [RunZtoHArg](#) (PHEAD WORD *fun, WORD *args)
- int [RunHtoZArg](#) (PHEAD WORD *fun, WORD *args)
- int [TestArgNum](#) (int n, int totarg, WORD *args)
- WORD [PutArgInScratch](#) (WORD *arg, UWORD *scrat)
- UBYTE * [ReadRange](#) (UBYTE *s, WORD *out, int par)
- int [FindRange](#) (PHEAD WORD *args, WORD *arg1, WORD *arg2, WORD totarg)

4.149.1 Detailed Description

Routines that deal with the transform statement and its varieties.

Definition in file [transform.c](#).

4.149.2 Function Documentation

4.149.2.1 CoTransform()

```
int CoTransform (
    UBYTE * in )
```

Definition at line 87 of file [transform.c](#).

4.149.2.2 RunTransform()

```
int RunTransform (
    PHEAD WORD * term,
    WORD * params )
```

Definition at line 750 of file [transform.c](#).

4.149.2.3 RunEncode()

```
int RunEncode (
    PHEAD WORD * fun,
    WORD * args,
    WORD * info )
```

Definition at line 1006 of file [transform.c](#).

4.149.2.4 RunDecode()

```
int RunDecode (
    PHEAD WORD * fun,
    WORD * args,
    WORD * info )
```

Definition at line 1193 of file [transform.c](#).

4.149.2.5 RunReplace()

```
int RunReplace (
    PHEAD WORD * fun,
    WORD * args,
    WORD * info )
```

Definition at line 1363 of file [transform.c](#).

4.149.2.6 RunImplode()

```
int RunImplode (
    WORD * fun,
    WORD * args )
```

Definition at line [1841](#) of file [transform.c](#).

4.149.2.7 RunExplode()

```
int RunExplode (
    PHEAD WORD * fun,
    WORD * args )
```

Definition at line [2042](#) of file [transform.c](#).

4.149.2.8 RunPermute()

```
int RunPermute (
    PHEAD WORD * fun,
    WORD * args,
    WORD * info )
```

Definition at line [2156](#) of file [transform.c](#).

4.149.2.9 RunReverse()

```
int RunReverse (
    PHEAD WORD * fun,
    WORD * args )
```

Definition at line [2383](#) of file [transform.c](#).

4.149.2.10 RunDedup()

```
int RunDedup (
    PHEAD WORD * fun,
    WORD * args )
```

Definition at line [2470](#) of file [transform.c](#).

4.149.2.11 RunCycle()

```
int RunCycle (
    PHEAD WORD * fun,
    WORD * args,
    WORD * info )
```

Definition at line [2559](#) of file [transform.c](#).

4.149.2.12 RunAddArg()

```
int RunAddArg (
    PHEAD WORD * fun,
    WORD * args )
```

Definition at line 2711 of file [transform.c](#).

4.149.2.13 RunMulArg()

```
int RunMulArg (
    PHEAD WORD * fun,
    WORD * args )
```

Definition at line 2800 of file [transform.c](#).

4.149.2.14 RunIsLyndon()

```
int RunIsLyndon (
    PHEAD WORD * fun,
    WORD * args,
    int par )
```

Definition at line 2941 of file [transform.c](#).

4.149.2.15 RunToLyndon()

```
WORD RunToLyndon (
    PHEAD WORD * fun,
    WORD * args,
    int par )
```

Definition at line 3019 of file [transform.c](#).

4.149.2.16 RunDropArg()

```
int RunDropArg (
    PHEAD WORD * fun,
    WORD * args )
```

Definition at line 3131 of file [transform.c](#).

4.149.2.17 RunSelectArg()

```
int RunSelectArg (
    PHEAD WORD * fun,
    WORD * args )
```

Definition at line 3157 of file [transform.c](#).

4.149.2.18 RunZtoHArg()

```
int RunZtoHArg (
    PHEAD WORD * fun,
    WORD * args )
```

Definition at line 3185 of file [transform.c](#).

4.149.2.19 RunHtoZArg()

```
int RunHtoZArg (
    PHEAD WORD * fun,
    WORD * args )
```

Definition at line 3241 of file [transform.c](#).

4.149.2.20 TestArgNum()

```
int TestArgNum (
    int n,
    int totarg,
    WORD * args )
```

Definition at line 3307 of file [transform.c](#).

4.149.2.21 PutArgInScratch()

```
WORD PutArgInScratch (
    WORD * arg,
    UWORD * scrat )
```

Definition at line 3376 of file [transform.c](#).

4.149.2.22 ReadRange()

```
UBYTE * ReadRange (
    UBYTE * s,
    WORD * out,
    int par )
```

Definition at line 3412 of file [transform.c](#).

4.149.2.23 FindRange()

```
int FindRange (
    PHEAD WORD * args,
    WORD * arg1,
    WORD * arg2,
    WORD totarg )
```

Definition at line 3572 of file [transform.c](#).

4.150 transform.c

[Go to the documentation of this file.](#)

```

00001
00005 /* #[] License : */
00006 /*
00007 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00008 * When using this file you are requested to refer to the publication
00009 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010 * This is considered a matter of courtesy as the development was paid
00011 * for by FOM the Dutch physics granting agency and we would like to
00012 * be able to track its scientific use to convince FOM of its value
00013 * for the community.
00014 *
00015 * This file is part of FORM.
00016 *
00017 * FORM is free software: you can redistribute it and/or modify it under the
00018 * terms of the GNU General Public License as published by the Free Software
00019 * Foundation, either version 3 of the License, or (at your option) any later
00020 * version.
00021 *
00022 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00025 * details.
00026 *
00027 * You should have received a copy of the GNU General Public License along
00028 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00029 */
00030 /* #[] License : */
00031 /*
00032     #[] Includes : transform.c
00033 */
00034
00035 #include "form3.h"
00036
00037 /*
00038     #[] Includes :
00039     #[] Transform :
00040     #[] Intro :
00041
00042     Here are the routines for the transform statement. This is a
00043     group of transformations on function arguments or groups of
00044     function arguments. The purpose of this command is that it
00045     avoids repetitive pattern matching.
00046     Syntax:
00047         Transform,SetOfFunctions,OneOrMoreTransformations;
00048     Each transformation is given by
00049         Replace(argfirst,arglast)=(,,)
00050         Encode(argfirst,arglast):base=#
00051         Decode(argfirst,arglast):base=#
00052         Implode(argfirst,arglast)
00053         Explode(argfirst,arglast)
00054         Permute(cycle)(cycle)(cycle)...(cycle)
00055         Reverse(argfirst,arglast)
00056         Dedup(argfirst,arglast)
00057         Cycle(argfirst,arglast)=+/-num
00058         IsLyndon(argfirst,arglast)=(yes,no)
00059         ToLyndon(argfirst,arglast)=(yes,no)
00060     In replace the extra information is
00061         a replace_() without the name of the replace_ function.
00062         This can be as in (0,1,1,0) or (xarg_,1-xarg_) to indicate
00063         a symbolic argument or (x,y,y,x) to exchange x and y, etc.
00064     In Encode and Decode argfirst is the most significant 'word' and
00065     arglast is the least significant 'word'.
00066     Note that we need to introduce the generic symbolic arguments xarg_,
00067     parg_, iarg_ and farg_.
00068     Examples:
00069         Transform,{H,E}
00070             ,Replace(1:'WEIGHT')=(0,1,1,0)
00071             ,Encode(1:'WEIGHT')=base(2);
00072         Transform,{H,E}
00073             ,Decode(1:'WEIGHT')=base(3)
00074             ,Replace(1:'WEIGHT')=(2,-1,1,0,0,1);
00075     Others that can be added:
00076         symmetrize?
00077
00078     6-may-2016: Changed MAXPOSITIVE2 into MAXPOSITIVE4. This makes room
00079                 for the use of dollar variables as arguments.
00080
00081     #[] Intro :
00082     #[] CoTransform :
00083 */
00084
00085 static WORD tranarray[10] = { SUBEXPRESSION, SUBEXPSIZE, 0, 1, 0, 0, 0, 0, 0, 0 };

```

```

00086
00087 int CoTransform(UBYTE *in)
00088 {
00089     GETIDENTITY
00090     UBYTE *s = in, c, *ss, *Tempbuf;
00091     WORD number, type, i, *work = AT.WorkPointer+2, *wp, range[2], one = 1;
00092     WORD numdol, *wstart;
00093     int error = 0, irhs;
00094     LONG x;
00095     while ( *in == ',' ) in++;
00096     wp = work + 1;
00097 /*
00098     #[ Sets :
00099
00100     First the set specification(s). No sets means all functions (dangerous!)
00101 */
00102     for(;;) {
00103         if ( *in == '{' ) {
00104             s = in+1;
00105             SKIPBRA2(in)
00106             number = DoTempSet(s,in);
00107             in++;
00108             if ( *in != ',' ) {
00109                 c = in[1]; in[1] = 0;
00110                 MesPrint("& %s: A set in a transform statement should be followed by a comma",s);
00111                 in[1] = c; in++;
00112                 if ( error == 0 ) error = 1;
00113             }
00114         }
00115         else if ( *in == '[' || FG.cTable[*in] == 0 ) {
00116             s = in;
00117             in = SkipAName(in);
00118             if ( *in != ',' ) break;
00119             c = *in; *in = 0;
00120             type = GetName(AC.varnames,s,&number,NOAUTO);
00121             if ( type == CFUNCTION ) {
00122 #ifdef WITHFLOAT
00123                 if ( (number+FUNCTION) == FLOATFUN ) {
00124                     MesPrint("&Illegal use of a transform statement and float_");
00125                     if ( error == 0 ) error = 1;
00126                 }
00127 #endif
00128                 number += MAXVARIABLES + FUNCTION; }
00129             else if ( type != CSET ) {
00130                 MesPrint("& %s: A transform statement starts with sets of functions",s);
00131                 if ( error == 0 ) error = 1;
00132             }
00133             *in++ = c;
00134         }
00135         else {
00136             MesPrint("&Illegal syntax in Transform statement",s);
00137             if ( error == 0 ) error = 1;
00138             return(error);
00139         }
00140         if ( number >= 0 ) {
00141             if ( number < MAXVARIABLES ) {
00142 /*
00143                 Check that this is a set of functions
00144 */
00145                 if ( Sets[number].type != CFUNCTION ) {
00146                     MesPrint("&A set in a transform statement should be a set of functions");
00147                     if ( error == 0 ) error = 1;
00148                 }
00149 #ifdef WITHFLOAT
00150                 WORD *r1, *r2;
00151                 r1 = SetElements + Sets[number].first;
00152                 r2 = SetElements + Sets[number].last;
00153                 while ( r1 < r2 ) {
00154                     if ( *r1++ == FLOATFUN ) {
00155                         MesPrint("&Illegal use of a transform statement and float_");
00156                         if ( error == 0 ) error = 1;
00157                     }
00158                 }
00159 #endif
00160             }
00161             else if ( error == 0 ) error = 1;
00162 /*
00163             Now write the number to the right place
00164 */
00165             *wp++ = number;
00166             while ( *in == ',' ) in++;
00167         }
00168         *work = wp - work;
00169         work = wp; wp++;
00170     }
00171 /*
00172     #] Sets :

```

```

00173
00174     Now we should loop over the various transformations
00175 */
00176     while ( *s ) {
00177         in = s;
00178         if ( FG.cTable[*in] != 0 ) {
00179             MesPrint("&Illegal character in Transform statement");
00180             if ( error == 0 ) error = 1;
00181             return(error);
00182         }
00183         in = SkipAName(in);
00184         if ( *in == '>' || *in == '<' || *in == '+' || *in == '-' ) in++;
00185         ss = in;
00186         c = *ss; *ss = 0;
00187         if ( c != '(' ) {
00188             MesPrint("&Illegal syntax in specifying a transformation inside a Transform statement");
00189             if ( error == 0 ) error = 1;
00190             return(error);
00191         }
00192     /*
00193     #[ replace :
00194     */
00195     if ( StrICmp(s, (UBYTE *) "replace") == 0 ) {
00196     /*
00197         Subkeys: (,,) as in replace_(",",)
00198         The idea here is to read the subkeys as the argument
00199         of a replace_ function.
00200         We put the whole together as in the multiply statement (which
00201         could just be a replace_(...)) and compile it.
00202         Then we expand the tree with Generator and check the complete
00203         expression for legality.
00204     */
00205         type = REPLACEARG;
00206     doreplace:
00207         *ss = c;
00208         if ( ( in = ReadRange(in, range, 0) ) == 0 ) {
00209             if ( error == 0 ) error = 1;
00210             return(error);
00211         }
00212         in++;
00213     /*
00214         We have replace(##)=(...), and we want dum_(...) (DUMFUN)
00215         to send to the compiler. The pointer is after the '=';
00216     */
00217         s = in;
00218         if ( *s != '(' ) {
00219             MesPrint("&");
00220             if ( error == 0 ) error = 1;
00221             return(error);
00222         }
00223         SKIPBRA3(in);
00224         if ( *in != ')' ) {
00225             MesPrint("&");
00226             if ( error == 0 ) error = 1;
00227             return(error);
00228         }
00229         in++;
00230         if ( *in != ',' && *in != '\0' ) {
00231             MesPrint("&");
00232             if ( error == 0 ) error = 1;
00233             return(error);
00234         }
00235         i = in - s;
00236         ss = Tempbuf = (UBYTE *) Malloc1(i+5, "CoTransform/replace");
00237         *ss++ = 'd'; *ss++ = 'u'; *ss++ = 'm'; *ss++ = '_';
00238         NCOPY(ss, s, i)
00239         *ss++ = 0;
00240         AC.ProtoType = tranarray;
00241         tranarray[4] = AC.cbufnum;
00242         irhs = CompileAlgebra(Tempbuf, RHSIDE, AC.ProtoType);
00243         M_free(Tempbuf, "CoTransform/replace");
00244         if ( irhs < 0 ) {
00245             if ( error == 0 ) error = 1;
00246             return(error);
00247         }
00248         tranarray[2] = irhs;
00249     /*
00250         The result of the compilation goes through Generator during
00251         execution, because that takes care of $-variables.
00252         This is why we could not use replace_ and had to use dum_.
00253     */
00254         *wp++ = ARGGRANGE;
00255         *wp++ = range[0];
00256         *wp++ = range[1];
00257         *wp++ = type;
00258         *wp++ = SUBEXPSIZE+4;
00259         for ( i = 0; i < SUBEXPSIZE; i++ ) *wp++ = tranarray[i];

```

```

00260         *wp++ = 1;
00261         *wp++ = 1;
00262         *wp++ = 3;
00263         *work = wp-work;
00264         work = wp; *wp++ = 0;
00265         s = in;
00266     }
00267     /*
00268     #] replace :
00269     #[ encode/decode :
00270     */
00271     else if ( StrICmp(s, (UBYTE *) "decode" ) == 0 ) {
00272         type = DECODEARG;
00273         goto doencode;
00274     }
00275     else if ( StrICmp(s, (UBYTE *) "encode" ) == 0 ) {
00276         type = ENCODEARG;
00277 doencode:
00278         *ss = c;
00279         if ( ( in = ReadRange(in, range, 2) ) == 0 ) {
00280             if ( error == 0 ) error = 1;
00281             return(error);
00282         }
00283         in++;
00284         s = in; while ( FG.cTable[*in] == 0 ) in++;
00285         c = *in; *in = 0;
00286     /*
00287     Subkeys: base=# or base=$var
00288     */
00289     if ( StrICmp(s, (UBYTE *) "base" ) == 0 ) {
00290         *in = c;
00291         if ( *in != '=' ) {
00292             MesPrint("&Illegal base specification in encode/decode transformation");
00293             if ( error == 0 ) error = 1;
00294             return(error);
00295         }
00296         in++;
00297         if ( *in == '$' ) {
00298             in++; ss = in;
00299             in = SkipAName(in);
00300             c = *in; *in = 0;
00301             if ( GetName(AC.dollarnames, ss, &numdol, NOAUTO) != CDOLLAR ) {
00302                 MesPrint("&%s is undefined", ss-1);
00303                 numdol = AddDollar(ss, DOLINDEX, &one, 1);
00304                 return(1);
00305             }
00306             *in = c;
00307             x = -numdol;
00308         }
00309         else {
00310             x = 0;
00311             while ( FG.cTable[*in] == 1 ) {
00312                 x = 10*x + *in++ - '0';
00313                 if ( x > MAXPOSITIVE4 ) {
00314                     MesPrint("&Illegal value for base in encode/decode transformation");
00315                     if ( error == 0 ) error = 1;
00316                     return(error);
00317                 }
00318             }
00319             if ( x <= 1 ) goto illsize;
00320         }
00321         if ( *in != ',' && *in != '\0' ) {
00322             MesPrint("&Illegal termination of transformation");
00323             if ( error == 0 ) error = 1;
00324             return(error);
00325         }
00326     }
00327     else {
00328         MesPrint("&Illegal option in encode/decode transformation");
00329         if ( error == 0 ) error = 1;
00330         return(error);
00331     }
00332     /*
00333     Now we can put the whole statement together
00334     We have the set(s) in work up to wp and the range in range.
00335     The base is in x and the type tells whether it is encode or decode.
00336     */
00337     *wp++ = ARGRANGE;
00338     *wp++ = range[0];
00339     *wp++ = range[1];
00340     *wp++ = type;
00341     *wp++ = 4;
00342     *wp++ = BASECODE;
00343     *wp++ = (WORD)x;
00344     *work = wp-work;
00345     work = wp; *wp++ = 0;
00346     s = in;

```

```

00347 /*
00348     #] encode/decode :
00349     #[ implode :
00350 */
00351     else if ( StrICmp(s, (UBYTE *)"implode") == 0
00352         || StrICmp(s, (UBYTE *)"tosumnotation") == 0 ) {
00353 /*
00354         Subkeys: ?
00355 */
00356         type = IMplodeARG;
00357         *ss = c;
00358         if ( ( in = ReadRange(in, range, 1) ) == 0 ) {
00359             if ( error == 0 ) error = 1;
00360             return(error);
00361         }
00362         *wp++ = ARGRANGE;
00363         *wp++ = range[0];
00364         *wp++ = range[1];
00365         *wp++ = type;
00366         *work = wp-work;
00367         work = wp; *wp++ = 0;
00368         s = in;
00369     }
00370 /*
00371     #] implode :
00372     #[ explode :
00373 */
00374     else if ( StrICmp(s, (UBYTE *)"explode") == 0
00375         || StrICmp(s, (UBYTE *)"tointegralnotation") == 0 ) {
00376 /*
00377         Subkeys: ?
00378 */
00379         type = EXPLODEARG;
00380         *ss = c;
00381         if ( ( in = ReadRange(in, range, 1) ) == 0 ) {
00382             if ( error == 0 ) error = 1;
00383             return(error);
00384         }
00385         *wp++ = ARGRANGE;
00386         *wp++ = range[0];
00387         *wp++ = range[1];
00388         *wp++ = type;
00389         *work = wp-work;
00390         work = wp; *wp++ = 0;
00391         s = in;
00392     }
00393 /*
00394     #] explode :
00395     #[ permute :
00396 */
00397     else if ( StrICmp(s, (UBYTE *)"permute") == 0 ) {
00398         type = PERMUTEARG;
00399         *ss = c;
00400         *wp++ = ARGRANGE;
00401         *wp++ = 1;
00402         *wp++ = MAXPOSITIVE4;
00403         *wp++ = type;
00404 /*
00405         Now a sequence of cycles
00406 */
00407         do {
00408             wstart = wp; wp++;
00409             do {
00410                 in++;
00411                 if ( *in == '$' ) {
00412                     WORD number; UBYTE *t;
00413                     in++; t = in;
00414                     while ( FG.cTable[*in] < 2 ) in++;
00415                     c = *in; *in = 0;
00416                     if ( ( number = GetDollar(t) ) < 0 ) {
00417                         MesPrint("&Undefined variable %s", t);
00418                         if ( !error ) error = 1;
00419                         number = AddDollar(t, 0, 0, 0);
00420                     }
00421                     *in = c;
00422                     *wp++ = -number-1;
00423                 }
00424             } else {
00425                 x = 0;
00426                 while ( FG.cTable[*in] == 1 ) {
00427                     x = 10*x + *in++ - '0';
00428                     if ( x > MAXPOSITIVE4 ) {
00429                         MesPrint("&value in permute transformation too large");
00430                         if ( error == 0 ) error = 1;
00431                         return(error);
00432                     }
00433                 }

```

```

00434         if ( x == 0 ) {
00435             MesPrint("&value 0 in permute transformation not allowed");
00436             if ( error == 0 ) error = 1;
00437             return(error);
00438         }
00439         *wp++ = (WORD)x-1;
00440     } while ( *in == ',' );
00441     if ( *in != ')' ) {
00442         MesPrint("&Illegal syntax in permute transformation");
00443         if ( error == 0 ) error = 1;
00444         return(error);
00445     }
00446     in++;
00447     if ( *in != ',' && *in != '(' && *in != '\\0' ) {
00448         MesPrint("&Illegal ending in permute transformation");
00449         if ( error == 0 ) error = 1;
00450         return(error);
00451     }
00452     *wstart = wp-wstart;
00453     if ( *wstart == 1 ) wstart--;
00454 } while ( *in == '(' );
00455 *work = wp-work;
00456 work = wp; *wp++ = 0;
00457 s = in;
00458 }
00459 /*
00460 #] permute :
00461 #[ reverse :
00462 */
00463 else if ( StrICmp(s, (UBYTE *) "reverse") == 0 ) {
00464     type = REVERSEARG;
00465     *ss = c;
00466     if ( ( in = ReadRange(in, range, 1) ) == 0 ) {
00467         if ( error == 0 ) error = 1;
00468         return(error);
00469     }
00470     *wp++ = ARGRANGE;
00471     *wp++ = range[0];
00472     *wp++ = range[1];
00473     *wp++ = type;
00474     *work = wp-work;
00475     work = wp; *wp++ = 0;
00476     s = in;
00477 }
00478 /*
00479 #] reverse :
00480 #[ dedup :
00481 */
00482 else if ( StrICmp(s, (UBYTE *) "dedup") == 0 ) {
00483     type = DEDUPARG;
00484     *ss = c;
00485     if ( ( in = ReadRange(in, range, 1) ) == 0 ) {
00486         if ( error == 0 ) error = 1;
00487         return(error);
00488     }
00489     *wp++ = ARGRANGE;
00490     *wp++ = range[0];
00491     *wp++ = range[1];
00492     *wp++ = type;
00493     *work = wp-work;
00494     work = wp; *wp++ = 0;
00495     s = in;
00496 }
00497 /*
00498 #] dedup :
00499 #[ cycle :
00500 */
00501 else if ( StrICmp(s, (UBYTE *) "cycle") == 0 ) {
00502     type = CYCLEARG;
00503     *ss = c;
00504     if ( ( in = ReadRange(in, range, 0) ) == 0 ) {
00505         if ( error == 0 ) error = 1;
00506         return(error);
00507     }
00508     *wp++ = ARGRANGE;
00509     *wp++ = range[0];
00510     *wp++ = range[1];
00511     *wp++ = type;
00512 }
00513 /*
00514 Now a sequence of cycles
00515 */
00516 in++;
00517 if ( *in == '+' ) {
00518 }
00519 else if ( *in == '-' ) {
00520     one = -1;

```



```

00521     }
00522     else {
00523         MesPrint("&Cycle in a Transform statement should be followed by +/-number/$");
00524         if ( error == 0 ) error = 1;
00525         return(error);
00526     }
00527     in++; x = 0;
00528     if ( *in == '$' ) {
00529         UBYTE *si = in;
00530         in++; si = in;
00531         while ( FG.cTable[*in] == 0 || FG.cTable[*in] == 1 ) in++;
00532         c = *in; *in = 0;
00533         if ( ( x = GetDollar(si) ) < 0 ) {
00534             MesPrint("&Undefined $-variable in transform,cycle statement.");
00535             error = 1;
00536         }
00537         *in = c;
00538         if ( one < 0 ) x += MAXPOSITIVE4;
00539         x += MAXPOSITIVE2;
00540         *wp++ = x;
00541     }
00542     else {
00543         while ( FG.cTable[*in] == 1 ) {
00544             x = 10*x + *in++ - '0';
00545             if ( x > MAXPOSITIVE4 ) {
00546                 MesPrint("&Number in cycle in a Transform statement too big");
00547                 if ( error == 0 ) error = 1;
00548                 return(error);
00549             }
00550         }
00551         *wp++ = x*one;
00552     }
00553     *work = wp-work;
00554     work = wp; *wp++ = 0;
00555     s = in;
00556 }
00557 /*
00558 #] cycle :
00559 #[ islyndon/tolyndon :
00560 */
00561 else if ( StrICmp(s,(UBYTE *)"islyndon" ) == 0 ) {
00562     type = ISLYNDON;
00563     goto doreplace;
00564 }
00565 else if ( StrICmp(s,(UBYTE *)"islyndon<" ) == 0 ) {
00566     type = ISLYNDON;
00567     goto doreplace;
00568 }
00569 else if ( StrICmp(s,(UBYTE *)"islyndon-" ) == 0 ) {
00570     type = ISLYNDON;
00571     goto doreplace;
00572 }
00573 else if ( StrICmp(s,(UBYTE *)"islyndon>" ) == 0 ) {
00574     type = ISLYNDONR;
00575     goto doreplace;
00576 }
00577 else if ( StrICmp(s,(UBYTE *)"islyndon+" ) == 0 ) {
00578     type = ISLYNDONR;
00579     goto doreplace;
00580 }
00581 else if ( StrICmp(s,(UBYTE *)"tolyndon" ) == 0 ) {
00582     type = TOLYNDON;
00583     goto doreplace;
00584 }
00585 else if ( StrICmp(s,(UBYTE *)"tolyndon<" ) == 0 ) {
00586     type = TOLYNDON;
00587     goto doreplace;
00588 }
00589 else if ( StrICmp(s,(UBYTE *)"tolyndon-" ) == 0 ) {
00590     type = TOLYNDON;
00591     goto doreplace;
00592 }
00593 else if ( StrICmp(s,(UBYTE *)"tolyndon>" ) == 0 ) {
00594     type = TOLYNDONR;
00595     goto doreplace;
00596 }
00597 else if ( StrICmp(s,(UBYTE *)"tolyndon+" ) == 0 ) {
00598     type = TOLYNDONR;
00599     goto doreplace;
00600 }
00601 /*
00602 #] islyndon/tolyndon :
00603 #[ addarg :
00604 */
00605 else if ( StrICmp(s,(UBYTE *)"addargs" ) == 0 ) {
00606     type = ADDARG;
00607     *ss = c;

```

```

00608         if ( ( in = ReadRange(in,range,1) ) == 0 ) {
00609             if ( error == 0 ) error = 1;
00610             return(error);
00611         }
00612         *wp++ = ARGRANGE;
00613         *wp++ = range[0];
00614         *wp++ = range[1];
00615         *wp++ = type;
00616         *work = wp-work;
00617         work = wp; *wp++ = 0;
00618         s = in;
00619     }
00620     /*
00621     #] addarg :
00622     #[ mularg :
00623     */
00624     else if ( ( StrICmp(s,(UBYTE *)"mulargs" ) == 0 )
00625         || ( StrICmp(s,(UBYTE *)"multiplyargs" ) == 0 ) ) {
00626         type = MULTIPLYARG;
00627         *ss = c;
00628         if ( ( in = ReadRange(in,range,1) ) == 0 ) {
00629             if ( error == 0 ) error = 1;
00630             return(error);
00631         }
00632         *wp++ = ARGRANGE;
00633         *wp++ = range[0];
00634         *wp++ = range[1];
00635         *wp++ = type;
00636         *work = wp-work;
00637         work = wp; *wp++ = 0;
00638         s = in;
00639     }
00640     /*
00641     #] mularg :
00642     #[ droparg :
00643     */
00644     else if ( StrICmp(s,(UBYTE *)"dropargs" ) == 0 ) {
00645         type = DROPARG;
00646         *ss = c;
00647         if ( ( in = ReadRange(in,range,1) ) == 0 ) {
00648             if ( error == 0 ) error = 1;
00649             return(error);
00650         }
00651         *wp++ = ARGRANGE;
00652         *wp++ = range[0];
00653         *wp++ = range[1];
00654         *wp++ = type;
00655         *work = wp-work;
00656         work = wp; *wp++ = 0;
00657         s = in;
00658     }
00659     /*
00660     #] droparg :
00661     #[ selectarg :
00662     */
00663     else if ( StrICmp(s,(UBYTE *)"selectargs" ) == 0 ) {
00664         type = SELECTARG;
00665         *ss = c;
00666         if ( ( in = ReadRange(in,range,1) ) == 0 ) {
00667             if ( error == 0 ) error = 1;
00668             return(error);
00669         }
00670         *wp++ = ARGRANGE;
00671         *wp++ = range[0];
00672         *wp++ = range[1];
00673         *wp++ = type;
00674         *work = wp-work;
00675         work = wp; *wp++ = 0;
00676         s = in;
00677     }
00678     /*
00679     #] selectarg :
00680     #[ ZtoH :
00681     */
00682     else if ( StrICmp(s,(UBYTE *)"ztoh" ) == 0 ) {
00683         /*
00684         Subkeys: ?
00685         */
00686         type = ZTOHARG;
00687         *ss = c;
00688         if ( ( in = ReadRange(in,range,1) ) == 0 ) {
00689             if ( error == 0 ) error = 1;
00690             return(error);
00691         }
00692         *wp++ = ARGRANGE;
00693         *wp++ = range[0];
00694         *wp++ = range[1];

```

```

00695         *wp++ = type;
00696         *work = wp-work;
00697         work = wp; *wp++ = 0;
00698         s = in;
00699     }
00700 /*
00701     #] ZtoH :
00702     #[ HtoZ :
00703 */
00704     else if ( StrICmp(s,(UBYTE *)"htoz") == 0 ) {
00705 /*
00706         Subkeys: ?
00707 */
00708         type = HTOZARG;
00709         *ss = c;
00710         if ( ( in = ReadRange(in,range,1) ) == 0 ) {
00711             if ( error == 0 ) error = 1;
00712             return(error);
00713         }
00714         *wp++ = ARGRANGE;
00715         *wp++ = range[0];
00716         *wp++ = range[1];
00717         *wp++ = type;
00718         *work = wp-work;
00719         work = wp; *wp++ = 0;
00720         s = in;
00721     }
00722 /*
00723     #] HtoZ :
00724 */
00725     else {
00726         MesPrint("&Unknown transformation inside a Transform statement: %s",s);
00727         *ss = c;
00728         if ( error == 0 ) error = 1;
00729         return(error);
00730     }
00731     while ( *s == ',' ) s++;
00732 }
00733 AT.WorkPointer[0] = TYPETRANSFORM;
00734 AT.WorkPointer[1] = i = wp - AT.WorkPointer;
00735 AddNtoL(i,AT.WorkPointer);
00736 return(error);
00737 }
00738
00739 /*
00740     #] CoTransform :
00741     #[ RunTransform :
00742
00743     Executes the transform statement.
00744     This routine hunts down the functions and sends them to the various
00745     action routines.
00746     params: size,#set1,...,#setn, transformations
00747
00748 */
00749 int RunTransform(PHEAD WORD *term, WORD *params)
00750 {
00751     WORD *t, *tstop, *w, *m, *out, *in, *tt, retval;
00752     WORD *fun, *args, *info, *infoend, *onetransform, *funs, *endfun;
00753     WORD *thearg = 0, *iterm, *newterm, *nt, *oldwork = AT.WorkPointer, sign = 1;
00754     int i;
00755     out = tstop = term + *term;
00756     tstop -= ABS(tstop[-1]);
00757     in = term;
00758     t = term + 1;
00759     while ( t < tstop ) {
00760         endfun = onetransform = params + *params;
00761         funs = params + 1;
00762         if ( *t < FUNCTION ) {}
00763         else if ( funs == endfun ) { /* we do all functions */
00764             hit;;
00765             #ifdef WITHFLOAT
00766                 if ( *t == FLOATFUN ) goto next;
00767             #endif
00768             while ( in < t ) *out++ = *in++;
00769             tt = t + t[1]; fun = out;
00770             while ( in < tt ) *out++ = *in++;
00771             do {
00772                 args = onetransform + 1;
00773                 info = args; while ( *info <= MAXRANGEINDICATOR ) {
00774                     if ( *info == ALLARGS ) info++;
00775                     else if ( *info == NUMARG ) info += 2;
00776                     else if ( *info == ARGRANGE ) info += 3;
00777                     else if ( *info == MAKEARGS ) info += 3;
00778                 }
00779                 switch ( *info ) {
00780                     case REPLACEARG:

```

```

00782         if ( RunReplace(BHEAD fun,args,info) ) goto abo;
00783         out = fun + fun[1];
00784         break;
00785     case ENCODEARG:
00786         if ( RunEncode(BHEAD fun,args,info) ) goto abo;
00787         out = fun + fun[1];
00788         break;
00789     case DECODEARG:
00790         if ( RunDecode(BHEAD fun,args,info) ) goto abo;
00791         out = fun + fun[1];
00792         break;
00793     case IMplodeARG:
00794         if ( RunImplode(fun,args) ) goto abo;
00795         out = fun + fun[1];
00796         break;
00797     case EXPLODEARG:
00798         if ( RunExplode(BHEAD fun,args) ) goto abo;
00799         out = fun + fun[1];
00800         break;
00801     case PERMUTEARG:
00802         if ( RunPermute(BHEAD fun,args,info) ) goto abo;
00803         out = fun + fun[1];
00804         break;
00805     case REVERSEARG:
00806         if ( RunReverse(BHEAD fun,args) ) goto abo;
00807         out = fun + fun[1];
00808         break;
00809     case DEDUPARG:
00810         if ( RunDedup(BHEAD fun,args) ) goto abo;
00811         out = fun + fun[1];
00812         break;
00813     case CYCLEARG:
00814         if ( RunCycle(BHEAD fun,args,info) ) goto abo;
00815         out = fun + fun[1];
00816         break;
00817     case ADDARG:
00818         if ( RunAddArg(BHEAD fun,args) ) goto abo;
00819         out = fun + fun[1];
00820         break;
00821     case MULTIPLYARG:
00822         if ( RunMulArg(BHEAD fun,args) ) goto abo;
00823         out = fun + fun[1];
00824         break;
00825     case ISLYNDON:
00826         if ( ( retval = RunIsLyndon(BHEAD fun,args,1) ) < -1 ) goto abo;
00827         goto returnvalues;
00828         break;
00829     case ISLYNDONR:
00830         if ( ( retval = RunIsLyndon(BHEAD fun,args,-1) ) < -1 ) goto abo;
00831         goto returnvalues;
00832         break;
00833     case TOLYNDON:
00834         if ( ( retval = RunToLyndon(BHEAD fun,args,1) ) < -1 ) goto abo;
00835         goto returnvalues;
00836         break;
00837     case TOLYNDONR:
00838         if ( ( retval = RunToLyndon(BHEAD fun,args,-1) ) < -1 ) goto abo;
00839     returnvalues:;
00840         out = fun + fun[1];
00841         if ( retval == -1 ) break;
00842     /*
00843         Work out the yes/no stuff
00844     */
00845     AT.WorkPointer += 2*AM.MaxTer;
00846     if ( AT.WorkPointer > AT.WorkTop ) {
00847         MLOCK(ErrorMessageLock);
00848         MesWork();
00849         MUNLOCK(ErrorMessageLock);
00850         return(-1);
00851     }
00852     iterm = AT.WorkPointer;
00853     info++;
00854     for ( i = 0; i < *info; i++ ) iterm[i] = info[i];
00855     AT.WorkPointer = iterm + *iterm;
00856     AR.Eside = LHSIDEX;
00857     NewSort(BHEAD0);
00858     if ( Generator(BHEAD iterm,AR.Cnumlhs) ) {
00859         LowerSortLevel();
00860         AT.WorkPointer = oldwork;
00861         return(-1);
00862     }
00863     newterm = AT.WorkPointer;
00864     if ( EndSort(BHEAD newterm,1) < 0 ) {}
00865     if ( ( *newterm && *(newterm+*newterm) != 0 ) || *newterm == 0 ) {
00866         MLOCK(ErrorMessageLock);
00867         MesPrint("&yes/no information in islyndon/tolyndon does not evaluate into
a single term");

```

```

00868             MUNLOCK(ErrorMessageLock);
00869             return(-1);
00870         }
00871         AR.Eside = RHSIDE;
00872         i = *newterm; tt = iterm; nt = newterm;
00873         NCOPY(tt,nt,i);
00874         AT.WorkPointer = iterm + *iterm;
00875         info = iterm + 1;
00876         infoend = info+info[1];
00877         info += FUNHEAD;
00878
00879         if ( retval == 0 ) {
00880             /*
00881                 Need second argument (=no)
00882             */
00883             if ( info >= infoend ) {
00884                 abortlyndon;;
00885                 MLOCK(ErrorMessageLock);
00886                 MesPrint("There should be a yes and a no argument in
islyndon/tolyndon");
00887                 MUNLOCK(ErrorMessageLock);
00888                 Terminate(-1);
00889             }
00890             NEXTARG(info)
00891             if ( info >= infoend ) goto abortlyndon;
00892             thearg = info;
00893         }
00894         else if ( retval == 1 ) {
00895             /*
00896                 Need first argument (=yes)
00897             */
00898             if ( info >= infoend ) goto abortlyndon;
00899             thearg = info;
00900             NEXTARG(info)
00901             if ( info >= infoend ) goto abortlyndon;
00902         }
00903         NEXTARG(info)
00904         if ( info < infoend ) goto abortlyndon;
00905         /*
00906             The argument in thearg needs to be copied
00907             We did not pull it through generator to guarantee
00908             that it is a single argument.
00909             The easiest way is to let the routine Normalize
00910             do the job and put everything in an exponent function
00911             with the power one.
00912         */
00913         if ( *thearg == -SNUMBER && thearg[1] == 0 ) {
00914             *term = 0; return(0);
00915         }
00916         if ( *thearg == -SNUMBER && thearg[1] == 1 ) { }
00917         else {
00918             fun = out;
00919             *out++ = EXPONENT; out++; *out++ = 1; FILLFUN3(out);
00920             COPY1ARG(out,thearg);
00921             *out++ = -SNUMBER; *out++ = 1;
00922             fun[1] = out-fun;
00923         }
00924         break;
00925     case DROPARG:
00926         if ( RunDropArg(BHEAD fun,args) ) goto abo;
00927         out = fun + fun[1];
00928         break;
00929     case SELECTARG:
00930         if ( RunSelectArg(BHEAD fun,args) ) goto abo;
00931         out = fun + fun[1];
00932         break;
00933     case ZTOHARG:
00934         {
00935             WORD s = RunZtoHArg(BHEAD fun,args);
00936             if ( s < 0 ) goto abo;
00937             if ( s == 1 ) sign = -sign;
00938             out = fun + fun[1];
00939         }
00940         break;
00941     case HTOZARG:
00942         {
00943             WORD s = RunHtoZArg(BHEAD fun,args);
00944             if ( s < 0 ) goto abo;
00945             if ( s == 1 ) sign = -sign;
00946             out = fun + fun[1];
00947         }
00948         break;
00949     default:
00950         MLOCK(ErrorMessageLock);
00951         MesPrint("Irregular code in execution of transform statement");
00952         MUNLOCK(ErrorMessageLock);
00953         Terminate(-1);

```

```

00954         }
00955         onetransform += *onetransform;
00956     } while ( *onetransform );
00957 }
00958 else {
00959     while ( funs < endfun ) { /* sum over sets */
00960         if ( *funs > MAXVARIABLES ) {
00961             if ( *t == *funs-MAXVARIABLES ) goto hit;
00962         }
00963         else {
00964             w = SetElements + Sets[*funs].first;
00965             m = SetElements + Sets[*funs].last;
00966             while ( w < m ) { /* sum over set elements */
00967                 if ( *w == *t ) goto hit;
00968                 w++;
00969             }
00970         }
00971         funs++;
00972     }
00973 }
00974 #ifdef WITHFLOAT
00975 next:
00976 #endif
00977     t += t[1];
00978 }
00979 tt = term + *term; while ( in < tt ) *out++ = *in++;
00980 if ( sign == -1 ) out[-1] = -out[-1];
00981 *tt = i = out - tt;
00982 /*
00983     Now copy the whole thing back
00984 */
00985     NCOPY(term,tt,i)
00986     return(0);
00987 abo:
00988     MLOCK(ErrorMessageLock);
00989     MesCall("RunTransform");
00990     MUNLOCK(ErrorMessageLock);
00991     return(-1);
00992 }
00993
00994 /*
00995     #] RunTransform :
00996     #[ RunEncode :
00997
00998     The info is given by
00999         ENCODEARG,size,BASECODE,num
01000     and possibly more codes to follow.
01001     Only one range is allowed and for now, it should be fully numerical
01002     If the range is in reverse order, we need to either revert it
01003     first or work with an array of pointers.
01004 */
01005
01006 int RunEncode(PHEAD WORD *fun, WORD *args, WORD *info)
01007 {
01008     WORD base, *f, *funstop, *fun1, *t, size1, size2, size3, *arg;
01009     int num, num1, num2, n, i, i1, i2;
01010     UWORD *scrat1, *scrat2, *scrat3;
01011     WORD *tt, *tstop, totarg, arg1, arg2;
01012     if ( functions[fun[0]-FUNCTION].spec != 0 ) return(0);
01013     if ( *args != ARGRANGE ) {
01014         MLOCK(ErrorMessageLock);
01015         MesPrint("Illegal range encountered in RunEncode");
01016         MUNLOCK(ErrorMessageLock);
01017         Terminate(-1);
01018     }
01019     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
01020     while ( tt < tstop ) { totarg++; NEXTARG(tt); }
01021     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
01022     if ( arg1 > totarg || arg2 > totarg ) return(0);
01023
01024     if ( info[2] == BASECODE ) {
01025         base = info[3];
01026         if ( base <= 0 ) { /* is a dollar variable */
01027             i1 = -base;
01028             base = DolToNumber(BHEAD i1);
01029             if ( AN.ErrorInDollar || base < 2 ) {
01030                 MLOCK(ErrorMessageLock);
01031                 MesPrint("%$s does not have a number value > 1 in base/encode/transform statement in
module %1",
01032                     DOLLARNAME(Dollars,i1),AC.CModule);
01033                 MUNLOCK(ErrorMessageLock);
01034                 Terminate(-1);
01035             }
01036         }
01037     /*
01038         Compute number of pointers needed and make sure there is space
01039     */

```

```

01040     if ( arg1 > arg2 ) { num1 = arg2; num2 = arg1; }
01041     else { num1 = arg1; num2 = arg2; }
01042     num = num2-num1+1;
01043     WantAddPointers(num);
01044     /*
01045     Collect the pointers in pWorkSpace
01046     */
01047     n = 1; funstop = fun+fun[1]; f = fun+FUNHEAD;
01048     while ( n < num1 ) {
01049         if ( f >= funstop ) return(0);
01050         NEXTARG(f);
01051         n++;
01052     }
01053     fun1 = f; i = 0;
01054     while ( n <= num2 ) {
01055         if ( f >= funstop ) return(0);
01056         if ( *f != -SNUMBER ) {
01057             if ( *f < 0 ) return(0);
01058             t = f + *f - 1;
01059             i1 = ABS(*t);
01060             if ( (*f-i1) != (ARGHEAD+1) ) return(0); /* Not numerical */
01061             i1 = (i1-1)/2 - 1;
01062             t--;
01063             while ( i1 > 0 ) {
01064                 if ( *t != 0 ) return(0); /* Not an integer */
01065                 t--; i1--;
01066             }
01067             AT.pWorkSpace[AT.pWorkPointer+i] = f;
01068             i++;
01069             NEXTARG(f);
01070             n++;
01071         }
01072     }
01073     /*
01074     f points now to after the arguments; fun1 at the first.
01075     Now check whether we need to revert the order
01076     */
01077     if ( arg1 > arg2 ) {
01078         i1 = 0; i2 = i-1;
01079         while ( i1 < i2 ) {
01080             t = AT.pWorkSpace[AT.pWorkPointer+i1];
01081             AT.pWorkSpace[AT.pWorkPointer+i1] = AT.pWorkSpace[AT.pWorkPointer+i2];
01082             AT.pWorkSpace[AT.pWorkPointer+i2] = t;
01083             i1++; i2--;
01084         }
01085     }
01086     /*
01087     Now we can put the thing together.
01088     x = arg1;
01089     x = base*x+arg2
01090     x = base*x+arg3 etc.
01091     We need three scratch arrays for long integers
01092     (see NumberMalloc in tools.c).
01093     */
01094     scrat1 = NumberMalloc("RunEncode");
01095     scrat2 = NumberMalloc("RunEncode");
01096     scrat3 = NumberMalloc("RunEncode");
01097     arg = AT.pWorkSpace[AT.pWorkPointer];
01098     size1 = PutArgInScratch(arg,scrat1);
01099     i--;
01100     while ( i > 0 ) {
01101         if ( MulLong(scrat1,size1,(UWORD *)(&base),1,scrat2,&size2) ) {
01102             NumberFree(scrat3,"RunEncode");
01103             NumberFree(scrat2,"RunEncode");
01104             NumberFree(scrat1,"RunEncode");
01105             goto CalledFrom;
01106         }
01107         NEXTARG(arg);
01108         size3 = PutArgInScratch(arg,scrat3);
01109         if ( AddLong(scrat2,size2,scrat3,size3,scrat1,&size1) ) {
01110             NumberFree(scrat3,"RunEncode");
01111             NumberFree(scrat2,"RunEncode");
01112             NumberFree(scrat1,"RunEncode");
01113             goto CalledFrom;
01114         }
01115         i--;
01116     }
01117     /*
01118     Now put the output in place. There are two cases, one being much
01119     faster than the other. Hence we program both.
01120     Fast: it fits inside the old location.
01121     Slow: it does not.
01122     The total space is f-fun1
01123     */
01124     if ( size1 == 0 ) { /* Fits! */
01125         *fun1++ = -SNUMBER; *fun1++ = 0;
01126         while ( f < funstop ) *fun1++ = *f++;

```

```

01127         fun[1] = funstop-fun;
01128     }
01129     else if ( size1 == 1 && scratl[0] <= MAXPOSITIVE ) { /* Fits! */
01130         *funl++ = -SNUMBER; *funl++ = scratl[0];
01131         while ( f < funstop ) *funl++ = *f++;
01132         fun[1] = funl-fun;
01133     }
01134     else if ( size1 == -1 && scratl[0] <= MAXPOSITIVE+1 ) { /* Fits! */
01135         *funl++ = -SNUMBER;
01136         if ( scratl[0] < MAXPOSITIVE ) *funl++ = scratl[0];
01137         else *funl++ = (WORD) (MAXPOSITIVE+1);
01138         while ( f < funstop ) *funl++ = *f++;
01139         fun[1] = funl-fun;
01140     }
01141     else if ( ABS(size1)*2+2+ARGHEAD <= f-funl ) { /* Fits! */
01142         if ( size1 < 0 ) { size2 = size1*2-1; size1 = -size1; size3 = -size2; }
01143         else { size2 = 2*size1+1; size3 = size2; }
01144         *funl++ = size3+ARGHEAD+1;
01145         *funl++ = 0; FILLARG(funl);
01146         *funl++ = size3+1;
01147         for ( i = 0; i < size1; i++ ) *funl++ = scratl[i];
01148         *funl++ = 1;
01149         for ( i = 1; i < size1; i++ ) *funl++ = 0;
01150         *funl++ = size2;
01151         while ( f < funstop ) *funl++ = *f++;
01152         fun[1] = funl-fun;
01153     }
01154     else { /* Does not fit */
01155         t = funstop;
01156         if ( size1 < 0 ) { size2 = size1*2-1; size1 = -size1; size3 = -size2; }
01157         else { size2 = 2*size1+1; size3 = size2; }
01158         *t++ = size3+ARGHEAD+1;
01159         *t++ = 0; FILLARG(t);
01160         *t++ = size3+1;
01161         for ( i = 0; i < size1; i++ ) *t++ = scratl[i];
01162         *t++ = 1;
01163         for ( i = 1; i < size1; i++ ) *t++ = 0;
01164         *t++ = size2;
01165         while ( f < funstop ) *t++ = *f++;
01166         f = funstop;
01167         while ( f < t ) *funl++ = *f++;
01168         fun[1] = funl-fun;
01169     }
01170     NumberFree(scrat3,"RunEncode");
01171     NumberFree(scrat2,"RunEncode");
01172     NumberFree(scratl,"RunEncode");
01173 }
01174 else {
01175     MLOCK(ErrorMessageLock);
01176     MesPrint("Unimplemented type of encoding encountered in RunEncode");
01177     MUNLOCK(ErrorMessageLock);
01178     Terminate(-1);
01179 }
01180 return(0);
01181 CalledFrom:
01182     MLOCK(ErrorMessageLock);
01183     MesCall("RunEncode");
01184     MUNLOCK(ErrorMessageLock);
01185     return(-1);
01186 }
01187
01188 /*
01189     #] RunEncode :
01190     #[ RunDecode :
01191 */
01192
01193 int RunDecode(PHEAD WORD *fun, WORD *args, WORD *info)
01194 {
01195     WORD base, num, num1, num2, n, *f, *funstop, *funl, size1, size2, size3, *t;
01196     WORD i1, i2, i, sig;
01197     UWORD *scratl, *scrat2, *scrat3;
01198     WORD *tt, *tstop, totarg, arg1, arg2;
01199     if ( functions[fun[0]-FUNCTION].spec != 0 ) return(0);
01200     if ( *args != ARGRANGE ) {
01201         MLOCK(ErrorMessageLock);
01202         MesPrint("Illegal range encountered in RunDecode");
01203         MUNLOCK(ErrorMessageLock);
01204         Terminate(-1);
01205     }
01206     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
01207     while ( tt < tstop ) { totarg++; NEXTARG(tt); }
01208     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
01209     if ( arg1 > totarg && arg2 > totarg ) return(0);
01210     if ( info[2] == BASECODE ) {
01211         base = info[3];
01212         if ( base <= 0 ) { /* is a dollar variable */
01213             i1 = -base;

```



```

01214         base = DolToNumber(BHEAD il);
01215         if ( AN.ErrorInDollar || base < 2 ) {
01216             MLOCK(ErrorMessageLock);
01217             MesPrint("%s does not have a number value > 1 in base/decode/transform statement in
module %1",
01218                     DOLLARNAME(Dollars,il),AC.CModule);
01219             MUNLOCK(ErrorMessageLock);
01220             Terminate(-1);
01221         }
01222     }
01223 /*
01224     Compute number of output arguments needed
01225 */
01226     if ( arg1 > arg2 ) { num1 = arg2; num2 = arg1; }
01227     else { num1 = arg1; num2 = arg2; }
01228     num = num2-num1+1;
01229     if ( num <= 1 ) return(0);
01230 /*
01231     Find argument num1
01232 */
01233     funstop = fun + fun[1];
01234     f = fun + FUNHEAD; n = 1;
01235     while ( f < funstop ) {
01236         if ( n == num1 ) break;
01237         NEXTARG(f); n++;
01238     }
01239     if ( f >= funstop ) return(0); /* not enough arguments */
01240 /*
01241     Check that f is integer
01242 */
01243     if ( *f == -SNUMBER ) {}
01244     else if ( *f < 0 ) return(0);
01245     else {
01246         t = f + *f - 1;
01247         il = ABS(*t);
01248         if ( (*f-il) != (ARGHEAD+1) ) return(0); /* Not numerical */
01249         il = (il-1)/2 - 1;
01250         t--;
01251         while ( il > 0 ) {
01252             if ( *t != 0 ) return(0); /* Not an integer */
01253             t--; il--;
01254         }
01255     }
01256     fun1 = f;
01257 /*
01258     The argument that should be decoded is in fun1
01259     We have to copy it to scratch
01260 */
01261     scrat1 = NumberMalloc("RunEncode");
01262     scrat2 = NumberMalloc("RunEncode");
01263     scrat3 = NumberMalloc("RunEncode");
01264     size1 = PutArgInScratch(fun1,scrat1);
01265     if ( size1 < 0 ) { sig = -1; size1 = -size1; }
01266     else sig = 1;
01267 /*
01268     We can check first whether this number can be decoded
01269 */
01270     scrat2[0] = base; size2 = 1;
01271     if ( RaisPow(BHEAD scrat2,&size2,num) ) {
01272         NumberFree(scrat3,"RunEncode");
01273         NumberFree(scrat2,"RunEncode");
01274         NumberFree(scrat1,"RunEncode");
01275         goto CalledFrom;
01276     }
01277     if ( BigLong(scrat1,size1,scrat2,size2) >= 0 ) { /* Number too big */
01278         NumberFree(scrat3,"RunEncode");
01279         NumberFree(scrat2,"RunEncode");
01280         NumberFree(scrat1,"RunEncode");
01281         return(0);
01282     }
01283 /*
01284     We need num*2 spaces
01285 */
01286     if ( *fun1 > num*2 ) { /* shrink space */
01287         t = fun1 + 2*num; f = fun1 + *fun1;
01288         while ( f < funstop ) *t++ = *f++;
01289         fun[1] = t - fun;
01290     }
01291     else if ( *fun1 < num*2 ) { /* case includes -SNUMBER */
01292         if ( *fun1 < 0 ) { /* expand space from -SNUMBER */
01293             fun[1] += (num-1)*2;
01294             t = funstop + (num-1)*2;
01295         }
01296         else { /* expand space from general argument */
01297             fun[1] += 2*num - *fun1;
01298             t = funstop + 2*num - *fun1;
01299         }
01300     }

```

```

01300         f = funstop;
01301         while ( f > fun1 ) *--t = *--f;
01302     }
01303 /*
01304     Now there is space for num -SNUMBER arguments filled from the top.
01305 */
01306     for ( i = num-1; i >= 0; i-- ) {
01307         DivLong(scrat1,size1,(UWORD *)(&base),1,scrat2,&size2,scrat3,&size3);
01308         fun1[2*i] = -SNUMBER;
01309         if ( size3 == 0 ) fun1[2*i+1] = 0;
01310         else fun1[2*i+1] = (WORD)(scrat3[0])*sig;
01311         for ( i1 = 0; i1 < size2; i1++ ) scrat1[i1] = scrat2[i1];
01312         size1 = size2;
01313     }
01314     if ( size2 != 0 ) {
01315         MLOCK(ErrorMessageLock);
01316         MesPrint("RunDecode: number to be decoded is too big");
01317         MUNLOCK(ErrorMessageLock);
01318         NumberFree(scrat3,"RunEncode");
01319         NumberFree(scrat2,"RunEncode");
01320         NumberFree(scrat1,"RunEncode");
01321         goto CalledFrom;
01322     }
01323 /*
01324     Now check whether we should change the order of the arguments
01325 */
01326     if ( arg1 > arg2 ) {
01327         i1 = 1; i2 = 2*num-1;
01328         while ( i2 > i1 ) {
01329             i = fun1[i1]; fun1[i1] = fun1[i2]; fun1[i2] = i;
01330             i1 += 2; i2 -= 2;
01331         }
01332     }
01333     NumberFree(scrat3,"RunEncode");
01334     NumberFree(scrat2,"RunEncode");
01335     NumberFree(scrat1,"RunEncode");
01336 }
01337 else {
01338     MLOCK(ErrorMessageLock);
01339     MesPrint("Unimplemented type of encoding encountered in RunDecode");
01340     MUNLOCK(ErrorMessageLock);
01341     Terminate(-1);
01342 }
01343 return(0);
01344 CalledFrom:
01345 MLOCK(ErrorMessageLock);
01346 MesCall("RunDecode");
01347 MUNLOCK(ErrorMessageLock);
01348 return(-1);
01349 }
01350
01351 /*
01352     #] RunDecode :
01353     #[ RunReplace :
01354
01355     Gets the function, passes the arguments and looks whether they
01356     need to be treated. If so, the exact treatment is found in info.
01357     The info is given as if it is a function of type REPLACEMENT but
01358     its name is REPLACEARG (which is NOT a function).
01359     It is performed on the arguments.
01360     The output is at first written after fun and in the end overwrites fun.
01361 */
01362
01363 int RunReplace(PHEAD WORD *fun, WORD *args, WORD *info)
01364 {
01365     int n = 0, i, dirty = 0, totarg, nfix, nwild, ngeneral;
01366     WORD *t, *tt, *u, *tstop, *info1, *infoend, *oldwork = AT.WorkPointer;
01367     WORD *term, *newterm, *nt, *term1, *term2;
01368     WORD wild[4], mask, *term3, *term4, *oldmask = AT.WildMask;
01369     WORD n1, n2, doanyway;
01370     info++;
01371     t = fun; tstop = fun + fun[1]; u = tstop;
01372     for ( i = 0; i < FUNHEAD; i++ ) *u++ = *t++;
01373     tt = t;
01374     if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
01375         totarg = 0;
01376         while ( tt < tstop ) { totarg++; NEXTARG(tt); }
01377     }
01378     else {
01379         totarg = tstop - tt;
01380     }
01381 /*
01382     Now get the info through Generator to bring it to standard form.
01383     info points at a single term that should be sent to Generator.
01384
01385     We want to put the information in the Workspace but fun etc lies there
01386     already. This means that we have to move the WorkPointer quite high up.

```

```

01387 */
01388     AT.WorkPointer += 2*AM.MaxTer;
01389     if ( AT.WorkPointer > AT.WorkTop ) {
01390         MLOCK(ErrorMessageLock);
01391         MesWork();
01392         MUNLOCK(ErrorMessageLock);
01393         return(-1);
01394     }
01395     term = AT.WorkPointer;
01396     for ( i = 0; i < *info; i++ ) term[i] = info[i];
01397     AT.WorkPointer = term + *term;
01398     AR.Eside = LHSIDEX;
01399     NewSort(BHEAD0);
01400     if ( Generator(BHEAD term,AR.Cnumlhs) ) {
01401         LowerSortLevel();
01402         AT.WorkPointer = oldwork;
01403         return(-1);
01404     }
01405     newterm = AT.WorkPointer;
01406     if ( EndSort(BHEAD newterm,1) < 0 ) {}
01407     if ( ( *newterm && *(newterm+*newterm) != 0 ) || *newterm == 0 ) {
01408         MLOCK(ErrorMessageLock);
01409         MesPrint("information in replace transformation does not evaluate into a single term");
01410         MUNLOCK(ErrorMessageLock);
01411         return(-1);
01412     }
01413     AR.Eside = RHSIDE;
01414     i = *newterm; tt = term; nt = newterm;
01415     NCOPY(tt,nt,i);
01416     AT.WorkPointer = term + *term;
01417     info = term + 1;
01418
01419     term1 = term + *term;
01420     term2 = term1+1;
01421     *term2++ = REPLACEMENT;
01422     term2++; FILLFUN(term2)
01423 /*
01424     First we count the different types of objects
01425 */
01426     infoend = info + info[1];
01427     info1 = info + FUNHEAD;
01428     nfix = nwild = ngeneral = 0;
01429     while ( info1 < infoend ) {
01430         if ( *info1 == -SNUMBER ) {
01431             nfix++;
01432             info1 += 2; NEXTARG(info1)
01433         }
01434         else if ( *info1 <= -FUNCTION ) {
01435             if ( *info1 == -WILDARGFUN ) {
01436                 nwild++;
01437                 info1++; NEXTARG(info1)
01438             }
01439             else {
01440                 *term2++ = *info1++; COPY1ARG(term2,info1)
01441                 ngeneral++;
01442             }
01443         }
01444         else if ( *info1 == -INDEX ) {
01445             if ( info1[1] == WILDARGINDEX + AM.OffsetIndex ) {
01446                 nwild++;
01447                 info1 += 2; NEXTARG(info1)
01448             }
01449             else {
01450                 *term2++ = *info1++; *term2++ = *info1++; COPY1ARG(term2,info1)
01451                 ngeneral++;
01452             }
01453         }
01454         else if ( *info1 == -SYMBOL ) {
01455             if ( info1[1] == WILDARGSYMBOL ) {
01456                 nwild++;
01457                 info1 += 2; NEXTARG(info1)
01458             }
01459             else {
01460                 *term2++ = *info1++; *term2++ = *info1++; COPY1ARG(term2,info1)
01461                 ngeneral++;
01462             }
01463         }
01464         else if ( *info1 == -MINVECTOR || *info1 == -VECTOR ) {
01465             if ( info1[1] == WILDARGVECTOR + AM.OffsetVector ) {
01466                 nwild++;
01467                 info1 += 2; NEXTARG(info1)
01468             }
01469             else {
01470                 *term2++ = *info1++; *term2++ = *info1++; COPY1ARG(term2,info1)
01471                 ngeneral++;
01472             }
01473         }

```

```

01474         else {
01475             MLOCK(ErrorMessageLock);
01476             MesPrint("&irregular code found in replace transformation (RunReplace)");
01477             MUNLOCK(ErrorMessageLock);
01478             Terminate(-1);
01479         }
01480     }
01481     AT.WorkPointer = term2;
01482     *term1 = term2 - term1;
01483     term1[2] = *term1 - 1;
01484     /*
01485     And now stepping through the arguments
01486     */
01487     while ( t < tstop ) {
01488         n++; /* The number of the argument. Now check whether we need it */
01489         if ( TestArgNum(n,totarg,args) == 0 ) {
01490             if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
01491                 if ( *t <= -FUNCTION ) { *u++ = *t++; }
01492                 else if ( *t < 0 ) { *u++ = *t++; *u++ = *t++; }
01493                 else { i = *t; NCOPY(u,t,i) }
01494             }
01495             else *u++ = *t++;
01496             continue;
01497         }
01498     /*
01499     Here we have in info effectively a replace_ function, but with
01500     additionally the possibility of integer arguments. We treat those first
01501     and for the rest we have to do some pattern matching.
01502     Note that the compilation routine should check that there is an
01503     even number of arguments in the replace function.
01504
01505     First we go for number -> something
01506     */
01507     doanyway = 0;
01508     if ( nfix > 0 ) {
01509         if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
01510             if ( *t == -SNUMBER ) {
01511                 infof = info + FUNHEAD;
01512                 while ( infof < infoend ) {
01513                     if ( *infof == -SNUMBER ) {
01514                         if ( infof[1] == t[1] ) {
01515                             if ( infof[2] == -SNUMBER ) {
01516                                 *u++ = -SNUMBER; *u++ = infof[3];
01517                                 infof += 4;
01518                             }
01519                             else {
01520                                 infof += 2;
01521                                 if ( infof[0] <= -FUNCTION ) i = 1;
01522                                 else if ( infof[0] < 0 ) i = 2;
01523                                 else i = *infof;
01524                                 NCOPY(u,infof,i)
01525                             }
01526                             t += 2; goto nextt;
01527                         }
01528                         infof += 2;
01529                         NEXTARG(infof);
01530                     }
01531                     else {
01532                         NEXTARG(infof);
01533                         NEXTARG(infof);
01534                     }
01535                 }
01536             /*
01537             Here we had no match in the style of 1->2. It could however
01538             be that xarg_ does something
01539             */
01540             doanyway = 1; n2 = t[1];
01541         }
01542     }
01543     else { /* Tensor */
01544         if ( *t < AM.OffsetIndex && *t >= 0 ) {
01545             infof = info + FUNHEAD;
01546             while ( infof < infoend ) {
01547                 if ( (*infof == -SNUMBER) && ( infof[1] == *t )
01548                     && ( ( infof[2] == -SNUMBER ) && ( infof[3] >= 0 )
01549                     && ( infof[3] < AM.OffsetIndex ) )
01550                     || ( infof[2] == -INDEX || infof[2] == -VECTOR
01551                     || infof[2] == -MINVECTOR ) ) ) {
01552                 *u++ = infof[3];
01553                 infof += 4;
01554                 t++; goto nextt;
01555             }
01556             else {
01557                 NEXTARG(infof);
01558                 NEXTARG(infof);
01559             }
01560         }

```

```

01561     }
01562     }
01563 }
01564 else if ( *t == -SNUMBER ) {
01565     doanyway = 1; n2 = t[1];
01566 }
01567 /*
01568 First we try to catch those elements that have an exact match
01569 in the traditional replace_ part.
01570 This means that *t should be less than zero and match an entry
01571 in the replace_ function that we prepared.
01572 */
01573 if ( ngeneral > 0 ) {
01574     if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
01575         if ( *t < 0 ) {
01576             term3 = term1 + *term1;
01577             term4 = term1 + FUNHEAD;
01578             while ( term4 < term3 ) {
01579                 if ( *term4 == *t && ( *t <= -FUNCTION ||
01580                     ( t[1] == term4[1] ) ) ) break;
01581                 NEXTARG(term4)
01582             }
01583             if ( term4 < term3 ) goto dothisnow;
01584         }
01585     }
01586     else {
01587         term3 = term1 + *term1;
01588         term4 = term1 + FUNHEAD;
01589         while ( term4 < term3 ) {
01590             if ( ( term4[1] == *t ) &&
01591                 ( ( *term4 == -INDEX || *term4 == -VECTOR ||
01592                   ( *term4 == -SYMBOL && term4[1] < AM.OffsetIndex
01593                     && term4[1] >= 0 ) ) ) ) break;
01594             NEXTARG(term4)
01595         }
01596         if ( term4 < term3 ) goto dothisnow;
01597     }
01598 }
01599 /*
01600 First we eliminate the fixed arguments and make a 'new info'
01601 If there is anything left we can continue.
01602 Now we look for whole argument wildcards (arg_,parg_,iarg_ or farg_)
01603 */
01604 if ( nwild > 0 ) {
01605     /*
01606         If we have f(a)*replace_(xarg_,b(xarg_)) this gives f(b(a))
01607         In testing the wildcard we have CheckWild do the work.
01608         This means that we have to set up the special variables
01609         (AT.WildMask,AN.WildValue,AN.NumWild)
01610     */
01611     wild[1] = 4;
01612     info1 = info + FUNHEAD;
01613     while ( info1 < infoend ) {
01614         if ( *info1 == -SYMBOL && info1[1] == WILDARGSYMBOL
01615             && ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) ) {
01616             wild[0] = SYMTOSUB;
01617             wild[2] = WILDARGSYMBOL;
01618             wild[3] = 0;
01619             AN.WildValue = wild;
01620             AT.WildMask = &mask;
01621             mask = 0;
01622             AN.NumWild = 1;
01623             if ( *t == -SYMBOL || ( *t > 0 && CheckWild(BHEAD WILDARGSYMBOL,SYMTOSUB,1,t) == 0
01624 )
01625             || doanyway ) {
01626                 /*
01627                     We put the part in replace in a function and make
01628                     a replace_(xarg_,(t argument)).
01629                 */
01630                 n1 = SYMBOL; n2 = WILDARGSYMBOL;
01631                 info1 += 2;
01632                 getthisone;;
01633                 term3 = term2+1;
01634                 if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
01635                     *term3++ = DUMFUN; term3++; FILLFUN(term3)
01636                     COPY1ARG(term3,info1)
01637                 }
01638                 else {
01639                     *term3++ = fun[0]; term3++; FILLFUN(term3)
01640                     *term3++ = *info1;
01641                 }
01642                 term2[2] = term3 - term2 - 1;
01643                 tt = term3;
01644                 *term3++ = REPLACEMENT;
01645                 term3++; FILLFUN(term3)
01646                 *term3++ = -n1;

```

```

01647         if ( n1 < FUNCTION ) *term3++ = n2;
01648         if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
01649             term4 = t;
01650             COPY1ARG(term3,term4)
01651         }
01652         else {
01653             *term3++ = *t;
01654         }
01655         tt[1] = term3 - tt;
01656         *term3++ = 1; *term3++ = 1; *term3++ = 3;
01657         *term2 = term3 - term2;
01658
01659         AT.WorkPointer = term3;
01660         NewSort(BHEAD0);
01661         if ( Generator(BHEAD term2,AR.Cnumlhs) ) {
01662             LowerSortLevel();
01663             AT.WorkPointer = oldwork;
01664             AT.WildMask = oldmask;
01665             return(-1);
01666         }
01667         term4 = AT.WorkPointer;
01668         if ( EndSort(BHEAD term4,1) < 0 ) {}
01669         if ( ( *term4 && *(term4+*term4) != 0 ) || *term4 == 0 ) {
01670             MLOCK(ErrorMessageLock);
01671             MesPrint("&information in replace transformation does not evaluate into a
single term");
01672             MUNLOCK(ErrorMessageLock);
01673             return(-1);
01674         }
01675         /*
01676         Now we can copy the new function argument to the output u
01677         */
01678         i = term4[2]-FUNHEAD;
01679         term3 = term4+FUNHEAD+1;
01680         NCOPY(u,term3,i)
01681         if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
01682             NEXTARG(t)
01683         }
01684         else t++;
01685         AT.WorkPointer = term2;
01686
01687         goto nextt;
01688     }
01689     info1 += 2; NEXTARG(info1)
01690 }
01691 else if ( ( *info1 == -INDEX )
01692     && ( info[1] == WILDARGINDEX + AM.OffsetIndex ) ) {
01693     wild[0] = INDTO SUB;
01694     wild[2] = WILDARGINDEX+AM.OffsetIndex;
01695     wild[3] = 0;
01696     AN.WildValue = wild;
01697     AT.WildMask = &mask;
01698     mask = 0;
01699     AN.NumWild = 1;
01700     if ( ( functions[fun[0]-FUNCTION].spec == TENSORFUNCTION )
01701         || ( *t == -INDEX || ( *t > 0 && CheckWild(BHEAD WILDARGINDEX,INDTO SUB,1,t) == 0 )
) ) {
01702         /*
01703         We put the part in replace in a function and make
01704         a replace_(xarg_,(t argument)).
01705         */
01706         n1 = INDEX; n2 = WILDARGINDEX+AM.OffsetIndex;
01707         info1 += 2;
01708         goto getthisone;
01709     }
01710     info1 += 2; NEXTARG(info1)
01711 }
01712 else if ( ( *info1 == -VECTOR )
01713     && ( info[1] == WILDARGVECTOR + AM.OffsetVector ) ) {
01714     wild[0] = VECTOSUB;
01715     wild[2] = WILDARGVECTOR+AM.OffsetVector;
01716     wild[3] = 0;
01717     AN.WildValue = wild;
01718     AT.WildMask = &mask;
01719     mask = 0;
01720     AN.NumWild = 1;
01721     if ( functions[fun[0]-FUNCTION].spec == TENSORFUNCTION ) {
01722         if ( *t < MINSPEC ) {
01723             n1 = VECTOR; n2 = WILDARGVECTOR+AM.OffsetVector;
01724             info1 += 2;
01725             goto getthisone;
01726         }
01727     }
01728     else if ( *t == -VECTOR || *t == -MINVECTOR ||
( *t > 0 && CheckWild(BHEAD WILDARGVECTOR,VECTOSUB,1,t) == 0 ) ) {
01729
01730         /*
01731         We put the part in replace in a function and make

```

```

01732             a replace_(xarg_,(t argument)).
01733 /*
01734             n1 = VECTOR; n2 = WILDARGVECTOR+AM.OffsetVector;
01735             info1 += 2;
01736             goto getthisone;
01737         }
01738         info1 += 2; NEXTARG(info1)
01739     }
01740     else if ( *info1 == -WILDARGFUN ) {
01741         wild[0] = FUNTOFUN;
01742         wild[2] = WILDARGFUN;
01743         wild[3] = 0;
01744         AN.WildValue = wild;
01745         AT.WildMask = &mask;
01746         mask = 0;
01747         AN.NumWild = 1;
01748         if ( *t <= -FUNCTION || ( *t > 0 && CheckWild(BHEAD WILDARGFUN,FUNTOFUN,1,t) == 0
    ) ) {
01749 /*
01750             We put the part in replace in a function and make
01751             a replace_(xarg_,(t argument)).
01752 */
01753             n2 = n1 = -WILDARGFUN; /* n2 is to keep the compiler quiet */
01754             info1++;
01755             goto getthisone;
01756         }
01757         info1++; NEXTARG(info1)
01758     }
01759     else {
01760         NEXTARG(info1) NEXTARG(info1)
01761     }
01762 }
01763 }
01764 if ( ngeneral > 0 ) {
01765 /*
01766     They are all in a replace_ function.
01767     Compose the whole thing into a term with replace_()*dum_(arg)
01768     which will be given to Generator.
01769     If we have f(a(x))*replace_(x,b) this gives f(a(b))
01770 */
01771 dothisnow;
01772 term3 = term2; term4 = term1; i = *term1;
01773 NCOPY(term3,term4,i)
01774 term4 = term3;
01775 if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
01776     *term3++ = DUMFUN; term3++; FILLFUN(term3);
01777     tt = t;
01778     COPY1ARG(term3,tt)
01779 }
01780 else {
01781     *term3++ = fun[0]; term3++; FILLFUN(term3); *term3++ = *t;
01782 }
01783 term4[1] = term3-term4;
01784 *term3++ = 1; *term3++ = 1; *term3++ = 3;
01785 *term2 = term3-term2;
01786 AT.WorkPointer = term3;
01787 NewSort(BHEAD0);
01788 if ( Generator(BHEAD term2,AR.Cnumlhs) ) {
01789     LowerSortLevel();
01790     AT.WorkPointer = oldwork;
01791     AT.WildMask = oldmask;
01792     return(-1);
01793 }
01794 term4 = AT.WorkPointer;
01795 if ( EndSort(BHEAD term4,1) < 0 ) {}
01796 if ( ( *term4 && *(term4+*term4) != 0 ) || *term4 == 0 ) {
01797     MLOCK(ErrorMessageLock);
01798     MesPrint("&information in replace transformation does not evaluate into a single
term");
01799     MUNLOCK(ErrorMessageLock);
01800     return(-1);
01801 }
01802 /*
01803     Now we can copy the new function argument to the output u
01804 */
01805 i = term4[2]-FUNHEAD;
01806 term3 = term4+FUNHEAD+1;
01807 NCOPY(u,term3,i)
01808 NEXTARG(t)
01809 AT.WorkPointer = term2;
01810 goto nextt;
01811 }
01812 }
01813
01814 /*
01815     No catch. Copy the argument and continue.
01816 */

```

```

01817         if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
01818             if ( *t <= -FUNCTION ) { *u++ = *t++; }
01819             else if ( *t < 0 ) { *u++ = *t++; *u++ = *t++; }
01820             else { i = *t; NCOPY(u,t,i) }
01821         }
01822         else {
01823             *u++ = *t++;
01824         }
01825     nextt++;
01826 }
01827 i = u - tstop; tstop[1] = i; tstop[2] = dirty;
01828 t = fun; u = tstop; NCOPY(t,u,i)
01829 AT.WorkPointer = oldwork;
01830 AT.WildMask = oldmask;
01831 return(0);
01832 }
01833
01834 /*
01835     #] RunReplace :
01836     #[ RunImplode :
01837
01838     Note that we restrict ourselves to short integers and/or single symbols
01839 */
01840
01841 int RunImplode(WORD *fun, WORD *args)
01842 {
01843     GETIDENTITY
01844     WORD *tt, *tstop, totarg, arg1, arg2, num1, num2, i1, n;
01845     WORD *f, *t, *ttt, *t4, *ff, *fff;
01846     WORD moveup, numzero, outspace;
01847     if ( functions[fun[0]-FUNCTION].spec != 0 ) return(0);
01848     if ( *args != ARGGRANGE ) {
01849         MLOCK(ErrorMessageLock);
01850         MesPrint("Illegal range encountered in RunImplode");
01851         MUNLOCK(ErrorMessageLock);
01852         Terminate(-1);
01853     }
01854     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
01855     while ( tt < tstop ) { totarg++; NEXTARG(tt); }
01856     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
01857 /*
01858     Get the proper range in forward direction and the number of arguments
01859 */
01860     if ( arg1 > arg2 ) { num1 = arg2; num2 = arg1; }
01861     else { num1 = arg1; num2 = arg2; }
01862     if ( num1 > totarg || num2 > totarg ) return(0);
01863 /*
01864     We need, for the most general case 4 spots for each:
01865         x,pow,coef,sign
01866     Hence we put these in the workspace above the term after tstop
01867 */
01868     n = 1; f = fun+FUNHEAD;
01869     while ( n < num1 ) {
01870         if ( f >= tstop ) return(0);
01871         NEXTARG(f);
01872         n++;
01873     }
01874     ff = f;
01875 /*
01876     We are now at the first argument to be done
01877     Go through the terms and test their validity.
01878     If one of them doesn't conform to the rules we don't do anything.
01879     The terms to be done are put in special notation after the function.
01880     Notation: numsymbol, power, [coef], sign
01881     If numsymbol is negative there is no symbol.
01882     We do it this way because otherwise stepping backwards (as in range=(4,1))
01883     would be very difficult.
01884 */
01885     tt = tstop;
01886     while ( n <= num2 ) {
01887         if ( f >= tstop ) return(0);
01888         if ( *f == -SNUMBER ) { *tt++ = -1; *tt++ = 0;
01889             if ( f[1] < 0 ) { *tt++ = -f[1]; *tt++ = -1; }
01890             else { *tt++ = f[1]; *tt++ = 1; }
01891             f += 2;
01892         }
01893         else if ( *f == -SYMBOL ) { *tt++ = f[1]; *tt++ = 1; *tt++ = 1; *tt++ = 1; f += 2; }
01894         else if ( *f < 0 ) return(0);
01895         else {
01896             if ( *f != ( f[ARGHEAD]+ARGHEAD ) ) return(0); /* Not a single term */
01897             t = f + *f - 1;
01898             i1 = ABS(*t);
01899             if ( ( i1 > 3 ) || ( t[-1] != 1 ) ) return(0); /* Not an integer or too big */
01900             if ( (UWORD)(t[-2]) > MAXPOSITIVE4 ) return(0); /* number too big */
01901             if ( f[ARGHEAD] == i1+1 ) { /* numerical which is fine */
01902                 *tt++ = -1; *tt++ = 0; *tt++ = t[-2];
01903                 if ( *t < 0 ) { *tt++ = -1; }

```



```

01904         else { *tt++ = 1; }
01905     }
01906     else if ( ( f[ARGHEAD+1] != SYMBOL )
01907         || ( f[ARGHEAD+2] != 4 )
01908         || ( ( f+ARGHEAD+1+f[ARGHEAD+2] ) < ( t-i1 ) ) ) return(0);
01909     /* not a single symbol with a coefficient */
01910     else {
01911         *tt++ = f[ARGHEAD+3];
01912         *tt++ = f[ARGHEAD+4];
01913         *tt++ = t[-2];
01914         if ( *t < 0 ) { *tt++ = -1; }
01915         else { *tt++ = 1; }
01916     }
01917     f += *f;
01918 }
01919 n++;
01920 }
01921 fff = f;
01922 /*
01923 At this point we can do the implosion.
01924 Requirement: no coefficient shall take more than one word.
01925 (a stricter requirement may be needed to keep the explosion contained)
01926 */
01927 if ( arg1 > arg2 ) {
01928 /*
01929     Work backward.
01930 */
01931     t = tt - 4; numzero = 0;
01932     while ( t >= tstop ) {
01933         if ( t[2] == 0 ) numzero++;
01934         else {
01935             if ( numzero > 0 ) {
01936                 t[2] += numzero;
01937                 t4 = t+4;
01938                 ttt = t4 + 4*numzero;
01939                 while ( ttt < tt ) *t4++ = *ttt++;
01940                 tt -= 4*numzero;
01941                 numzero = 0;
01942             }
01943         }
01944         t -= 4;
01945     }
01946 }
01947 else {
01948     t = tstop;
01949     numzero = 0; ttt = t;
01950     while ( t < tt ) {
01951         if ( t[2] == 0 ) numzero++;
01952         else {
01953             if ( numzero > 0 ) {
01954                 t[2] += numzero;
01955                 t4 = t;
01956                 while ( t4 < tt ) *ttt++ = *t4++;
01957                 tt -= 4*numzero;
01958                 t -= 4*numzero;
01959                 ttt = t + 4;
01960                 numzero = 0;
01961             }
01962             else {
01963                 ttt = t + 4;
01964             }
01965         }
01966         t += 4;
01967     }
01968 /*
01969     We may have numzero > 0 at the end. We leave them.
01970     Output space is currently from tstop to tt
01971 */
01972 }
01973 /*
01974 Now we compute the real output space needed
01975 */
01976 t = tstop; outspace = 0;
01977 while ( t < tt ) {
01978     if ( t[0] == -1 ) {
01979         if ( t[2] > MAXPOSITIVE4 ) { return(0); /* Number too big */ }
01980         outspace += 2;
01981     }
01982     else if ( t[1] == 1 && t[2] == 1 && t[3] == 1 ) { outspace += 2; }
01983     else { outspace += 8 + ARGHEAD; }
01984     t += 4;
01985 }
01986 if ( outspace < ( fff-ff ) ) {
01987     t = tstop;
01988     while ( t < tt ) {
01989         if ( t[0] == -1 ) { *ff++ = -SNUMBER; *ff++ = t[2]*t[3]; }
01990         else if ( t[1] == 1 && t[2] == 1 && t[3] == 1 ) {

```

```

01991         *ff++ = -SYMBOL; *ff++ = t[0];
01992     }
01993     else {
01994         *ff++ = 8+ARGHEAD; *ff++ = 0; FILLARG(ff);
01995         *ff++ = 8; *ff++ = SYMBOL; *ff++ = 4; *ff++ = t[0]; *ff++ = t[1];
01996         *ff++ = t[2]; *ff++ = 1; *ff++ = t[3] > 0 ? 3: -3;
01997     }
01998     t += 4;
01999 }
02000 while ( fff < tstop ) *ff++ = *fff++;
02001 fun[1] = ff - fun;
02002 }
02003 else if ( outspace > (fff-ff) ) {
02004 /*
02005     Move the answer up by the required amount.
02006     Move the tail to its new location
02007     Move in things as for outspace == (fff-ff)
02008 */
02009     moveup = outspace-(fff-ff);
02010     ttt = tt + moveup;
02011     t = tt;
02012     while ( t > fff ) *--ttt = *--t;
02013     tt += moveup; tstop += moveup;
02014     fff += moveup;
02015     fun[1] += moveup;
02016     goto moveinto;
02017 }
02018 else {
02019 moveinto:
02020     t = tstop;
02021     while ( t < tt ) {
02022         if ( t[0] == -1 ) { *ff++ = -SNUMBER; *ff++ = t[2]*t[3]; }
02023         else if ( t[1] == 1 && t[2] == 1 && t[3] == 1 ) {
02024             *ff++ = -SYMBOL; *ff++ = t[0];
02025         }
02026         else {
02027             *ff++ = 8+ARGHEAD; *ff++ = 0; FILLARG(ff);
02028             *ff++ = 8; *ff++ = SYMBOL; *ff++ = 4; *ff++ = t[0]; *ff++ = t[1];
02029             *ff++ = t[2]; *ff++ = 1; *ff++ = t[3] > 0 ? 3: -3;
02030         }
02031         t += 4;
02032     }
02033 }
02034 return(0);
02035 }
02036
02037 /*
02038     #] RunImplode :
02039     #[ RunExplode :
02040 */
02041
02042 int RunExplode(PHEAD WORD *fun, WORD *args)
02043 {
02044     WORD arg1, arg2, num1, num2, *tt, *tstop, totarg, *tonew, *newfun;
02045     WORD *ff, *f;
02046     int reverse = 0, iarg, i, numzero;
02047     if ( functions[fun[0]-FUNCTION].spec != 0 ) return(0);
02048     if ( *args != ARGGRANGE ) {
02049         MLOCK(ErrorMessageLock);
02050         MesPrint("Illegal range encountered in RunExplode");
02051         MUNLOCK(ErrorMessageLock);
02052         Terminate(-1);
02053     }
02054     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
02055     while ( tt < tstop ) { totarg++; NEXTARG(tt); }
02056     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02057 /*
02058     Get the proper range in forward direction and the number of arguments
02059 */
02060     if ( arg1 > arg2 ) { num1 = arg2; num2 = arg1; reverse = 1; }
02061     else { num1 = arg1; num2 = arg2; }
02062     if ( num1 > totarg || num2 > totarg ) return(0);
02063     if ( tstop + AM.MaxTer > AT.WorkTop ) goto OverWork;
02064 /*
02065     We will make the new function after the old one in the workspace
02066     Find the first argument
02067 */
02068     tonew = newfun = tstop;
02069     ff = fun + FUNHEAD; iarg = 0;
02070     while ( ff < tstop ) {
02071         iarg++;
02072         if ( iarg == num1 ) {
02073             i = ff - fun; f = fun;
02074             NCOPY(tonew,f,i)
02075             break;
02076         }
02077         NEXTARG(ff)

```

```

02078     }
02079     /*
02080     We have reached the first argument to be done
02081     */
02082     while ( iarg <= num2 ) {
02083         if ( *ff == -SYMBOL || ( *ff == -SNUMBER && ff[1] == 0 ) )
02084             { *tonew++ = *ff++; *tonew++ = *ff++; }
02085         else if ( *ff == -SNUMBER ) {
02086             numzero = ABS(ff[1])-1;
02087             if ( reverse ) {
02088                 *tonew++ = -SNUMBER; *tonew++ = ff[1] < 0 ? -1: 1;
02089                 while ( numzero > 0 ) {
02090                     *tonew++ = -SNUMBER; *tonew++ = 0; numzero--;
02091                 }
02092             }
02093             else {
02094                 while ( numzero > 0 ) {
02095                     *tonew++ = -SNUMBER; *tonew++ = 0; numzero--;
02096                 }
02097                 *tonew++ = -SNUMBER; *tonew++ = ff[1] < 0 ? -1: 1;
02098             }
02099             ff += 2;
02100         }
02101         else if ( *ff < 0 ) { return(0); }
02102         else {
02103             if ( *ff != ARGHEAD+8 || ff[ARGHEAD] != 8
02104                 || ff[ARGHEAD+1] != SYMBOL || ABS(ff[ARGHEAD+7]) != 3
02105                 || ff[ARGHEAD+6] != 1 ) return(0);
02106             numzero = ff[ARGHEAD+5];
02107             if ( numzero >= MAXPOSITIVE4 ) return(0);
02108             numzero--;
02109             if ( reverse ) {
02110                 if ( ff[ARGHEAD+7] > 0 ) { *tonew++ = -SNUMBER; *tonew++ = 1; }
02111                 else {
02112                     *tonew++ = ARGHEAD+8; *tonew++ = 0; FILLARG(tonew)
02113                     *tonew++ = 8; *tonew++ = SYMBOL; *tonew++ = ff[ARGHEAD+3];
02114                     *tonew++ = ff[ARGHEAD+4]; *tonew++ = 1; *tonew++ = 1;
02115                     *tonew++ = -3;
02116                 }
02117                 while ( numzero > 0 ) {
02118                     *tonew++ = -SNUMBER; *tonew++ = 0; numzero--;
02119                 }
02120             }
02121             else {
02122                 while ( numzero > 0 ) {
02123                     *tonew++ = -SNUMBER; *tonew++ = 0; numzero--;
02124                 }
02125                 *tonew++ = ARGHEAD+8; *tonew++ = 0; FILLARG(tonew)
02126                 *tonew++ = 8; *tonew++ = SYMBOL; *tonew++ = 4;
02127                 *tonew++ = ff[ARGHEAD+3]; *tonew++ = ff[ARGHEAD+4];
02128                 *tonew++ = 1; *tonew++ = 1;
02129                 if ( ff[ARGHEAD+7] > 0 ) *tonew++ = 3;
02130                 else *tonew++ = -3;
02131             }
02132             ff += *ff;
02133         }
02134         if ( tonew > AT.WorkTop ) goto OverWork;
02135         iarg++;
02136     }
02137     /*
02138     Copy the tail, settle the size and copy the whole thing back.
02139     */
02140     while ( ff < tstop ) *tonew++ = *ff++;
02141     i = newfun[1] = tonew-newfun;
02142     NCOPY(fun,newfun,i)
02143     return(0);
02144 OverWork:;
02145     MLOCK(ErrorMessageLock);
02146     MesWork();
02147     MUNLOCK(ErrorMessageLock);
02148     return(-1);
02149 }
02150
02151 /*
02152     #] RunExplode :
02153     #[ RunPermute :
02154     */
02155
02156 int RunPermute(PHEAD WORD *fun, WORD *args, WORD *info)
02157 {
02158     WORD *tt, totarg, *tstop, arg1, arg2, n, num, i, *f, *f1, *f2, *infostop;
02159     WORD *in, *iw, withdollar;
02160     DOLLARS d;
02161     if ( *args != ARGRANGE ) {
02162         MLOCK(ErrorMessageLock);
02163         MesPrint("Illegal range encountered in RunPermute");
02164         MUNLOCK(ErrorMessageLock);

```

```

02165         Terminate(-1);
02166     }
02167     if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
02168         tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
02169         while ( tt < tstop ) { totarg++; NEXTARG(tt); }
02170         arg1 = 1; arg2 = totarg;
02171     /*
02172         We need to:
02173         1: get pointers to the arguments
02174         2: permute the pointers
02175         3: copy the arguments to safe territory in the new order
02176         4: copy this new order back in situ.
02177     */
02178     num = arg2-arg1+1;
02179     WantAddPointers(num); /* Guarantees the presence of enough pointers */
02180     f = fun+FUNHEAD; n = 1; i = 0;
02181     while ( n < arg1 ) { n++; NEXTARG(f) }
02182     f1 = f;
02183     while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; NEXTARG(f) }
02184 /*
02185     Now the permutations
02186 */
02187     info++;
02188     while ( *info ) {
02189         infostop = info + *info;
02190         info++;
02191         if ( *info > totarg ) return(0);
02192     /*
02193         Now we have a look whether there are dollar variables to be expanded
02194         We also shift out all values that are out of range.
02195     */
02196     withdollar = 0; in = info;
02197     while ( in < infostop ) {
02198         if ( *in < 0 ) { /* Dollar variable -(number+1) */
02199             d = Dollars - *in - 1;
02200 #ifdef WITHPTHREADS
02201             {
02202                 int nummodopt, dtype = -1, numdollar = -*in-1;
02203                 if ( AS.MultiThreaded && ( AC.mparallelflag == PARALLELFLAG ) ) {
02204                     for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
02205                         if ( numdollar == ModOptdollars[nummodopt].number ) break;
02206                     }
02207                     if ( nummodopt < NumModOptdollars ) {
02208                         dtype = ModOptdollars[nummodopt].type;
02209                         if ( dtype == MODLOCAL ) {
02210                             d = ModOptdollars[nummodopt].dstruct+AT.identity;
02211                         }
02212                         else {
02213                             LOCK(d->pthreadlockread);
02214                         }
02215                     }
02216                 }
02217             }
02218 #endif
02219             if ( ( d->type == DOLNUMBER || d->type == DOLTERMS )
02220                 && d->where[0] == 4 && d->where[4] == 0 ) {
02221                 if ( d->where[3] < 0 || d->where[2] != 1 || d->where[1] > totarg ) return(0);
02222             }
02223             else if ( d->type == DOLWILDARGS ) {
02224                 iw = d->where+1;
02225                 while ( *iw ) {
02226                     if ( *iw == -SNUMBER ) {
02227                         if ( iw[1] <= 0 || iw[1] > totarg ) return(0);
02228                     }
02229                     else goto IllType;
02230                     iw += 2;
02231                 }
02232             }
02233             else {
02234 IllType:
02235                 MLOCK(ErrorMessageLock);
02236                 MesPrint("Illegal type of $-variable in RunPermute");
02237                 MUNLOCK(ErrorMessageLock);
02238                 Terminate(-1);
02239             }
02240             withdollar++;
02241         }
02242         else if ( *in > totarg ) return(0);
02243         in++;
02244     }
02245     if ( withdollar ) { /* We need some space for a copy */
02246         WORD *incopy, *tocopy;
02247         incopy = TermMalloc("RunPermute");
02248         tocopy = incopy+1; in = info;
02249         while ( in < infostop ) {
02250             if ( *in < 0 ) {
02251                 d = Dollars - *in - 1;

```

```

02252 #ifdef WITHPTHREADS
02253 {
02254     int nummodopt, dtype = -1, numdollar = -*in-1;
02255     if ( AS.MultiThreaded && ( AC.mparallelfld == PARALLELFLAG ) ) {
02256         for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
02257             if ( numdollar == ModOptdollars[nummodopt].number ) break;
02258         }
02259         if ( nummodopt < NumModOptdollars ) {
02260             dtype = ModOptdollars[nummodopt].type;
02261             if ( dtype == MODLOCAL ) {
02262                 d = ModOptdollars[nummodopt].dstruct+AT.identity;
02263             }
02264             else {
02265                 LOCK(d->pthreadlockread);
02266             }
02267         }
02268     }
02269 }
02270 #endif
02271     if ( d->type == DOLNUMBER || d->type == DOLTERMS ) {
02272         *tocopy++ = d->where[1] - 1;
02273     }
02274     else if ( d->type == DOLWILDARGS ) {
02275         iw = d->where+1;
02276         while ( *iw ) {
02277             *tocopy++ = iw[1] - 1;
02278             iw += 2;
02279         }
02280     }
02281     in++;
02282 }
02283     else *tocopy++ = *in++;
02284 }
02285 *tocopy = 0;
02286 *incopy = tocopy - incopy;
02287 in = incopy+1;
02288 tt = AT.pWorkSpace[AT.pWorkPointer+*in];
02289 in++;
02290 while ( in < tocopy ) {
02291     if ( *in > totarg ) return(0);
02292     AT.pWorkSpace[AT.pWorkPointer+in-1] = AT.pWorkSpace[AT.pWorkPointer+*in];
02293     in++;
02294 }
02295 AT.pWorkSpace[AT.pWorkPointer+in-1] = tt;
02296 TermFree(incopy,"RunPermute");
02297 info = infostop;
02298 }
02299 else {
02300     tt = AT.pWorkSpace[AT.pWorkPointer+*info];
02301     info++;
02302     while ( info < infostop ) {
02303         if ( *info > totarg ) return(0);
02304         AT.pWorkSpace[AT.pWorkPointer+info-1] = AT.pWorkSpace[AT.pWorkPointer+*info];
02305         info++;
02306     }
02307     AT.pWorkSpace[AT.pWorkPointer+info-1] = tt;
02308 }
02309 }
02310 /*
02311     info++;
02312     while ( *info ) {
02313         infostop = info + *info;
02314         info++;
02315         if ( *info > totarg ) return(0);
02316         tt = AT.pWorkSpace[AT.pWorkPointer+*info];
02317         info++;
02318         while ( info < infostop ) {
02319             if ( *info > totarg ) return(0);
02320             AT.pWorkSpace[AT.pWorkPointer+info-1] = AT.pWorkSpace[AT.pWorkPointer+*info];
02321             info++;
02322         }
02323         AT.pWorkSpace[AT.pWorkPointer+info-1] = tt;
02324     }
02325 */
02326 /*
02327     And the final cleanup
02328 */
02329     if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
02330     f2 = tstop;
02331     for ( i = 0; i < num; i++ ) { f = AT.pWorkSpace[AT.pWorkPointer+i]; COPY1ARG(f2,f) }
02332     i = f2 - tstop;
02333     NCOPY(f1,tstop,i)
02334 }
02335 else { /* tensors */
02336     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = tstop-tt;
02337     arg1 = 1; arg2 = totarg;
02338     num = arg2-arg1+1;

```

```

02339     WantAddPointers(num); /* Guarantees the presence of enough pointers */
02340     f = fun+FUNHEAD; n = 1; i = 0;
02341     while ( n < arg1 ) { n++; f++; }
02342     f1 = f;
02343     while ( n <= arg2 ) { AT.pWorkspace[AT.pWorkPointer+i++] = f; n++; f++; }
02344 /*
02345     Now the permutations
02346 */
02347     info++;
02348     while ( *info ) {
02349         infostop = info + *info;
02350         info++;
02351         if ( *info > totarg ) return(0);
02352         tt = AT.pWorkspace[AT.pWorkPointer+*info];
02353         info++;
02354         while ( info < infostop ) {
02355             if ( *info > totarg ) return(0);
02356             AT.pWorkspace[AT.pWorkPointer+info[-1]] = AT.pWorkspace[AT.pWorkPointer+*info];
02357             info++;
02358         }
02359         AT.pWorkspace[AT.pWorkPointer+info[-1]] = tt;
02360     }
02361 /*
02362     And the final cleanup
02363 */
02364     if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
02365     f2 = tstop;
02366     for ( i = 0; i < num; i++ ) { f = AT.pWorkspace[AT.pWorkPointer+i]; *f2++ = *f++; }
02367     i = f2 - tstop;
02368     NCOPY(f1,tstop,i)
02369 }
02370 return(0);
02371 OverWork:;
02372 MLOCK(ErrorMessageLock);
02373 MesWork();
02374 MUNLOCK(ErrorMessageLock);
02375 return(-1);
02376 }
02377
02378 /*
02379     #] RunPermute :
02380     #[ RunReverse :
02381 */
02382
02383 int RunReverse(PHEAD WORD *fun, WORD *args)
02384 {
02385     WORD *tt, totarg, *tstop, arg1, arg2, n, num, i, *f, *f1, *f2, i1, i2;
02386     if ( *args != ARGRANGE ) {
02387         MLOCK(ErrorMessageLock);
02388         MesPrint("Illegal range encountered in RunReverse");
02389         MUNLOCK(ErrorMessageLock);
02390         Terminate(-1);
02391     }
02392     if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
02393         tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
02394         while ( tt < tstop ) { totarg++; NEXTARG(tt); }
02395         if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02396 /*
02397         We need to:
02398         1: get pointers to the arguments
02399         2: reverse the order of the pointers
02400         3: copy the arguments to safe territory in the new order
02401         4: copy this new order back in situ.
02402 */
02403         if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
02404         if ( arg2 > totarg ) return(0);
02405
02406         num = arg2-arg1+1;
02407         WantAddPointers(num); /* Guarantees the presence of enough pointers */
02408         f = fun+FUNHEAD; n = 1; i = 0;
02409         while ( n < arg1 ) { n++; NEXTARG(f) }
02410         f1 = f;
02411         while ( n <= arg2 ) { AT.pWorkspace[AT.pWorkPointer+i++] = f; n++; NEXTARG(f) }
02412         i1 = i-1; i2 = 0;
02413         while ( i1 > i2 ) {
02414             tt = AT.pWorkspace[AT.pWorkPointer+i1];
02415             AT.pWorkspace[AT.pWorkPointer+i1] = AT.pWorkspace[AT.pWorkPointer+i2];
02416             AT.pWorkspace[AT.pWorkPointer+i2] = tt;
02417             i1--; i2++;
02418         }
02419         if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
02420         f2 = tstop;
02421         for ( i = 0; i < num; i++ ) { f = AT.pWorkspace[AT.pWorkPointer+i]; COPY1ARG(f2,f) }
02422         i = f2 - tstop;
02423         NCOPY(f1,tstop,i)
02424     }
02425     else { /* Tensors */

```

```

02426     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = tstop - tt;
02427     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02428 /*
02429     We need to:
02430         1: get pointers to the arguments
02431         2: reverse the order of the pointers
02432         3: copy the arguments to safe territory in the new order
02433         4: copy this new order back in situ.
02434 */
02435     if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
02436     if ( arg2 > totarg ) return(0);
02437
02438     num = arg2-arg1+1;
02439     WantAddPointers(num); /* Guarantees the presence of enough pointers */
02440     f = fun+FUNHEAD; n = 1; i = 0;
02441     while ( n < arg1 ) { n++; f++; }
02442     f1 = f;
02443     while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; f++; }
02444     i1 = i-1; i2 = 0;
02445     while ( i1 > i2 ) {
02446         tt = AT.pWorkSpace[AT.pWorkPointer+i1];
02447         AT.pWorkSpace[AT.pWorkPointer+i1] = AT.pWorkSpace[AT.pWorkPointer+i2];
02448         AT.pWorkSpace[AT.pWorkPointer+i2] = tt;
02449         i1--; i2++;
02450     }
02451     if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
02452     f2 = tstop;
02453     for ( i = 0; i < num; i++ ) { f = AT.pWorkSpace[AT.pWorkPointer+i]; *f2++ = *f++; }
02454     i = f2 - tstop;
02455     NCOPY(f1,tstop,i)
02456 }
02457 return(0);
02458 OverWork:;
02459 MLOCK(ErrorMessageLock);
02460 MesWork();
02461 MUNLOCK(ErrorMessageLock);
02462 return(-1);
02463 }
02464
02465 /*
02466     #] RunReverse :
02467     #[ RunDedup :
02468 */
02469
02470 int RunDedup(PHEAD WORD *fun, WORD *args)
02471 {
02472     WORD *tt, totarg, *tstop, arg1, arg2, n, i, j,k, *f, *f1, *f2, *fd, *fstart;
02473     if ( *args != ARGRANGE ) {
02474         MLOCK(ErrorMessageLock);
02475         MesPrint("Illegal range encountered in RunDedup");
02476         MUNLOCK(ErrorMessageLock);
02477         Terminate(-1);
02478     }
02479     if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
02480         tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
02481         while ( tt < tstop ) { totarg++; NEXTARG(tt); }
02482         if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02483
02484         if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
02485         if ( arg2 > totarg ) return(0);
02486
02487         f = fun+FUNHEAD; n = 1;
02488         while ( n < arg1 ) { n++; NEXTARG(f) }
02489         f1 = f; // fast forward to first element in range
02490         i = 0; // new argument count
02491         fstart = f1;
02492
02493         for (; n <= arg2; n++ ) {
02494             f2 = fstart;
02495             for ( j = 0; j < i; j++ ) { // check all previous terms
02496                 fd = f2;
02497                 NEXTARG(fd)
02498                 for ( k = 0; k < fd-f2; k++ ) // byte comparison of args
02499                     if ( f2[k] != f[k] ) break;
02500
02501                 if ( k == fd-f2 ) break; // duplicate arg
02502                 f2 = fd;
02503             }
02504
02505             if ( j == i ) {
02506                 // unique factor, copy in situ
02507                 COPY1ARG(f1,f)
02508                 i++;
02509             } else {
02510                 NEXTARG(f)
02511             }
02512         }

```

```

02513
02514 // move the terms from after the range
02515 for (j = n; j <= totarg; j++) {
02516     COPY1ARG(f1,f)
02517 }
02518
02519 fun[1] = f1 - fun; // resize function
02520 }
02521 else { /* Tensors */
02522     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = tstop - tt;
02523     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02524
02525     if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
02526     if ( arg2 > totarg ) return(0);
02527
02528     f = fun+FUNHEAD;
02529     i = arg1; // new argument count
02530     n = i;
02531
02532     for (; n <= arg2; n++) {
02533         for ( j = arg1; j < i; j++) { // check all previous terms
02534             if ( f[n-1] == f[j-1] ) break; // duplicate arg
02535         }
02536
02537         if ( j == i ) {
02538             // unique factor, copy in situ
02539             f[i-1] = f[n-1];
02540             i++;
02541         }
02542     }
02543
02544 // move the terms from after the range
02545 for (j = n; j <= totarg; j++, i++) {
02546     f[i-1] = f[j-1];
02547 }
02548
02549 fun[1] = f + i - 1 - fun; // resize function
02550 }
02551 return(0);
02552 }
02553
02554 /*
02555     #] RunDedup :
02556     #[ RunCycle :
02557 */
02558
02559 int RunCycle(PHEAD WORD *fun, WORD *args, WORD *info)
02560 {
02561     WORD *tt, totarg, *tstop, arg1, arg2, n, num, i, j, *f, *f1, *f2, x, ncyc, cc;
02562     if ( *args != ARGGRANGE ) {
02563         MLOCK(ErrorMessageLock);
02564         MesPrint("Illegal range encountered in RunCycle");
02565         MUNLOCK(ErrorMessageLock);
02566         Terminate(-1);
02567     }
02568     ncyc = info[1];
02569     if ( ncyc >= MAXPOSITIVE2 ) { /* $ variable */
02570         ncyc -= MAXPOSITIVE2;
02571         if ( ncyc >= MAXPOSITIVE4 ) {
02572             ncyc -= MAXPOSITIVE4; /* -$ */
02573             cc = -1;
02574         }
02575         else cc = 1;
02576         ncyc = DolToNumber(BHEAD ncyc);
02577         if ( AN.ErrorInDollar ) {
02578             MesPrint(" Error in Dollar variable in transform,cycle()=$");
02579             return(-1);
02580         }
02581         if ( ncyc >= MAXPOSITIVE4 || ncyc <= -MAXPOSITIVE4 ) {
02582             MesPrint(" Illegal value from Dollar variable in transform,cycle()=$");
02583             return(-1);
02584         }
02585         ncyc *= cc;
02586     }
02587     if ( functions[fun[0]-FUNCTION].spec != TENSORFUNCTION ) {
02588         tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
02589         while ( tt < tstop ) { totarg++; NEXTARG(tt); }
02590         if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02591         if ( arg1 > arg2 ) { n = arg1; arg1 = arg2; arg2 = n; }
02592         if ( arg2 > totarg ) return(0);
02593     /*
02594     We need to:
02595     1: get pointers to the arguments
02596     2: cycle the pointers
02597     3: copy the arguments to safe territory in the new order
02598     4: copy this new order back in situ.
02599     */

```



```

02600     num = arg2-arg1+1;
02601     WantAddPointers(num); /* Guarantees the presence of enough pointers */
02602     f = fun+FUNHEAD; n = 1; i = 0;
02603     while ( n < arg1 ) { n++; NEXTARG(f) }
02604     f1 = f;
02605     while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; NEXTARG(f) }
02606 /*
02607     Now the cycle(s). First minimize the number of cycles.
02608 */
02609     x = ncyc;
02610     if ( x >= i ) {
02611         x %= i;
02612         if ( x > i/2 ) x -= i;
02613     }
02614     else if ( x <= -i ) {
02615         x = -((-x) % i);
02616         if ( x <= -i/2 ) x += i;
02617     }
02618     while ( x ) {
02619         if ( x > 0 ) {
02620             tt = AT.pWorkSpace[AT.pWorkPointer+i-1];
02621             for ( j = i-1; j > 0; j-- )
02622                 AT.pWorkSpace[AT.pWorkPointer+j] = AT.pWorkSpace[AT.pWorkPointer+j-1];
02623             AT.pWorkSpace[AT.pWorkPointer] = tt;
02624             x--;
02625         }
02626         else {
02627             tt = AT.pWorkSpace[AT.pWorkPointer];
02628             for ( j = 1; j < i; j++ )
02629                 AT.pWorkSpace[AT.pWorkPointer+j-1] = AT.pWorkSpace[AT.pWorkPointer+j];
02630             AT.pWorkSpace[AT.pWorkPointer+j-1] = tt;
02631             x++;
02632         }
02633     }
02634 /*
02635     And the final cleanup
02636 */
02637     if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
02638     f2 = tstop;
02639     for ( i = 0; i < num; i++ ) { f = AT.pWorkSpace[AT.pWorkPointer+i]; COPY1ARG(f2,f) }
02640     i = f2 - tstop;
02641     NCOPY(f1,tstop,i)
02642 }
02643 else { /* Tensors */
02644     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = tstop - tt;
02645     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02646     if ( arg1 > arg2 ) { n = arg1; arg1 = arg2; arg2 = n; }
02647     if ( arg2 > totarg ) return(0);
02648 /*
02649     We need to:
02650     1: get pointers to the arguments
02651     2: cycle the pointers
02652     3: copy the arguments to safe territory in the new order
02653     4: copy this new order back in situ.
02654 */
02655     num = arg2-arg1+1;
02656     WantAddPointers(num); /* Guarantees the presence of enough pointers */
02657     f = fun+FUNHEAD; n = 1; i = 0;
02658     while ( n < arg1 ) { n++; f++; }
02659     f1 = f;
02660     while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; f++; }
02661 /*
02662     Now the cycle(s). First minimize the number of cycles.
02663 */
02664     x = ncyc;
02665     if ( x >= i ) {
02666         x %= i;
02667         if ( x > i/2 ) x -= i;
02668     }
02669     else if ( x <= -i ) {
02670         x = -((-x) % i);
02671         if ( x <= -i/2 ) x += i;
02672     }
02673     while ( x ) {
02674         if ( x > 0 ) {
02675             tt = AT.pWorkSpace[AT.pWorkPointer+i-1];
02676             for ( j = i-1; j > 0; j-- )
02677                 AT.pWorkSpace[AT.pWorkPointer+j] = AT.pWorkSpace[AT.pWorkPointer+j-1];
02678             AT.pWorkSpace[AT.pWorkPointer] = tt;
02679             x--;
02680         }
02681         else {
02682             tt = AT.pWorkSpace[AT.pWorkPointer];
02683             for ( j = 1; j < i; j++ )
02684                 AT.pWorkSpace[AT.pWorkPointer+j-1] = AT.pWorkSpace[AT.pWorkPointer+j];
02685             AT.pWorkSpace[AT.pWorkPointer+j-1] = tt;
02686             x++;

```

```

02687     }
02688     }
02689     /*
02690     And the final cleanup
02691     */
02692     if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
02693     f2 = tstop;
02694     for ( i = 0; i < num; i++ ) { f = AT.pWorkspace[AT.pWorkPointer+i]; *f2++ = *f++; }
02695     i = f2 - tstop;
02696     NCOPY(f1,tstop,i)
02697     }
02698     return(0);
02699 OverWork:;
02700     MLOCK(ErrorMessageLock);
02701     MesWork();
02702     MUNLOCK(ErrorMessageLock);
02703     return(-1);
02704 }
02705
02706 /*
02707     #] RunCycle :
02708     #[ RunAddArg :
02709     */
02710
02711 int RunAddArg(PHEAD WORD *fun, WORD *args)
02712 {
02713     WORD *tt, totarg, *tstop, arg1, arg2, n, num, *f, *f1, *f2;
02714     WORD scribble[10+ARGHEAD];
02715     LONG space;
02716     if ( *args != ARGRANGE ) {
02717         MLOCK(ErrorMessageLock);
02718         MesPrint("Illegal range encountered in RunAddArg");
02719         MUNLOCK(ErrorMessageLock);
02720         Terminate(-1);
02721     }
02722     if ( functions[fun[0]-FUNCTION].spec == TENSORFUNCTION ) {
02723         MLOCK(ErrorMessageLock);
02724         MesPrint("Illegal attempt to add arguments of a tensor in AddArg");
02725         MUNLOCK(ErrorMessageLock);
02726         Terminate(-1);
02727     }
02728     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
02729     while ( tt < tstop ) { totarg++; NEXTARG(tt); }
02730     /* ignore functions with no arguments */
02731     if ( totarg == 0 ) return(0);
02732     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02733     /*
02734     We need to:
02735         1: establish that we actually need to add something
02736         2: start a sort
02737         3: if needed, convert arguments to long arguments
02738         4: send (terms in) argument to StoreTerm
02739         5: EndSort and copy the result back into the function
02740     Note that the function is in the workspace, above the term and no
02741     relevant information is trailing it.
02742     */
02743     if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
02744     if ( arg2 > totarg ) return(0);
02745     num = arg2-arg1+1;
02746     if ( num == 1 ) return(0);
02747     f = fun+FUNHEAD; n = 1;
02748     while ( n < arg1 ) { n++; NEXTARG(f) }
02749     f1 = f;
02750     NewSort(BHEAD0);
02751     while ( n <= arg2 ) {
02752         if ( *f > 0 ) {
02753             f2 = f + *f; f += ARGHEAD;
02754             while ( f < f2 ) { StoreTerm(BHEAD f); f += *f; }
02755         }
02756         else if ( *f == -SNUMBER && f[1] == 0 ) {
02757             f+= 2;
02758         }
02759         else {
02760             ToGeneral(f,scribble,1);
02761             StoreTerm(BHEAD scribble);
02762             NEXTARG(f);
02763         }
02764         n++;
02765     }
02766     if ( EndSort(BHEAD tstop+ARGHEAD,1) < 0 ) return(-1);
02767     num = 0;
02768     f2 = tstop+ARGHEAD;
02769     while ( *f2 ) { f2 += *f2; num++; }
02770     *tstop = f2-tstop;
02771     for ( n = 1; n < ARGHEAD; n++ ) tstop[n] = 0;
02772     if ( num == 1 && ToFast(tstop,tstop) == 1 ) {
02773         f2 = tstop; NEXTARG(f2);

```

```

02774     }
02775     if ( *tstop == ARGHEAD ) {
02776         *tstop = -SNUMBER; tstop[1] = 0;
02777         f2 = tstop+2;
02778     }
02779     /*
02780     Copy the trailing arguments after the new argument, then copy the whole back.
02781     */
02782     while ( f < tstop ) *f2++ = *f++;
02783     while ( f < f2 ) *f1++ = *f++;
02784     space = f1 - fun;
02785     if ( (space+8)*sizeof(WORD) > (UWORD)AM.MaxTer ) {
02786         MLOCK(ErrorMessageLock);
02787         MesWork();
02788         MUNLOCK(ErrorMessageLock);
02789         return(-1);
02790     }
02791     fun[1] = (WORD)space;
02792     return(0);
02793 }
02794
02795 /*
02796     #] RunAddArg :
02797     #[ RunMulArg :
02798 */
02799
02800 int RunMulArg(PHEAD WORD *fun, WORD *args)
02801 {
02802     WORD *t, totarg, *tstop, arg1, arg2, n, *f, nb, *m, i, *w;
02803     WORD *scratch, argbuf[20], argsize, *where, *newterm;
02804     LONG oldcpointer_pos;
02805     CBUF *C = cbuf + AT.ebufnum;
02806     if ( *args != ARGGRANGE ) {
02807         MLOCK(ErrorMessageLock);
02808         MesPrint("Illegal range encountered in RunMulArg");
02809         MUNLOCK(ErrorMessageLock);
02810         Terminate(-1);
02811     }
02812     if ( functions[fun[0]-FUNCTION].spec == TENSORFUNCTION ) {
02813         MLOCK(ErrorMessageLock);
02814         MesPrint("Illegal attempt to multiply arguments of a tensor in MulArg");
02815         MUNLOCK(ErrorMessageLock);
02816         Terminate(-1);
02817     }
02818     t = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
02819     while ( t < tstop ) { totarg++; NEXTARG(t); }
02820     /* ignore functions with no arguments */
02821     if ( totarg == 0 ) return(0);
02822     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02823     if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
02824     if ( arg1 > totarg ) return(0);
02825     if ( arg2 < 1 ) return(0);
02826     if ( arg1 < 1 ) arg1 = 1;
02827     if ( arg2 > totarg ) arg2 = totarg;
02828     if ( arg1 == arg2 ) return(0);
02829     /*
02830     Now we move the arguments to a compiler buffer
02831     Then we create a term in the workspace that is the product of
02832     subexpression pointers to the objects in the compiler buffer.
02833     Next we let Generator work out that term.
02834     Finally we pick up the results from EndSort and put it in the function.
02835     */
02836     f = fun+FUNHEAD; n = 1;
02837     while ( n < arg1 ) { n++; NEXTARG(f) }
02838     t = f;
02839     if ( fun >= AT.WorkSpace && fun < AT.WorkTop ) {
02840         if ( AT.WorkPointer < fun+fun[1] ) AT.WorkPointer = fun+fun[1];
02841     }
02842     scratch = AT.WorkPointer;
02843     w = scratch+1;
02844     oldcpointer_pos = C->Pointer-C->Buffer;
02845     nb = C->numrhs;
02846     while ( n <= arg2 ) {
02847         if ( *t > 0 ) {
02848             argsize = *t - ARGHEAD; where = t + ARGHEAD; t += *t;
02849         }
02850         else if ( *t <= -FUNCTION ) {
02851             argbuf[0] = FUNHEAD+4; argbuf[1] = -*t++; argbuf[2] = FUNHEAD;
02852             for ( i = 2; i < FUNHEAD; i++ ) argbuf[i+1] = 0;
02853             argbuf[FUNHEAD+1] = 1;
02854             argbuf[FUNHEAD+2] = 1;
02855             argbuf[FUNHEAD+3] = 3;
02856             argsize = argbuf[0];
02857             where = argbuf;
02858         }
02859         else if ( *t == -SYMBOL ) {
02860             argbuf[0] = 8; argbuf[1] = SYMBOL; argbuf[2] = 4;

```

```

02861         argbuf[3] = t[1]; argbuf[4] = 1;
02862         argbuf[5] = 1; argbuf[6] = 1; argbuf[7] = 3;
02863         argsize = 8; t += 2;
02864         where = argbuf;
02865     }
02866     else if ( *t == -VECTOR || *t == -MINVECTOR ) {
02867         argbuf[0] = 7; argbuf[1] = INDEX; argbuf[2] = 3;
02868         argbuf[3] = t[1];
02869         argbuf[4] = 1; argbuf[5] = 1;
02870         if ( *t == -MINVECTOR ) argbuf[6] = -3;
02871         else argbuf[6] = 3;
02872         argsize = 7; t += 2;
02873         where = argbuf;
02874     }
02875     else if ( *t == -INDEX ) {
02876         argbuf[0] = 7; argbuf[1] = INDEX; argbuf[2] = 3;
02877         argbuf[3] = t[1];
02878         argbuf[4] = 1; argbuf[5] = 1; argbuf[6] = 3;
02879         argsize = 7; t += 2;
02880         where = argbuf;
02881     }
02882     else if ( *t == -SNUMBER ) {
02883         if ( t[1] < 0 ) {
02884             argbuf[0] = 4; argbuf[1] = -t[1]; argbuf[2] = 1; argbuf[3] = -3;
02885         }
02886         else {
02887             argbuf[0] = 4; argbuf[1] = t[1]; argbuf[2] = 1; argbuf[3] = 3;
02888         }
02889         argsize = 4; t += 2;
02890         where = argbuf;
02891     }
02892     else {
02893         /* unreachable */
02894         return(1);
02895     }
02896 /*
02897     Now add the argbuf to AT.ebufnum
02898 */
02899     m = AddRHS(AT.ebufnum,1);
02900     while ( (m + argsize + 10) > C->Top ) m = DoubleCbuffer(AT.ebufnum,m,17);
02901     for ( i = 0; i < argsize; i++ ) m[i] = where[i];
02902     m[i] = 0;
02903     C->Pointer = m + i + 1;
02904     n++;
02905     *w++ = SUBEXPRESSION; *w++ = SUBEXPSIZE; *w++ = C->numrhs; *w++ = 1;
02906     *w++ = AT.ebufnum; FILLSUB(w);
02907 }
02908 *w++ = 1; *w++ = 1; *w++ = 3;
02909 *scratch = w-scratch;
02910 AT.WorkPointer = w;
02911 NewSort(BHEAD0);
02912 Generator(BHEAD scratch,AR.Cnumlhs);
02913 newterm = AT.WorkPointer;
02914 EndSort(BHEAD newterm+ARGHEAD,1);
02915 C->Pointer = C->Buffer+oldcpointer_pos;
02916 C->numrhs = nb;
02917 w = newterm+ARGHEAD; while ( *w ) w += *w;
02918 *newterm = w-newterm; newterm[1] = 0;
02919 if ( ToFast(newterm,newterm) ) {
02920     if ( *newterm <= -FUNCTION ) w = newterm+1;
02921     else w = newterm+2;
02922 }
02923 while ( t < tstop ) *w++ = *t++;
02924 i = w - newterm;
02925 t = newterm; NCOPY(f,t,i);
02926 fun[1] = f-fun;
02927 AT.WorkPointer = scratch;
02928 if ( AT.WorkPointer > AT.WorkSpace && AT.WorkPointer < f ) AT.WorkPointer = f;
02929 return(0);
02930 }
02931
02932 /*
02933     #] RunMulArg :
02934     #[ RunIsLyndon :
02935
02936     Determines whether the range constitutes a Lyndon word.
02937     The two cases of ordering are distinguished by the order of
02938     the numbers of the arguments in the range.
02939 */
02940
02941 int RunIsLyndon(PHEAD WORD *fun, WORD *args, int par)
02942 {
02943     WORD *tt, totarg, *tstop, arg1, arg2, arg, num, *f, n, i;
02944     /* WORD *f1; */
02945     WORD sign, i1, i2, retval;
02946     if ( fun[0] <= GAMMASEVEN && fun[0] >= GAMMA ) return(0);
02947     if ( *args != ARGRANGE ) {

```

```

02948     MLOCK(ErrorMessageLock);
02949     MesPrint("Illegal range encountered in RunIsLyndon");
02950     MUNLOCK(ErrorMessageLock);
02951     Terminate(-1);
02952 }
02953 tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
02954 while ( tt < tstop ) { totarg++; NEXTARG(tt); }
02955 if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
02956 if ( arg1 > totarg || arg2 > totarg ) return(-1);
02957 /*
02958 Now make a list of the relevant arguments.
02959 */
02960 if ( arg1 == arg2 ) return(1);
02961 if ( arg2 < arg1 ) { /* greater, rather than smaller */
02962     arg = arg1; arg1 = arg2; arg2 = arg; sign = 1;
02963 }
02964 else sign = 0;
02965
02966 num = arg2-arg1+1;
02967 WantAddPointers(num); /* Guarantees the presence of enough pointers */
02968 f = fun+FUNHEAD; n = 1; i = 0;
02969 while ( n < arg1 ) { n++; NEXTARG(f) }
02970 /* f1 = f; */
02971 while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; NEXTARG(f) }
02972 /*
02973 If sign == 1 we should alter the order of the pointers first
02974 */
02975 if ( sign ) {
02976     i1 = i-1; i2 = 0;
02977     while ( i1 > i2 ) {
02978         tt = AT.pWorkSpace[AT.pWorkPointer+i1];
02979         AT.pWorkSpace[AT.pWorkPointer+i1] = AT.pWorkSpace[AT.pWorkPointer+i2];
02980         AT.pWorkSpace[AT.pWorkPointer+i2] = tt;
02981         i1--; i2++;
02982     }
02983 }
02984 /*
02985 The argument range is from f1 to f and the num pointers to the arguments
02986 are in AT.pWorkSpace[AT.pWorkPointer] to AT.pWorkSpace[AT.pWorkPointer+num-1]
02987 */
02988 for ( i1 = 1; i1 < num; i1++ ) {
02989     retval = par * CompArg(AT.pWorkSpace[AT.pWorkPointer+i1],
02990         AT.pWorkSpace[AT.pWorkPointer]);
02991     if ( retval > 0 ) continue;
02992     if ( retval < 0 ) return(0);
02993     for ( i2 = 1; i2 < num; i2++ ) {
02994         retval = par * CompArg(AT.pWorkSpace[AT.pWorkPointer+(i1+i2)%num],
02995             AT.pWorkSpace[AT.pWorkPointer+i2]);
02996         if ( retval < 0 ) return(0);
02997         if ( retval > 0 ) goto nextil;
02998     }
02999 /*
03000 If we come here the sequence is not unique.
03001 */
03002     return(0);
03003 nextil:;
03004 }
03005 return(1);
03006 }
03007
03008 /*
03009 #] RunIsLyndon :
03010 #[ RunToLyndon :
03011
03012 Determines whether the range constitutes a Lyndon word.
03013 If not, we rotate it to a Lyndon word. If this is not possible
03014 we return the noLyndon condition.
03015 The two cases of ordering are distinguished by the order of
03016 the numbers of the arguments in the range.
03017 */
03018
03019 WORD RunToLyndon(PHEAD WORD *fun, WORD *args, int par)
03020 {
03021     WORD *ttt, totarg, *tstop, arg1, arg2, arg, num, *f, *f1, *f2, n, i;
03022     WORD sign, i1, i2, retval, unique;
03023     if ( fun[0] <= GAMMASEVEN && fun[0] >= GAMMA ) return(0);
03024     if ( *args != ARGRANGE ) {
03025         MLOCK(ErrorMessageLock);
03026         MesPrint("Illegal range encountered in RunToLyndon");
03027         MUNLOCK(ErrorMessageLock);
03028         Terminate(-1);
03029     }
03030     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
03031     while ( tt < tstop ) { totarg++; NEXTARG(tt); }
03032     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
03033     if ( arg1 > totarg || arg2 > totarg ) return(-1);
03034 /*

```

```

03035     Now make a list of the relevant arguments.
03036 */
03037     if ( arg1 == arg2 ) return(1);
03038     if ( arg2 < arg1 ) { /* greater, rather than smaller */
03039         arg = arg1; arg1 = arg2; arg2 = arg; sign = 1;
03040     }
03041     else sign = 0;
03042
03043     num = arg2-arg1+1;
03044     WantAddPointers((2*num)); /* Guarantees the presence of enough pointers */
03045     f = fun+FUNHEAD; n = 1; i = 0;
03046     while ( n < arg1 ) { n++; NEXTARG(f) }
03047     f1 = f;
03048     while ( n <= arg2 ) { AT.pWorkSpace[AT.pWorkPointer+i++] = f; n++; NEXTARG(f) }
03049 */
03050     If sign == 1 we should alter the order of the pointers first
03051 */
03052     if ( sign ) {
03053         i1 = i-1; i2 = 0;
03054         while ( i1 > i2 ) {
03055             tt = AT.pWorkSpace[AT.pWorkPointer+i1];
03056             AT.pWorkSpace[AT.pWorkPointer+i1] = AT.pWorkSpace[AT.pWorkPointer+i2];
03057             AT.pWorkSpace[AT.pWorkPointer+i2] = tt;
03058             i1--; i2++;
03059         }
03060     }
03061 */
03062     The argument range is from f1 to f and the num pointers to the arguments
03063     are in AT.pWorkSpace[AT.pWorkPointer] to AT.pWorkSpace[AT.pWorkPointer+num-1]
03064 */
03065     unique = 1;
03066     for ( i1 = 1; i1 < num; i1++ ) {
03067         retval = par * CompArg(AT.pWorkSpace[AT.pWorkPointer+i1],
03068                               AT.pWorkSpace[AT.pWorkPointer]);
03069         if ( retval > 0 ) continue;
03070         if ( retval < 0 ) {
03071 Rotate:;
03072 /*
03073         Rotate so that i1 becomes the zero element. Then start again.
03074 */
03075         for ( i2 = 0; i2 < num; i2++ ) {
03076             AT.pWorkSpace[AT.pWorkPointer+num+i2] =
03077                 AT.pWorkSpace[AT.pWorkPointer+(i1+i2)%num];
03078         }
03079         for ( i2 = 0; i2 < num; i2++ ) {
03080             AT.pWorkSpace[AT.pWorkPointer+i2] =
03081                 AT.pWorkSpace[AT.pWorkPointer+i2+num];
03082         }
03083         i1 = 0;
03084         goto nextil;
03085     }
03086     for ( i2 = 1; i2 < num; i2++ ) {
03087         retval = par * CompArg(AT.pWorkSpace[AT.pWorkPointer+(i1+i2)%num],
03088                               AT.pWorkSpace[AT.pWorkPointer+i2]);
03089         if ( retval < 0 ) goto Rotate;
03090         if ( retval > 0 ) goto nextil;
03091     }
03092 */
03093     If we come here the sequence is not unique.
03094 */
03095     unique = 0;
03096 nextil:;
03097     }
03098     if ( sign ) {
03099         i1 = i-1; i2 = 0;
03100         while ( i1 > i2 ) {
03101             tt = AT.pWorkSpace[AT.pWorkPointer+i1];
03102             AT.pWorkSpace[AT.pWorkPointer+i1] = AT.pWorkSpace[AT.pWorkPointer+i2];
03103             AT.pWorkSpace[AT.pWorkPointer+i2] = tt;
03104             i1--; i2++;
03105         }
03106     }
03107 */
03108     Now rewrite the arguments into the proper order
03109 */
03110     if ( tstop+(f-f1) > AT.WorkTop ) goto OverWork;
03111     f2 = tstop;
03112     for ( i = 0; i < num; i++ ) { f = AT.pWorkSpace[AT.pWorkPointer+i]; COPY1ARG(f2,f) }
03113     i = f2 - tstop;
03114     NCOPY(f1,tstop,i)
03115 */
03116     The return value indicates whether we have a Lyndon word
03117 */
03118     return(unique);
03119 OverWork:;
03120     MLOCK(ErrorMessageLock);
03121     MesWork();

```

```

03122     MUNLOCK(ErrorMessageLock);
03123     return(-2);
03124 }
03125
03126 /*
03127     #] RunToLyndon :
03128     #[ RunDropArg :
03129 */
03130
03131 int RunDropArg(PHEAD WORD *fun, WORD *args)
03132 {
03133     WORD *t, *tstop, *f, totarg, arg1, arg2, n;
03134
03135     t = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
03136     while ( t < tstop ) { totarg++; NEXTARG(t); }
03137     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
03138     if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
03139     if ( arg1 > totarg ) return(0);
03140     if ( arg2 < 1 ) return(0);
03141     if ( arg1 < 1 ) arg1 = 1;
03142     if ( arg2 > totarg ) arg2 = totarg;
03143     f = fun+FUNHEAD; n = 1;
03144     while ( n < arg1 ) { n++; NEXTARG(f) }
03145     t = f;
03146     while ( n <= arg2 ) { n++; NEXTARG(t) }
03147     while ( t < tstop ) *f++ = *t++;
03148     fun[1] = f-fun;
03149     return(0);
03150 }
03151
03152 /*
03153     #] RunDropArg :
03154     #[ RunSelectArg :
03155 */
03156
03157 int RunSelectArg(PHEAD WORD *fun, WORD *args)
03158 {
03159     WORD *t, *tstop, *f, *tt, totarg, arg1, arg2, n;
03160
03161     t = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
03162     while ( t < tstop ) { totarg++; NEXTARG(t); }
03163     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
03164     if ( arg2 < arg1 ) { n = arg1; arg1 = arg2; arg2 = n; }
03165     if ( arg1 > totarg ) return(0);
03166     if ( arg2 < 1 ) return(0);
03167     if ( arg1 < 1 ) arg1 = 1;
03168     if ( arg2 > totarg ) arg2 = totarg;
03169     f = fun+FUNHEAD; n = 1; t = f;
03170     while ( n < arg1 ) { n++; NEXTARG(t) }
03171     while ( n <= arg2 ) {
03172         tt = t; NEXTARG(tt)
03173         while ( t < tt ) *f++ = *t++;
03174         n++;
03175     }
03176     fun[1] = f-fun;
03177     return(0);
03178 }
03179
03180 /*
03181     #] RunSelectArg :
03182     #[ RunZtoHArg :
03183 */
03184
03185 int RunZtoHArg(PHEAD WORD *fun, WORD *args)
03186 {
03187     WORD *tt, totarg, *tstop, arg1, arg2, n, i, *f, *f1;
03188     int sign = 0;
03189     WORD *t, *t1, *t2, *t3;
03190     if ( *args != ARGGRANGE ) {
03191         MLOCK(ErrorMessageLock);
03192         MesPrint("Illegal range encountered in RunZtoHArg.");
03193         MUNLOCK(ErrorMessageLock);
03194         Terminate(-1);
03195     }
03196     if ( functions[fun[0]-FUNCTION].spec != 0 ) {
03197         MLOCK(ErrorMessageLock);
03198         MesPrint("The ZtoH transformation can only be executed on regular functions with nonzero
integer arguments.");
03199         MUNLOCK(ErrorMessageLock);
03200         Terminate(-1);
03201     }
03202     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
03203     while ( tt < tstop ) { totarg++; NEXTARG(tt); }
03204     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
03205     /*
03206     Check the arguments. Should be -SNUMBER x!=0
03207     */

```

```

03208     f = fun+FUNHEAD; n = 1;
03209     while ( n < arg1 ) { n++; NEXTARG(f) }
03210     f1 = f;
03211     for ( i = arg1; i <= arg2; i++, f += 2 ) {
03212         if ( *f != -SNUMBER || f[1] == 0 ) return(-1);
03213     }
03214     /*
03215     Now we need a copy.
03216     */
03217     t = f1; t1 = t2 = tt = TermMalloc("RunZtoHArg");
03218     while ( t < f ) { *t1++ = *t++; *t1++ = *t++; }
03219     t = f1;
03220     while ( t2 < t1 ) {
03221         t += 2;
03222         if ( t2[1] < 0 ) {
03223             t3 = t;
03224             while ( t3 < f ) { t3[1] = -t3[1]; t3 += 2; }
03225         }
03226         t2 += 2;
03227     }
03228     TermFree(tt,"RunZtoHArg");
03229     /*
03230     Now the overall sign.
03231     */
03232     while ( f1 < f ) { if ( f1[1] < 0 ) sign = 1-sign; f1 += 2; }
03233     return(sign);
03234 }
03235
03236 /*
03237     #] RunZtoHArg :
03238     #[ RunHtoZArg :
03239     */
03240
03241 int RunHtoZArg(PHEAD WORD *fun, WORD *args)
03242 {
03243     WORD *tt, totarg, *tstop, arg1, arg2, n, i, *f, *f1, *f2;
03244     int sign = 0;
03245     WORD *t, *t1, *t2;
03246     if ( *args != ARGRANGE ) {
03247         MLOCK(ErrorMessageLock);
03248         MesPrint("Illegal range encountered in RunZtoHArg.");
03249         MUNLOCK(ErrorMessageLock);
03250         Terminate(-1);
03251     }
03252     if ( functions[fun[0]-FUNCTION].spec != 0 ) {
03253         MLOCK(ErrorMessageLock);
03254         MesPrint("The HtoZ transformation can only be executed on regular functions with nonzero
integer arguments.");
03255         MUNLOCK(ErrorMessageLock);
03256         Terminate(-1);
03257     }
03258     tt = fun+FUNHEAD; tstop = fun+fun[1]; totarg = 0;
03259     while ( tt < tstop ) { totarg++; NEXTARG(tt); }
03260     if ( FindRange(BHEAD args,&arg1,&arg2,totarg) ) return(-1);
03261     /*
03262     Check the arguments. Should be -SNUMBER x!=0
03263     */
03264     f = fun+FUNHEAD; n = 1;
03265     while ( n < arg1 ) { n++; NEXTARG(f) }
03266     f2 = f1 = f;
03267     for ( i = arg1; i <= arg2; i++, f += 2 ) {
03268         if ( *f != -SNUMBER || f[1] == 0 ) return(-1);
03269     }
03270     /*
03271     First the overall sign.
03272     */
03273     while ( f2 < f ) { if ( f2[1] < 0 ) sign = 1-sign; f2 += 2; }
03274     /*
03275     Now we need a copy.
03276     */
03277     t = f1; t1 = tt = TermMalloc("RunHtoZArg");
03278     while ( t < f ) { *t1++ = *t++; *t1++ = *t++; }
03279     /*
03280     Now the transformation.
03281     */
03282     t = f1; t2 = tt + 2;
03283     while ( t2 < t1 ) {
03284         t += 2;
03285         if ( t2[-1] < 0 ) t[1] = -t[1];
03286         t2 += 2;
03287     }
03288     TermFree(tt,"RunHtoZArg");
03289     return(sign);
03290 }
03291
03292 /*
03293     #] RunHtoZArg :

```



```

03294     #] TestArgNum :
03295
03296     Looks whether argument n is contained in any of the ranges
03297     specified in args. Args contains objects of the types
03298         ALLARGS
03299         NUMARG,num
03300         ARGRANGE,num1,num2
03301     The object MAKEARGS,num1,num2 is skipped
03302     Any other object terminates the range specifications.
03303
03304     Currently only ARGRANGE is used (10-may-2016)
03305 */
03306
03307 int TestArgNum(int n, int totarg, WORD *args)
03308 {
03309     GETIDENTITY
03310     WORD x1, x2;
03311     for(;;) {
03312         switch ( *args ) {
03313             case ALLARGS:
03314                 return(1);
03315             case NUMARG:
03316                 if ( n == args[1] ) return(1);
03317                 if ( args[1] >= MAXPOSITIVE4 ) {
03318                     x1 = args[1]-MAXPOSITIVE4;
03319                     if ( totarg-x1 == n ) return(1);
03320                 }
03321                 args += 2;
03322                 break;
03323             case ARGRANGE:
03324                 if ( args[1] >= MAXPOSITIVE2 ) {
03325                     x1 = args[1] - MAXPOSITIVE2;
03326                     if ( x1 > MAXPOSITIVE4 ) {
03327                         x1 = x1 - MAXPOSITIVE4;
03328                         x1 = DolToNumber(BHEAD x1);
03329                         x1 = totarg - x1;
03330                     }
03331                     else {
03332                         x1 = DolToNumber(BHEAD x1);
03333                     }
03334                 }
03335                 else if ( args[1] >= MAXPOSITIVE4 ) {
03336                     x1 = totarg-(args[1]-MAXPOSITIVE4);
03337                 }
03338                 else x1 = args[1];
03339                 if ( args[2] >= MAXPOSITIVE2 ) {
03340                     x2 = args[2] - MAXPOSITIVE2;
03341                     if ( x2 > MAXPOSITIVE4 ) {
03342                         x2 = x2 - MAXPOSITIVE4;
03343                         x2 = DolToNumber(BHEAD x2);
03344                         x2 = totarg - x2;
03345                     }
03346                     else {
03347                         x2 = DolToNumber(BHEAD x2);
03348                     }
03349                 }
03350                 else if ( args[2] >= MAXPOSITIVE4 ) {
03351                     x2 = totarg-(args[2]-MAXPOSITIVE4);
03352                 }
03353                 else x2 = args[2];
03354                 if ( x1 >= x2 ) {
03355                     if ( n >= x2 && n <= x1 ) return(1);
03356                 }
03357                 else {
03358                     if ( n >= x1 && n <= x2 ) return(1);
03359                 }
03360                 args += 3;
03361                 break;
03362             case MAKEARGS:
03363                 args += 3;
03364                 break;
03365             default:
03366                 return(0);
03367         }
03368     }
03369 }
03370
03371 /*
03372     #] TestArgNum :
03373     #] PutArgInScratch :
03374 */
03375
03376 WORD PutArgInScratch(WORD *arg,UWORD *scrat)
03377 {
03378     WORD size, *t, i;
03379     if ( *arg == -SNUMBER ) {
03380         scrat[0] = ABS(arg[1]);

```

```

03381         if ( arg[1] < 0 ) size = -1;
03382         else size = 1;
03383     }
03384     else {
03385         t = arg+*arg-1;
03386         if ( *t < 0 ) { i = ((-t)-1)/2; size = -i; }
03387         else { i = ( *t -1)/2; size = i; }
03388         t = arg+ARGHEAD+1;
03389         NCOPY(scrat,t,i);
03390     }
03391     return(size);
03392 }
03393
03394 /*
03395     #] PutArgInScratch :
03396     #[ ReadRange :
03397
03398     Comes in at the bracket and leaves at the = sign
03399     Ranges can be:
03400         #1,#2 with # numbers. If the second is smaller than the
03401             first we work it backwards.
03402         first,#2 or #2,first
03403         #1,last or last,#1
03404         first,last or last,first
03405     First is represented by 1. Last is represented by MAXPOSITIVE4.
03406
03407     par = 0: we need the = after.
03408     par = 1: we need a , or '\0' after.
03409     par = 2: we need a :
03410 */
03411
03412 UBYTE *ReadRange(UBYTE *s, WORD *out, int par)
03413 {
03414     UBYTE *in = s, *ss, c;
03415     LONG x1, x2;
03416
03417     SKIPBRA3(in)
03418     if ( par == 0 && in[1] != '=' ) {
03419         MesPrint("%A range in this type of transform statement should be followed by an = sign");
03420         return(0);
03421     }
03422     else if ( par == 1 && in[1] != ',' && in[1] != '\0' ) {
03423         MesPrint("%A range in this type of transform statement should be followed by a comma or
03424 end-of-statement");
03425         return(0);
03426     }
03427     else if ( par == 2 && in[1] != ':' ) {
03428         MesPrint("%A range in this type of transform statement should be followed by a :");
03429         return(0);
03430     }
03431     s++;
03432     if ( FG.cTable[*s] == 0 ) {
03433         ss = s; while ( FG.cTable[*s] == 0 ) s++;
03434         c = *s; *s = 0;
03435         if ( StrICmp(ss,(UBYTE *)"first") == 0 ) {
03436             *s = c;
03437             x1 = 1;
03438         }
03439         else if ( StrICmp(ss,(UBYTE *)"last") == 0 ) {
03440             *s = c;
03441             if ( c == '-' ) {
03442                 s++;
03443                 if ( *s == '$' ) {
03444                     s++; ss = s;
03445                     while ( FG.cTable[*s] == 0 || FG.cTable[*s] == 1 ) s++;
03446                     c = *s; *s = 0;
03447                     if ( ( x1 = GetDollar(ss) ) < 0 ) goto Error;
03448                     *s = c;
03449                     x1 += MAXPOSITIVE2;
03450                 }
03451                 else {
03452                     x1 = 0;
03453                     while ( *s >= '0' && *s <= '9' ) {
03454                         x1 = 10*x1 + *s++ - '0';
03455                         if ( x1 >= MAXPOSITIVE4 ) {
03456                             MesPrint("&Fixed range indicator bigger than %1", (LONG)MAXPOSITIVE4);
03457                             return(0);
03458                         }
03459                     }
03460                 }
03461             }
03462             x1 += MAXPOSITIVE4;
03463         }
03464         else x1 = MAXPOSITIVE4;
03465     }
03466     else {
03467         MesPrint("&Illegal keyword inside range specification");
03468         return(0);
03469     }

```

```

03467     }
03468 }
03469 else if ( FG.cTable[*s] == 1 ) {
03470     x1 = 0;
03471     while ( *s >= '0' && *s <= '9' ) {
03472         x1 = x1*10 + *s++ - '0';
03473         if ( x1 >= MAXPOSITIVE4 ) {
03474             MesPrint("&Fixed range indicator bigger than %1", (LONG)MAXPOSITIVE4);
03475             return(0);
03476         }
03477     }
03478 }
03479 else if ( *s == '$' ) {
03480     s++; ss = s;
03481     while ( FG.cTable[*s] == 0 || FG.cTable[*s] == 1 ) s++;
03482     c = *s; *s = 0;
03483     if ( ( x1 = GetDollar(ss) ) < 0 ) goto Error;
03484     *s = c;
03485     x1 += MAXPOSITIVE2;
03486 }
03487 else {
03488     MesPrint("&Illegal character in range specification");
03489     return(0);
03490 }
03491 if ( *s != ',' ) {
03492     MesPrint("&A range is two indicators, separated by a comma or blank");
03493     return(0);
03494 }
03495 s++;
03496 if ( FG.cTable[*s] == 0 ) {
03497     ss = s; while ( FG.cTable[*s] == 0 ) s++;
03498     c = *s; *s = 0;
03499     if ( StrICmp(ss, (UBYTE *)"first") == 0 ) {
03500         *s = c;
03501         x2 = 1;
03502     }
03503     else if ( StrICmp(ss, (UBYTE *)"last") == 0 ) {
03504         *s = c;
03505         if ( c == '-' ) {
03506             s++;
03507             if ( *s == '$' ) {
03508                 s++; ss = s;
03509                 while ( FG.cTable[*s] == 0 || FG.cTable[*s] == 1 ) s++;
03510                 c = *s; *s = 0;
03511                 if ( ( x2 = GetDollar(ss) ) < 0 ) goto Error;
03512                 *s = c;
03513                 x2 += MAXPOSITIVE2;
03514             }
03515             else {
03516                 x2 = 0;
03517                 while ( *s >= '0' && *s <= '9' ) {
03518                     x2 = 10*x2 + *s++ - '0';
03519                     if ( x2 >= MAXPOSITIVE4 ) {
03520                         MesPrint("&Fixed range indicator bigger than %1", (LONG)MAXPOSITIVE4);
03521                         return(0);
03522                     }
03523                 }
03524             }
03525             x2 += MAXPOSITIVE4;
03526         }
03527         else x2 = MAXPOSITIVE4;
03528     }
03529     else {
03530         MesPrint("&Illegal keyword inside range specification");
03531         return(0);
03532     }
03533 }
03534 else if ( FG.cTable[*s] == 1 ) {
03535     x2 = 0;
03536     while ( *s >= '0' && *s <= '9' ) {
03537         x2 = x2*10 + *s++ - '0';
03538         if ( x2 >= MAXPOSITIVE4 ) {
03539             MesPrint("&Fixed range indicator bigger than %1", (LONG)MAXPOSITIVE4);
03540             return(0);
03541         }
03542     }
03543 }
03544 else if ( *s == '$' ) {
03545     s++; ss = s;
03546     while ( FG.cTable[*s] == 0 || FG.cTable[*s] == 1 ) s++;
03547     c = *s; *s = 0;
03548     if ( ( x2 = GetDollar(ss) ) < 0 ) goto Error;
03549     *s = c;
03550     x2 += MAXPOSITIVE2;
03551 }
03552 else {
03553     MesPrint("&Illegal character in range specification");

```

```

03554         return(0);
03555     }
03556     if ( s < in ) {
03557         MesPrint("%A range is two indicators, separated by a comma or blank between parentheses");
03558         return(0);
03559     }
03560     out[0] = x1; out[1] = x2;
03561     return(in+1);
03562 Error:
03563     MesPrint("&Undefined variable %s in range",ss);
03564     return(0);
03565 }
03566 /*
03567     #] ReadRange :
03568     #[ FindRange :
03569 */
03571
03572 int FindRange(PHEAD WORD *args, WORD *arg1, WORD *arg2, WORD totarg)
03573 {
03574     WORD n[2], fromlast, i;
03575     for ( i = 0; i < 2; i++ ) {
03576         n[i] = args[i+1];
03577         fromlast = 0;
03578         if ( n[i] >= MAXPOSITIVE2 ) { /* This is a dollar variable */
03579             n[i] -= MAXPOSITIVE2;
03580             if ( n[i] >= MAXPOSITIVE4 ) {
03581                 fromlast = 1;
03582                 n[i] -= MAXPOSITIVE4; /* Now we have the number of the dollar variable */
03583             }
03584             n[i] = DolToNumber(BHEAD n[i]);
03585             if ( AN.ErrorInDollar ) {
03586                 MLOCK(ErrorMessageLock);
03587                 MesPrint("Illegal $ value in range while executing transform statement.");
03588                 MUNLOCK(ErrorMessageLock);
03589                 return(-1);
03590             }
03591             if ( fromlast ) n[i] = totarg-n[i];
03592         }
03593         else if ( n[i] >= MAXPOSITIVE4 ) { n[i] = totarg-(n[i]-MAXPOSITIVE4); }
03594         if ( n[i] <= 0 ) {
03595             MLOCK(ErrorMessageLock);
03596             MesPrint("Illegal non-positive value in range (%d) while executing transform.",
03597 i+1);
03597             MUNLOCK(ErrorMessageLock);
03598             return(-1);
03599         }
03600     }
03601     *arg1 = n[0];
03602     *arg2 = n[1];
03603     return(0);
03604 }
03605 /*
03606     #] FindRange :
03607     #[ Transform :
03608 */
03609 */

```

4.151 unix.h File Reference

Macros

- #define [LINEFEED](#) '\n'
- #define [CARRIAGEReturn](#) 0x0D
- #define [WITHPIPE](#)
- #define [WITHSYSTEM](#)
- #define [WITHEXTERNALCHANNEL](#)
- #define [TRAPsignals](#)
- #define [P_term](#)(code) exit(((int)(code<0?-code:code))
- #define [SEPARATOR](#) '/'
- #define [ALTSEPARATOR](#) '/'
- #define [PATHSEPARATOR](#) ':'
- #define [WITH_ENV](#)
- #define [WITH_ALARM](#)

4.151.1 Detailed Description

Settings for Unix-like systems.

Definition in file [unix.h](#).

4.151.2 Macro Definition Documentation

4.151.2.1 LINEFEED

```
#define LINEFEED '\n'
```

Definition at line 33 of file [unix.h](#).

4.151.2.2 CARRIAGEReturn

```
#define CARRIAGEReturn 0x0D
```

Definition at line 34 of file [unix.h](#).

4.151.2.3 WITHPIPE

```
#define WITHPIPE
```

Definition at line 36 of file [unix.h](#).

4.151.2.4 WITHSYSTEM

```
#define WITHSYSTEM
```

Definition at line 37 of file [unix.h](#).

4.151.2.5 WITHEXTERNALCHANNEL

```
#define WITHEXTERNALCHANNEL
```

Definition at line 46 of file [unix.h](#).

4.151.2.6 TRAPsignals

```
#define TRAPsignals
```

Definition at line 49 of file [unix.h](#).

4.151.2.7 P_term

```
#define P_term(  
    code )  exit((int) (code<0?-code:code))
```

Definition at line 52 of file [unix.h](#).

4.151.2.8 SEPARATOR

```
#define SEPARATOR '/'
```

Definition at line 54 of file [unix.h](#).

4.151.2.9 ALTSEPARATOR

```
#define ALTSEPARATOR '/'
```

Definition at line 55 of file [unix.h](#).

4.151.2.10 PATHSEPARATOR

```
#define PATHSEPARATOR ':'
```

Definition at line 56 of file [unix.h](#).

4.151.2.11 WITH_ENV

```
#define WITH_ENV
```

Definition at line 57 of file [unix.h](#).

4.151.2.12 WITH_ALARM

```
#define WITH_ALARM
```

Definition at line 58 of file [unix.h](#).

4.152 unix.h

[Go to the documentation of this file.](#)

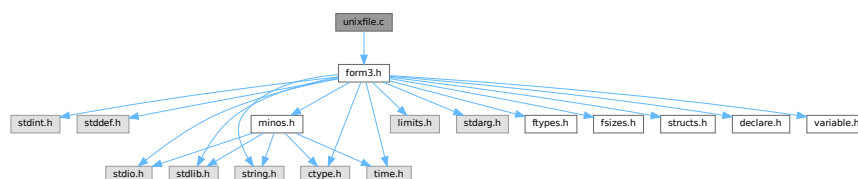
```

00001
00006 /* #[ License : */
00007 /*
00008 *   Copyright (C) 1984-2026 J.A.M. Vermaseren
00009 *   When using this file you are requested to refer to the publication
00010 *   J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00011 *   This is considered a matter of courtesy as the development was paid
00012 *   for by FOM the Dutch physics granting agency and we would like to
00013 *   be able to track its scientific use to convince FOM of its value
00014 *   for the community.
00015 *
00016 *   This file is part of FORM.
00017 *
00018 *   FORM is free software: you can redistribute it and/or modify it under the
00019 *   terms of the GNU General Public License as published by the Free Software
00020 *   Foundation, either version 3 of the License, or (at your option) any later
00021 *   version.
00022 *
00023 *   FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00024 *   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00025 *   FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00026 *   details.
00027 *
00028 *   You should have received a copy of the GNU General Public License along
00029 *   with FORM. If not, see <http://www.gnu.org/licenses/>.
00030 */
00031 /* #[ License : */
00032
00033 #define LINEFEED '\n'
00034 #define CARRIAGERETURN 0x0D
00035
00036 #define WITHPIPE
00037 #define WITHSYSTEM
00038
00039 /*[13jul2005 mt]:*/
00040 /*With SAFESIGNAL defined, write() and read() syscalls are wrapped by
00041 the errno checkup*/
00042 /**define SAFESIGNAL*/
00043 /*:[13jul2005 mt]*/
00044
00045 /*[29apr2004 mt]:*/
00046 #define WITHEXTERNALCHANNEL
00047 /*
00048 */
00049 #define TRAP SIGNALS
00050 /*:[29apr2004 mt]*/
00051
00052 #define P_term(code)    exit((int)(code<0?-code:code))
00053
00054 #define SEPARATOR '/'
00055 #define ALTSEPARATOR '/'
00056 #define PATHSEPARATOR ':'
00057 #define WITH_ENV
00058 #define WITH_ALARM

```

4.153 unixfile.c File Reference

```
#include "form3.h"
Include dependency graph for unixfile.c:
```



4.153.1 Detailed Description

The interface to a fast variety of file routines in the unix system.

Definition in file [unixfile.c](#).

4.154 unixfile.c

[Go to the documentation of this file.](#)

```

00001
00005 /* #[ License : */
00006 /*
00007  * Copyright (C) 1984-2026 J.A.M. Vermaseren
00008  * When using this file you are requested to refer to the publication
00009  * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00010  * This is considered a matter of courtesy as the development was paid
00011  * for by FOM the Dutch physics granting agency and we would like to
00012  * be able to track its scientific use to convince FOM of its value
00013  * for the community.
00014  *
00015  * This file is part of FORM.
00016  *
00017  * FORM is free software: you can redistribute it and/or modify it under the
00018  * terms of the GNU General Public License as published by the Free Software
00019  * Foundation, either version 3 of the License, or (at your option) any later
00020  * version.
00021  *
00022  * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00023  * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00024  * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00025  * details.
00026  *
00027  * You should have received a copy of the GNU General Public License along
00028  * with FORM. If not, see <http://www.gnu.org/licenses/>.
00029 */
00030 /* #[ License : */
00031 /*
00032  *#[ Includes :
00033
00034  *File with direct interface to the UNIX functions.
00035  *This makes a big difference in speed!
00036 */
00037 #include "form3.h"
00038
00039
00040 /*[13jul2005 mt]:*/
00041 #ifdef SAFESIGNAL
00042 /*[15oct2004 mt]:*/
00043 /*To access errno variable and EINTR constant:*/
00044 #include <errno.h>
00045 /*:[15oct2004 mt]:*/
00046 #endif
00047 /*:[13jul2005 mt]:*/
00048
00049 #ifdef UFILES
00050
00051 FILES TheStdout = { 1 };
00052 FILES *Ustdout = &TheStdout;
00053
00054 #ifdef GPP
00055 extern "C" open();
00056 extern "C" close();
00057 extern "C" read();
00058 extern "C" write();
00059 extern "C" lseek();
00060 #endif
00061
00062 /*
00063  *#[ Includes :
00064  *#[ Uopen :
00065 */
00066
00067 FILES *Uopen(char *filename, char *mode)
00068 {
00069     FILES *f = (FILES *)Malloc1(sizeof(FILES), "Uopen");
00070     int flags = 0, rights = 0644;
00071     while ( *mode ) {
00072         if ( *mode == 'r' ) { flags |= O_RDONLY; }

```



```

00073         else if ( *mode == 'w' ) { flags |= O_CREAT | O_TRUNC; }
00074         else if ( *mode == 'a' ) { flags |= O_APPEND; }
00075         else if ( *mode == 'b' ) { }
00076         else if ( *mode == '+' ) { flags |= O_RDWR; }
00077         mode++;
00078     }
00079     f->descriptor = open(filename,flags,rights);
00080     if ( f->descriptor >= 0 ) return(f);
00081     if ( ( flags & O_APPEND ) != 0 ) {
00082         flags |= O_CREAT;
00083         f->descriptor = open(filename,flags,rights);
00084         if ( f->descriptor >= 0 ) return(f);
00085     }
00086     M_free(f,"Uopen");
00087     return(0);
00088 }
00089
00090 /*
00091  #] Uopen :
00092  #[ Uclose :
00093  */
00094
00095 int Uclose(FILE *f)
00096 {
00097     int retval;
00098     if ( f ) {
00099         retval = close(f->descriptor);
00100         M_free(f,"Uclose");
00101         return(retval);
00102     }
00103     return(EOF);
00104 }
00105
00106 /*
00107  #] Uclose :
00108  #[ Uread :
00109  */
00110
00111 size_t Uread(char *ptr, size_t size, size_t nobj, FILE *f)
00112 {
00113     /*[13jul2005 mt]:*/
00114     #ifdef SAFESIGNAL
00115     /*[15oct2004 mt]:*/
00116     /*Operation read() can be interrupted by a signal. Note, this is rather unlikely,
00117     so we do not save size*nobj for future attempts*/
00118     size_t ret;
00119     /*If the syscall is interrupted by a signal before it
00120     succeeded in getting any progress, it must be repeated:*/
00121     while( ( ret=read(f->descriptor,ptr,size*nobj)<1)&&(errno == EINTR) );
00122     return(ret);
00123     #else
00124     #ifdef DEEPDEBUG
00125     {
00126         POSITION pos;
00127         SETBASEPOSITION(pos,lseek(f->descriptor,0L,SEEK_CUR));
00128         MesPrint("handle %d: reading %ld bytes from position %p\n",f->descriptor,size*nobj,pos);
00129     }
00130     #endif
00131     return(read(f->descriptor,ptr,size*nobj));
00132     #endif
00133 }
00134
00135 /*
00136  #] Uread :
00137  #[ Uwrite :
00138  */
00139
00140 size_t Uwrite(char *ptr, size_t size, size_t nobj, FILE *f)
00141 {
00142     /*[13jul2005 mt]:*/
00143     #ifdef SAFESIGNAL
00144     /*[15oct2004 mt]:*/
00145     /*Operation write() can be interrupted by a signal. */
00146     size_t ret;
00147     thesize=size*nobj;
00148     /*If the syscall is interrupted by a signal before it
00149     succeeded in getting any progress, it must be repeated:*/
00150     while( ( ret=write(f->descriptor,ptr,thesize)<1 )&&(errno == EINTR) );
00151     return(ret);
00152     #else
00153     return(write(f->descriptor,ptr,size*nobj));
00154     #endif
00155 }

```

```

00160 #ifdef DEEPDEBUG
00161 {
00162     POSITION pos;
00163     SETBASEPOSITION(pos,lseek(f->descriptor,0L,SEEK_CUR));
00164     MesPrint("handle %d: writing %ld bytes to position %p\n",f->descriptor,size*nobj,pos);
00165 }
00166 #endif
00167     return(write(f->descriptor,ptr,size*nobj));
00168
00169 /*:[15oct2004 mt]*/
00170 #endif
00171 /*:[13jul2005 mt]*/
00172 }
00173
00174 /*
00175     #] Uwrite :
00176     #[ Useek :
00177 */
00178
00179 int Useek(FILE *f, off_t offset, int origin)
00180 {
00181     if ( f && ( lseek(f->descriptor,offset,origin) >= 0 ) ) return(0);
00182     return(-1);
00183 }
00184
00185 /*
00186     #] Useek :
00187     #[ Utell :
00188 */
00189
00190 off_t Utell(FILE *f)
00191 {
00192     if ( f ) return((off_t)lseek(f->descriptor,0L,SEEK_CUR));
00193     else return(-1);
00194 }
00195
00196 /*
00197     #] Utell :
00198     #[ Uflush :
00199 */
00200
00201 void Uflush(FILE *f) { DUMMYUSE(f); }
00202
00203 /*
00204     #] Uflush :
00205     #[ Ugetpos :
00206 */
00207
00208 int Ugetpos(FILE *f, fpos_t *ptr)
00209 {
00210     DUMMYUSE(f); DUMMYUSE(ptr);
00211     return(-1);
00212 }
00213
00214 /*
00215     #] Ugetpos :
00216     #[ Usetpos :
00217 */
00218
00219 int Usetpos(FILE *f,fpos_t *ptr)
00220 {
00221     DUMMYUSE(f); DUMMYUSE(ptr);
00222     return(-1);
00223 }
00224
00225 /*
00226     #] Usetpos :
00227     #[ Usetbuf :
00228 */
00229
00230 void Usetbuf(FILE *f, char *ptr) { DUMMYUSE(f); DUMMYUSE(ptr); }
00231
00232 /*
00233     #] Usetbuf :
00234 */
00235 #endif

```

4.155 variable.h File Reference

This graph shows which files directly or indirectly include this file:



Macros

- #define [chartype](#) FG.cTable
- #define [Procedures](#) (([PROCEDURE](#) *) (AP.ProcList.lijst))
- #define [NumProcedures](#) AP.ProcList.num
- #define [MaxProcedures](#) AP.ProcList.maxnum
- #define [DoLoops](#) (([DOLOOP](#) *) (AP.LoopList.lijst))
- #define [NumDoLoops](#) AP.LoopList.num
- #define [MaxDoLoops](#) AP.LoopList.maxnum
- #define [PreVar](#) (([PREVAR](#) *) (AP.PreVarList.lijst))
- #define [NumPre](#) AP.PreVarList.num
- #define [MaxNumPre](#) AP.PreVarList.maxnum
- #define [SetElements](#) (([WORD](#) *) (AC.SetElementList.lijst))
- #define [Sets](#) (([SETS](#)) (AC.SetList.lijst))
- #define [functions](#) (([FUNCTIONS](#)) (AC.FunctionList.lijst))
- #define [indices](#) (([INDICES](#)) (AC.IndexList.lijst))
- #define [symbols](#) (([SYMBOLS](#)) (AC.SymbolList.lijst))
- #define [vectors](#) (([VECTORS](#)) (AC.VectorList.lijst))
- #define [tablebases](#) (([DBASE](#) *) (AC.TableBaseList.lijst))
- #define [NumFunctions](#) AC.FunctionList.num
- #define [NumIndices](#) AC.IndexList.num
- #define [NumSymbols](#) AC.SymbolList.num
- #define [NumVectors](#) AC.VectorList.num
- #define [NumSets](#) AC.SetList.num
- #define [NumSetElements](#) AC.SetElementList.num
- #define [NumTableBases](#) AC.TableBaseList.num
- #define [GlobalFunctions](#) AC.FunctionList.numglobal
- #define [GlobalIndices](#) AC.IndexList.numglobal
- #define [GlobalSymbols](#) AC.SymbolList.numglobal
- #define [GlobalVectors](#) AC.VectorList.numglobal
- #define [GlobalSets](#) AC.SetList.numglobal
- #define [GlobalSetElements](#) AC.SetElementList.numglobal
- #define [cbuf](#) (([CBUF](#) *) (AC.cbufList.lijst))
- #define [channels](#) (([CHANNEL](#) *) (AC.ChannelList.lijst))
- #define [NumOutputChannels](#) AC.ChannelList.num
- #define [Dollars](#) (([DOLLARS](#)) (AP.DollarList.lijst))
- #define [NumDollars](#) AP.DollarList.num
- #define [Dubious](#) (([DUBIOUSV](#)) (AC.DubiousList.lijst))
- #define [NumDubious](#) AC.DubiousList.num
- #define [Expressions](#) (([EXPRESSIONS](#)) (AC.ExpressionList.lijst))
- #define [NumExpressions](#) AC.ExpressionList.num
- #define [autofunctions](#) (([FUNCTIONS](#)) (AC.AutoFunctionList.lijst))
- #define [autoindices](#) (([INDICES](#)) (AC.AutoIndexList.lijst))
- #define [autosymbols](#) (([SYMBOLS](#)) (AC.AutoSymbolList.lijst))
- #define [autovectors](#) (([VECTORS](#)) (AC.AutoVectorList.lijst))

- `#define xsymbol (cbuf[AM.sbufnum].rhs)`
- `#define numxsymbol (cbuf[AM.sbufnum].numrhs)`
- `#define PotModdollars ((WORD *) (AC.PotModDolList.lijst))`
- `#define NumPotModdollars AC.PotModDolList.num`
- `#define ModOptdollars ((MODOPTDOLLAR *) (AC.ModOptDolList.lijst))`
- `#define NumModOptdollars AC.ModOptDolList.num`
- `#define BUG A.bug;`
- `#define AC A.Cc`
- `#define AM A.M`
- `#define AN A.N`
- `#define AO A.O`
- `#define AP A.P`
- `#define AR A.R`
- `#define AS A.S`
- `#define AT A.T`
- `#define AX A.X`

Variables

- `WRITEBUFTOEXTCHANNEL` **writeBufToExtChannel**
- `GETCFROMEXTCHANNEL` **getcFromExtChannel**
- `SETTERMINATORFOREXTCHANNEL` **setTerminatorForExternalChannel**
- `SETKILLMODEFOREXTCHANNEL` **setKillModeForExternalChannel**
- `WRITEFILE` **WriteFile**
- `ALLGLOBALS` **A**
- `FIXEDGLOBALS` **FG**
- `FIXEDSET` **fixedsets []**
- `char *` **setupfilename**

4.155.1 Detailed Description

Contains a number of defines to make the coding easier. Especially the defines for the use of the lists are very nice. And of course the AC for A.Cc and AT for either A.T or B->T are indispensable to keep FORM and TFORM in one set of sources.

Definition in file [variable.h](#).

4.155.2 Macro Definition Documentation

4.155.2.1 chartype

```
#define chartype FG.cTable
```

Definition at line 77 of file [variable.h](#).

4.155.2.2 Procedures

```
#define Procedures ((PROCEDURE *) (AP.ProcList.lijst))
```

Definition at line 79 of file [variable.h](#).

4.155.2.3 NumProcedures

```
#define NumProcedures AP.ProcList.num
```

Definition at line 80 of file [variable.h](#).

4.155.2.4 MaxProcedures

```
#define MaxProcedures AP.ProcList.maxnum
```

Definition at line 81 of file [variable.h](#).

4.155.2.5 DoLoops

```
#define DoLoops ((DOLOOP *) (AP.LoopList.lijst))
```

Definition at line 82 of file [variable.h](#).

4.155.2.6 NumDoLoops

```
#define NumDoLoops AP.LoopList.num
```

Definition at line 83 of file [variable.h](#).

4.155.2.7 MaxDoLoops

```
#define MaxDoLoops AP.LoopList.maxnum
```

Definition at line 84 of file [variable.h](#).

4.155.2.8 PreVar

```
#define PreVar ((PREVAR *) (AP.PreVarList.lijst))
```

Definition at line 85 of file [variable.h](#).

4.155.2.9 NumPre

```
#define NumPre AP.PreVarList.num
```

Definition at line 86 of file [variable.h](#).

4.155.2.10 MaxNumPre

```
#define MaxNumPre AP.PreVarList.maxnum
```

Definition at line 87 of file [variable.h](#).

4.155.2.11 SetElements

```
#define SetElements ((WORD *) (AC.SetElementList.lijt))
```

Definition at line 88 of file [variable.h](#).

4.155.2.12 Sets

```
#define Sets ((SETS) (AC.SetList.lijt))
```

Definition at line 89 of file [variable.h](#).

4.155.2.13 functions

```
#define functions ((FUNCTIONS) (AC.FunctionList.lijt))
```

Definition at line 90 of file [variable.h](#).

4.155.2.14 indices

```
#define indices ((INDICES) (AC.IndexList.lijt))
```

Definition at line 91 of file [variable.h](#).

4.155.2.15 symbols

```
#define symbols ((SYMBOLS) (AC.SymbolList.lijt))
```

Definition at line 92 of file [variable.h](#).

4.155.2.16 vectors

```
#define vectors ((VECTORS) (AC.VectorList.lijt))
```

Definition at line 93 of file [variable.h](#).

4.155.2.17 tablebases

```
#define tablebases ((DBASE *) (AC.TableBaseList.lijt))
```

Definition at line 94 of file [variable.h](#).

4.155.2.18 NumFunctions

```
#define NumFunctions AC.FunctionList.num
```

Definition at line 95 of file [variable.h](#).

4.155.2.19 NumIndices

```
#define NumIndices AC.IndexList.num
```

Definition at line 96 of file [variable.h](#).

4.155.2.20 NumSymbols

```
#define NumSymbols AC.SymbolList.num
```

Definition at line 97 of file [variable.h](#).

4.155.2.21 NumVectors

```
#define NumVectors AC.VectorList.num
```

Definition at line 98 of file [variable.h](#).

4.155.2.22 NumSets

```
#define NumSets AC.SetList.num
```

Definition at line 99 of file [variable.h](#).

4.155.2.23 NumSetElements

```
#define NumSetElements AC.SetElementList.num
```

Definition at line 100 of file [variable.h](#).

4.155.2.24 NumTableBases

```
#define NumTableBases AC.TableBaseList.num
```

Definition at line 101 of file [variable.h](#).

4.155.2.25 GlobalFunctions

```
#define GlobalFunctions AC.FunctionList.numglobal
```

Definition at line 102 of file [variable.h](#).

4.155.2.26 GlobalIndices

```
#define GlobalIndices AC.IndexList.numglobal
```

Definition at line 103 of file [variable.h](#).

4.155.2.27 GlobalSymbols

```
#define GlobalSymbols AC.SymbolList.numglobal
```

Definition at line 104 of file [variable.h](#).

4.155.2.28 GlobalVectors

```
#define GlobalVectors AC.VectorList.numglobal
```

Definition at line 105 of file [variable.h](#).

4.155.2.29 GlobalSets

```
#define GlobalSets AC.SetList.numglobal
```

Definition at line 106 of file [variable.h](#).

4.155.2.30 GlobalSetElements

```
#define GlobalSetElements AC.SetElementList.numglobal
```

Definition at line 107 of file [variable.h](#).

4.155.2.31 cbuf

```
#define cbuf ((CBUF *) (AC.cbufList.lijt))
```

Definition at line 108 of file [variable.h](#).

4.155.2.32 channels

```
#define channels ((CHANNEL *) (AC.ChannelList.lijt))
```

Definition at line 109 of file [variable.h](#).

4.155.2.33 NumOutputChannels

```
#define NumOutputChannels AC.Channellist.num
```

Definition at line 110 of file [variable.h](#).

4.155.2.34 Dollars

```
#define Dollars ((DOLLARS) (AP.DollarList.lijt))
```

Definition at line 111 of file [variable.h](#).

4.155.2.35 NumDollars

```
#define NumDollars AP.DollarList.num
```

Definition at line 112 of file [variable.h](#).

4.155.2.36 Dubious

```
#define Dubious ((DUBIOUSV)(AC.DubiousList.lijst))
```

Definition at line 113 of file [variable.h](#).

4.155.2.37 NumDubious

```
#define NumDubious AC.DubiousList.num
```

Definition at line 114 of file [variable.h](#).

4.155.2.38 Expressions

```
#define Expressions ((EXPRESSIONS)(AC.ExpressionList.lijst))
```

Definition at line 115 of file [variable.h](#).

4.155.2.39 NumExpressions

```
#define NumExpressions AC.ExpressionList.num
```

Definition at line 116 of file [variable.h](#).

4.155.2.40 autofunctions

```
#define autofunctions ((FUNCTIONS)(AC.AutoFunctionList.lijst))
```

Definition at line 117 of file [variable.h](#).

4.155.2.41 autoindices

```
#define autoindices ((INDICES)(AC.AutoIndexList.lijst))
```

Definition at line 118 of file [variable.h](#).

4.155.2.42 autosymbols

```
#define autosymbols ((SYMBOLS)(AC.AutoSymbolList.lijst))
```

Definition at line 119 of file [variable.h](#).

4.155.2.43 autovectors

```
#define autovectors ((VECTORS) (AC.AutoVectorList.lijt))
```

Definition at line 120 of file [variable.h](#).

4.155.2.44 xsymbol

```
#define xsymbol (cbuf[AM.sbufnum].rhs)
```

Definition at line 121 of file [variable.h](#).

4.155.2.45 numxsymbol

```
#define numxsymbol (cbuf[AM.sbufnum].numrhs)
```

Definition at line 122 of file [variable.h](#).

4.155.2.46 PotModdollars

```
#define PotModdollars ((WORD *) (AC.PotModDolList.lijt))
```

Definition at line 124 of file [variable.h](#).

4.155.2.47 NumPotModdollars

```
#define NumPotModdollars AC.PotModDolList.num
```

Definition at line 125 of file [variable.h](#).

4.155.2.48 ModOptdollars

```
#define ModOptdollars ((MODOPTDOLLAR *) (AC.ModOptDolList.lijt))
```

Definition at line 126 of file [variable.h](#).

4.155.2.49 NumModOptdollars

```
#define NumModOptdollars AC.ModOptDolList.num
```

Definition at line 127 of file [variable.h](#).

4.155.2.50 BUG

```
#define BUG A.bug;
```

Definition at line 129 of file [variable.h](#).

4.155.2.51 AC

```
#define AC A.Cc
```

Definition at line 145 of file [variable.h](#).

4.155.2.52 AM

```
#define AM A.M
```

Definition at line 146 of file [variable.h](#).

4.155.2.53 AN

```
#define AN A.N
```

Definition at line 147 of file [variable.h](#).

4.155.2.54 AO

```
#define AO A.O
```

Definition at line 148 of file [variable.h](#).

4.155.2.55 AP

```
#define AP A.P
```

Definition at line 149 of file [variable.h](#).

4.155.2.56 AR

```
#define AR A.R
```

Definition at line 150 of file [variable.h](#).

4.155.2.57 AS

```
#define AS A.S
```

Definition at line 151 of file [variable.h](#).

4.155.2.58 AT

```
#define AT A.T
```

Definition at line 152 of file [variable.h](#).

4.155.2.59 AX

```
#define AX A.X
```

Definition at line [153](#) of file [variable.h](#).

4.155.3 Variable Documentation

4.155.3.1 WriteFile

```
WRITEFILE WriteFile [extern]
```

Definition at line [1359](#) of file [tools.c](#).

4.155.3.2 A

```
ALLGLOBALS A [extern]
```

Definition at line [133](#) of file [inivar.h](#).

4.155.3.3 FG

```
FIXEDGLOBALS FG [extern]
```

Definition at line [34](#) of file [inivar.h](#).

4.155.3.4 fixedsets

```
FIXEDSET fixedsets[] [extern]
```

Definition at line [259](#) of file [inivar.h](#).

4.155.3.5 setupfilename

```
char* setupfilename [extern]
```

Definition at line [277](#) of file [inivar.h](#).

4.156 variable.h

[Go to the documentation of this file.](#)

```

00001 #ifndef __VARIABLE__
00002
00003 #define __VARIABLE__
00004
00013 /* #[ License : */
00014 /*
00015  * Copyright (C) 1984-2026 J.A.M. Vermaseren
00016  * When using this file you are requested to refer to the publication
00017  * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00018  * This is considered a matter of courtesy as the development was paid
00019  * for by FOM the Dutch physics granting agency and we would like to
00020  * be able to track its scientific use to convince FOM of its value
00021  * for the community.
00022  *
00023  * This file is part of FORM.
00024  *
00025  * FORM is free software: you can redistribute it and/or modify it under the
00026  * terms of the GNU General Public License as published by the Free Software
00027  * Foundation, either version 3 of the License, or (at your option) any later
00028  * version.
00029  *
00030  * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00031  * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00032  * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00033  * details.
00034  *
00035  * You should have received a copy of the GNU General Public License along
00036  * with FORM. If not, see <http://www.gnu.org/licenses/>.
00037  */
00038 /* #] License : */
00039
00040 /*See the file extcmd.c*/
00041
00042 #ifdef REMOVEDBY_MT
00043 extern int (*writeBufToExtChannel)(char *buf, size_t n);
00044 extern int (*getcFromExtChannel)();
00045 extern int (*setTerminatorForExternalChannel)(char *newterminator);
00046 #endif
00047 extern WRITEBUFTOEXTCHANNEL writeBufToExtChannel;
00048 extern GETCFROMEXTCHANNEL getcFromExtChannel;
00049 extern SETTERMINATORFOREXTCHANNEL setTerminatorForExternalChannel;
00050 extern SETKILLMODEFOREXTCHANNEL setKillModeForExternalChannel;
00051
00052 /*
00053 extern LONG (*WriteFile)(int handle, UBYTE *buffer, LONG number);
00054 */
00055 extern WRITEFILE WriteFile;
00056 /*:[17nov2005 mt]*/
00057
00058 extern ALLGLOBALS A;
00059 #ifdef WITHPTHREADS
00060 extern ALLPRIVATES **AB;
00061 #endif
00062
00063 extern FIXEDGLOBALS FG;
00064 extern FIXEDSET fixedsets[];
00065
00066 extern char *setupfilename;
00067
00068 EXTERNLOCK(ErrorMessageLock)
00069 EXTERNLOCK(ErrorReadLock)
00070 EXTERNLOCK(dummylock)
00071
00072 #ifdef VMS
00073 #include <stdio.h>
00074 extern FILE **FileStructs;
00075 #endif
00076
00077 #define chartype FG.cTable
00078
00079 #define Procedures ((PROCEDURE *) (AP.ProcList.lijst))
00080 #define NumProcedures AP.ProcList.num
00081 #define MaxProcedures AP.ProcList.maxnum
00082 #define DoLoops ((DOLOOP *) (AP.LoopList.lijst))
00083 #define NumDoLoops AP.LoopList.num
00084 #define MaxDoLoops AP.LoopList.maxnum
00085 #define PreVar ((PREVAR *) (AP.PreVarList.lijst))
00086 #define NumPre AP.PreVarList.num
00087 #define MaxNumPre AP.PreVarList.maxnum
00088 #define SetElements ((WORD *) (AC.SetElementList.lijst))
00089 #define Sets ((SETS) (AC.SetList.lijst))
00090 #define functions ((FUNCTIONS) (AC.FunctionList.lijst))

```

```

00091 #define indices ((INDICES) (AC.IndexList.lijst))
00092 #define symbols ((SYMBOLS) (AC.SymbolList.lijst))
00093 #define vectors ((VECTORS) (AC.VectorList.lijst))
00094 #define tablebases ((DBASE *) (AC.TableBaseList.lijst))
00095 #define NumFunctions AC.FunctionList.num
00096 #define NumIndices AC.IndexList.num
00097 #define NumSymbols AC.SymbolList.num
00098 #define NumVectors AC.VectorList.num
00099 #define NumSets AC.SetList.num
00100 #define NumSetElements AC.SetElementList.num
00101 #define NumTableBases AC.TableBaseList.num
00102 #define GlobalFunctions AC.FunctionList.numglobal
00103 #define GlobalIndices AC.IndexList.numglobal
00104 #define GlobalSymbols AC.SymbolList.numglobal
00105 #define GlobalVectors AC.VectorList.numglobal
00106 #define GlobalSets AC.SetList.numglobal
00107 #define GlobalSetElements AC.SetElementList.numglobal
00108 #define cbuf ((CBUF *) (AC.cbufList.lijst))
00109 #define channels ((CHANNEL *) (AC.ChannelList.lijst))
00110 #define NumOutputChannels AC.ChannelList.num
00111 #define Dollars ((DOLLARS) (AP.DollarList.lijst))
00112 #define NumDollars AP.DollarList.num
00113 #define Dubious ((DUBIOUSV) (AC.DubiousList.lijst))
00114 #define NumDubious AC.DubiousList.num
00115 #define Expressions ((EXPRESSIONS) (AC.ExpressionList.lijst))
00116 #define NumExpressions AC.ExpressionList.num
00117 #define autofunctions ((FUNCTIONS) (AC.AutoFunctionList.lijst))
00118 #define autoindices ((INDICES) (AC.AutoIndexList.lijst))
00119 #define autosymbols ((SYMBOLS) (AC.AutoSymbolList.lijst))
00120 #define autovectors ((VECTORS) (AC.AutoVectorList.lijst))
00121 #define xsymbol (cbuf[AM.sbufnum].rhs)
00122 #define numxsymbol (cbuf[AM.sbufnum].numrhs)
00123
00124 #define PotModdollars ((WORD *) (AC.PotModDolList.lijst))
00125 #define NumPotModdollars AC.PotModDolList.num
00126 #define ModOptdollars ((MODOPTDOLLAR *) (AC.ModOptDolList.lijst))
00127 #define NumModOptdollars AC.ModOptDolList.num
00128
00129 #define BUG A.bug;
00130
00131 #ifndef WITHPTHREADS
00132 #define AC A.Cc
00133 #define AM A.M
00134 #define AO A.O
00135 #define AP A.P
00136 #define AS A.S
00137 #define AX A.X
00138 #define AN B->N
00139 #define AR B->R
00140 #define AT B->T
00141 #define AN0 B0->N
00142 #define AR0 B0->R
00143 #define AT0 B0->T
00144 #else
00145 #define AC A.Cc
00146 #define AM A.M
00147 #define AN A.N
00148 #define AO A.O
00149 #define AP A.P
00150 #define AR A.R
00151 #define AS A.S
00152 #define AT A.T
00153 #define AX A.X
00154 #endif
00155
00156 #endif

```

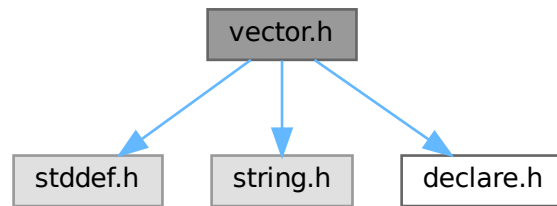
4.157 vector.h File Reference

```

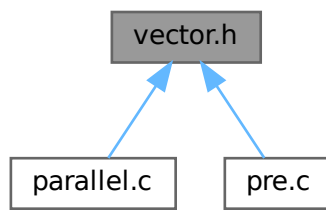
#include <stddef.h>
#include <string.h>
#include "declare.h"

```

Include dependency graph for vector.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define [VectorStruct](#)(T)
- #define [Vector](#)(T, X) [VectorStruct](#)(T) X = { NULL, 0, 0 }
- #define [DeclareVector](#)(T, X) [VectorStruct](#)(T) X
- #define [VectorInit](#)(X)
- #define [VectorFree](#)(X)
- #define [VectorPtr](#)(X) ((X).ptr)
- #define [VectorFront](#)(X) ((X).ptr[0])
- #define [VectorBack](#)(X) ((X).ptr[(X).size - 1])
- #define [VectorSize](#)(X) ((X).size)
- #define [VectorCapacity](#)(X) ((X).capacity)
- #define [VectorEmpty](#)(X) ((X).size == 0)
- #define [VectorClear](#)(X) do { (X).size = 0; } while (0)
- #define [VectorReserve](#)(X, newcapacity)
- #define [VectorPushBack](#)(X, x)
- #define [VectorPushBacks](#)(X, src, n)
- #define [VectorPopBack](#)(X) do { (X).size --; } while (0)
- #define [VectorInsert](#)(X, index, x)
- #define [VectorInserts](#)(X, index, src, n)
- #define [VectorErase](#)(X, index)
- #define [VectorErases](#)(X, index, n)

4.157.1 Detailed Description

An implementation of dynamic array.

Example:

```
size_t i;
Vector(int, vec);
VectorPushBack(vec, 1);
VectorPushBack(vec, 2);
VectorPushBack(vec, 3);
for ( i = 0; i < VectorSize(vec); i++ )
    printf("%d\n", VectorPtr(vec)[i]);
VectorFree(vec);
```

Definition in file [vector.h](#).

4.157.2 Macro Definition Documentation

4.157.2.1 VectorStruct

```
#define VectorStruct(
    T )
```

Value:

```
struct { \
    T *ptr; \
    size_t size; \
    size_t capacity; \
}
```

A struct for vector objects.

Parameters

<i>T</i>	the type of elements.

Definition at line 65 of file [vector.h](#).

4.157.2.2 Vector

```
#define Vector(
    T,
    X ) VectorStruct(T) X = { NULL, 0, 0 }
```

Defines and initialises a vector X of the type T. The user must call [VectorFree\(X\)](#) after the use of X.

Parameters

<i>T</i>	the type of elements.
<i>X</i>	the name of vector object

Definition at line 84 of file [vector.h](#).

4.157.2.3 DeclareVector

```
#define DeclareVector(  
    T,  
    X )  VectorStruct (T) X
```

Declares a vector X of the type T. The user must call [VectorInit\(X\)](#) before the use of X.

Parameters

<i>T</i>	the type of elements.
<i>X</i>	the name of vector object

Definition at line 99 of file [vector.h](#).

4.157.2.4 VectorInit

```
#define VectorInit(  
    X )
```

Value:

```
do { \  
    (X).ptr = NULL; \  
    (X).size = 0; \  
    (X).capacity = 0; \  
} while (0)
```

Initialises a vector X of the type T. The user must call [VectorFree\(X\)](#) after the use of X.

Parameters

<i>X</i>	the vector object.

Definition at line 113 of file [vector.h](#).

4.157.2.5 VectorFree

```
#define VectorFree(  
    X )
```

Value:

```
do { \  
    M_free((X).ptr, "VectorFree:" #X); \  
    (X).ptr = NULL; \  
    (X).size = 0; \  
    (X).capacity = 0; \  
} while (0)
```

Frees the buffer allocated by the vector X.

Parameters

<i>X</i>	the vector object.

Definition at line 130 of file [vector.h](#).

4.157.2.6 VectorPtr

```
#define VectorPtr(  
    X )    ((X).ptr)
```

Returns the pointer to the buffer for the vector X. NULL when [VectorCapacity\(X\)](#) == 0.

Parameters

X	the vector object.

Returns

the pointer to the allocated buffer for the vector.

Definition at line 150 of file [vector.h](#).

4.157.2.7 VectorFront

```
#define VectorFront(  
    X )    ((X).ptr[0])
```

Returns the first element of the vector X. Undefined when [VectorSize\(X\)](#) == 0.

Parameters

X	the vector object.

Returns

the first element of the vector.

Definition at line 165 of file [vector.h](#).

4.157.2.8 VectorBack

```
#define VectorBack(  
    X )    ((X).ptr[(X).size - 1])
```

Returns the last element of the vector X. Undefined when [VectorSize\(X\)](#) == 0.

Parameters

X	the vector object.

Returns

the last element of the vector.

Definition at line 180 of file [vector.h](#).

4.157.2.9 VectorSize

```
#define VectorSize(  
    X )    ((X).size)
```

Returns the size of the vector X.

Parameters

<i>X</i>	the vector object.

Returns

the size of the vector.

Definition at line 194 of file [vector.h](#).

4.157.2.10 VectorCapacity

```
#define VectorCapacity(  
    X )    ((X).capacity)
```

Returns the capacity (the number of the already allocated elements) of the vector X.

Parameters

<i>X</i>	the vector object.

Returns

the capacity of the vector.

Definition at line 208 of file [vector.h](#).

4.157.2.11 VectorEmpty

```
#define VectorEmpty(  
    X )    ((X).size == 0)
```

Returns true the size of the vector X is zero.

Parameters

<i>X</i>	the vector object.

Returns

true if the vector has no elements, false otherwise.

Definition at line 222 of file [vector.h](#).

4.157.2.12 VectorClear

```
#define VectorClear(  
    X )  do { (X).size = 0; } while (0)
```

Sets the size of the vector *X* to zero.

Parameters

<i>X</i>	the vector object.

Definition at line 235 of file [vector.h](#).

4.157.2.13 VectorReserve

```
#define VectorReserve(  
    X,  
    newcapacity )
```

Value:

```
do { \
    size_t v_tmp_newcapacity_ = (newcapacity); \
    if ( (X).capacity < v_tmp_newcapacity_ ) { \
        void *v_tmp_newptr_; \
        v_tmp_newcapacity_ = (v_tmp_newcapacity_ * 3) / 2; \
        if ( v_tmp_newcapacity_ < 4 ) v_tmp_newcapacity_ = 4; \
        v_tmp_newptr_ = Malloc1(sizeof((X).ptr[0]) * v_tmp_newcapacity_, "VectorReserve:" #X); \
        if ( (X).ptr != NULL ) { \
            memcpy(v_tmp_newptr_, (X).ptr, (X).size * sizeof((X).ptr[0])); \
            M_free((X).ptr, "VectorReserve:" #X); \
        } \
        (X).ptr = v_tmp_newptr_; \
        (X).capacity = v_tmp_newcapacity_; \
    } \
} while (0)
```

Requires that the capacity of the vector *X* is equal to or larger than *newcapacity*.

Parameters

<i>X</i>	the vector object.
<i>newcapacity</i>	the capacity to be reserved.

Definition at line 249 of file [vector.h](#).

4.157.2.14 VectorPushBack

```
#define VectorPushBack(  
    X,  
    x )
```

Value:

```
do { \
    VectorReserve((X), (X).size + 1); \
    (X).ptr[(X).size++] = (x); \
} while (0)
```

Adds an element *x* at the end of the vector *X*.

Parameters

<i>X</i>	the vector object.
<i>x</i>	the element to be added.

Definition at line 277 of file [vector.h](#).

4.157.2.15 VectorPushBacks

```
#define VectorPushBacks(  
    X,  
    src,  
    n )
```

Value:

```
do { \
    size_t v_tmp_n_ = (n); \
    VectorReserve((X), (X).size + v_tmp_n_); \
    memcpy((X).ptr + (X).size, (src), v_tmp_n_ * sizeof((X).ptr[0])); \
    (X).size += v_tmp_n_; \
} while (0)
```

Adds an *n* elements of *src* at the end of the vector *X*.

Parameters

<i>X</i>	the vector object.
<i>src</i>	the starting address of the buffer storing elements to be added.
<i>n</i>	the number of elements to be added.

Definition at line 295 of file [vector.h](#).

4.157.2.16 VectorPopBack

```
#define VectorPopBack(  
    X ) do { (X).size --; } while (0)
```

Removes the last element of the vector *X*. [VectorSize\(X\)](#) must be > 0 .

Parameters

<i>X</i>	the vector object.

Definition at line 314 of file [vector.h](#).

4.157.2.17 VectorInsert

```
#define VectorInsert(
    X,
    index,
    x )
```

Value:

```
do { \
    size_t v_tmp_index_ = (index); \
    VectorReserve((X), (X).size + 1); \
    memmove((X).ptr + v_tmp_index_ + 1, (X).ptr + v_tmp_index_, ((X).size - v_tmp_index_) *
    sizeof((X).ptr[0])); \
    (X).ptr[v_tmp_index_] = (x); \
    (X).size++; \
} while (0)
```

Inserts an element *x* at the specified index of the vector *X*. The index must be $0 \leq \text{index} < \text{VectorSize}(X)$.

Parameters

<i>X</i>	the vector object.
<i>index</i>	the position at which the element will be inserted.
<i>x</i>	the element to be inserted.

Definition at line 330 of file [vector.h](#).

4.157.2.18 VectorInserts

```
#define VectorInserts(
    X,
    index,
    src,
    n )
```

Value:

```
do { \
    size_t v_tmp_index_ = (index), v_tmp_n_ = (n); \
    VectorReserve((X), (X).size + v_tmp_n_); \
    memmove((X).ptr + v_tmp_index_ + v_tmp_n_, (X).ptr + v_tmp_index_, ((X).size - v_tmp_index_) *
    sizeof((X).ptr[0])); \
    memcpy((X).ptr + v_tmp_index_, (src), v_tmp_n_ * sizeof((X).ptr[0])); \
    (X).size += v_tmp_n_; \
} while (0)
```

Inserts an *n* elements of *src* at the specified index of the vector *X*. The index must be $0 \leq \text{index} < \text{VectorSize}(X)$.

Parameters

<i>X</i>	the vector object.
<i>index</i>	the position at which the elements will be inserted.
<i>src</i>	the starting address of the buffer storing elements to be inserted.
<i>n</i>	the number of elements to be inserted.

Definition at line 353 of file [vector.h](#).

4.157.2.19 VectorErase

```
#define VectorErase(  
    X,  
    index )
```

Value:

```
do { \
    size_t v_tmp_index_ = (index); \
    memmove((X).ptr + v_tmp_index_, (X).ptr + v_tmp_index_ + 1, ((X).size - v_tmp_index_ - 1) * \
    sizeof((X).ptr[0])); \
    (X).size--; \
} while (0)
```

Removes an element at the specified index of the vector X. The index must be $0 \leq \text{index} < \text{VectorSize}(X)$.

Parameters

<i>X</i>	the vector object.
<i>index</i>	the position of the element to be removed.

Definition at line 374 of file [vector.h](#).

4.157.2.20 VectorErases

```
#define VectorErases(  
    X,  
    index,  
    n )
```

Value:

```
do { \
    size_t v_tmp_index_ = (index), v_tmp_n_ = (n); \
    memmove((X).ptr + v_tmp_index_, (X).ptr + v_tmp_index_ + v_tmp_n_, ((X).size - v_tmp_index_ - 1) * \
    sizeof((X).ptr[0])); \
    (X).size -= v_tmp_n_; \
} while (0)
```

Removes an n elements at the specified index of the vector X. The index must be $0 \leq \text{index} < \text{VectorSize}(X) - n + 1$.

Parameters

<i>X</i>	the vector object.
<i>index</i>	the starting position of the elements to be removed.
<i>n</i>	the number of elements to be removed.

Definition at line 394 of file [vector.h](#).

4.158 vector.h

[Go to the documentation of this file.](#)

```

00001 #ifndef VECTOR_H_
00002 #define VECTOR_H_
00003
00020 /* # [ License : */
00021 /*
00022 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00023 * When using this file you are requested to refer to the publication
00024 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00025 * This is considered a matter of courtesy as the development was paid
00026 * for by FOM the Dutch physics granting agency and we would like to
00027 * be able to track its scientific use to convince FOM of its value
00028 * for the community.
00029 *
00030 * This file is part of FORM.
00031 *
00032 * FORM is free software: you can redistribute it and/or modify it under the
00033 * terms of the GNU General Public License as published by the Free Software
00034 * Foundation, either version 3 of the License, or (at your option) any later
00035 * version.
00036 *
00037 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00038 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00039 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00040 * details.
00041 *
00042 * You should have received a copy of the GNU General Public License along
00043 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00044 */
00045 /* # [ License : */
00046 /*
00047 # [ Includes :
00048 */
00049
00050 #include <stddef.h>
00051 #include <string.h>
00052 #include "declare.h"
00053
00054 /*
00055 # ] Includes :
00056 # [ Vector :
00057 # [ VectorStruct :
00058 */
00059
00065 #define VectorStruct(T) \
00066     struct { \
00067         T *ptr; \
00068         size_t size; \
00069         size_t capacity; \
00070     }
00071
00072 /*
00073 # ] VectorStruct :
00074 # [ Vector :
00075 */
00076
00084 #define Vector(T, X) \
00085     VectorStruct(T) X = { NULL, 0, 0 }
00086
00087 /*
00088 # ] Vector :
00089 # [ DeclareVector :
00090 */
00091
00099 #define DeclareVector(T, X) \
00100     VectorStruct(T) X
00101
00102 /*
00103 # ] DeclareVector :
00104 # [ VectorInit :
00105 */
00106
00113 #define VectorInit(X) \
00114     do { \
00115         (X).ptr = NULL; \
00116         (X).size = 0; \
00117         (X).capacity = 0; \
00118     } while (0)
00119
00120 /*
00121 # ] VectorInit :
00122 # [ VectorFree :
00123 */
00124
00130 #define VectorFree(X) \
00131     do { \
00132         M_free((X).ptr, "VectorFree:" #X); \
00133         (X).ptr = NULL; \

```



```

00134         (X).size = 0; \
00135         (X).capacity = 0; \
00136     } while (0)
00137
00138     /*
00139         #] VectorFree :
00140         #[ VectorPtr :
00141     */
00142
00150 #define VectorPtr(X) \
00151     ((X).ptr)
00152
00153     /*
00154         #] VectorPtr :
00155         #[ VectorFront :
00156     */
00157
00165 #define VectorFront(X) \
00166     ((X).ptr[0])
00167
00168     /*
00169         #] VectorFront :
00170         #[ VectorBack :
00171     */
00172
00180 #define VectorBack(X) \
00181     ((X).ptr[(X).size - 1])
00182
00183     /*
00184         #] VectorBack :
00185         #[ VectorSize :
00186     */
00187
00194 #define VectorSize(X) \
00195     ((X).size)
00196
00197     /*
00198         #] VectorSize :
00199         #[ VectorCapacity :
00200     */
00201
00208 #define VectorCapacity(X) \
00209     ((X).capacity)
00210
00211     /*
00212         #] VectorCapacity :
00213         #[ VectorEmpty :
00214     */
00215
00222 #define VectorEmpty(X) \
00223     ((X).size == 0)
00224
00225     /*
00226         #] VectorEmpty :
00227         #[ VectorClear :
00228     */
00229
00235 #define VectorClear(X) \
00236     do { (X).size = 0; } while (0)
00237
00238     /*
00239         #] VectorClear :
00240         #[ VectorReserve :
00241     */
00242
00249 #define VectorReserve(X, newcapacity) \
00250     do { \
00251         size_t v_tmp_newcapacity_ = (newcapacity); \
00252         if ( (X).capacity < v_tmp_newcapacity_ ) { \
00253             void *v_tmp_newptr_; \
00254             v_tmp_newcapacity_ = (v_tmp_newcapacity_ * 3) / 2; \
00255             if ( v_tmp_newcapacity_ < 4 ) v_tmp_newcapacity_ = 4; \
00256             v_tmp_newptr_ = Malloc1(sizeof((X).ptr[0]) * v_tmp_newcapacity_, "VectorReserve:" #X); \
00257             if ( (X).ptr != NULL ) { \
00258                 memcpy(v_tmp_newptr_, (X).ptr, (X).size * sizeof((X).ptr[0])); \
00259                 M_free((X).ptr, "VectorReserve:" #X); \
00260             } \
00261             (X).ptr = v_tmp_newptr_; \
00262             (X).capacity = v_tmp_newcapacity_; \
00263         } \
00264     } while (0)
00265
00266     /*
00267         #] VectorReserve :
00268         #[ VectorPushBack :
00269     */
00270

```

```

00277 #define VectorPushBack(X, x) \
00278     do { \
00279         VectorReserve((X), (X).size + 1); \
00280         (X).ptr[(X).size++] = (x); \
00281     } while (0)
00282
00283 /*
00284     #] VectorPushBack :
00285     #[ VectorPushBacks :
00286 */
00287
00295 #define VectorPushBacks(X, src, n) \
00296     do { \
00297         size_t v_tmp_n_ = (n); \
00298         VectorReserve((X), (X).size + v_tmp_n_); \
00299         memcpy((X).ptr + (X).size, (src), v_tmp_n_ * sizeof((X).ptr[0])); \
00300         (X).size += v_tmp_n_; \
00301     } while (0)
00302
00303 /*
00304     #] VectorPushBacks :
00305     #[ VectorPopBack :
00306 */
00307
00314 #define VectorPopBack(X) \
00315     do { (X).size --; } while (0)
00316
00317 /*
00318     #] VectorPopBack :
00319     #[ VectorInsert :
00320 */
00321
00330 #define VectorInsert(X, index, x) \
00331     do { \
00332         size_t v_tmp_index_ = (index); \
00333         VectorReserve((X), (X).size + 1); \
00334         memmove((X).ptr + v_tmp_index_ + 1, (X).ptr + v_tmp_index_, ((X).size - v_tmp_index_) *
sizeof((X).ptr[0])); \
00335         (X).ptr[v_tmp_index_] = (x); \
00336         (X).size++; \
00337     } while (0)
00338
00339 /*
00340     #] VectorInsert :
00341     #[ VectorInserts :
00342 */
00343
00353 #define VectorInserts(X, index, src, n) \
00354     do { \
00355         size_t v_tmp_index_ = (index), v_tmp_n_ = (n); \
00356         VectorReserve((X), (X).size + v_tmp_n_); \
00357         memmove((X).ptr + v_tmp_index_ + v_tmp_n_, (X).ptr + v_tmp_index_, ((X).size - v_tmp_index_) *
sizeof((X).ptr[0])); \
00358         memcpy((X).ptr + v_tmp_index_, (src), v_tmp_n_ * sizeof((X).ptr[0])); \
00359         (X).size += v_tmp_n_; \
00360     } while (0)
00361
00362 /*
00363     #] VectorInserts :
00364     #[ VectorErase :
00365 */
00366
00374 #define VectorErase(X, index) \
00375     do { \
00376         size_t v_tmp_index_ = (index); \
00377         memmove((X).ptr + v_tmp_index_, (X).ptr + v_tmp_index_ + 1, ((X).size - v_tmp_index_ - 1) *
sizeof((X).ptr[0])); \
00378         (X).size--; \
00379     } while (0)
00380
00381 /*
00382     #] VectorErase :
00383     #[ VectorErases :
00384 */
00385
00394 #define VectorErases(X, index, n) \
00395     do { \
00396         size_t v_tmp_index_ = (index), v_tmp_n_ = (n); \
00397         memmove((X).ptr + v_tmp_index_, (X).ptr + v_tmp_index_ + v_tmp_n_, ((X).size - v_tmp_index_ -
1) * sizeof((X).ptr[0])); \
00398         (X).size -= v_tmp_n_; \
00399     } while (0)
00400
00401 /*
00402     #] VectorErases :
00403     #[ Vector :
00404 */

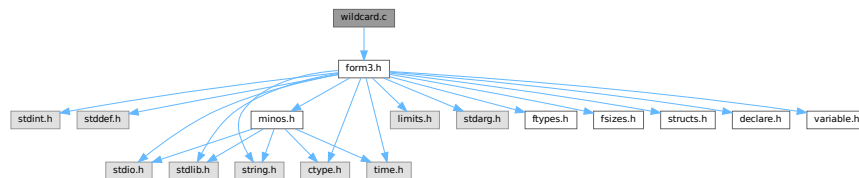
```

```
00405 #endif /* VECTOR_H_ */
```

4.159 wildcard.c File Reference

```
#include "form3.h"
```

Include dependency graph for wildcard.c:



Macros

- `#define` [DEBUG\(x\)](#)

Functions

- `WORD` [WildFill](#) (PHEAD WORD *to, WORD *from, WORD *sub)
- `int` [ResolveSet](#) (PHEAD WORD *from, WORD *to, WORD *subs)
- `void` [ClearWild](#) (PHEAD0)
- `int` [AddWild](#) (PHEAD WORD oldnumber, WORD type, WORD newnumber)
- `int` [CheckWild](#) (PHEAD WORD oldnumber, WORD type, WORD newnumber, WORD *newval)
- `int` [DenToFunction](#) (WORD *term, WORD numfun)

4.159.1 Detailed Description

Contains the functions that deal with the wildcards. During the pattern matching there are two steps: 1: check that a wildcard substitution is correct (if there was already an assignment for this variable, it is the same; it is part of the proper set; it is the proper type of variables, etc.) 2: make the assignment In addition we have to be able to clear assignments. During execution we have to make the actual replacements (WildFill)

Definition in file [wildcard.c](#).

4.159.2 Macro Definition Documentation

4.159.2.1 DEBUG

```
#define DEBUG(
    x )
```

Definition at line 44 of file [wildcard.c](#).

4.159.3 Function Documentation

4.159.3.1 WildFill()

```
WORD WildFill (
    PHEAD WORD * to,
    WORD * from,
    WORD * sub )
```

Definition at line 65 of file [wildcard.c](#).

4.159.3.2 ResolveSet()

```
int ResolveSet (
    PHEAD WORD * from,
    WORD * to,
    WORD * subs )
```

Definition at line 1361 of file [wildcard.c](#).

4.159.3.3 ClearWild()

```
void ClearWild (
    PHEAD0 )
```

Definition at line 1518 of file [wildcard.c](#).

4.159.3.4 AddWild()

```
int AddWild (
    PHEAD WORD oldnumber,
    WORD type,
    WORD newnumber )
```

Definition at line 1544 of file [wildcard.c](#).

4.159.3.5 CheckWild()

```
int CheckWild (
    PHEAD WORD oldnumber,
    WORD type,
    WORD newnumber,
    WORD * newval )
```

Definition at line 1791 of file [wildcard.c](#).

4.159.3.6 DenToFunction()

```
int DenToFunction (
    WORD * term,
    WORD numFun )
```

Definition at line 2557 of file [wildcard.c](#).

4.160 wildcard.c

[Go to the documentation of this file.](#)

```
00001
00012 /* #[ License : */
00013 /*
00014 * Copyright (C) 1984-2026 J.A.M. Vermaseren
00015 * When using this file you are requested to refer to the publication
00016 * J.A.M.Vermaseren "New features of FORM" math-ph/0010025
00017 * This is considered a matter of courtesy as the development was paid
00018 * for by FOM the Dutch physics granting agency and we would like to
00019 * be able to track its scientific use to convince FOM of its value
00020 * for the community.
00021 *
00022 * This file is part of FORM.
00023 *
00024 * FORM is free software: you can redistribute it and/or modify it under the
00025 * terms of the GNU General Public License as published by the Free Software
00026 * Foundation, either version 3 of the License, or (at your option) any later
00027 * version.
00028 *
00029 * FORM is distributed in the hope that it will be useful, but WITHOUT ANY
00030 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
00031 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
00032 * details.
00033 *
00034 * You should have received a copy of the GNU General Public License along
00035 * with FORM. If not, see <http://www.gnu.org/licenses/>.
00036 */
00037 /* #] License : */
00038 /*
00039 #[ Includes : wildcard.c
00040 */
00041
00042 #include "form3.h"
00043
00044 #define DEBUG(x)
00045
00046 /*
00047 #define DEBUG(x) x
00048
00049 #] Includes :
00050 #[ Wildcards :
00051 #[ WildFill : WORD WildFill(to,from,sub)
00052
00053 Takes the term in from and puts it into to while
00054 making wildcard substitutions.
00055 The return value is the number of words put in to.
00056 The length as the first word of from is not copied.
00057
00058 There are two possible algorithms:
00059 1: For each element in `from': scan sub.
00060 2: For each wildcard in sub replace elements in term.
00061 The original algorithm used 1:
00062
00063 */
00064
00065 WORD WildFill(PHEAD WORD *to, WORD *from, WORD *sub)
00066 {
00067 GETBIDENTITY
00068 WORD i, j, *s, *t, *m, len, dflag, odirt, adirt;
00069 WORD *r, *u, *v, *w, *z, *zst, *zz, *subs, *accu, na, dirty = 0, *tstop;
00070 WORD *temp = 0, *uu, *oldcpointer, sgn;
00071 WORD subcount, setflag, *setlist = 0, si;
00072 accu = oldcpointer = AR.CompressPointer;
00073 t = sub;
00074 t += sub[1];
00075 s = sub + SUBEXPSIZE;
00076 i = 0;
```

```

00077     while ( s < t && *s != FROMBRAC ) {
00078         i++; s += s[1];
00079     }
00080     if ( !i ) {          /* No wildcards -> done quickly */
00081         j = i = *from;
00082         NCOPY(to,from,i);
00083         if ( dirty ) AN.WildDirt = dirty;
00084         return(j);
00085     }
00086     sgn = 0;
00087     subs = sub + SUBEXPSIZE;
00088     t = from;
00089     GETSTOP(t,r);
00090     t++;
00091     m = to + 1;
00092     if ( t < r ) do {
00093         uu = u = t + t[1];
00094         setflag = 0;
00095     ReSwitch:
00096         switch ( *t ) {
00097             case SYMBOL:
00098             /*
00099                 #[ SYMBOLS :
00100             */
00101                 z = accu;
00102                 *m++ = *t++;
00103                 *m++ = *t++;
00104                 v = m;
00105                 while ( t < u ) {
00106                     *m = *t;
00107                     for ( si = 0; si < setflag; si += 2 ) {
00108                         if ( t == temp + setlist[si] ) goto sspow;
00109                     }
00110                     s = subs;
00111                     for ( j = 0; j < i; j++ ) {
00112                         if ( *t == s[2] ) {
00113                             if ( *s == SYMTOSYM ) {
00114                                 *m = s[3]; dirty = 1;
00115                                 break;
00116                             }
00117                             else if ( *s == SYMTONUM ) {
00118                                 dirty = 1;
00119                                 zst = z;
00120                                 *z++ = SNUMBER;
00121                                 *z++ = 4;
00122                                 *z++ = s[3];
00123                                 w = z;
00124                                 *z++ = *t;
00125                                 if ( ABS(*t) >= 2*MAXPOWER ) {
00126     DoPow:
00127                                     s = subs;
00128                                     for ( j = 0; j < i; j++ ) {
00129                                         if ( ( *s == SYMTONUM ) &&
00130                                             ( ABS(*t) - 2*MAXPOWER ) == s[2] ) {
00131                                             dirty = 1;
00132                                             *w = s[3];
00133                                             if ( *t < 0 ) *w = -*w;
00134                                             break;
00135                                         }
00136                                         if ( ( *s == SYMTOSYM ) &&
00137                                             ( ABS(*t) - 2*MAXPOWER ) == s[2] ) {
00138                                             dirty = 1;
00139                                             zz = z;
00140                                             while ( --zz >= zst ) {
00141                                                 zz[1+FUNHEAD+ARGHEAD] = *zz;
00142                                             }
00143                                             w += 1+FUNHEAD+ARGHEAD;
00144                                             *zst = EXPONENT;
00145                                             zst[2] = DIRTYFLAG;
00146                                             zst[FUNHEAD+ARGHEAD] = WORDDIF(z,zst)+4;
00147                                             zst[1+FUNHEAD] = 1;
00148                                             zst[FUNHEAD] = WORDDIF(z,zst)+4+ARGHEAD;
00149                                             z += FUNHEAD+ARGHEAD+1;
00150                                             *w = 1; /* exponent -> 1 */
00151                                             *z++ = 1;
00152                                             *z++ = 1;
00153                                             *z++ = 3;
00154                                             if ( *t > 0 ) {
00155                                                 *z++ = -SYMBOL;
00156                                                 *z++ = s[3];
00157                                             }
00158                                             else {
00159                                                 *z++ = ARGHEAD+8;
00160                                                 *z++ = 1;
00161                                                 *z++ = 8;
00162                                                 *z++ = SYMBOL;
00163                                                 *z++ = 4;
00164                                                 *z++ = s[3];

```

```

00164             *z++ = 1;
00165             *z++ = 1;
00166             *z++ = 1;
00167             *z++ = -3;
00168         }
00169         zst[1] = WORDDIF(z,zst);
00170         break;
00171     }
00172     if ( *s == SYMTOSUB &&
00173         ( ABS(*t) - 2*MAXPOWER ) == s[2] ) {
00174         dirty = 1;
00175         zz = z;
00176         while ( --zz >= zst ) {
00177             zz[1+FUNHEAD+ARGHEAD] = *zz;
00178         }
00179         w += 1+FUNHEAD+ARGHEAD;
00180         *zst = EXPONENT;
00181         zst[2] = DIRTYFLAG;
00182         zst[FUNHEAD+ARGHEAD] = WORDDIF(z,zst)+4;
00183         zst[1+FUNHEAD] = 1;
00184         zst[FUNHEAD] = WORDDIF(z,zst)+4+ARGHEAD;
00185         z += FUNHEAD+ARGHEAD+1;
00186         *w = 1; /* exponent -> 1 */
00187         *z++ = 1;
00188         *z++ = 1;
00189         *z++ = 3;
00190         *z++ = 4+SUBEXPSIZE+ARGHEAD;
00191         *z++ = 1;
00192         *z++ = 4+SUBEXPSIZE;
00193         *z++ = SUBEXPRESSION;
00194         *z++ = SUBEXPSIZE;
00195         *z++ = s[3];
00196         *z++ = 1;
00197         *z++ = AT.ebufnum;
00198         FILLSUB(z)
00199         *z++ = 1;
00200         *z++ = 1;
00201         *z++ = *t > 0 ? 3: -3;
00202         zst[1] = WORDDIF(z,zst);
00203         break;
00204     }
00205     s += s[1];
00206 }
00207 }
00208 if ( !*w ) z = w - 3;
00209 t++;
00210 goto Seven;
00211 }
00212 else if ( *s == SYMTOSUB ) {
00213     dirty = 1;
00214     zst = z;
00215     *z++ = SUBEXPRESSION;
00216     *z++ = SUBEXPSIZE;
00217     *z++ = s[3];
00218     w = z;
00219     *z++ = *++t;
00220     *z++ = AT.ebufnum;
00221     FILLSUB(z)
00222     goto DoPow;
00223 }
00224 }
00225 s += s[1];
00226 }
00227 sspow:
00228 s = subs;
00229 *++m = *++t;
00230 for ( si = 0; si < setflag; si += 2 ) {
00231     if ( t == temp + setlist[si] ) {
00232         t++; m++;
00233         goto Seven;
00234     }
00235 }
00236 for ( j = 0; j < i; j++ ) {
00237     if ( ( ABS(*t) - 2*MAXPOWER ) == s[2] ) {
00238         if ( *s == SYMTONUM ) {
00239             dirty = 1;
00240             *m = s[3];
00241             if ( *t < 0 ) *m = -*m;
00242             break;
00243         }
00244         else if ( *s == SYMTOSYM ) {
00245             dirty = 1;
00246             *z++ = EXPONENT;
00247             if ( *t < 0 ) *z++ = FUNHEAD+ARGHEAD+10;
00248             else *z++ = 4+FUNHEAD;
00249             *z++ = 0;
00250             FILLFUN3(z)

```

```

00251             *z++ = -SYMBOL;
00252             *z++ = m[-1];
00253             if ( *t < 0 ) {
00254                 *z++ = ARGHEAD+8;
00255                 *z++ = 0;
00256                 *z++ = 8;
00257                 *z++ = SYMBOL;
00258                 *z++ = 4;
00259                 *z++ = s[3];
00260                 *z++ = 1;
00261                 *z++ = 1;
00262                 *z++ = 1;
00263                 *z = -3;
00264             }
00265             else {
00266                 *z++ = -SYMBOL;
00267                 *z++ = s[3];
00268             }
00269             m -= 2;
00270             break;
00271         }
00272         else if ( *s == SYMTOSUB ) {
00273             zst = z;
00274             *z++ = SYMBOL;
00275             *z++ = 4;
00276             *z++ = *-m;
00277             w = z;
00278             *z++ = *t;
00279             goto MakeExp;
00280         }
00281     }
00282     s += s[1];
00283 }
00284 t++;
00285 if ( *m ) m++;
00286 else m--;
00287 Seven;;
00288 }
00289 j = WORDDIF(m,v);
00290 if ( !j ) m -= 2;
00291 else v[-1] = j + 2;
00292 s = accu;
00293 while ( s < z ) *m++ = *s++;
00294 break;
00295 /*
00296 #] SYMBOLS :
00297 */
00298 case DOTPRODUCT:
00299 /*
00300 #[ DOTPRODUCTS :
00301 */
00302     *m++ = *t++;
00303     *m++ = *t++;
00304     v = m;
00305     z = accu;
00306     while ( t < u ) {
00307         *m = *t;
00308         subcount = 0;
00309         /* Process the first vector of the DOTPRODUCT */
00310         for ( si = 0; si < setflag; si += 2 ) {
00311             if ( t == temp + setlist[si] ) goto ss2;
00312         }
00313         s = subs;
00314         for ( j = 0; j < i; j++ ) {
00315             if ( *t == s[2] ) {
00316                 if ( *s == VECTOVEC ) {
00317                     *m = s[3]; dirty = 1; break;
00318                 }
00319                 if ( *s == VECTOMIN ) {
00320                     *m = s[3];
00321                     dirty = 1;
00322                     if ( ( ABS(t[2]) - 2*MAXPOWER ) < 0 ) {
00323                         /* The power is a number */
00324                         sgn += t[2];
00325                     }
00326                     else {
00327                         /* The power is a wildcard. Put a -, resolve later. */
00328                         sgn++;
00329                     }
00330                     break;
00331                 }
00332                 if ( *s == VECTOSUB ) {
00333                     *m = s[3]; dirty = 1; subcount = 1; break;
00334                 }
00335             }
00336             s += s[1];
00337         }

```



```

00338 ss2:
00339     ***m = ***t;
00340     s = subs;
00341     /* Process the second vector of the DOTPRODUCT */
00342     for ( si = 0; si < setflag; si += 2 ) {
00343         if ( t == temp + setlist[si] ) goto ss3;
00344     }
00345     for ( j = 0; j < i; j++ ) {
00346         if ( *t == s[2] ) {
00347             if ( *s == VECTOVEC ) {
00348                 *m = s[3]; dirty = 1; break;
00349             }
00350             if ( *s == VECTOMIN ) {
00351                 *m = s[3];
00352                 dirty = 1;
00353                 if ( ( ABS(t[1]) - 2*MAXPOWER ) < 0 ) {
00354                     /* The power is a number */
00355                     sgn += t[1];
00356                 }
00357                 else {
00358                     /* The power is a wildcard. Put a -, resolve later. */
00359                     sgn++;
00360                 }
00361                 break;
00362             }
00363             if ( *s == VECTOSUB ) {
00364                 *m = s[3]; dirty = 1; subcount += 2; break;
00365             }
00366         }
00367         s += s[1];
00368     }
00369 ss3:
00370     ***m = ***t;
00371     /* Process the power */
00372     if ( ( ABS(*t) - 2*MAXPOWER ) < 0 ) goto RegPow;
00373     s = subs;
00374     for ( j = 0; j < i; j++ ) {
00375         if ( ( ABS(*t) - 2*MAXPOWER ) == s[2] ) {
00376             if ( *s == SYMTONUM ) {
00377                 *m = s[3];
00378                 /* Since the power is a wildcard, sgn is 0,1,2 depending whether
00379                    there were 0,1,2 VECTOMIN in the DOTPRODUCT. Multiply by the
00380                    power, which is positive currently. */
00381                 sgn *= *m;
00382                 /* Now flip the sign of the power, if the wildcard came with a - */
00383                 if ( *t < 0 ) *m = -*m;
00384                 dirty = 1;
00385                 break;
00386             }
00387             if ( *s <= SYMTOSUB ) {
00388
00389                 Here we put together a power function with the proper
00390                 arguments. Note that a p?.q? resolves to a single power.
00391
00392                 m -= 2;
00393                 *z++ = EXPONENT;
00394                 w = z;
00395                 if ( subcount == 0 ) {
00396                     *z++ = 17+FUNHEAD+2*ARGHEAD;
00397                     *z++ = DIRTYFLAG;
00398                     FILLFUN3(z)
00399                     *z++ = 9+ARGHEAD;
00400                     *z++ = 0;
00401                     FILLARG(z)
00402                     *z++ = 9;
00403                     *z++ = DOTPRODUCT;
00404                     *z++ = 5;
00405                     *z++ = *m;
00406                     *z++ = m[1];
00407                     *z++ = 1;
00408                     *z++ = 1;
00409                     *z++ = 1;
00410                     *z++ = 3;
00411                     if ( *s == SYMTOSYM ) {
00412                         *z++ = 8+ARGHEAD;
00413                         *z++ = 0;
00414                         FILLARG(z)
00415                         *z++ = 8;
00416                         *z++ = SYMBOL;
00417                         *z++ = 4;
00418                         *z++ = s[3];
00419                         *z++ = 1;
00420                     }
00421                     else {
00422                         *z++ = 4+SUBEXPSIZE+ARGHEAD;
00423                         *z++ = 1;
00424                         FILLARG(z)
00425                         *z++ = 4+SUBEXPSIZE;

```

```

00425         *z++ = SUBEXPRESSION;
00426         *z++ = SUBEXPSIZE;
00427         *z++ = s[3];
00428         *z++ = 1;
00429         *z++ = AT.ebufnum;
00430         FILLSUB(z)
00431     }
00432     *z++ = 1; *z++ = 1;
00433     *z++ = ( s[2] > 0 ) ? 3: -3;
00434 }
00435 else if ( subcount == 3 ) {
00436     *z++ = 20+2*SUBEXPSIZE+FUNHEAD+2*ARGHEAD;
00437     *z++ = DIRTYFLAG;
00438     FILLFUN3(z)
00439     *z++ = 12+2*SUBEXPSIZE+ARGHEAD;
00440     *z++ = 1;
00441     *z++ = 12+2*SUBEXPSIZE;
00442     *z++ = SUBEXPRESSION;
00443     *z++ = 4+SUBEXPSIZE;
00444     *z++ = *m + 1;
00445     *z++ = 1;
00446     *z++ = AT.ebufnum;
00447     FILLSUB(z)
00448     *z++ = INDTOIND;
00449     *z++ = 4;
00450     *z++ = FUNNYVEC;
00451     *z++ = ++AR.CurDum;

00452
00453     *z++ = SUBEXPRESSION;
00454     *z++ = 4+SUBEXPSIZE;
00455     *z++ = m[1] + 1;
00456     *z++ = 1;
00457     *z++ = AT.ebufnum;
00458     FILLSUB(z)
00459     *z++ = INDTOIND;
00460     *z++ = 4;
00461     *z++ = FUNNYVEC;
00462     *z++ = AR.CurDum;
00463     *z++ = 1; *z++ = 1; *z++ = 3;
00464 }
00465 else {
00466     if ( subcount == 2 ) {
00467         j = *m; *m = m[1]; m[1] = j;
00468     }
00469     *z++ = 16+SUBEXPSIZE+FUNHEAD+2*ARGHEAD;
00470     *z++ = DIRTYFLAG;
00471     FILLFUN3(z)
00472     *z++ = 8+SUBEXPSIZE+ARGHEAD;
00473     *z++ = 1;
00474     *z++ = 8+SUBEXPSIZE;
00475     *z++ = SUBEXPRESSION;
00476     *z++ = 4+SUBEXPSIZE;
00477     *z++ = *m + 1;
00478     *z++ = 1;
00479     *z++ = AT.ebufnum;
00480     FILLSUB(z)
00481     *z++ = INDTOIND;
00482     *z++ = 4;
00483     *z++ = FUNNYVEC;
00484     *z++ = m[1];
00485     *z++ = 1; *z++ = 1; *z++ = 3;
00486 }
00487 if ( *s == SYMTOSYM ) {
00488     if ( s[2] > 0 ) {
00489         *z++ = -SYMBOL;
00490         *z++ = s[3];
00491         t++;
00492         *w = z-w+1;
00493         goto NextDot;
00494     }
00495     *z++ = 8+ARGHEAD;
00496     *z++ = 0;
00497     *z++ = 8;
00498     *z++ = SYMBOL;
00499     *z++ = 4;
00500     *z++ = s[3];
00501     *z++ = 1;
00502 }
00503 else {
00504     *z++ = 4+SUBEXPSIZE+ARGHEAD;
00505     *z++ = 1;
00506     *z++ = 4+SUBEXPSIZE;
00507     *z++ = SUBEXPRESSION;
00508     *z++ = SUBEXPSIZE;
00509     *z++ = s[3];
00510     *z++ = 1;
00511     *z++ = AT.ebufnum;

```

```

00512             FILLSUB(z)
00513         }
00514         *z++ = 1; *z++ = 1;
00515         *z++ = ( s[2] > 0 ) ? 3: -3;
00516         t++;
00517         *w = z-w+1;
00518         goto NextDot;
00519     }
00520 }
00521 s += s[1];
00522 }
00523 RegPow:
00524 if ( *m ) m++;
00525 else { m -= 2; subcount = 0; }
00526 t++;
00527 if ( subcount ) {
00528     m -= 3;
00529     if ( subcount == 3 ) {
00530         if ( m[2] < 0 ) {
00531             j = (-m[2]) * (2*SUBEXPSIZE+8);
00532             *z++ = DENOMINATOR;
00533             *z++ = j + 8 + FUNHEAD + ARGHEAD;
00534             *z++ = DIRTYFLAG;
00535             FILLFUN3(z)
00536             *z++ = j + 8 + ARGHEAD;
00537             *z++ = 1;
00538             *z++ = j + 8;
00539             while ( m[2] < 0 ) {
00540                 (m[2])++;
00541                 *z++ = SUBEXPRESSION;
00542                 *z++ = 4+SUBEXPSIZE;
00543                 *z++ = *m + 1;
00544                 *z++ = 1;
00545                 *z++ = AT.ebufnum;
00546                 FILLSUB(z)
00547                 *z++ = INDTOIND;
00548                 *z++ = 4;
00549                 *z++ = FUNNYVEC;
00550                 *z++ = ++AR.CurDum;
00551                 *z++ = SUBEXPRESSION;
00552                 *z++ = 8+SUBEXPSIZE;
00553                 *z++ = m[1] + 1;
00554                 *z++ = 1;
00555                 *z++ = AT.ebufnum;
00556                 FILLSUB(z)
00557                 *z++ = INDTOIND;
00558                 *z++ = 4;
00559                 *z++ = FUNNYVEC;
00560                 *z++ = AR.CurDum;
00561                 *z++ = SYMTOSYM; /* Needed to avoid */
00562                 *z++ = 4; /* problems with */
00563                 *z++ = 1000; /* conversion to */
00564                 *z++ = 1000; /* square of subexp*/
00565             }
00566             *z++ = 1; *z++ = 1; *z++ = 3;
00567         }
00568         else {
00569             while ( m[2] > 0 ) {
00570                 (m[2])--;
00571                 *z++ = SUBEXPRESSION;
00572                 *z++ = 4+SUBEXPSIZE;
00573                 *z++ = *m + 1;
00574                 *z++ = 1;
00575                 *z++ = AT.ebufnum;
00576                 FILLSUB(z)
00577                 *z++ = INDTOIND;
00578                 *z++ = 4;
00579                 *z++ = FUNNYVEC;
00580                 *z++ = ++AR.CurDum;
00581                 *z++ = SUBEXPRESSION;
00582                 *z++ = 4+SUBEXPSIZE;
00583                 *z++ = m[1] + 1;
00584                 *z++ = 1;
00585                 *z++ = AT.ebufnum;
00586                 FILLSUB(z)
00587                 *z++ = INDTOIND;
00588                 *z++ = 4;
00589                 *z++ = FUNNYVEC;
00590                 *z++ = AR.CurDum;
00591             }
00592         }
00593     }
00594     else {
00595         if ( subcount == 2 ) {
00596             j = *m; *m = m[1]; m[1] = j;
00597         }
00598         if ( m[2] < 0 ) {
00599             *z++ = DENOMINATOR;

```

```

00599             *z++ = 8+SUBEXPSIZE+FUNHEAD+ARGHEAD;
00600             *z++ = DIRTYFLAG;
00601             FILLFUN3(z)
00602             *z++ = 8+SUBEXPSIZE+ARGHEAD;
00603             *z++ = 1;
00604             *z++ = 8+SUBEXPSIZE;
00605         }
00606         *z++ = SUBEXPRESSION;
00607         *z++ = 4+SUBEXPSIZE;
00608         *z++ = *m + 1;
00609         *z++ = ABS(m[2]);
00610         *z++ = AT.ebufnum;
00611         FILLSUB(z)
00612         *z++ = INDTOIND;
00613         *z++ = 4;
00614         *z++ = FUNNYVEC;
00615         *z++ = m[1];
00616         if ( m[2] < 0 ) {
00617             *z++ = 1; *z++ = 1; *z++ = 3;
00618         }
00619     }
00620 }
00621 NextDot;;
00622     }
00623     if ( m <= v ) m = v - 2;
00624     else v[-1] = WORDDIF(m,v) + 2;
00625     if ( z > accu ) {
00626         j = WORDDIF(z,accu);
00627         z = accu;
00628         NCOPY(m,z,j);
00629     }
00630     break;
00631 /*
00632 #] DOTPRODUCTS :
00633 */
00634 case SETSET:
00635 /*
00636 #[ SETS :
00637 */
00638     temp = accu + ((AR.ComprTop - accu)»1)&(-2));
00639     if ( ResolveSet(BHEAD t,temp,sub) ) {
00640         Terminate(-1);
00641     }
00642     setlist = t + 2 + t[3];
00643     setflag = t[1] - 2 - t[3]; /* Number of elements * 2 */
00644     t = temp; u = t + t[1];
00645     goto ReSwitch;
00646 /*
00647 #] SETS :
00648 */
00649 case VECTOR:
00650 /*
00651 #[ VECTORS :
00652 */
00653     *m++ = *t++;
00654     *m++ = *t++;
00655     v = m;
00656     z = accu;
00657     while ( t < u ) {
00658         *m = *t;
00659         for ( si = 0; si < setflag; si += 2 ) {
00660             if ( t == temp + setlist[si] ) goto ss4;
00661         }
00662         s = subs;
00663         for ( j = 0; j < i; j++ ) {
00664             if ( *t == s[2] ) {
00665                 if ( *s == INDTOIND || *s == VECTOVEC ) {
00666                     *m = s[3]; dirty = 1; break;
00667                 }
00668                 if ( *s == VECTOMIN ) {
00669                     *m = s[3]; dirty = 1; sgn++; break;
00670                 }
00671                 else if ( *s == VECTOSUB ) {
00672                     *z++ = SUBEXPRESSION;
00673                     *z++ = 4+SUBEXPSIZE;
00674                     *z++ = s[3]+1;
00675                     *z++ = 1;
00676                     *z++ = AT.ebufnum;
00677                     FILLSUB(z)
00678                     *z++ = VECTOVEC;
00679                     *z++ = 4;
00680                     *z++ = FUNNYVEC;
00681                     *z++ = *++t;
00682                     m--;
00683                     s = subs;
00684                     for ( j = 0; j < i; j++ ) {
00685                         if ( z[-1] == s[2] ) {

```

```

00686         if ( *s == INDTOIND || *s == VECTOVEC ) {
00687             z[-1] = s[3];
00688             break;
00689         }
00690         if ( *s == INDTOSUB || *s == VECTOSUB ) {
00691             z[-1] = ++AR.CurDum;
00692             *z++ = SUBEXPRESSION;
00693             *z++ = 4+SUBEXPSIZE;
00694             *z++ = s[3]+1;
00695             *z++ = 1;
00696             *z++ = AT.ebufnum;
00697             FILLSUB(z)
00698             if ( *s == INDTOSUB ) *z++ = INDTOIND;
00699             else *z++ = VECTOSUB;
00700             *z++ = 4;
00701             *z++ = FUNNYVEC;
00702             *z++ = AR.CurDum;
00703             break;
00704         }
00705     }
00706     s += s[1];
00707 }
00708 dirty = 1;
00709 break;
00710 }
00711 else if ( *s == INDTOSUB ) {
00712     *z++ = SUBEXPRESSION;
00713     *z++ = 4+SUBEXPSIZE;
00714     *z++ = s[3]+1;
00715     *z++ = 1;
00716     *z++ = AT.ebufnum;
00717     FILLSUB(z)
00718     *z++ = INDTOIND;
00719     *z++ = 4;
00720     *z++ = FUNNYVEC;
00721     m -= 2;
00722     *z++ = m[1];
00723     dirty = 1;
00724     t++;
00725     break;
00726 }
00727 }
00728 s += s[1];
00729 }
00730 ss4:
00731     m++; t++;
00732     if ( m <= v ) m = v-2;
00733     else v[-1] = WORDDIF(m,v)+2;
00734     if ( z > accu ) {
00735         j = WORDDIF(z,accu); z = accu;
00736         NCOPY(m,z,j);
00737     }
00738     break;
00739 /*
00740 #] VECTORS :
00741 */
00742 case INDEX:
00743 /*
00744 #[ INDEX :
00745 */
00746     *m++ = *t++;
00747     *m++ = *t++;
00748     v = m;
00749     z = accu;
00750     while ( t < u ) {
00751         *m = *t;
00752         for ( si = 0; si < setflag; si += 2 ) {
00753             if ( t == temp + setlist[si] ) goto ss5;
00754         }
00755         s = subs;
00756         for ( j = 0; j < i; j++ ) {
00757             if ( *t == s[2] ) {
00758                 if ( *s == INDTOIND || *s == VECTOVEC )
00759                     { *m = s[3]; dirty = 1; break; }
00760                 if ( *s == VECTOMIN )
00761                     { *m = s[3]; dirty = 1; sgn++; break; }
00762                 else if ( *s == VECTOSUB || *s == INDTOSUB ) {
00763                     *z++ = SUBEXPRESSION;
00764                     *z++ = SUBEXPSIZE;
00765                     *z++ = s[3];
00766                     *z++ = 1;
00767                     *z++ = AT.ebufnum;
00768                     FILLSUB(z)
00769                     m--;
00770                     dirty = 1;
00771                     break;
00772                 }

```

```

00773         }
00774         s += s[1];
00775     }
00776 ss5:      m++; t++;
00777     }
00778     if ( m <= v ) m = v-2;
00779     else v[-1] = WORDDIF(m,v)+2;
00780     if ( z > accu ) {
00781         j = WORDDIF(z,accu); z = accu;
00782         NCOPY(m,z,j);
00783     }
00784     break;
00785 /*
00786     #] INDEX :
00787 */
00788     case DELTA:
00789     case LEVICIVITA:
00790     case GAMMA:
00791 /*
00792     #[ SPECIALS :
00793 */
00794     v = m;
00795     *m++ = *t++;
00796     *m++ = *t++;
00797 #if FUNHEAD > 2
00798     if ( t[-2] != DELTA ) *m++ = *t++;
00799 #endif
00800 Tensors:
00801     COPYFUN3(m,t)
00802     z = accu;
00803     while ( t < u ) {
00804         *m = *t;
00805         for ( si = 0; si < setflag; si += 2 ) {
00806             if ( t == temp + setlist[si] ) goto ss6;
00807         }
00808         s = subs;
00809         if ( *m == FUNNYWILD ) {
00810             CBUF *C = cbuf+AT.ebufnum;
00811             t++;
00812             for ( j = 0; j < i; j++ ) {
00813                 if ( *s == ARGTOARG && *t == s[2] ) {
00814                     v[2] |= DIRTYFLAG;
00815                     if ( s[3] < 0 ) { /* empty */
00816                         t++; break;
00817                     }
00818                     w = C->rhs[s[3]];
00819                     DEBUG(MesPrint("Thread %w(a): s[3] = %d, w=(%d,%d,%d,%d)",s[3],w[0],w[1],w[2],w[3]));
00820                     j = *w++;
00821                     if ( j > 0 ) {
00822                         NCOPY(m,w,j);
00823                     }
00824                     else {
00825                         while ( *w ) {
00826                             if ( *w == -INDEX || *w == -VECTOR
00827                                 || *w == -MINVECTOR
00828                                 || ( *w == -SNUMBER && w[1] >= 0
00829                                     && w[1] < AM.OffsetIndex ) ) {
00830                                 if ( *w == -MINVECTOR ) sgn++;
00831                                 w++;
00832                                 *m++ = *w++;
00833                             }
00834                             else {
00835                                 MLOCK(ErrorMessageLock);
00836                                 DEBUG(MesPrint("Thread %w(aa): *w = %d",*w));
00837                                 MesPrint("Illegal substitution of argument field in
00838 tensor");
00839                                 MUNLOCK(ErrorMessageLock);
00840                                 SETERROR(-1)
00841                             }
00842                         }
00843                         t++;
00844                         break;
00845                     }
00846                     s += s[1];
00847                 }
00848             }
00849             else {
00850                 for ( j = 0; j < i; j++ ) {
00851                     if ( *t == s[2] ) {
00852                         if ( *s == INDTOIND || *s == VECTOVEC )
00853                             { *m = s[3]; dirty = 1; break; }
00854                         if ( *s == VECTOMIN )
00855                             { *m = s[3]; dirty = 1; sgn++; break; }
00856                         else if ( *s == VECTOSUB || *s == INDTOSUB ) {
00857                             *m = ++AR.CurDum;
00858                             *z++ = SUBEXPRESSION;

```

```

00859                                     *z++ = 4+SUBEXPSIZE;
00860                                     *z++ = s[3]+1;
00861                                     *z++ = 1;
00862                                     *z++ = AT.ebufnum;
00863                                     FILLSUB(z)
00864                                     *z++ = INDTOIND;
00865                                     *z++ = 4;
00866                                     *z++ = FUNNYVEC;
00867                                     *z++ = AR.CurDum;
00868                                     dirty = 1;
00869                                     break;
00870                                 }
00871                             }
00872                             s += s[1];
00873                         }
00874                         if ( j < i && *v != DELTA ) v[2] |= DIRTYFLAG;
00875 ss6:                             m++; t++;
00876                     }
00877                 }
00878                 v[1] = WORDDIF(m,v);
00879                 if ( z > accu ) {
00880                     j = WORDDIF(z,accu); z = accu;
00881                     NCOPY(m,z,j);
00882                 }
00883                 break;
00884 /*
00885 #] SPECIALS :
00886 */
00887 case SUBEXPRESSION:
00888 /*
00889 #[ SUBEXPRESSION :
00890 */
00891     dirty = 1;
00892     tstop = t + t[1];
00893     *m++ = *t++;
00894     *m++ = *t++;
00895     *m++ = *t++;
00896     *m++ = *t++;
00897     if ( t[-1] >= 2*MAXPOWER || t[-1] <= -2*MAXPOWER ) {
00898         s = subs;
00899         for ( j = 0; j < i; j++ ) {
00900             if ( *s == SYMTONUM &&
00901                 ( ABS(t[-1]) - 2*MAXPOWER ) == s[2] ) {
00902                 m[-1] = s[3];
00903                 if ( t[-1] < 0 ) m[-1] = -m[-1];
00904                 break;
00905             }
00906             s += s[1];
00907         }
00908     }
00909     *m++ = *t++;
00910     COPYSUB(m,t)
00911     while ( t < tstop ) {
00912         for ( si = 0; si < setflag; si += 2 ) {
00913             if ( t == temp + setlist[si] - 2 ) goto ss7;
00914         }
00915         s = subs;
00916         for ( j = 0; j < i; j++ ) {
00917             if ( s[2] == t[2] ) {
00918                 if ( ( *s <= SYMTOSUB && *t <= SYMTOSUB )
00919                     || ( *s == *t && *s < FROMBRAC )
00920                     || ( *s == VECTOVEC && ( *t == VECTOSUB || *t == VECTOMIN ) )
00921                     || ( *s == VECTOSUB && ( *t == VECTOVEC || *t == VECTOMIN ) )
00922                     || ( *s == VECTOMIN && ( *t == VECTOSUB || *t == VECTOVEC ) )
00923                     || ( *s == INDTOIND && *t == INDTOIND )
00924                     || ( *s == INDTOIND && *t == INDTOIND ) ) {
00925                     WORD *vv = m;
00926 /*
00927                     *t = *s; Wrong!!! Overwrites compiler buffer */
00928                     j = t[1];
00929                     NCOPY(m,t,j);
00930                     vv[0] = s[0];
00931                     vv[3] = s[3];
00932                     goto sr7;
00933                 }
00934             }
00935             s += s[1];
00936 ss7:             j = t[1];
00937             NCOPY(m,t,j);
00938 sr7:;
00939         }
00940         break;
00941 /*
00942 #] SUBEXPRESSION :
00943 */
00944 case EXPRESSION:
00945 /*

```

```

00946      #[] EXPRESSION :
00947      */
00948          dirty = 1;
00949          tstop = t + t[1];
00950          v = m;
00951          *m++ = *t++;
00952          *m++ = *t++;
00953          *m++ = *t++;
00954          *m++ = *t++;
00955          s = subs;
00956          for ( j = 0; j < i; j++ ) {
00957              if ( ( ABS(t[-1]) - 2*MAXPOWER ) == s[2] ) {
00958                  if ( *s == SYMTONUM ) {
00959                      m[-1] = s[3];
00960                      if ( t[-1] < 0 ) m[-1] = -m[-1];
00961                      break;
00962                  }
00963                  else if ( *s <= SYMTOSUB ) {
00964                      MLOCK(ErrorMessageLock);
00965                      MesPrint("Wildcard power of expression should be a number");
00966                      MUNLOCK(ErrorMessageLock);
00967                      SETERROR(-1)
00968                  }
00969              }
00970              s += s[1];
00971          }
00972          *m++ = *t++;
00973          COPYSUB(m,t)
00974          while ( t < tstop && *t != WILDCARDS ) {
00975              j = t[1];
00976              NCOPY(m,t,j);
00977          }
00978          if ( t < tstop && *t == WILDCARDS ) {
00979              *m++ = *t;
00980              s = sub;
00981              j = s[1];
00982              *m++ = j+2;
00983              NCOPY(m,s,j);
00984              t += t[1];
00985          }
00986          if ( t < tstop && *t == FROMBRAC ) {
00987              w = m;
00988              *m++ = *t;
00989              *m++ = t[1];
00990              if ( WildFill(BHEAD m,t+2,sub) < 0 ) {
00991                  MLOCK(ErrorMessageLock);
00992                  MesCall("WildFill");
00993                  MUNLOCK(ErrorMessageLock);
00994                  SETERROR(-1)
00995              }
00996              m += *m;
00997              w[1] = m - w;
00998              t += t[1];
00999          }
01000          while ( t < tstop ) {
01001              j = t[1];
01002              NCOPY(m,t,j);
01003          }
01004          v[1] = m-v;
01005          break;
01006      /*
01007      #[] EXPRESSION :
01008      */
01009      default:
01010      /*
01011      #[] FUNCTIONS :
01012      */
01013          if ( *t >= FUNCTION ) {
01014              dflag = 0;
01015              na = 0;
01016              *m = *t;
01017              for ( si = 0; si < setflag; si += 2 ) {
01018                  if ( t == temp + setlist[si] ) {
01019                      dflag = DIRTYFLAG; goto ss8;
01020                  }
01021              }
01022              s = subs;
01023              for ( j = 0; j < i; j++ ) {
01024                  if ( *s == FUNTOFUN && *t == s[2] )
01025                      { *m = s[3]; dirty = 1; dflag = DIRTYFLAG; break; }
01026                  s += s[1];
01027              }
01028          ss8:
01029              v = m;
01030              if ( *t >= FUNCTION && functions[*t-FUNCTION].spec
01031                  >= TENSORFUNCTION ) {
01032                  if ( *m < FUNCTION || functions[*m-FUNCTION].spec
01033                      < TENSORFUNCTION ) {

```



```

01033             MLOCK(ErrorMessageLock);
01034             MesPrint("Illegal wildcarding of regular function to tensorfunction");
01035             MUNLOCK(ErrorMessageLock);
01036             SETERROR(-1)
01037         }
01038         m++; t++;
01039         *m++ = *t++;
01040         *m++ = *t++ | dflag;
01041         goto Tensors;
01042     }
01043     m++; t++;
01044     *m++ = *t++;
01045     *m++ = *t++ | dflag;
01046     COPYFUN3(m,t)
01047     z = accu;
01048     while ( t < u ) {          /* do an argument */
01049         if ( *t < 0 ) {
01050             /*
01051              *#[ Simple arguments :
01052              */
01053             CBUF *C = cbuf+AT.ebufnum;
01054             for ( si = 0; si < setflag; si += 2 ) {
01055                 if ( *t <= -FUNCTION ) {
01056                     if ( t == temp + setlist[si] ) {
01057                         v[2] |= DIRTYFLAG; goto ss10; }
01058                     }
01059                     else {
01060                         if ( t == temp + setlist[si]-1 ) {
01061                             v[2] |= DIRTYFLAG; goto ss9; }
01062                         }
01063                     }
01064                 if ( *t == -ARGWILD ) {
01065                     s = subs;
01066                     for ( j = 0; j < i; j++ ) {
01067                         if ( *s == ARGTOARG && s[2] == t[1] ) break;
01068                         s += s[1];
01069                     }
01070                     v[2] |= DIRTYFLAG;
01071                     w = C->rhs[s[3]];
01072                     DEBUG(MesPrint("Thread %w(b): s[3] = %d, w=(%d,%d,%d,%d)",s[3],w[0],w[1],w[2],w[3]));
01073                     if ( *w == 0 ) {
01074                         w++;
01075                         while ( *w ) {
01076                             if ( *w > 0 ) j = *w;
01077                             else if ( *w <= -FUNCTION ) j = 1;
01078                             else j = 2;
01079                             NCOPY(m,w,j);
01080                         }
01081                     }
01082                     else {
01083                         j = *w++;
01084                         while ( --j >= 0 ) {
01085                             if ( *w < MINSPEC ) *m++ = -VECTOR;
01086                             else if ( *w >= 0 && *w < AM.OffsetIndex )
01087                                 *m++ = -SNUMBER;
01088                             else *m++ = -INDEX;
01089                             *m++ = *w++;
01090                         }
01091                     }
01092                     t += 2;
01093                     dirty = 1;
01094                     if ( ( *v == NUMARGSFUN || *v == NUMTERMSFUN )
01095                         && t >= u && m == v + FUNHEAD ) {
01096                         m = v;
01097                         *m++ = SNUMBER; *m++ = 3; *m++ = 0;
01098                         break;
01099                     }
01100                 }
01101                 else if ( *t <= -FUNCTION ) {
01102                     *m = *t;
01103                     s = subs;
01104                     for ( j = 0; j < i; j++ ) {
01105                         if ( -*t == s[2] ) {
01106                             if ( *s == FUNTOFUN )
01107                                 { *m = -s[3]; dirty = 1; v[2] |= DIRTYFLAG; break; }
01108                             }
01109                         s += s[1];
01110                     }
01111                     m++; t++;
01112                 }
01113                 else if ( *t == -SYMBOL ) {
01114                     *m++ = *t++;
01115                     *m = *t;
01116                     s = subs;
01117                     for ( j = 0; j < i; j++ ) {
01118                         if ( *t == s[2] && *s <= SYMTOSUB ) {
01119                             dirty = 1; v[2] |= DIRTYFLAG;

```

```

01120         if ( AR.PolyFunType == 2 && v[0] == AR.PolyFun )
01121             v[2] |= MUSTCLEANPRF;
01122         if ( *s == SYMTOSYM ) *m = s[3];
01123     else if ( *s == SYMTONUM ) {
01124         m[-1] = -SNUMBER;
01125         *m = s[3];
01126     }
01127     else if ( *s == SYMTOSUB ) {
01128         ToSub:
01129         m--;
01130         w = C->rhs[s[3]];
01131         DEBUG(MesPrint("Thread %w(c): s[3] = %d, w=(%d,%d,%d,%d)",s[3],w[0],w[1],w[2],w[3]));
01132         s = m;
01133         m += 2;
01134         while ( *w ) {
01135             j = *w;
01136             NCOPY(m,w,j);
01137         }
01138         *s = WORDDIF(m,s);
01139         s[1] = 0;
01140         *m = 0;
01141         if ( t[-1] == -MINVECTOR ) {
01142             w = s+2;
01143             while ( *w ) {
01144                 w += *w;
01145                 w[-1] = -w[-1];
01146             }
01147             if ( ToFast(s,s) ) {
01148                 if ( *s <= -FUNCTION ) m = s;
01149                 else m = s + 1;
01150             }
01151             else m--;
01152         }
01153         break;
01154     }
01155     s += s[1];
01156 }
01157 m++; t++;
01158 }
01159 else if ( *t == -INDEX ) {
01160     *m++ = *t++;
01161     *m = *t;
01162     s = subs;
01163     for ( j = 0; j < i; j++ ) {
01164         if ( *t == s[2] ) {
01165             if ( *s == INDTOIND || *s == VECTOVEC ) {
01166                 *m = s[3];
01167                 if ( *m < MINSPEC ) m[-1] = -VECTOR;
01168                 else if ( *m >= 0 && *m < AM.OffsetIndex )
01169                     m[-1] = -SNUMBER;
01170                 else m[-1] = -INDEX;
01171             }
01172             else if ( *s == VECTOSUB || *s == INDTOSUB ) {
01173                 m[-1] = -INDEX;
01174                 *m = ++AR.CurDum;
01175                 *z++ = SUBEXPRESSION;
01176                 *z++ = 4+SUBEXPSIZE;
01177                 *z++ = s[3]+1;
01178                 *z++ = 1;
01179                 *z++ = AT.ebufnum;
01180                 FILLSUB(z);
01181                 *z++ = INDTOIND;
01182                 *z++ = 4;
01183                 *z++ = FUNNYVEC;
01184                 *z++ = AR.CurDum;
01185             }
01186             v[2] |= DIRTYFLAG; dirty = 1;
01187             break;
01188         }
01189         s += s[1];
01190     }
01191     m++; t++;
01192 }
01193 else if ( *t == -VECTOR || *t == -MINVECTOR ) {
01194     *m++ = *t++;
01195     *m = *t;
01196     s = subs;
01197     for ( j = 0; j < i; j++ ) {
01198         if ( *t == s[2] ) {
01199             if ( *s == VECTOVEC ) *m = s[3];
01200             else if ( *s == VECTOMIN ) {
01201                 *m = s[3];
01202                 if ( t[-1] == -VECTOR )
01203                     m[-1] = -MINVECTOR;
01204                 else
01205                     m[-1] = -VECTOR;
01206             }

```

```

01207         else if ( *s == VECTOSUB ) goto ToSub;
01208         dirty = 1; v[2] |= DIRTYFLAG;
01209         break;
01210     }
01211     s += s[1];
01212 }
01213 m++; t++;
01214 }
01215 else if ( *t == -SNUMBER ) {
01216     *m++ = *t++;
01217     *m = *t;
01218     s = subs;
01219     for ( j = 0; j < i; j++ ) {
01220         if ( *t == s[2] && *s >= NUMTONUM && *s <= NUMTOSUB ) {
01221             dirty = 1; v[2] |= DIRTYFLAG;
01222             if ( *s == NUMTONUM ) *m = s[3];
01223             else if ( *s == NUMTOSYM ) {
01224                 m[-1] = -SYMBOL;
01225                 *m = s[3];
01226             }
01227             else if ( *s == NUMTOIND ) {
01228                 m[-1] = -INDEX;
01229                 *m = s[3];
01230             }
01231             else if ( *s == NUMTOSUB ) goto ToSub;
01232             break;
01233         }
01234         s += s[1];
01235     }
01236     m++; t++;
01237 }
01238 else {
01239     *m++ = *t++;
01240     *m++ = *t++;
01241 }
01242 na = WORDDIF(z,accu);
01243 /*
01244 #] Simple arguments :
01245 */
01246 }
01247 else {
01248     w = m;
01249     zz = t;
01250     NEXTARG(zz)
01251     odirt = AN.WildDirt; AN.WildDirt = 0;
01252     AR.CompressPointer = accu + na;
01253     for ( j = 0; j < ARGHEAD; j++ ) *m++ = *t++;
01254     j = 0;
01255     adirt = 0;
01256     while ( t < zz ) { /* do a term */
01257         if ( ( len = WildFill(BHEAD m,t,sub) ) < 0 ) {
01258             MLOCK(ErrorMessageLock);
01259             MesCall("WildFill");
01260             MUNLOCK(ErrorMessageLock);
01261             SETERROR(-1)
01262         }
01263         if ( AN.WildDirt ) {
01264             adirt = AN.WildDirt;
01265             AN.WildDirt = 0;
01266         }
01267         m += len;
01268         t += *t;
01269     }
01270     *w = WORDDIF(m,w); /* Fill parameter length */
01271     if ( adirt ) {
01272         dirty = w[1] = 1; v[2] |= DIRTYFLAG;
01273         if ( AR.PolyFunType == 2 && v[0] == AR.PolyFun )
01274             v[2] |= MUSTCLEANPRF;
01275         AN.WildDirt = adirt;
01276     }
01277     else {
01278         AN.WildDirt = odirt;
01279     }
01280     if ( ToFast(w,w) ) {
01281         if ( *w <= -FUNCTION ) {
01282             if ( *w == NUMARGSFUN || *w == NUMTERMSFUN ) {
01283                 *w = -SNUMBER; w[1] = 0; m = w + 2;
01284             }
01285             else m = w+1;
01286         }
01287         else m = w+2;
01288     }
01289     AR.CompressPointer = oldcpointer;
01290 }
01291 }
01292 v[1] = WORDDIF(m,v); /* Fill function length */
01293 s = accu;

```

```

01294          NCOPY(m,s,na);
01295  /*
01296          Now some code to speed up a few special cases
01297  */
01298      if ( v[0] == EXPONENT ) {
01299          if ( v[1] == FUNHEAD+4 && v[FUNHEAD] == -SYMBOL &&
01300              v[FUNHEAD+2] == -SNUMBER && v[FUNHEAD+3] < MAXPOWER
01301              && v[FUNHEAD+3] > -MAXPOWER ) {
01302              v[0] = SYMBOL;
01303              v[1] = 4;
01304              v[2] = v[FUNHEAD+1];
01305              v[3] = v[FUNHEAD+3];
01306              m = v+4;
01307          }
01308          else if ( v[1] == FUNHEAD+ARGHEAD+11
01309              && v[FUNHEAD] == ARGHEAD+9
01310              && v[FUNHEAD+ARGHEAD] == 9
01311              && v[FUNHEAD+ARGHEAD+1] == DOTPRODUCT
01312              && v[FUNHEAD+ARGHEAD+8] == 3
01313              && v[FUNHEAD+ARGHEAD+7] == 1
01314              && v[FUNHEAD+ARGHEAD+6] == 1
01315              && v[FUNHEAD+ARGHEAD+5] == 1
01316              && v[FUNHEAD+ARGHEAD+9] == -SNUMBER
01317              && v[FUNHEAD+ARGHEAD+10] < MAXPOWER
01318              && v[FUNHEAD+ARGHEAD+10] > -MAXPOWER ) {
01319              v[0] = DOTPRODUCT;
01320              v[1] = 5;
01321              v[2] = v[FUNHEAD+ARGHEAD+3];
01322              v[3] = v[FUNHEAD+ARGHEAD+4];
01323              v[4] = v[FUNHEAD+ARGHEAD+10];
01324              m = v+5;
01325          }
01326      }
01327  }
01328  else { while ( t < u ) *m++ = *t++; }
01329  /*
01330      #] FUNCTIONS :
01331  */
01332      }
01333      t = uu;
01334  } while ( t < r );
01335  t = from; /* Copy coefficient */
01336  t += *t;
01337  if ( r < t ) do { *m++ = *r++; } while ( r < t );
01338  if ( ( sgn & 1 ) != 0 ) m[-1] = -m[-1];
01339  *to = WORDDIF(m,to);
01340  if ( dirty ) AN.WildDirt = dirty;
01341  return(*to);
01342 }
01343
01344 /*
01345      #] WildFill :
01346      #[ ResolveSet :          WORD ResolveSet(from,to,subs)
01347
01348      The set syntax is:
01349      SET,length,subterm,where,whichmember[,where,whichmember]
01350
01351      setlength is 2*n+1 with n the number of set substitutions.
01352      length = setlength + subtermlength + 2
01353
01354      At 'where' is the number of the set and 'whichmember' is the
01355      number of the element. This is still a symbol/dollar and we
01356      have to find the substitution in the wildcards.
01357      The output is the subterm in which the setelements have been
01358      substituted. This is ready for further wildcard substitutions.
01359  */
01360
01361 int ResolveSet(PHEAD WORD *from, WORD *to, WORD *subs)
01362 {
01363     GETBIDENTITY
01364     WORD *m, *s, *w, j, i, ii, i3, flag, num;
01365     DOLLARS d = 0;
01366 #ifdef WITHPTHREADS
01367     int nummodopt, dtype = -1;
01368 #endif
01369     m = to; /* pointer in output */
01370     s = from + 2;
01371     w = s + s[1];
01372     while ( s < w ) *m++ = *s++;
01373     j = (from[1] - WORDDIF(w,from) ) » 1;
01374     m = subs + subs[1];
01375     subs += SUBEXPSIZE;
01376     s = subs;
01377     i = 0;
01378     while ( s < m ) { i++; s += s[1]; }
01379     m = to;
01380     if ( *m >= FUNCTION && functions[*m-FUNCTION].spec

```

```

01381     >= TENSORFUNCTION ) flag = 0;
01382     else flag = 1;
01383     while ( --j >= 0 ) {
01384         if ( w[l] >= 0 ) {
01385             s = subs;
01386             for ( ii = 0; ii < i; ii++ ) {
01387                 if ( *s == SYMTONUM && s[2] == w[l] ) { num = s[3]; goto GotOne; }
01388                 s += s[1];
01389             }
01390             MLOCK(ErrorMessageLock);
01391             MesPrint(" Unresolved setelement during substitution");
01392             MUNLOCK(ErrorMessageLock);
01393             return(-1);
01394         }
01395         else { /* Dollar ! */
01396             d = Dollars - w[l];
01397             #ifdef WITHPTHREADS
01398                 if ( AS.MultiThreaded ) {
01399                     for ( nummodopt = 0; nummodopt < NumModOptdollars; nummodopt++ ) {
01400                         if ( -w[l] == ModOptdollars[nummodopt].number ) break;
01401                     }
01402                     if ( nummodopt < NumModOptdollars ) {
01403                         dtype = ModOptdollars[nummodopt].type;
01404                         if ( dtype == MODLOCAL ) {
01405                             d = ModOptdollars[nummodopt].dstruct+AT.identity;
01406                         }
01407                         else {
01408                             LOCK(d->pthreadslackread);
01409                         }
01410                     }
01411                 }
01412             #endif
01413             if ( d->type == DOLNUMBER || d->type == DOLTERMS ) {
01414                 if ( d->where[0] == 4 && d->where[3] == 3 && d->where[2] == 1
01415                     && d->where[1] > 0 && d->where[4] == 0 ) {
01416                     num = d->where[1]; goto GotOne;
01417                 }
01418             }
01419             else if ( d->type == DOLINDEX ) {
01420                 if ( d->index > 0 && d->index < AM.OffsetIndex ) {
01421                     num = d->index; goto GotOne;
01422                 }
01423             }
01424             else if ( d->type == DOLARGUMENT ) {
01425                 if ( d->where[0] == -SNUMBER && d->where[1] > 0 ) {
01426                     num = d->where[1]; goto GotOne;
01427                 }
01428             }
01429             else if ( d->type == DOLWILDARGS ) {
01430                 if ( d->where[0] == 1 &&
01431                     d->where[1] > 0 && d->where[1] < AM.OffsetIndex ) {
01432                     num = d->where[1]; goto GotOne;
01433                 }
01434                 if ( d->where[0] == 0 && d->where[1] < 0 && d->where[3] == 0 ) {
01435                     if ( ( d->where[1] == -SNUMBER && d->where[2] > 0 )
01436                         || ( d->where[1] == -INDEX && d->where[2] > 0
01437                             && d->where[2] < AM.OffsetIndex ) ) {
01438                         num = d->where[2]; goto GotOne;
01439                     }
01440                 }
01441             }
01442             #ifdef WITHPTHREADS
01443                 if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslackread); }
01444             #endif
01445             MLOCK(ErrorMessageLock);
01446             MesPrint("Unusable type of variable %s in set substitution",
01447                 AC.dollarnames->namebuffer+d->name);
01448             MUNLOCK(ErrorMessageLock);
01449             return(-1);
01450         }
01451     GotOne:;
01452     #ifdef WITHPTHREADS
01453         if ( dtype > 0 && dtype != MODLOCAL ) { UNLOCK(d->pthreadslackread); }
01454     #endif
01455     ii = m[*w];
01456     if ( ii >= 2*MAXPOWER ) i3 = ii - 2*MAXPOWER;
01457     else if ( ii <= -2*MAXPOWER ) i3 = -ii - 2*MAXPOWER;
01458     else i3 = ( ii >= 0 ) ? ii: -ii - 1;
01459
01460     if ( num > ( Sets[i3].last - Sets[i3].first ) || num <= 0 ) {
01461         MLOCK(ErrorMessageLock);
01462         MesPrint("Array bound check during set substitution");
01463         MesPrint("    value is %d",num);
01464         MUNLOCK(ErrorMessageLock);
01465         return(-1);
01466     }
01467     m[*w] = (SetElements+Sets[i3].first)[num-1];

```

```

01468         if ( Sets[i3].type == CSYMBOL && m[*w] > MAXPOWER ) {
01469             if ( ii >= 2*MAXPOWER ) m[*w] -= 2*MAXPOWER;
01470             else if ( ii <= -2*MAXPOWER ) m[*w] = -(m[*w] - 2*MAXPOWER);
01471             else {
01472                 m[*w] -= MAXPOWER;
01473                 if ( m[*w] < MAXPOWER ) m[*w] -= 2*MAXPOWER;
01474                 if ( flag ) MakeDirty(m,m+*w,1);
01475             }
01476         }
01477         else if ( Sets[i3].type == CSYMBOL ) {
01478             if ( ii >= 2*MAXPOWER ) m[*w] += 2*MAXPOWER;
01479             else if ( ii <= -2*MAXPOWER ) m[*w] = -m[*w] - 2*MAXPOWER;
01480             else if ( ii < 0 ) m[*w] = - m[*w];
01481         }
01482         else if ( ii < 0 ) m[*w] = - m[*w];
01483         w += 2;
01484     }
01485     m = to;
01486     if ( *m >= FUNCTION && functions[*m-FUNCTION].spec
01487     >= TENSORFUNCTION ) {
01488         w = from + 2 + from[3];
01489         if ( *w == 0 ) { /* We had function -> tensor */
01490             m = from + 2 + FUNHEAD; s = to + FUNHEAD;
01491             while ( m < w ) {
01492                 if ( *m == -INDEX || *m == -VECTOR ) {}
01493                 else if ( *m == -ARGWILD ) { *s++ = FUNNYWILD; }
01494                 else {
01495                     MLOCK(ErrorMessageLock);
01496                     MesPrint("Illegal argument in tensor after set substitution");
01497                     MUNLOCK(ErrorMessageLock);
01498                     SETERROR(-1)
01499                 }
01500                 *s++ = m[1];
01501                 m += 2;
01502             }
01503             to[1] = WORDDIFF(s,to);
01504         }
01505     }
01506     return(0);
01507 }
01508
01509 /*
01510     #] ResolveSet :
01511     #[ ClearWild :          void ClearWild()
01512
01513     Clears the current wildcard settings and makes them ready for
01514     CheckWild and AddWild.
01515
01516 */
01517 void ClearWild(PHEAD0)
01518 {
01519     GETBIDENTITY
01520     WORD n, nn, *w;
01521     n = (AN.WildValue[-SUBEXPSIZE+1]-SUBEXPSIZE)/4; /* Number of wildcards */
01522     AN.NumWild = nn = n;
01523     if ( n > 0 ) {
01524         w = AT.WildMask;
01525         do { *w++ = 0; } while ( --n > 0 );
01526         w = AN.WildValue;
01527         do {
01528             if ( *w == SYMTONUM ) *w = SYMTOSYM;
01529             w += w[1];
01530         } while ( --nn > 0 );
01531     }
01532 }
01533
01534
01535 /*
01536     #] ClearWild :
01537     #[ AddWild :          WORD AddWild(oldnumber,type,newnumber)
01538
01539     Adds a wildcard assignment.
01540     Extra parameter in AN.argaddress;
01541
01542 */
01543 int AddWild(PHEAD WORD oldnumber, WORD type, WORD newnumber)
01544 {
01545     GETBIDENTITY
01546     WORD *w, *m, n, k, i = -1;
01547     CBUF *C = cbuf+AT.ebufnum;
01548     WORD eattensor = type & EATTENSOR;
01549     type = type & ~EATTENSOR;
01550     DEBUG(WORD *mm;)
01551     AN.WildReserve = 0;
01552     m = AT.WildMask;
01553     w = AN.WildValue;

```

```

01555     n = AN.NumWild;
01556     if ( n <= 0 ) { return(-1); }
01557     if ( type <= SYMTOSUB ) {
01558         do {
01559             if ( w[2] == oldnumber && *w <= SYMTOSUB ) {
01560                 if ( n > 1 && w[4] == SETTONUM ) i = w[7];
01561                 *w = type;
01562                 if ( *m != 2 ) *m = 1;
01563                 if ( type != SYMTOSUB ) {
01564                     if ( type == SYMTONUM ) AN.MaskPointer = m;
01565                     w[3] = newnumber;
01566                     goto FlipOn;
01567                 }
01568                 m = AddrRHS(AT.ebufnum,1);
01569                 w[3] = C->numrhs;
01570                 w = AN.argaddress;
01571             DEBUG(mm = m;);
01572             n = *w - ARGHEAD;
01573             w += ARGHEAD;
01574             while ( (m + n + 10) > C->Top ) m = DoubleCbuffer(AT.ebufnum,m,4);
01575             while ( --n >= 0 ) *m++ = *w++;
01576             *m++ = 0;
01577             C->rhs[C->numrhs+1] = m;
01578             DEBUG(MesPrint("Thread %w(d): m=(%d,%d,%d,%d) (%d)",mm[0],mm[1],mm[2],mm[3],C->numrhs);)
01579             C->Pointer = m;
01580             goto FlipOn;
01581         }
01582         m++; w += w[1];
01583     } while ( --n > 0 );
01584 }
01585 else if ( type == ARGTOARG ) {
01586     do {
01587         if ( w[2] == oldnumber && *w == ARGTOARG ) {
01588             *m = 1;
01589             m = AddrRHS(AT.ebufnum,1);
01590             w[3] = C->numrhs;
01591             w = AN.argaddress;
01592             DEBUG(mm=m;);
01593             if ( eattensor ) {
01594                 n = newnumber;
01595                 *m++ = n;
01596                 w = AN.argaddress;
01597             }
01598             else {
01599                 while ( --newnumber >= 0 ) { NEXTARG(w) }
01600                 n = WORDDIF(w,AN.argaddress);
01601                 w = AN.argaddress;
01602                 *m++ = 0;
01603             }
01604             while ( (m + n + 10) > C->Top ) m = DoubleCbuffer(AT.ebufnum,m,5);
01605             DEBUG(if ( mm != m-1 ) MesPrint("Thread %w(e): Alarm!"); mm = m-1;);
01606             while ( --n >= 0 ) *m++ = *w++;
01607             *m++ = 0;
01608             C->rhs[C->numrhs+1] = m;
01609             C->Pointer = m;
01610             DEBUG(MesPrint("Thread %w(e): w=(%d,%d,%d,%d) (%d)",mm[0],mm[1],mm[2],mm[3],C->numrhs);)
01611             return(0);
01612         }
01613         m++; w += w[1];
01614     } while ( --n > 0 );
01615 }
01616 else if ( type == ARLTOARL ) {
01617     do {
01618         if ( w[2] == oldnumber && *w == ARGTOARG ) {
01619             WORD **a;
01620             *m = 1;
01621             m = AddrRHS(AT.ebufnum,1);
01622             w[3] = C->numrhs;
01623             DEBUG(mm=m;);
01624             a = (WORD **)(AN.argaddress); n = 0; k = newnumber;
01625             while ( --newnumber >= 0 ) {
01626                 w = *a++;
01627                 if ( *w > 0 ) n += *w;
01628                 else if ( *w <= -FUNCTION ) n++;
01629                 else n += 2;
01630             }
01631             *m++ = 0;
01632             while ( (m + n + 10) > C->Top ) m = DoubleCbuffer(AT.ebufnum,m,6);
01633             DEBUG(if ( mm != m-1 ) MesPrint("Thread %w(f): Alarm!"); mm = m-1;);
01634             a = (WORD **)(AN.argaddress);
01635             while ( --k >= 0 ) {
01636                 w = *a++;
01637                 if ( *w > 0 ) { n = *w; NCOPY(m,w,n); }
01638                 else if ( *w <= -FUNCTION ) *m++ = *w++;
01639                 else { *m++ = *w++; *m++ = *w++; }
01640             }
01641             *m++ = 0;

```

```

01642             C->rhs[C->numrhs+1] = m;
01643 DEBUG(MesPrint("Thread %w(f): w=(%d,%d,%d,%d) (%d) ",mm[0],mm[1],mm[2],mm[3],C->numrhs);)
01644             C->Pointer = m;
01645             return(0);
01646         }
01647         m++; w += w[1];
01648     } while ( --n > 0 );
01649 }
01650 else if ( type == VECTOSUB || type == INDTOISUB ) {
01651     WORD *ss, *sstop, *tt, *ttstop, j, *v1, *v2 = 0;
01652     do {
01653         if ( w[2] == oldnumber && ( *w == type ||
01654             ( type == VECTOSUB && ( *w == VECTOVEC || *w == VECTOMIN ) )
01655             || ( type == INDTOISUB && *w == INDTOIND ) ) ) {
01656             if ( n > 1 && w[4] == SETTONUM ) i = w[7];
01657             *w = type;
01658             *m = 1;
01659             m = AddRHS(AT.ebufnum,1);
01660             w[3] = C->numrhs;
01661             w = AN.argaddress;
01662             n = *w - ARGHEAD;
01663             w += ARGHEAD;
01664             while ( (m + n + 10) > C->Top ) m = DoubleCbuffer(AT.ebufnum,m,7);
01665             while ( --n >= 0 ) *m++ = *w++;
01666             *m++ = 0;
01667             C->rhs[C->numrhs+1] = m;
01668             C->Pointer = m;
01669             m = AddRHS(AT.ebufnum,1);
01670             w = AN.argaddress;
01671             n = *w - ARGHEAD;
01672             w += ARGHEAD;
01673             while ( (m + n + 10) > C->Top ) m = DoubleCbuffer(AT.ebufnum,m,8);
01674             sstop = w + n;
01675             while ( w < sstop ) { /* Run over terms */
01676                 tt = w + *w; ttstop = tt - ABS(tt[-1]);
01677                 ss = m; m++; w++;
01678                 while ( w < ttstop ) { /* Subterms */
01679                     if ( *w != INDEX ) {
01680                         j = w[1];
01681                         NCOPY(m,w,j);
01682                     }
01683                     else {
01684                         v1 = m;
01685                         *m++ = *w++;
01686                         *m++ = j = *w++;
01687                         j -= 2;
01688                         while ( --j >= 0 ) {
01689                             if ( *w >= MINSPEC ) *m++ = *w++;
01690                             else v2 = w++;
01691                         }
01692                         j = WORDDIFF(m,v1);
01693                         if ( j != v1[1] ) {
01694                             if ( j <= 2 ) m -= 2;
01695                             else v1[1] = j;
01696                             *m++ = VECTOR;
01697                             *m++ = 4;
01698                             *m++ = *v2;
01699                             *m++ = FUNNYVEC;
01700                         }
01701                     }
01702                 }
01703                 while ( w < tt ) *m++ = *w++;
01704                 *ss = WORDDIFF(m,ss);
01705             }
01706             *m++ = 0;
01707             C->rhs[C->numrhs+1] = m;
01708             C->Pointer = m;
01709             if ( m > C->Top ) {
01710                 MLOCK(ErrorMessageLock);
01711                 MesPrint("Internal problems with extra compiler buffer");
01712                 MUNLOCK(ErrorMessageLock);
01713                 Terminate(-1);
01714             }
01715             goto FlipOn;
01716         }
01717         m++; w += w[1];
01718     } while ( --n > 0 );
01719 }
01720 else {
01721     do {
01722         if ( w[2] == oldnumber && ( *w == type || ( type == VECTOVEC
01723             && ( *w == VECTOMIN || *w == VECTOSUB ) ) || ( type == VECTOMIN
01724             && ( *w == VECTOVEC || *w == VECTOSUB ) )
01725             || ( type == INDTOIND && *w == INDTOISUB ) ) ) {
01726             if ( n > 1 && w[4] == SETTONUM ) i = w[7];
01727             *w = type;
01728             w[3] = newnumber;

```



```

01729             *m = 1;
01730             goto FlipOn;
01731         }
01732         m++; w += w[1];
01733     } while ( --n > 0 );
01734 }
01735 MLOCK(ErrorMessageLock);
01736 MesPrint("Bug in AddWild.");
01737 MUNLOCK(ErrorMessageLock);
01738 return(-1);
01739 FlipOn:
01740 if ( i >= 0 ) {
01741     m = AT.WildMask;
01742     w = AN.WildValue;
01743     n = AN.NumWild;
01744     while ( --n >= 0 ) {
01745         if ( w[2] == i && *w == SYMTONUM ) {
01746             *m = 2;
01747             return(0);
01748         }
01749         m++; w += w[1];
01750     }
01751     MLOCK(ErrorMessageLock);
01752     MesPrint(" Bug in AddWild with passing set[i]");
01753     MUNLOCK(ErrorMessageLock);
01754     /*
01755      For the moment we want to crash here. That is easier with debugging.
01756     */
01757     #ifdef WITHPTHREADS
01758     { WORD *s = 0;
01759       *s++ = 1;
01760     }
01761     #endif
01762     Terminate(-1);
01763 }
01764 return(0);
01765 }
01766
01767 /*
01768 #] AddWild :
01769 #[ CheckWild :      WORD CheckWild(oldnumber,type,newnumber,newval)
01770
01771 Tests whether a wildcard assignment is allowed.
01772 A return value of zero means that it is allowed (nihil obstat).
01773 If the variable has been assigned already its existing
01774 assignment is returned in AN.oldvalue and AN.oldtype, which are
01775 global variables.
01776
01777 Note the special problem with name?set[i]. Here we have to pass
01778 an extra assignment. This cannot be done via globals as we
01779 call CheckWild sometimes twice before calling AddWild.
01780 Trick: Check the assignment of the number and if OK put it
01781 in place, but don't alter the used flag (if needed).
01782 Then AddWild can alter the used flag but the value is there.
01783 As long as this trick is 'hanging' we turn on the flag:
01784 'AN.WildReserve' which is either turned off by AddWild or by
01785 a failing call to CheckWild.
01786
01787 With ARGTOARG the tensors give the number of arguments
01788 or-ed with EATTENSOR which is at least 8192.
01789 */
01790
01791 int CheckWild(PHEAD WORD oldnumber, WORD type, WORD newnumber, WORD *newval)
01792 {
01793     GETBIDENTITY
01794     WORD *w, *m, *s, n, old2, inset;
01795     WORD n2, oldval, dirty, i, j;
01796     int notflag = 0, retblock = 0;
01797     CBUF *C = cbuf+AT.ebufnum;
01798     WORD eattensor = type & EATTENSOR;
01799     type = type & ~EATTENSOR;
01800     m = AT.WildMask;
01801     w = AN.WildValue;
01802     n = AN.NumWild;
01803     if ( n <= 0 ) { AN.oldtype = -1; AN.WildReserve = 0; return(-1); }
01804     switch ( type ) {
01805     case SYMTONUM :
01806         *newval = newnumber;
01807         do {
01808             if ( w[2] == oldnumber && *w <= SYMTOSUB ) {
01809                 old2 = *w;
01810                 if ( !*m ) goto TestSet;
01811                 AN.MaskPointer = m;
01812                 if ( *w == SYMTONUM && w[3] == newnumber ) {
01813                     return(0);
01814                 }
01815                 AN.oldtype = old2; AN.oldvalue = w[3]; goto NoMatch;

```

```

01816         }
01817         m++; w += w[1];
01818     } while ( --n > 0 );
01819     break;
01820 case SYMTOSYM :
01821     *newval = newnumber;
01822     do {
01823         if ( w[2] == oldnumber && *w <= SYMTOSUB ) {
01824             old2 = *w;
01825             if ( *w == SYMTOSYM ) {
01826                 if ( !*m ) goto TestSet;
01827                 if ( newnumber >= 0 && (w+4) < AN.WildStop
01828                     && ( w[4] == FROMSET || w[4] == SETTONUM )
01829                     && w[7] >= 0 ) goto TestSet;
01830                 if ( w[3] == newnumber ) return(0);
01831             }
01832             else {
01833                 if ( !*m ) goto TestSet;
01834             }
01835             goto NoM;
01836         }
01837         m++; w += w[1];
01838     } while ( --n > 0 );
01839     break;
01840 case SYMTOSUB :
01841     /*
01842     Now newval contains the pointer to the argument.
01843     */
01844     {
01845     /*
01846     Search for vector or index nature. If so: reject.
01847     */
01848     WORD *ss, *sstop, *tt, *ttstop;
01849     ss = newval;
01850     sstop = ss + *ss;
01851     ss += ARGHEAD;
01852     while ( ss < sstop ) {
01853         tt = ss + *ss;
01854         ttstop = tt - ABS(tt[-1]);
01855         ss++;
01856         while ( ss < ttstop ) {
01857             if ( *ss == INDEX ) goto NoMatch;
01858             ss += ss[1];
01859         }
01860         ss = tt;
01861     }
01862 }
01863 do {
01864     if ( w[2] == oldnumber && *w <= SYMTOSUB ) {
01865         old2 = *w;
01866         if ( *w == SYMTONUM || *w == SYMTOSYM ) {
01867             if ( !*m ) {
01868                 s = w + w[1];
01869                 if ( s >= AN.WildStop || *s != SETTONUM )
01870                     goto TestSet;
01871             }
01872         }
01873         else if ( *w == SYMTOSUB ) {
01874             if ( !*m ) {
01875                 s = w + w[1];
01876                 if ( s >= AN.WildStop || *s != SETTONUM )
01877                     goto TestSet;
01878             }
01879             n = *newval - 2;
01880             newval += 2;
01881             m = C->rhs[w[3]];
01882             if ( (C->rhs[w[3]+1] - m - 1) == n ) {
01883                 while ( n > 0 ) {
01884                     if ( *m != *newval ) {
01885                         m++; newval++; break;
01886                     }
01887                     m++; newval++;
01888                     n--;
01889                 }
01890                 if ( n <= 0 ) return(0);
01891             }
01892         }
01893         AN.oldtype = old2; AN.oldvalue = w[3]; goto NoMatch;
01894     }
01895     m++; w += w[1];
01896 } while ( --n > 0 );
01897 break;
01898 case ARGTOARG :
01899     do {
01900         if ( w[2] == oldnumber && *w == ARGTOARG ) {
01901             if ( !*m ) return(0); /* nihil obstat */
01902             m = C->rhs[w[3]];

```

```

01903         if ( eattensor ) {
01904             n = newnumber;
01905             if ( *m != 0 ) {
01906                 if ( n == *m ) {
01907                     m++;
01908                     while ( --n >= 0 ) {
01909                         if ( *m != *newval ) {
01910                             m++; newval++; break;
01911                         }
01912                         m++; newval++;
01913                     }
01914                     if ( n < 0 ) return(0);
01915                 }
01916             }
01917             else {
01918                 m++;
01919                 while ( --n >= 0 ) {
01920                     if ( *newval != m[1] || ( *m != -INDEX
01921                         && *m != -VECTOR && *m != -SNUMBER ) ) break;
01922                     m += 2;
01923                     newval++;
01924                 }
01925                 if ( n < 0 && *m == 0 ) return(0);
01926             }
01927         }
01928         else {
01929             i = newnumber;
01930             if ( *m != 0 ) { /* Tensor field */
01931                 if ( *m == i ) {
01932                     m++;
01933                     while ( --i >= 0 ) {
01934                         if ( *m != newval[1]
01935                             || ( *newval != -VECTOR
01936                                 && *newval != -INDEX
01937                                 && *newval != -SNUMBER ) ) break;
01938                         newval += 2;
01939                         m++;
01940                     }
01941                     if ( i < 0 ) return(0);
01942                 }
01943             }
01944             else {
01945                 m++;
01946                 s = newval;
01947                 while ( --i >= 0 ) { NEXTARG(s) }
01948                 n = WORDDIF(s,newval);
01949                 while ( --n >= 0 ) {
01950                     if ( *m != *newval ) {
01951                         m++; newval++; break;
01952                     }
01953                     m++; newval++;
01954                 }
01955                 if ( n < 0 && *m == 0 ) return(0);
01956             }
01957         }
01958         AN.oldtype = *w; AN.oldvalue = w[3]; goto NoMatch;
01959     }
01960     m++; w += w[1];
01961 } while ( --n > 0 );
01962 break;
01963 case ARLTOARL :
01964     do {
01965         if ( w[2] == oldnumber && *w == ARGTOARG ) {
01966             WORD **a;
01967             if ( !*m ) return(0); /* nihil obstat */
01968             m = C->rhs[w[3]];
01969             i = newnumber;
01970             a = (WORD **)newval;
01971             if ( *m != 0 ) { /* Tensor field */
01972                 if ( *m == i ) {
01973                     m++;
01974                     while ( --i >= 0 ) {
01975                         s = *a++;
01976                         if ( *m != s[1]
01977                             || ( *s != -VECTOR
01978                                 && *s != -INDEX
01979                                 && *s != -SNUMBER ) ) break;
01980                         m++;
01981                     }
01982                     if ( i < 0 ) return(0);
01983                 }
01984             }
01985             else {
01986                 m++;
01987                 while ( --i >= 0 ) {
01988                     s = *a++;
01989                     if ( *s > 0 ) {

```

```

01990             n = *s;
01991             while ( --n >= 0 ) {
01992                 if ( *s != *m ) {
01993                     s++; m++; break;
01994                 }
01995                 s++; m++;
01996             }
01997             if ( n >= 0 ) break;
01998         }
01999         else if ( *s <= -FUNCTION ) {
02000             if ( *s != *m ) {
02001                 s++; m++; break;
02002             }
02003             s++; m++;
02004         }
02005         else {
02006             if ( *s != *m ) {
02007                 s++; m++; break;
02008             }
02009             s++; m++;
02010             if ( *s != *m ) {
02011                 s++; m++; break;
02012             }
02013             s++; m++;
02014         }
02015     }
02016     if ( i < 0 && *m == 0 ) return(0);
02017 }
02018 AN.oldtype = *w; AN.oldvalue = w[3]; goto NoMatch;
02019 }
02020 m++; w += w[1];
02021 } while ( --n > 0 );
02022 break;
02023 case VECTOSUB :
02024 case INDTOSUB :
02025 /*
02026     Now newval contains the pointer to the argument(s).
02027 */
02028 {
02029 /*
02030     Search for vector or index nature. If not so: reject.
02031 */
02032     WORD *ss, *sstop, *tt, *ttstop, count, jt;
02033     ss = newval;
02034     sstop = ss + *ss;
02035     ss += ARGHEAD;
02036     while ( ss < sstop ) {
02037         tt = ss + *ss;
02038         ttstop = tt - ABS(tt[-1]);
02039         ss++;
02040         count = 0;
02041         while ( ss < ttstop ) {
02042             if ( *ss == INDEX ) {
02043                 jt = ss[1] - 2; ss += 2;
02044                 while ( --jt >= 0 ) {
02045                     if ( *ss < MINSPEC ) count++;
02046                     ss++;
02047                 }
02048             }
02049             else ss += ss[1];
02050         }
02051         if ( count != 1 ) goto NoMatch;
02052         ss = tt;
02053     }
02054 }
02055 do {
02056     if ( w[2] == oldnumber ) {
02057         old2 = *w;
02058         if ( ( type == VECTOSUB && ( *w == VECTOVEC || *w == VECTOMIN ) )
02059             || ( type == INDTOSUB && *w == INDTOIND ) ) {
02060             if ( !*m ) goto TestSet;
02061             AN.oldtype = old2; AN.oldvalue = w[3]; goto NoMatch;
02062         }
02063         else if ( *w == type ) {
02064             if ( !*m ) goto TestSet;
02065             if ( type != INDTOIND && type != INDTOSUB ) { /* Prevent double index */
02066                 n = *newval - 2;
02067                 newval += 2;
02068                 m = C->rhs[w[3]];
02069                 if ( (C->rhs[w[3]+1] - m - 1) == n ) {
02070                     while ( n > 0 ) {
02071                         if ( *m != *newval ) {
02072                             m++; newval++; break;
02073                         }
02074                         m++; newval++;
02075                         n--;
02076                     }

```

```

02077             if ( n <= 0 ) return(0);
02078         }
02079     }
02080     AN.oldtype = old2; AN.oldvalue = w[3]; goto NoMatch;
02081 }
02082 }
02083     m++; w += w[1];
02084 } while ( --n > 0 );
02085 break;
02086 default :
02087     *newval = newnumber;
02088     do {
02089         if ( w[2] == oldnumber ) {
02090             if ( *w == type ) {
02091                 old2 = *w;
02092                 if ( !*m ) goto TestSet;
02093                 if ( newnumber >= 0 && (w+4) < AN.WildStop &&
02094                     ( w[4] == FROMSET || w[4] == SETTONUM )
02095                     && w[7] >= 0 ) goto TestSet;
02096                 if ( newnumber < 0 && *w == VECTOVEC
02097                     && (w+4) < AN.WildStop && ( w[4] == FROMSET
02098                     || w[4] == SETTONUM ) && w[7] >= 0 ) goto TestSet;
02099             /*
02100              The next statement kills multiple indices -> vector
02101             */
02102             if ( *w == INDTOIND && w[3] < 0 ) goto NoMatch;
02103             if ( w[3] == newnumber ) {
02104                 if ( *w != FUNTOFUN || newnumber < FUNCTION
02105                     || functions[newnumber-FUNCTION].spec ==
02106                     functions[oldnumber-FUNCTION].spec )
02107                     return(0);
02108             }
02109             AN.oldtype = old2; AN.oldvalue = w[3]; goto NoMatch;
02110         }
02111         else if ( ( type == VECTOVEC &&
02112             ( *w == VECTOSUB || *w == VECTOMIN ) )
02113             || ( type == INDTOIND && *w == INDTOSUB ) ) {
02114             if ( *m ) goto NoMatch;
02115             old2 = *w;
02116             goto TestSet;
02117         }
02118         else if ( type == VECTOMIN &&
02119             ( *w == VECTOSUB || *w == VECTOVEC ) ) {
02120             if ( *m ) goto NoMatch;
02121             old2 = *w;
02122             goto TestSet;
02123         }
02124     }
02125     m++; w += w[1];
02126     if ( n > 1 && ( *w == FROMSET
02127         || *w == SETTONUM ) ) { n--; m++; w += w[1]; }
02128 } while ( --n > 0 );
02129 break;
02130 }
02131 AN.oldtype = -1;
02132 AN.oldvalue = -1;
02133 AN.WildReserve = 0;
02134 MLOCK(ErrorMessageLock);
02135 MesPrint("Inconsistency in Wildcard prototype.");
02136 MUNLOCK(ErrorMessageLock);
02137 return(-1);
02138 NoMatch:
02139 AN.WildReserve = 0;
02140 return(1+retblock);
02141 /*
02142 Here we test the compatibility with a set specification.
02143 */
02144 TestSet:
02145     dirty = *m;
02146     oldval = w[3];
02147     w += w[1];
02148     if ( w < AN.WildStop && ( *w == FROMSET || *w == SETTONUM ) ) {
02149         WORD k;
02150         s = w;
02151         j = w[2]; n2 = w[3];
02152     /*
02153         if SETTONUM:  x?j[n2]
02154         if FROMSET:   x?j?n2    or x?j and n2 = -WOLDOFFSET.
02155     */
02156     if ( j > WILDOFFSET ) {
02157         j -= 2*WILDOFFSET;
02158         notflag = 1;
02159     /*
02160         ??????
02161     */
02162     AN.oldtype = -1;
02163     AN.oldvalue = -1;

```

```

02164     }
02165     if ( j < AM.NumFixedSets ) {      /* special set */
02166         retblock = 1;
02167         switch ( j ) {
02168             case POS_:
02169                 if ( type != SYMTONUM ||
02170                     newnumber <= 0 ) goto NoMnot;
02171                 break;
02172             case POS0_:
02173                 if ( type != SYMTONUM ||
02174                     newnumber < 0 ) goto NoMnot;
02175                 break;
02176             case NEG_:
02177                 if ( type != SYMTONUM ||
02178                     newnumber >= 0 ) goto NoMnot;
02179                 break;
02180             case NEG0_:
02181                 if ( type != SYMTONUM ||
02182                     newnumber > 0 ) goto NoMnot;
02183                 break;
02184             case EVEN_:
02185                 if ( type != SYMTONUM ||
02186                     ( newnumber & 1 ) != 0 ) goto NoMnot;
02187                 break;
02188             case ODD_:
02189                 if ( type != SYMTONUM ||
02190                     ( newnumber & 1 ) == 0 ) goto NoMnot;
02191                 break;
02192             case Z_:
02193                 if ( type != SYMTONUM ) goto NoMnot;
02194                 break;
02195             case SYMBOL_:
02196                 if ( type != SYMTOSYM ) goto NoMnot;
02197                 break;
02198             case FIXED_:
02199                 if ( type != INDTOIND ||
02200                     newnumber >= AM.OffsetIndex ||
02201                     newnumber < 0 ) goto NoMnot;
02202                 break;
02203             case INDEX_:
02204                 if ( type != INDTOIND ||
02205                     newnumber < 0 ) goto NoMnot;
02206                 break;
02207             case Q_:
02208                 if ( type == SYMTONUM ) break;
02209                 if ( type == SYMTOSUB ) {
02210                     WORD *ss, *sstop;
02211                     ss = newval;
02212                     sstop = ss + *ss;
02213                     ss += ARGHEAD;
02214                     if ( ss >= sstop ) break;
02215                     if ( ss + *ss < sstop ) goto NoMnot;
02216                     if ( ABS(sstop[-1]) == ss[0]-1 ) break;
02217                 }
02218                 goto NoMnot;
02219             case DUMMYINDEX_:
02220                 if ( type != INDTOIND ||
02221                     newnumber < AM.IndDum || newnumber >= AM.IndDum+MAXDUMMIES ) goto NoMnot;
02222                 break;
02223             case VECTOR_:
02224                 if ( type != VECTOVEC ) goto NoMnot;
02225                 break;
02226             default:
02227                 goto NoMnot;
02228         }
02229     Mnot:
02230         if ( notflag ) goto NoM;
02231         return(0);
02232     NoMnot:
02233         if ( !notflag ) goto NoM;
02234         return(0);
02235     }
02236     else if ( Sets[j].type == CRANGE ) {
02237         if ( ( type == SYMTONUM )
02238             || ( type == INDTOIND && ( newnumber > 0
02239                 && newnumber <= AM.OffsetIndex ) ) ) {
02240             if ( Sets[j].first < MAXPOWER ) {
02241                 if ( newnumber >= Sets[j].first ) goto NoMnot;
02242             }
02243             else if ( Sets[j].first < 3*MAXPOWER ) {
02244                 if ( newnumber+2*MAXPOWER > Sets[j].first ) goto NoMnot;
02245             }
02246             if ( Sets[j].last > -MAXPOWER ) {
02247                 if ( newnumber <= Sets[j].last ) goto NoMnot;
02248             }
02249             else if ( Sets[j].last > -3*MAXPOWER ) {
02250                 if ( newnumber-2*MAXPOWER < Sets[j].last ) goto NoMnot;

```

```

02251         }
02252         goto Mnot;
02253     }
02254     goto NoMnot;
02255 }
02256 /*
02257     Now we have to determine which set element
02258 */
02259 w = SetElements + Sets[j].first;
02260 m = SetElements + Sets[j].last;
02261 if ( w == m ) {
02262 /*
02263     The set is empty! This is not a match unless we have !{}, in
02264     which case it is.
02265 */
02266     if ( notflag ) return 0;
02267     AN.oldtype = old2;
02268     AN.oldvalue = oldval;
02269     goto NoMatch;
02270 }
02271
02272 if ( ( Sets[j].flags & ORDEREDSET ) == ORDEREDSET ) {
02273 /*
02274     We search first and ask questions later
02275 */
02276     i = BinarySearch(w, Sets[j].last - Sets[j].first, newnumber);
02277     if ( i < 0 ) { /* no matter what, it is not in the set. */
02278         goto NoMnot;
02279     }
02280     else {
02281 /*
02282         We can set the proper parameters now to make only the
02283         checks for the given set element.
02284         After that we jump into the appropriate loop.
02285 */
02286         w = m = SetElements + i;
02287         i++;
02288         if ( Sets[j].type == -1 || Sets[j].type == CNUMBER ) {
02289             goto insideloop1;
02290         }
02291         else {
02292             goto insideloop2;
02293         }
02294     }
02295 }
02296 i = 1;
02297 if ( Sets[j].type == -1 || Sets[j].type == CNUMBER ) {
02298     do {
02299         insideloop1:
02300         if ( notflag ) {
02301             switch ( type ) {
02302                 case SYMTOSYM:
02303                     if ( Sets[j].type == CNUMBER ) {}
02304                     else {
02305                         if ( *w == newnumber ) goto NoMatch;
02306                     }
02307                     break;
02308                 case SYMTONUM:
02309                 case INDTOIND:
02310                     if ( *w == newnumber ) goto NoMatch;
02311                     break;
02312                 default:
02313                     break;
02314             }
02315         }
02316         else if ( type != SYMTONUM && type != INDTOIND
02317             && type != SYMTOSYM ) goto NoMatch;
02318         else if ( type == SYMTOSYM && Sets[j].type == CNUMBER ) goto NoMatch;
02319         else if ( *w == newnumber ) {
02320             if ( *s == SETTONUM ) {
02321                 if ( n2 == oldnumber && type
02322                     <= SYMTOSUB ) goto NoMatch;
02323                 m = AT.WildMask;
02324                 w = AN.WildValue;
02325                 n = AN.NumWild;
02326                 while ( --n >= 0 ) {
02327                     if ( w[2] == n2 && *w <= SYMTOSUB ) {
02328                         if ( !*m ) {
02329                             *w = SYMTONUM;
02330                             w[3] = i;
02331                             AN.WildReserve = 1;
02332                             return(0);
02333                         }
02334                         if ( *w != SYMTONUM )
02335                             goto NoMatch;
02336                         if ( w[3] == i ) return(0);
02337                         i = w[3];

```

```

02338             j = (SetElements + Sets[j].first)[i];
02339             if ( j == n2 ) return(0);
02340             goto NoMatch;
02341         }
02342         m++; w += w[1];
02343     }
02344 }
02345 else if ( n2 >= 0 ) {
02346     *newval = *(w - Sets[j].first + Sets[n2].first);
02347     if ( *newval > MAXPOWER ) *newval -= 2*MAXPOWER;
02348     if ( dirty && *newval != oldval ) {
02349         *newval = oldval; goto NoMatch;
02350     }
02351 }
02352 return(0);
02353 }
02354 i++;
02355 } while ( ++w < m );
02356 }
02357 else {
02358     do {
02359         insideloop2:
02360         inset = *w;
02361         if ( notflag ) {
02362             switch ( type ) {
02363                 case SYMTONUM:
02364                 case SYMTOSYM:
02365                     if ( ( type == SYMTOSYM && *w == newnumber )
02366                         || ( type == SYMTONUM && *w-2*MAXPOWER == newnumber ) ) {
02367                         goto NoMatch;
02368                     }
02369                     /* fall through */
02370                 case SYMTOSUB:
02371                     if ( *w < 0 ) {
02372                         WORD *mm = AT.WildMask, *mmm, *part;
02373                         WORD *ww = AN.WildValue;
02374                         WORD nn = AN.NumWild;
02375                         k = -*w;
02376                         while ( --nn >= 0 ) {
02377                             if ( *mm && ww[2] == k && ww[0] == type ) {
02378                                 if ( type != SYMTOSUB ) {
02379                                     if ( ww[3] == newnumber ) goto NoMatch;
02380                                 }
02381                                 else {
02382                                     mmm = C->rhs[ww[3]];
02383                                     nn = *newval-2;
02384                                     part = newval+2;
02385                                     if ( (C->rhs[ww[3]+1]-mmm-1) == nn ) {
02386                                         while ( --nn >= 0 ) {
02387                                             if ( *mmm != *part ) {
02388                                                 mmm++; part++; break;
02389                                             }
02390                                             mmm++; part++;
02391                                         }
02392                                         if ( nn < 0 ) goto NoMatch;
02393                                     }
02394                                 }
02395                                 break;
02396                             }
02397                             mm++; ww += ww[1];
02398                         }
02399                     }
02400                     break;
02401                 case VECTOMIN:
02402                     if ( type == VECTOMIN ) {
02403                         if ( inset >= AM.OffsetVector ) { i++; continue; }
02404                         inset += WILDMASK;
02405                     }
02406                     /* fall through */
02407                 case VECTOVEC:
02408                     if ( inset == newnumber ) goto NoMatch;
02409                     /* fall through */
02410                 case VECTOSUB:
02411                     if ( inset - WILDOFFSET >= AM.OffsetVector ) {
02412                         WORD *mm = AT.WildMask, *mmm, *part;
02413                         WORD *ww = AN.WildValue;
02414                         WORD nn = AN.NumWild;
02415                         k = inset - WILDOFFSET;
02416                         while ( --nn >= 0 ) {
02417                             if ( *mm && ww[2] == k && ww[0] == type ) {
02418                                 if ( type == VECTOVEC ) {
02419                                     if ( ww[3] == newnumber ) goto NoMatch;
02420                                 }
02421                                 else {
02422                                     mmm = C->rhs[ww[3]];
02423                                     nn = *newval-2;
02424                                     part = newval+2;

```



```

02425         if ( (C->rhs[ww[3]+1]-mmm-1) == nn ) {
02426             while ( --nn >= 0 ) {
02427                 if ( *mmm != *part ) {
02428                     mmm++; part++; break;
02429                 }
02430                 mmm++; part++;
02431             }
02432             if ( nn < 0 ) goto NoMatch;
02433         }
02434     }
02435     break;
02436 }
02437 mm++; ww += ww[1];
02438 }
02439 }
02440 break;
02441 case INDTOIND:
02442     if ( *w == newnumber ) goto NoMatch;
02443     /* fall through */
02444 case INTOSUB:
02445     if ( *w - (WORD)WILDMASK >= AM.OffsetIndex ) {
02446         WORD *mm = AT.WildMask, *mmm, *part;
02447         WORD *ww = AN.WildValue;
02448         WORD nn = AN.NumWild;
02449         k = *w - WILDMASK;
02450         while ( --nn >= 0 ) {
02451             if ( *mm && ww[2] == k && ww[0] == type ) {
02452                 if ( type == INDTOIND ) {
02453                     if ( ww[3] == newnumber ) goto NoMatch;
02454                 }
02455                 else {
02456                     mmm = C->rhs[ww[3]];
02457                     nn = *newval-2;
02458                     part = newval+2;
02459                     if ( (C->rhs[ww[3]+1]-mmm-1) == nn ) {
02460                         while ( --nn >= 0 ) {
02461                             if ( *mmm != *part ) {
02462                                 mmm++; part++; break;
02463                             }
02464                             mmm++; part++;
02465                         }
02466                         if ( nn < 0 ) goto NoMatch;
02467                     }
02468                 }
02469                 break;
02470             }
02471             mm++; ww += ww[1];
02472         }
02473     }
02474     break;
02475 case FUNTOFUN:
02476     if ( *w == newnumber ) goto NoMatch;
02477     if ( ( type == FUNTOFUN &&
02478           ( k = *w - WILDMASK ) > FUNCTION ) ) {
02479         WORD *mm = AT.WildMask;
02480         WORD *ww = AN.WildValue;
02481         WORD nn = AN.NumWild;
02482         while ( --nn >= 0 ) {
02483             if ( *mm && ww[2] == k && ww[0] == type ) {
02484                 if ( ww[3] == newnumber ) goto NoMatch;
02485                 break;
02486             }
02487             mm++; ww += ww[1];
02488         }
02489     }
02490 default:
02491     break;
02492 }
02493 }
02494 else {
02495     if ( type == VECTOMIN ) {
02496         if ( inset >= AM.OffsetVector ) { i++; continue; }
02497         inset += WILDMASK;
02498     }
02499     if ( ( inset == newnumber && type != SYMTONUM ) ||
02500         ( type == SYMTONUM && inset-2*MAXPOWER == newnumber ) ) {
02501         if ( *s == SETTONUM ) {
02502             if ( n2 == oldnumber && type
02503                 <= SYMTOSUB ) goto NoMatch;
02504             m = AT.WildMask;
02505             w = AN.WildValue;
02506             n = AN.NumWild;
02507             while ( --n >= 0 ) {
02508                 if ( w[2] == n2 && *w <= SYMTOSUB ) {
02509                     if ( !*m ) {
02510                         *w = SYMTONUM;
02511                         w[3] = i;

```

```

02512             AN.WildReserve = 1;
02513             return(0);
02514         }
02515         if ( *w != SYMTONUM )
02516             goto NoMatch;
02517         if ( w[3] == i ) return(0);
02518         i = w[3];
02519         j = (SetElements + Sets[j].first)[i];
02520         if ( j == n2 ) return(0);
02521         goto NoMatch;
02522     }
02523     m++; w += w[1];
02524 }
02525 }
02526 else if ( n2 >= 0 ) {
02527     *newval = *(w - Sets[j].first + Sets[n2].first);
02528     if ( *newval > MAXPOWER ) *newval -= 2*MAXPOWER;
02529     if ( dirty && *newval != oldval ) {
02530         *newval = oldval; goto NoMatch;
02531     }
02532 }
02533 return(0);
02534 }
02535 }
02536 i++;
02537 } while ( ++w < m );
02538 }
02539 if ( notflag ) return(0);
02540 AN.oldtype = old2; AN.oldvalue = oldval; goto NoMatch;
02541 }
02542 else { return(0); }
02543
02544 NoM:
02545 AN.oldtype = old2; AN.oldvalue = w[3]; goto NoMatch;
02546 }
02547
02548 /*
02549     #] CheckWild :
02550     #] Wildcards :
02551     #[ DenToFunction :
02552
02553     Renames the denominator function into a function with the given number.
02554     For the syntax see   Denominators,function;
02555 */
02556
02557 int DenToFunction(WORD *term, WORD numfun)
02558 {
02559     int action = 0;
02560     WORD *t, *tstop, *tnext, *arg, *argstop, *targ;
02561     t = term+1;
02562     tstop = term + *term; tstop -= ABS(tstop[-1]);
02563     while ( t < tstop ) {
02564         if ( *t == DENOMINATOR ) {
02565             *t = numfun; t[2] |= DIRTYFLAG; action = 1;
02566         }
02567         tnext = t + t[1];
02568         if ( *t >= FUNCTION && functions[*t-FUNCTION].spec <= 0 ) {
02569             arg = t + FUNHEAD;
02570             while ( arg < tnext ) {
02571                 if ( *arg > 0 ) {
02572                     targ = arg + ARGHEAD; argstop = arg + *arg;
02573                     while ( targ < argstop ) {
02574                         if ( DenToFunction(targ,numfun) ) {
02575                             arg[1] |= DIRTYFLAG; t[2] |= DIRTYFLAG; action = 1;
02576                         }
02577                         targ += *targ;
02578                     }
02579                     arg = argstop;
02580                 }
02581                 else if ( *arg <= -FUNCTION ) arg++;
02582                 else arg += 2;
02583             }
02584         }
02585         t = tnext;
02586     }
02587     return(action);
02588 }
02589
02590 /*
02591     #] DenToFunction :
02592 */

```

Index

- [_CHECK](#)
 - [parallel.c, 2058](#)
 - [__CHECK](#)
 - [parallel.c, 2059](#)
 - [~AEdge](#)
 - [AEdge, 8](#)
 - [~ANode](#)
 - [ANode, 12](#)
 - [~AStack](#)
 - [AStack, 27](#)
 - [~Assign](#)
 - [Assign, 17](#)
 - [~ECand](#)
 - [ECand, 94](#)
 - [~EEdge](#)
 - [EEdge, 96](#)
 - [~EGraph](#)
 - [EGraph, 104](#)
 - [~ENode](#)
 - [ENode, 116](#)
 - [~EStack](#)
 - [EStack, 119](#)
 - [~Interaction](#)
 - [Interaction, 159](#)
 - [~MCBlock](#)
 - [MCBlock, 192](#)
 - [~MCBridge](#)
 - [MCBridge, 194](#)
 - [~MCEdge](#)
 - [MCEdge, 195](#)
 - [~MCOpi](#)
 - [MCOpi, 204](#)
 - [~MConn](#)
 - [MConn, 197](#)
 - [~MGraph](#)
 - [MGraph, 208](#)
 - [~MNodeClass](#)
 - [MNodeClass, 223](#)
 - [~MOrbits](#)
 - [MOrbits, 240](#)
 - [~Model](#)
 - [Model, 231](#)
 - [~NCand](#)
 - [NCand, 269](#)
 - [~NStack](#)
 - [NStack, 278](#)
 - [~Options](#)
 - [Options, 301](#)
 - [~Output](#)
 - [Output, 310](#)
 - [~PNodeClass](#)
 - [PNodeClass, 340](#)
 - [~Particle](#)
 - [Particle, 329](#)
 - [~Process](#)
 - [Process, 369](#)
 - [~SGroup](#)
 - [SGroup, 391](#)
 - [~SProcess](#)
 - [SProcess, 409](#)
 - [~fmpz](#)
 - [flint::fmpz, 138](#)
 - [~mpoly](#)
 - [flint::mpoly, 242](#)
 - [~mpoly_ctx](#)
 - [flint::mpoly_ctx, 243](#)
 - [~mpoly_factor](#)
 - [flint::mpoly_factor, 244](#)
 - [~poly](#)
 - [flint::poly, 343](#)
 - [poly, 346](#)
 - [~poly_factor](#)
 - [flint::poly_factor, 362](#)
- [A](#)
 - [inivar.h, 1673](#)
 - [variable.h, 3196](#)
- [a](#)
 - [MODNUM, 237](#)
- [a3](#)
 - [TrAcEs, 464](#)
- [a4](#)
 - [TrAcEs, 464](#)
- [ABS](#)
 - [declare.h, 788](#)
- [ABSFUNCTION](#)
 - [ftypes.h, 1392](#)
- [AC](#)
 - [variable.h, 3194](#)
- [accu](#)
 - [TrAcEn, 459](#)
 - [TrAcEs, 461](#)
- [AccumGCD](#)
 - [declare.h, 812](#)
 - [reken.c, 2549](#)
- [accup](#)
 - [TrAcEn, 459](#)
 - [TrAcEs, 461](#)
- [acode](#)

- Particle, 331
- PInput, 338
- ACOSFUNCTION
 - ftypes.h, 1396
- ACOSHFUNCTION
 - ftypes.h, 1397
- active
 - FiLe, 130
 - PF_BUFFER, 336
- activenames
 - C_const, 46
- ad
 - TrAcEs, 464
- add
 - COST, 76
 - Fraction, 140
 - poly, 354
- Add2Com
 - declare.h, 803
- Add2ComStrings
 - compcomm.c, 605
 - declare.h, 917
- ADD2POS
 - declare.h, 800
- Add3Com
 - declare.h, 803
- Add4Com
 - declare.h, 803
- Add5Com
 - declare.h, 803
- add_factor
 - factorized_poly, 126
- ADDARG
 - ftypes.h, 1446
- AddArgs
 - declare.h, 812
 - sort.c, 2717
- AddArrayIndex
 - declare.h, 811
 - sch.c, 2648
- addArtic
 - MConn, 199
- addBlock
 - MConn, 199
- addBlockSelf
 - MConn, 199
- addBridge
 - MConn, 198
- addCEdge
 - MConn, 198
- AddCoef
 - declare.h, 813
 - sort.c, 2715
- AddComString
 - compcomm.c, 605
 - declare.h, 945
- AddDictionary
 - declare.h, 1010
- dict.c, 1067
- AddDollar
 - declare.h, 880
 - names.c, 1826
- AddDubious
 - declare.h, 881
 - names.c, 1827
- addEdge
 - Assign, 18
- AddExpression
 - declare.h, 880
 - names.c, 1827
- addFLine
 - EGraph, 108
- AddFloats
 - float.c, 1284
- AddFunction
 - declare.h, 881
 - names.c, 1824
- addGroup
 - SGroup, 392
- AddIndex
 - declare.h, 881
 - names.c, 1823
- addInteraction
 - Model, 231
- addInteractionEnd
 - Model, 232
- AddLHS
 - comtool.c, 753
 - declare.h, 914
- AddLineFeed
 - declare.h, 790
- AddLong
 - declare.h, 813
 - reken.c, 2550
- AddName
 - declare.h, 877
 - names.c, 1819
- AddNtoC
 - comtool.c, 755
 - declare.h, 915
- AddNtoL
 - comtool.c, 754
 - declare.h, 915
- AddObject
 - declare.h, 976
 - minos.c, 1724
- addOPic
 - MConn, 198
- addParticle
 - Model, 231
- addParticleEnd
 - Model, 231
- AddPLon
 - declare.h, 813
 - reken.c, 2550
- AddPoly

- declare.h, [814](#)
- sort.c, [2716](#)
- ADDPOS
 - declare.h, [799](#)
- AddPotModdollar
 - declare.h, [957](#)
 - dollar.c, [1092](#)
- AddRat
 - declare.h, [815](#)
 - reken.c, [2548](#)
- AddRatToFloat
 - float.c, [1284](#)
- AddRHS
 - comtool.c, [754](#)
 - declare.h, [914](#)
- AddSet
 - declare.h, [882](#)
 - names.c, [1825](#)
- AddSymbol
 - declare.h, [880](#)
 - names.c, [1822](#)
- AddTableName
 - declare.h, [976](#)
 - minos.c, [1723](#)
- AddToCB
 - declare.h, [794](#)
- AddToCom
 - compcomm.c, [605](#)
 - declare.h, [917](#)
- AddToDictionary
 - declare.h, [1010](#)
 - dict.c, [1067](#)
- AddToDollarBuffer
 - declare.h, [951](#)
 - dollar.c, [1085](#)
- AddToIndex
 - declare.h, [974](#)
 - minos.c, [1724](#)
- AddToLine
 - declare.h, [815](#)
 - sch.c, [2646](#)
- AddToPreTypes
 - declare.h, [913](#)
 - pre.c, [2324](#)
- AddToScratch
 - declare.h, [1014](#)
 - store.c, [2824](#)
- AddToString
 - declare.h, [905](#)
 - tools.c, [3079](#)
- AddVector
 - declare.h, [881](#)
 - names.c, [1823](#)
- AddWild
 - declare.h, [815](#)
 - wildcard.c, [3212](#)
- AddWildcardName
 - declare.h, [912](#)
- names.c, [1822](#)
- AddWithFloat
 - float.c, [1291](#)
- adjMat
 - MGraph, [213](#)
- AdjustRenumScratch
 - declare.h, [835](#)
 - reshuf.c, [2605](#)
- aebufnum
 - T_const, [439](#)
- AEdge, [7](#)
 - ~AEdge, [8](#)
 - AEdge, [8](#)
 - cand, [8](#)
 - DUMMYPADDING, [8](#)
 - nlegs, [8](#)
 - nodes, [8](#)
 - ptcl, [8](#)
- aedges
 - ANode, [13](#)
- aelegs
 - ANode, [13](#)
- afterwards
 - Stream, [424](#)
- AGMFUNCTION
 - ftypes.h, [1402](#)
- agraph
 - AStack, [28](#)
 - SProcess, [412](#)
- agrcount
 - Process, [370](#)
 - SProcess, [414](#)
- ald
 - EGraph, [110](#)
- alfatable1
 - compiler.c, [715](#)
- all_variables
 - poly, [350](#)
- ALLARGS
 - ftypes.h, [1444](#)
- allAssigned
 - Assign, [18](#)
- allbufnum
 - T_const, [439](#)
- ALLDIRTY
 - ftypes.h, [1419](#)
- ALLFUNCTIONS
 - ftypes.h, [1403](#)
- ALLFUNPOWERS
 - ftypes.h, [1439](#)
- ALLGLOBALS
 - structs.h, [2898](#)
- AllGlobals, [9](#)
 - Cc, [10](#)
 - M, [10](#)
 - N, [10](#)
 - O, [11](#)
 - P, [11](#)

- R, [10](#)
- S, [10](#)
- T, [11](#)
- X, [11](#)
- ALLINTEGERDOUBLE
 - ftypes.h, [1384](#)
- AllLoops
 - declare.h, [1019](#)
 - lus.c, [1679](#)
- ALLMZVFUNCTIONS
 - ftypes.h, [1403](#)
- AllocFileHandle
 - declare.h, [962](#)
 - setfile.c, [2686](#)
- AllocSetups
 - declare.h, [892](#)
 - setfile.c, [2685](#)
- AllocSort
 - declare.h, [892](#)
 - setfile.c, [2686](#)
- AllocSortFileName
 - declare.h, [892](#)
 - setfile.c, [2686](#)
- AllOnePI
 - lus.c, [1680](#)
- AllowDelay
 - P_const, [320](#)
- allPart
 - Model, [235](#)
- allParticles
 - Model, [233](#)
- AllPaths
 - declare.h, [1020](#)
 - lus.c, [1680](#)
- allsign
 - TrAcEn, [460](#)
 - TrAcEs, [465](#)
- ALLVARIABLES
 - ftypes.h, [1374](#)
- ALSODOLLARS
 - ftypes.h, [1442](#)
- ALSOFUNARGS
 - ftypes.h, [1442](#)
- ALSOPOWERS
 - ftypes.h, [1442](#)
- ALSOREVERSE
 - ftypes.h, [1385](#)
- AltCollectFun
 - C_const, [66](#)
- ALTSEPARATOR
 - fwin.h, [1497](#)
 - unix.h, [3182](#)
- AM
 - variable.h, [3195](#)
- AN
 - variable.h, [3195](#)
- aname
 - Particle, [330](#)
- PInput, [338](#)
- ANDCOND
 - ftypes.h, [1436](#)
- ANNOUNCE
 - checkpoint.c, [529](#)
- ANode, [11](#)
 - ~ANode, [12](#)
 - aedges, [13](#)
 - aelegs, [13](#)
 - ANode, [12](#)
 - anodes, [13](#)
 - cand, [13](#)
 - deg, [13](#)
 - newleg, [12](#)
 - nlegs, [13](#)
- anodes
 - ANode, [13](#)
- antiParticle
 - Model, [233](#)
- ANTISYMMETRIC
 - ftypes.h, [1431](#)
- ANYTYPE
 - ftypes.h, [1375](#)
- AO
 - variable.h, [3195](#)
- AP
 - variable.h, [3195](#)
- aparticle
 - Particle, [329](#)
- Apply
 - declare.h, [826](#)
 - tables.c, [2958](#)
- ApplyExec
 - declare.h, [826](#)
 - tables.c, [2958](#)
- ApplyReset
 - declare.h, [826](#)
 - tables.c, [2958](#)
- AR
 - variable.h, [3195](#)
- AreArgsEqual
 - declare.h, [889](#)
 - tools.c, [3085](#)
- arg1
 - optimization, [293](#)
- arg2
 - optimization, [294](#)
- argaddress
 - N_const, [249](#)
- argag
 - Options, [306](#)
- ARGBUFFER, [14](#)
 - buffer, [14](#)
 - dollar, [14](#)
 - dummy, [15](#)
 - size, [14](#)
 - type, [15](#)
- ArgDotproductMerge

- argument.c, [480](#)
- declare.h, [921](#)
- argemg
 - Options, [306](#)
- ArgFactorize
 - argument.c, [479](#)
 - declare.h, [918](#)
- ARGFIELD
 - ftypes.h, [1388](#)
- ARGHEAD
 - ftypes.h, [1419](#)
- arglevel
 - C_const, [65](#)
- arglist
 - N_const, [253](#)
- arglistsize
 - N_const, [257](#)
- argmg
 - Options, [306](#)
- argnames
 - pReVaR, [365](#)
- ARRANGE
 - ftypes.h, [1444](#)
- args
 - PaRtl, [326](#)
- argsize
 - NeStInG, [272](#)
- argstack
 - C_const, [52](#)
- argsumcheck
 - C_const, [64](#)
- ArgSymbolMerge
 - argument.c, [480](#)
 - declare.h, [920](#)
- argtail
 - TaBIEs, [450](#)
- ARGTOARG
 - ftypes.h, [1416](#)
- argument.c, [477](#), [483](#)
 - ArgDotproductMerge, [480](#)
 - ArgFactorize, [479](#)
 - ArgSymbolMerge, [480](#)
 - ArgumentExplode, [478](#)
 - ArgumentImplode, [478](#)
 - CleanupArgCache, [479](#)
 - execarg, [478](#)
 - execterm, [478](#)
 - FindArg, [479](#)
 - InsertArg, [479](#)
 - MakeInteger, [481](#)
 - MakeMod, [481](#)
 - NEWORDER, [478](#)
 - SortWeights, [482](#)
 - TakeArgContent, [480](#)
- argument_to_poly
 - poly, [352](#)
- ArgumentExplode
 - argument.c, [478](#)
- declare.h, [971](#)
- ArgumentImplode
 - argument.c, [478](#)
 - declare.h, [971](#)
- ARGWILD
 - ftypes.h, [1387](#)
- ARLTOARL
 - ftypes.h, [1416](#)
- arraysize
 - POLYMOD, [363](#)
- articals
 - MConn, [200](#)
- AS
 - variable.h, [3195](#)
- ASINFUNCTION
 - ftypes.h, [1396](#)
- ASINHFUNCTION
 - ftypes.h, [1397](#)
- Assign, [15](#)
 - ~Assign, [17](#)
 - addEdge, [18](#)
 - allAssigned, [18](#)
 - Assign, [17](#)
 - assignAllVertices, [17](#)
 - assignVertex, [20](#)
 - assignPLeg, [21](#)
 - assignVertex, [18](#)
 - astack, [24](#)
 - candPart, [20](#)
 - candPartClassify, [21](#)
 - checkAG, [17](#)
 - checkCand, [23](#)
 - checkNode, [23](#)
 - checkOrderCpl, [21](#)
 - checkpoint0, [26](#)
 - cmpNodes, [22](#)
 - cmpPermGraph, [22](#)
 - connect, [19](#)
 - cpleft, [26](#)
 - detEdge, [21](#)
 - edges, [26](#)
 - edgeSym, [22](#)
 - egraph, [23](#)
 - fillEGraph, [19](#)
 - fromMGraph, [18](#)
 - getLegParticle, [19](#)
 - isIsomorphic, [22](#)
 - isOrdLegs, [22](#)
 - isOrdPLeg, [21](#)
 - legEdgeParticle, [19](#)
 - legPart, [20](#)
 - mgraph, [23](#)
 - model, [24](#)
 - nAGraphs, [25](#)
 - nAOPI, [25](#)
 - nEdges, [25](#)
 - nETotal, [25](#)
 - nExtern, [25](#)

- nNodes, [24](#)
- nodes, [25](#)
- opt, [24](#)
- orbits, [24](#)
- pnclass, [24](#)
- prCand, [17](#)
- proc, [24](#)
- reordLeg, [19](#)
- restoreCouple, [23](#)
- saveCouple, [22](#)
- selectLeg, [18](#)
- selectVertex, [18](#)
- selectVertexSimp, [18](#)
- selUnAssLeg, [20](#)
- selUnAssVertex, [20](#)
- selUnAssVertexSimp, [20](#)
- sproc, [24](#)
- subCouple, [23](#)
- updateCandNode, [21](#)
- wAGraphs, [25](#)
- wAOPI, [25](#)
- assign
 - SProcess, [410](#)
- assignAllVertices
 - Assign, [17](#)
- AssignDollar
 - declare.h, [951](#)
 - dollar.c, [1084](#)
- assigned
 - EGraph, [110](#)
- assignIVertex
 - Assign, [20](#)
- assignPLeg
 - Assign, [21](#)
- assignVertex
 - Assign, [18](#)
- AStack, [26](#)
 - ~AStack, [27](#)
 - agraph, [28](#)
 - AStack, [27](#)
 - checkPoint, [27](#)
 - eSize, [29](#)
 - eStack, [29](#)
 - eStackP, [29](#)
 - nSize, [29](#)
 - nStack, [28](#)
 - nStackP, [29](#)
 - prStack, [28](#)
 - pushEdge, [28](#)
 - pushNode, [28](#)
 - restore, [28](#)
 - restoreMsg, [28](#)
 - setAGraph, [27](#)
- astack
 - Assign, [24](#)
 - Process, [372](#)
 - SProcess, [411](#)
- AT
 - variable.h, [3195](#)
- ATAN2FUNCTION
 - ftypes.h, [1397](#)
- ATANFUNCTION
 - ftypes.h, [1397](#)
- ATANHFUNCTION
 - ftypes.h, [1397](#)
- ATENDOFMODULE
 - ftypes.h, [1373](#)
- atstartup
 - M_const, [189](#)
- AutoDeclareFlag
 - C_const, [54](#)
- AutoFunctionList
 - C_const, [45](#)
- autofunctions
 - variable.h, [3193](#)
- AutoIndexList
 - C_const, [45](#)
- autoindices
 - variable.h, [3193](#)
- AUTONAMES
 - ftypes.h, [1443](#)
- autonames
 - C_const, [46](#)
- AutoSymbolList
 - C_const, [45](#)
- autosymbols
 - variable.h, [3193](#)
- AutoVectorList
 - C_const, [45](#)
- autovectors
 - variable.h, [3193](#)
- auxr1
 - evaluate.c, [1139](#)
- auxr2
 - evaluate.c, [1139](#)
- auxr3
 - evaluate.c, [1139](#)
- auxr4
 - evaluate.c, [1139](#)
- auxr5
 - evaluate.c, [1139](#)
- AWORDMASK
 - form3.h, [1328](#)
- AX
 - variable.h, [3195](#)
- BACKINOUT
 - declare.h, [794](#)
- balance
 - NaMeNode, [263](#)
- Balancing
 - S_const, [387](#)
- BASECODE
 - ftypes.h, [1447](#)
- BASEPOSITION
 - declare.h, [801](#)
- bconn

- EGraph, [113](#)
- begin
 - Options, [304](#)
- beginProc
 - Options, [304](#)
- beginSubProc
 - Options, [304](#)
- Bernoulli
 - declare.h, [832](#)
 - reken.c, [2557](#)
- BERNOULLIFUNCTION
 - ftypes.h, [1393](#)
- bernoullis
 - T_const, [435](#)
- BETAVERSION
 - form3.h, [1326](#)
- BHEAD
 - structs.h, [2893](#)
- BHEAD0
 - structs.h, [2894](#)
- bicol
 - MGraph, [217](#)
- biconnE
 - EGraph, [106](#)
- biconnME
 - MGraph, [210](#)
- bicount
 - EGraph, [114](#)
 - MGraph, [217](#)
- bidef
 - EGraph, [113](#)
 - MGraph, [217](#)
- BigLong
 - declare.h, [815](#)
 - reken.c, [2550](#)
- biinitE
 - EGraph, [106](#)
- bilow
 - EGraph, [113](#)
 - MGraph, [217](#)
- BinarySearch
 - declare.h, [963](#)
 - sort.c, [2724](#)
- BinomGen
 - declare.h, [816](#)
 - ratio.c, [2497](#)
- BINOMIAL
 - ftypes.h, [1391](#)
- BIPART
 - ftypes.h, [1456](#)
- bipart
 - MGraph, [216](#)
- bisearchE
 - EGraph, [107](#)
- bisearchME
 - MGraph, [210](#)
- bit_0
 - bit_field, [30](#)
- bit_1
 - bit_field, [30](#)
- bit_2
 - bit_field, [30](#)
- bit_3
 - bit_field, [30](#)
- bit_4
 - bit_field, [30](#)
- bit_5
 - bit_field, [31](#)
- bit_6
 - bit_field, [31](#)
- bit_7
 - bit_field, [31](#)
- bit_field, [29](#)
 - bit_0, [30](#)
 - bit_1, [30](#)
 - bit_2, [30](#)
 - bit_3, [30](#)
 - bit_4, [30](#)
 - bit_5, [31](#)
 - bit_6, [31](#)
 - bit_7, [31](#)
- BITSINLONG
 - form3.h, [1327](#)
- BITSINWORD
 - form3.h, [1327](#)
- blksp
 - MConn, [200](#)
- blkstk
 - MConn, [201](#)
- blkstkptr
 - MConn, [203](#)
- blnce
 - tree, [467](#)
- BLOCK
 - ftypes.h, [1401](#)
- block
 - MGraph, [216](#)
- blocks
 - MConn, [200](#)
- BlockSpaces
 - O_const, [285](#)
- BOOL
 - form3.h, [1333](#)
- boomlijst
 - CbUf, [73](#)
 - TaBIEs, [450](#)
- BottomLevel
 - C_const, [56](#)
- bounds
 - TaBIEs, [452](#)
- BrackBuf
 - T_const, [434](#)
- bracket
 - BrAcKeTiNdEx, [32](#)
 - O_const, [282](#)
- bracketbuffer

- BrAcKeTiNfO, 33
- bracketbuffersize
 - BrAcKeTiNfO, 34
- BRACKETCURRENTEXPR
 - execute.c, 1152
- BracketFactors
 - M_const, 191
- bracketfill
 - BrAcKeTiNfO, 34
- BrAcKeTiNdEx, 31
 - bracket, 32
 - next, 32
 - start, 32
 - termsinbracket, 32
- bracketindexflag
 - C_const, 59
 - T_const, 440
- BrAcKeTiNfO, 33
 - bracketbuffer, 33
 - bracketbuffersize, 34
 - bracketfill, 34
 - indexbuffer, 33
 - indexbuffersize, 34
 - indexfill, 34
 - SortType, 34
- BracketInfo, 35
 - BracketInfo, 35
 - dummy, 36
 - num_terms, 36
 - operator<, 36
 - p, 36
 - pattern, 36
- bracketinfo
 - ExPrEsSiOn, 122
 - T_const, 437
- BracketNormalize
 - C_const, 59
 - declare.h, 846
 - normal.c, 1869
- BracketOn
 - R_const, 379
- BRACKETOTHEREXPR
 - execute.c, 1152
- bridges
 - MConn, 200
- buff
 - PF_BUFFER, 334
- Buffer
 - CbUf, 71
- buffer
 - ARGBUFFER, 14
 - INSIDEINFO, 157
 - longMultiStruct, 166
 - PRELOAD, 364
 - StOrEcAcHe, 417
 - StreaM, 422
- bufferedInd
 - O_const, 288
- BufferForOutput
 - inivar.h, 1674
- buffernum
 - SuBbUf, 425
- bufferposition
 - StreaM, 423
- buffers
 - TaBIes, 450
- buffersfill
 - TaBIes, 453
- BufferSize
 - CbUf, 73
- buffersize
 - StreaM, 423
- bufferssize
 - TaBIes, 453
- bufIPstruct, 37
 - e, 37
 - i, 37
- bufnum
 - TaBIes, 453
- bufpos
 - longMultiStruct, 166
- BUG
 - variable.h, 3194
- bugtool.c, 523, 524
 - ExprStatus, 524
- build_Horner_tree
 - optimize.cc, 1987
- buildTree
 - optimize.cc, 1991
- BUILTINDOLLARS
 - ftypes.h, 1405
- BUILTININDICES
 - ftypes.h, 1404
- BUILTINSYMBOLS
 - ftypes.h, 1404
- BUILTINVECTORS
 - ftypes.h, 1405
- bzero
 - form3.h, 1332
- c1PI
 - MGraph, 214
- c1PINoTadBlock
 - MGraph, 214
- C_const, 38
 - activenames, 46
 - AltCollectFun, 66
 - arglevel, 65
 - argstack, 52
 - argsumcheck, 64
 - AutoDeclareFlag, 54
 - AutoFunctionList, 45
 - AutoIndexList, 45
 - autonames, 46
 - AutoSymbolList, 45
 - AutoVectorList, 45
 - BottomLevel, 56

bracketindexflag, 59
BracketNormalize, 59
cbufList, 45
cbufnum, 54
ChannelList, 43
CheckpointFlag, 63
CheckpointInterval, 54
CheckpointRunAfter, 52
CheckpointRunBefore, 52
CheckpointStamp, 53
cmod, 47
CModule, 53
Cnumpows, 67
CodesFlag, 58
CollectFun, 66
CollectPercentage, 69
ComDefer, 66
Commercial, 70
CommutInSet, 52
CompileLevel, 57
compiletype, 54
CurrentStream, 47
debugFlags, 70
DidClean, 67
DirtPow, 66
dollarnames, 42
dolooplevel, 62
doloopnest, 51
doloopstack, 51
doloopstacksize, 62
DubiousList, 43
DumNum, 65
dumnumflag, 59
endoftokens, 50
ExpressionList, 43
exprfillwarning, 60
exprnames, 43
extrasym, 51
extrasymbols, 69
ffbufnum, 61
FinalStats, 58
firstconstindex, 54
firstctypemessage, 56
FixIndices, 49
FlintPolyFlag, 62
Fortran90Kind, 51
FunctionList, 43
Functions, 46
funpowers, 57
halfmod, 48
HideLevel, 68
HumanStatsFlag, 62
iBuffer, 49
iBufferSize, 53
idoption, 56
IfCount, 48
IfHeap, 48
IfLevel, 67
IfStack, 49
IfSumCheck, 52
IndexList, 44
Indices, 46
inexprlevel, 65
inexprstack, 53
inexprsumcheck, 64
InnerTest, 64
inparallelf, 60
insidefirst, 55
insidelevel, 65
insidestack, 52
insidesumcheck, 64
iPointer, 49
IsFortran90, 61
iStop, 49
LabelNames, 49
Labels, 50
IDefDim, 55
IDefDim4, 55
Ihdollarflag, 61
LineLength, 68
LogHandle, 68
IPolyFun, 68
IPolyFunExp, 69
IPolyFunInv, 68
IPolyFunPow, 69
IPolyFunType, 69
IPolyFunVar, 69
ISortType, 58
IUniTrace, 64
IUnitTrace, 66
MaxIf, 56
MaxLabels, 55
MaxNumStreams, 56
MaxSwitch, 68
maxtermlevel, 59
MemDebugFlag, 61
minsidefirst, 55
ModelLevel, 63
models, 47
modelspace, 63
modinverses, 51
modmode, 66
ModOptDolList, 45
modpowers, 48
mparallelf, 60
mProcessBucketSize, 53
MultiBracketBuf, 51
MultiBracketLevels, 61
MustTestTable, 65
NamesFlag, 57
ncmod, 65
nhalfmod, 66
NoCompress, 61
NoShowInput, 54
npowmod, 65
NumLabels, 55

- nummodels, 63
- NumStreams, 56
- NumWildcardNames, 55
- NwildC, 66
- OldFactArgFlag, 61
- OldGCDflag, 62
- OldParallelStats, 58
- origin, 63
- OutNumberType, 67
- OutputMode, 67
- OutputSpaces, 67
- outsidefun, 57
- parallelflag, 59
- partodoflag, 60
- PolyRatFunChanged, 69
- PotModDolList, 44
- powmod, 48
- PrintBacktraceFlag, 62
- ProcessBucketSize, 53
- ProcessStats, 59
- properorderflag, 60
- ProtoType, 48
- RepLevel, 64
- RepSumCheck, 64
- separators, 42
- SetElementList, 44
- SetList, 44
- SetupFlag, 58
- ShortStats, 54
- ShortStatsMax, 64
- SizeCommutelnSet, 63
- SortReallocateFlag, 62
- SortType, 58
- StatsFlag, 57
- StoreFileSize, 42
- StoreHandle, 68
- Streams, 46
- SwitchArray, 47
- SwitchHeap, 47
- SwitchlnArray, 68
- SwitchLevel, 67
- SymbolList, 44
- Symbols, 46
- SymChangeFlag, 69
- TableBaseList, 45
- tablecheck, 56
- tablefilling, 60
- termlevel, 65
- termstortstack, 47
- termstack, 47
- termsumcheck, 50
- TestValue, 52
- ThreadBalancing, 59
- ThreadBucketSize, 53
- ThreadsFlag, 58
- ThreadSortFileSynch, 59
- ThreadStats, 58
- ToBelnFactors, 70
- tokenarglevel, 51
- tokens, 50
- TokensWriteFlag, 57
- topolynomialflag, 61
- toptokens, 50
- TransEname, 53
- UnsureDollarMode, 57
- varnames, 43
- vectorlikeLHS, 63
- VectorList, 44
- Vectors, 46
- vetofilling, 60
- vetotablebasefill, 60
- WarnFlag, 57
- WhileLevel, 67
- WildC, 48
- WildcardBufferSize, 56
- WildcardNames, 50
- wildflag, 55
- WTimeStatsFlag, 62
- cache_buffer
 - checkpoint.c, 533
- cache_fill
 - checkpoint.c, 533
- CACHE_SIZE
 - checkpoint.c, 526
- CACHED_SNAPSHOT
 - checkpoint.c, 526
- CacheNumberFree
 - declare.h, 805
- CacheNumberMalloc
 - declare.h, 805
- CacheNumberMallocAddMemory
 - declare.h, 989
 - tools.c, 3083
- CacheNumberMemHeap
 - T_const, 437
- CacheNumberMemMax
 - T_const, 440
- CacheNumberMemTop
 - T_const, 440
- calcHash
 - node, 275
- CalculateEuler
 - float.c, 1287
- CalculateMZV
 - float.c, 1287
- CalculateMZVhalf
 - float.c, 1286
- CanCommu
 - CbUf, 72
- cand
 - AEdge, 8
 - ANode, 13
- candPart
 - Assign, 20
- candPartClassify
 - Assign, 21

- CARRIAGEReturn
 - fwin.h, [1496](#)
 - unix.h, [3181](#)
- caseoffset
 - SWITCH, [427](#)
- CatchDollar
 - declare.h, [951](#)
 - dollar.c, [1084](#)
- CBUF
 - structs.h, [2897](#)
- CbUf, [70](#)
 - boomlijst, [73](#)
 - Buffer, [71](#)
 - BufferSize, [73](#)
 - CanCommu, [72](#)
 - dimension, [73](#)
 - lhs, [72](#)
 - maxlhs, [73](#)
 - maxrhs, [74](#)
 - MaxTreeSize, [74](#)
 - mnumlhs, [74](#)
 - mnumrhs, [74](#)
 - numdum, [72](#)
 - numlhs, [73](#)
 - numrhs, [73](#)
 - NumTerms, [72](#)
 - numtree, [74](#)
 - Pointer, [71](#)
 - rhs, [72](#)
 - rootnum, [74](#)
 - Top, [71](#)
- cbuf
 - variable.h, [3192](#)
- cBuffer
 - sOrT, [400](#)
- cBufferSize
 - sOrT, [404](#)
- cbufList
 - C_const, [45](#)
- cbufnum
 - C_const, [54](#)
- Cc
 - AllGlobals, [10](#)
- cComChar
 - P_const, [322](#)
- cdeg
 - PGInput, [337](#)
- CDELETE
 - ftypes.h, [1374](#)
- CDELTA
 - ftypes.h, [1376](#)
- cDiag
 - MGraph, [214](#)
- CDOLLAR
 - ftypes.h, [1376](#)
- CDOTPRODUCT
 - ftypes.h, [1375](#)
- CDOUBLEDOT
 - ftypes.h, [1377](#)
- CDUBIOUS
 - ftypes.h, [1376](#)
- cedges
 - MConn, [200](#)
- CEXPRESSION
 - ftypes.h, [1375](#)
- CFD
 - minos.c, [1720](#)
- CFUN
 - ftypes.h, [1457](#)
- CFUNCTION
 - ftypes.h, [1375](#)
- cgen
 - SGroup, [394](#)
- ChainIn
 - declare.h, [971](#)
 - function.c, [1472](#)
- ChainOut
 - declare.h, [971](#)
 - function.c, [1473](#)
- CHANNEL
 - structs.h, [2897](#)
- ChAnNeL, [75](#)
 - handle, [75](#)
 - name, [75](#)
- ChannelList
 - C_const, [43](#)
- channels
 - variable.h, [3192](#)
- characters
 - DICTIONARY, [83](#)
- CharOut
 - declare.h, [893](#)
 - pre.c, [2312](#)
- chartype
 - variable.h, [3188](#)
- CHECK
 - parallel.c, [2058](#)
- check_memory
 - poly, [348](#)
- checkAG
 - Assign, [17](#)
- checkCand
 - Assign, [23](#)
- CHECKEXTERN
 - ftypes.h, [1454](#)
- CheckFloat
 - float.c, [1289](#)
- CHECKLOGTYPE
 - ftypes.h, [1386](#)
- checkNode
 - Assign, [23](#)
- checkOrderCpl
 - Assign, [21](#)
- checkPoint
 - AStack, [27](#)
- checkpoint.c, [525](#), [533](#)

- ANNOUNCE, [529](#)
- cache_buffer, [533](#)
- cache_fill, [533](#)
- CACHE_SIZE, [526](#)
- CACHED_SNAPSHOT, [526](#)
- CheckRecoveryFile, [530](#)
- DeleteRecoveryFile, [530](#)
- DoCheckpoint, [532](#)
- DoRecovery, [531](#)
- flush_cache, [531](#)
- fwrite_cached, [531](#)
- InitRecovery, [530](#)
- R_COPY_B, [527](#)
- R_COPY_LIST, [528](#)
- R_COPY_NAMETREE, [529](#)
- R_COPY_S, [528](#)
- R_FREE, [526](#)
- R_FREE_NAMETREE, [527](#)
- R_FREE_STREAM, [527](#)
- R_SET, [527](#)
- RecoveryFilename, [530](#)
- S_FLUSH_B, [528](#)
- S_WRITE_B, [527](#)
- S_WRITE_DOLLAR, [529](#)
- S_WRITE_LIST, [528](#)
- S_WRITE_NAMETREE, [529](#)
- S_WRITE_S, [528](#)
- checkpoint0
 - Assign, [26](#)
- CheckpointFlag
 - C_const, [63](#)
- CheckpointInterval
 - C_const, [54](#)
- CheckpointRunAfter
 - C_const, [52](#)
- CheckpointRunBefore
 - C_const, [52](#)
- CheckpointStamp
 - C_const, [53](#)
- CHECKPOLY
 - token.c, [3041](#)
- CheckPower
 - O_const, [284](#)
- CheckRecoveryFile
 - checkpoint.c, [530](#)
 - declare.h, [986](#)
- CheckTableDeclarations
 - declare.h, [826](#)
 - tables.c, [2956](#)
- CheckWild
 - declare.h, [816](#)
 - wildcard.c, [3212](#)
- childs
 - tree_node, [469](#)
- CHISHOLM
 - ftypes.h, [1385](#)
- Chisholm
 - declare.h, [816](#)
- opera.c, [1951](#)
- chkMomConsv
 - EGraph, [108](#)
- Chop
 - float.c, [1291](#)
- CINDEX
 - ftypes.h, [1375](#)
- cl2mcl
 - PNodeClass, [342](#)
- cl2nd
 - PNodeClass, [342](#)
 - SProcess, [412](#)
- clCmp
 - MNodeClass, [223](#)
- cldeg
 - NCInput, [271](#)
 - ToPoTyPe, [457](#)
- CleanDollarFactors
 - declare.h, [955](#)
 - dollar.c, [1090](#)
- CleanExpr
 - declare.h, [816](#)
 - execute.c, [1152](#)
- CLEANFLAG
 - ftypes.h, [1418](#)
- CleanUp
 - declare.h, [816](#)
 - startup.c, [2794](#)
- cleanup
 - flintinterface.h, [1266](#)
- cleanup_master
 - flintinterface.h, [1266](#)
- CleanupArgCache
 - argument.c, [479](#)
 - declare.h, [920](#)
- CleanUpSort
 - declare.h, [962](#)
 - sort.c, [2724](#)
- CleanupTerm
 - declare.h, [966](#)
 - execute.c, [1154](#)
- CLEAR
 - ftypes.h, [1437](#)
- ClearBracketIndex
 - declare.h, [969](#)
 - index.c, [1664](#)
- clearcbuf
 - comtool.c, [753](#)
 - declare.h, [961](#)
- ClearfFloat
 - evaluate.c, [1141](#)
- clearGroup
 - SGroup, [392](#)
- ClearMacro
 - declare.h, [896](#)
 - pre.c, [2315](#)
- CLEARMODULE
 - ftypes.h, [1366](#)

- ClearMZVTables
 - float.c, [1289](#)
- clearnamefill
 - NaMeTree, [268](#)
- clearnodefill
 - NaMeTree, [268](#)
- ClearOptimize
 - declare.h, [959](#)
 - optimize.cc, [2000](#)
- ClearPushback
 - declare.h, [892](#)
 - pre.c, [2311](#)
- ClearSpectators
 - declare.h, [1013](#)
 - spectator.c, [2782](#)
- ClearStore
 - M_const, [191](#)
- ClearTableTree
 - declare.h, [960](#)
 - tables.c, [2954](#)
- ClearTree
 - comtool.c, [756](#)
 - declare.h, [950](#)
- ClearWild
 - declare.h, [817](#)
 - wildcard.c, [3212](#)
- ClearWildcardNames
 - declare.h, [912](#)
 - names.c, [1822](#)
- clext
 - ToPoTyPe, [457](#)
- clist
 - Interaction, [160](#)
 - MGraph, [212](#)
 - MNodeClass, [225](#)
 - Process, [371](#)
 - SProcess, [413](#)
- clmat
 - MNodeClass, [225](#)
- clnum
 - NCInput, [271](#)
 - ToPoTyPe, [457](#)
- clord
 - MNodeClass, [225](#)
- closeAllExternalChannels
 - declare.h, [979](#)
 - extcmd.c, [1189](#)
- CloseChannel
 - declare.h, [886](#)
 - tools.c, [3077](#)
- closeExternalChannel
 - declare.h, [979](#)
 - extcmd.c, [1188](#)
- CloseFile
 - declare.h, [883](#)
 - tools.c, [3074](#)
- CloseStream
 - declare.h, [876](#)
- tools.c, [3073](#)
- class
 - MNode, [221](#)
- cltyp
 - NCInput, [271](#)
- cmaxd
 - NCInput, [271](#)
 - ToPoTyPe, [458](#)
- cmaxdeg
 - MNode, [221](#)
 - MNodeClass, [225](#)
 - Particle, [331](#)
 - PNodeClass, [341](#)
- cmind
 - NCInput, [271](#)
 - ToPoTyPe, [458](#)
- cminddeg
 - MNode, [221](#)
 - MNodeClass, [225](#)
 - Particle, [331](#)
 - PNodeClass, [341](#)
- cmmod
 - C_const, [47](#)
 - N_const, [254](#)
- CMODE
 - ftypes.h, [1383](#)
- CMODEL
 - ftypes.h, [1376](#)
- CModule
 - C_const, [53](#)
- cmp
 - node, [275](#)
- CmpArray
 - declare.h, [890](#)
 - tools.c, [3078](#)
- cmpMNCArray
 - MNodeClass, [224](#)
- cmpMom
 - EGraph, [107](#)
- cmpNodes
 - Assign, [22](#)
- cmpPermGraph
 - Assign, [22](#)
- cname
 - OptQGDef, [308](#)
- cnamlist
 - MInput, [219](#)
- cnlist
 - Model, [234](#)
- cNoTadBlock
 - MGraph, [214](#)
- cNoTadpole
 - MGraph, [214](#)
- cnum
 - PGInput, [337](#)
- CNUMBER
 - ftypes.h, [1376](#)
- Cnumlhs

- R_const, 378
- Cnumpows
 - C_const, 67
- CoAntiBracket
 - compcomm.c, 612
 - declare.h, 922
- CoAntiPutInside
 - compcomm.c, 619
 - declare.h, 924
- CoAntiSymmetrize
 - compcomm.c, 608
 - declare.h, 922
- CoApply
 - declare.h, 925
 - tables.c, 2957
- CoArgExplode
 - compcomm.c, 615
 - declare.h, 922
- CoArgImplode
 - compcomm.c, 615
 - declare.h, 922
- CoArgToExtraSymbol
 - compcomm.c, 616
 - declare.h, 996
- CoArgument
 - compcomm.c, 606
 - declare.h, 923
- CoAssign
 - comexpr.c, 573
 - declare.h, 949
- CoAuto
 - declare.h, 942
 - names.c, 1826
- CoBracket
 - compcomm.c, 612
 - declare.h, 924
- CoBreak
 - compcomm.c, 619
 - declare.h, 943
- CoCanonicalize
 - declare.h, 823
 - diagrams.c, 1041
- CoCase
 - compcomm.c, 619
 - declare.h, 943
- CoCFunction
 - declare.h, 924
 - names.c, 1824
- CoChainin
 - compcomm.c, 610
 - declare.h, 942
- CoChainout
 - compcomm.c, 610
 - declare.h, 942
- CoChisholm
 - compcomm.c, 609
 - declare.h, 941
- CoChop
 - float.c, 1290
- CoClearTable
 - compcomm.c, 616
 - declare.h, 941
- CoClearUserFlag
 - compcomm.c, 620
 - declare.h, 1018
- CoCollect
 - compcomm.c, 600
 - declare.h, 925
- CoCommuteInSet
 - declare.h, 882
 - names.c, 1824
- CoCompress
 - compcomm.c, 600
 - declare.h, 925
- CoContract
 - compcomm.c, 605
 - declare.h, 925
- CoCopySpectator
 - declare.h, 1013
 - spectator.c, 2782
- CoCreateAll
 - compcomm.c, 620
 - declare.h, 1019
- CoCreateAllLoops
 - compcomm.c, 620
 - declare.h, 1018
- CoCreateAllPaths
 - compcomm.c, 620
 - declare.h, 1018
- CoCreateSpectator
 - declare.h, 1012
 - spectator.c, 2781
- CoCTable
 - declare.h, 940
 - names.c, 1825
- CoCTensor
 - declare.h, 924
 - names.c, 1825
- CoCycleSymmetrize
 - compcomm.c, 608
 - declare.h, 925
- code
 - OPTIMIZERESULT, 298
- CoDeallocateTable
 - comexpr.c, 573
 - declare.h, 955
- CodeFactors
 - compiler.c, 714
 - declare.h, 950
- CoDefault
 - compcomm.c, 619
 - declare.h, 943
- CodeGenerator
 - compiler.c, 714
 - declare.h, 949
- CoDelete

- compcomm.c, 601
- declare.h, 925
- CoDenominators
 - compcomm.c, 616
 - declare.h, 926
- CodesFlag
 - C_const, 58
- codesize
 - OPTIMIZERESULT, 298
- CodeToLine
 - declare.h, 810
 - sch.c, 2647
- CoDimension
 - declare.h, 926
 - names.c, 1823
- CoDiscard
 - compcomm.c, 605
 - declare.h, 926
- CoDisorder
 - comexpr.c, 571
 - declare.h, 926
- CoDo
 - compcomm.c, 617
 - declare.h, 996
- CoDrop
 - compcomm.c, 603
 - declare.h, 926
- CoDropCoefficient
 - compcomm.c, 616
 - declare.h, 926
- CoDropSymbols
 - compcomm.c, 616
 - declare.h, 926
- coeff
 - optimization, 294
- COEFFI
 - ftypes.h, 1434
- coefficient
 - poly, 351
- coefficient_list_divmod
 - poly, 353
- coefficients_modulo
 - poly, 351
- COEFFICIENTSONLY
 - ftypes.h, 1442
- COEFFSYMBOL
 - ftypes.h, 1403
- coefs
 - POLYMOD, 363
- CoElse
 - compcomm.c, 612
 - declare.h, 927
- CoElseif
 - compcomm.c, 613
 - declare.h, 927
- CoEmptySpectator
 - declare.h, 1013
 - spectator.c, 2782
- CoEndArgument
 - compcomm.c, 606
 - declare.h, 927
- CoEndDo
 - compcomm.c, 617
 - declare.h, 996
- CoEndIf
 - compcomm.c, 613
 - declare.h, 927
- CoEndInExpression
 - compcomm.c, 611
 - declare.h, 924
- CoEndInside
 - compcomm.c, 606
 - declare.h, 927
- CoEndModel
 - declare.h, 1018
 - model.c, 1749
- CoEndRepeat
 - compcomm.c, 611
 - declare.h, 927
- CoEndSwitch
 - compcomm.c, 619
 - declare.h, 943
- CoEndTerm
 - compcomm.c, 614
 - declare.h, 927
- CoEndWhile
 - compcomm.c, 613
 - declare.h, 928
- CoEvaluate
 - float.c, 1291
- CoExit
 - compcomm.c, 610
 - declare.h, 928
- CoExtraSymbols
 - compcomm.c, 617
 - declare.h, 996
- CoFactArg
 - compcomm.c, 607
 - declare.h, 928
- CoFactDollar
 - compcomm.c, 617
 - declare.h, 928
- CoFactorize
 - compcomm.c, 618
 - declare.h, 928
- CoFill
 - comexpr.c, 572
 - declare.h, 929
- CoFillExpression
 - comexpr.c, 572
 - declare.h, 929
- CoFindLoop
 - compcomm.c, 613
 - declare.h, 953
- CoFixIndex
 - compcomm.c, 602

- declare.h, 929
- CoFlags
 - compcomm.c, 601
 - declare.h, 934
- CoFormat
 - compcomm.c, 600
 - declare.h, 929
- CoFromPolynomial
 - compcomm.c, 616
 - declare.h, 995
- CoFunction
 - declare.h, 882
 - names.c, 1824
- CoFunPowers
 - compcomm.c, 614
 - declare.h, 953
- CoGlobal
 - comexpr.c, 570
 - declare.h, 929
- CoGlobalFactorized
 - comexpr.c, 570
 - declare.h, 930
- CoGoTo
 - compcomm.c, 605
 - declare.h, 930
- CoHide
 - compcomm.c, 604
 - declare.h, 938
- Cold
 - comexpr.c, 571
 - declare.h, 930
- ColdExpression
 - comexpr.c, 572
 - declare.h, 949
- ColdNew
 - comexpr.c, 571
 - declare.h, 930
- ColdOld
 - comexpr.c, 570
 - declare.h, 930
- Colf
 - compcomm.c, 612
 - declare.h, 930
- ColfMatch
 - comexpr.c, 571
 - declare.h, 930
- ColfNoMatch
 - comexpr.c, 571
 - declare.h, 931
- ColIndex
 - declare.h, 931
 - names.c, 1823
- ColnExpression
 - compcomm.c, 611
 - declare.h, 923
- ColnParallel
 - compcomm.c, 610
 - declare.h, 923
- Colnside
 - compcomm.c, 606
 - declare.h, 923
- ColnsideFirst
 - compcomm.c, 601
 - declare.h, 931
- ColnsideHide
 - compcomm.c, 604
 - declare.h, 938
- CoKeep
 - compcomm.c, 601
 - declare.h, 931
- CoLabel
 - compcomm.c, 606
 - declare.h, 931
- CollectFun
 - C_const, 66
- CollectOverFlag
 - S_const, 388
- CollectPercentage
 - C_const, 69
- CoLoad
 - declare.h, 931
 - store.c, 2824
- CoLocal
 - comexpr.c, 570
 - declare.h, 931
- CoLocalFactorized
 - comexpr.c, 570
 - declare.h, 932
- CoMakeInteger
 - compcomm.c, 607
 - declare.h, 934
- CoMany
 - comexpr.c, 571
 - declare.h, 932
- combilast
 - SHvariables, 396
- ComChar
 - P_const, 322
- ComDefer
 - C_const, 66
- CoMerge
 - compcomm.c, 615
 - declare.h, 932
- CoMetric
 - compcomm.c, 602
 - declare.h, 932
- comexpr.c, 569, 573
 - CoAssign, 573
 - CoDeallocateTable, 573
 - CoDisorder, 571
 - CoFill, 572
 - CoFillExpression, 572
 - CoGlobal, 570
 - CoGlobalFactorized, 570
 - Cold, 571
 - ColdExpression, 572

- ColdNew, [571](#)
- ColdOld, [570](#)
- ColfMatch, [571](#)
- ColfNoMatch, [571](#)
- CoLocal, [570](#)
- CoLocalFactorized, [570](#)
- CoMany, [571](#)
- CoMulti, [571](#)
- CoMultiply, [572](#)
- CoOnce, [572](#)
- CoOnly, [572](#)
- CoPrintTable, [573](#)
- CoSelect, [572](#)
- DoExpr, [570](#)
- comfun
 - T_const, [442](#)
- COMFUNPOWERS
 - ftypes.h, [1439](#)
- comind
 - T_const, [442](#)
- commentchar
 - setfile.c, [2687](#)
- Commercial
 - C_const, [70](#)
- COMMERCIALSIZE
 - fsizes.h, [1342](#)
- Commute
 - declare.h, [817](#)
 - normal.c, [1867](#)
- commute
 - FUN_INFO, [142](#)
 - FuNcTiOn, [144](#)
- CommutelnSet
 - C_const, [52](#)
- comnum
 - T_const, [442](#)
- CoModel
 - declare.h, [1017](#)
 - model.c, [1748](#)
- CoModOption
 - declare.h, [932](#)
 - module.c, [1756](#)
- CoModuleOption
 - declare.h, [932](#)
 - module.c, [1756](#)
- CoModulus
 - compcomm.c, [611](#)
 - declare.h, [933](#)
- CompactifyTree
 - declare.h, [879](#)
 - names.c, [1821](#)
- COMPARE
 - structs.h, [2899](#)
- Compare1
 - declare.h, [818](#)
 - sort.c, [2718](#)
- compare_degree_vector
 - poly, [354](#)
- CompareArgs
 - declare.h, [889](#)
 - tools.c, [3085](#)
- COMPAREDUMMY
 - structs.h, [2897](#)
- CompareFunctions
 - declare.h, [817](#)
 - normal.c, [1867](#)
- CompareHSymbols
 - declare.h, [980](#)
 - sort.c, [2719](#)
- CompareRoutine
 - R_const, [377](#)
- CompareSymbols
 - declare.h, [980](#)
 - sort.c, [2718](#)
- CompareTerms
 - declare.h, [808](#)
- CompArg
 - declare.h, [817](#)
 - tools.c, [3086](#)
- compbuffer
 - SWITCHTABLE, [429](#)
- CompCoef
 - declare.h, [818](#)
 - reken.c, [2555](#)
- compcomm.c, [597](#), [621](#)
 - Add2ComStrings, [605](#)
 - AddComString, [605](#)
 - AddToCom, [605](#)
 - CoAntiBracket, [612](#)
 - CoAntiPutInside, [619](#)
 - CoAntiSymmetrize, [608](#)
 - CoArgExplode, [615](#)
 - CoArgImplode, [615](#)
 - CoArgToExtraSymbol, [616](#)
 - CoArgument, [606](#)
 - CoBracket, [612](#)
 - CoBreak, [619](#)
 - CoCase, [619](#)
 - CoChainin, [610](#)
 - CoChainout, [610](#)
 - CoChisholm, [609](#)
 - CoClearTable, [616](#)
 - CoClearUserFlag, [620](#)
 - CoCollect, [600](#)
 - CoCompress, [600](#)
 - CoContract, [605](#)
 - CoCreateAll, [620](#)
 - CoCreateAllLoops, [620](#)
 - CoCreateAllPaths, [620](#)
 - CoCycleSymmetrize, [608](#)
 - CoDefault, [619](#)
 - CoDelete, [601](#)
 - CoDenominators, [616](#)
 - CoDiscard, [605](#)
 - CoDo, [617](#)
 - CoDrop, [603](#)

CoDropCoefficient, 616
 CoDropSymbols, 616
 CoElse, 612
 CoElseIf, 613
 CoEndArgument, 606
 CoEndDo, 617
 CoEndIf, 613
 CoEndInExpression, 611
 CoEndInside, 606
 CoEndRepeat, 611
 CoEndSwitch, 619
 CoEndTerm, 614
 CoEndWhile, 613
 CoExit, 610
 CoExtraSymbols, 617
 CoFactArg, 607
 CoFactDollar, 617
 CoFactorize, 618
 CoFindLoop, 613
 CoFixIndex, 602
 CoFlags, 601
 CoFormat, 600
 CoFromPolynomial, 616
 CoFunPowers, 614
 CoGoTo, 605
 CoHide, 604
 Colf, 612
 ColnExpression, 611
 ColnParallel, 610
 ColnInside, 606
 ColnInsideFirst, 601
 ColnIntoHide, 604
 CoKeep, 601
 CoLabel, 606
 CoMakeInteger, 607
 CoMerge, 615
 CoMetric, 602
 CoModulus, 611
 CoMultiBracket, 612
 CoNFactorize, 618
 CoNoDrop, 603
 CoNoHide, 604
 CoNoIntoHide, 604
 CoNormalize, 606
 CoNoSkip, 604
 CoNotInParallel, 610
 CoNoUnHide, 604
 CoNPrint, 602
 CoNUnFactorize, 618
 CoNWrite, 608
 CoOff, 601
 CoOn, 601
 CoOptimizeOption, 618
 CoPolyFun, 614
 CoPolyRatFun, 614
 CoPopHide, 603
 CoPrint, 602
 CoPrintB, 602
 CoProcessBucket, 615
 CoProperCount, 601
 CoPushHide, 602
 CoPutInside, 618
 CoRatio, 608
 CoRCycleSymmetrize, 608
 CoRedefine, 608
 CoRenumber, 609
 CoRepeat, 611
 CoReplaceLoop, 613
 CoSetExitFlag, 611
 CoSetUserFlag, 620
 CoSkip, 603
 CoSort, 614
 CoSplitArg, 607
 CoSplitFirstArg, 607
 CoSplitLastArg, 607
 CoShuffle, 615
 CoSum, 609
 CoSwitch, 619
 CoSymmetrize, 607
 CoTerm, 614
 CoThreadBucket, 615
 CoToPolynomial, 616
 CoToTensor, 609
 CoToVector, 609
 CoTrace4, 609
 CoTraceN, 609
 CoTryReplace, 611
 CoUnFactorize, 618
 CoUnHide, 604
 CoUnitTrace, 614
 CountComp, 612
 CoWhile, 613
 CoWrite, 608
 DoArgPlode, 615
 DoArgument, 606
 DoBrackets, 612
 DoChain, 610
 DoFactorize, 618
 DoFindLoop, 613
 DoInParallel, 610
 DoPrint, 602
 DoPutInside, 619
 DoSymmetrize, 607
 GetDoParam, 617
 GetIfDollarFactor, 617
 SetExpr, 603
 SetExprCases, 603
 setonoff, 600
 CompGroup
 declare.h, 818
 function.c, 1472
 CompileAlgebra
 compiler.c, 713
 declare.h, 945
 CompileLevel
 C_const, 57

- compiler.c, [710](#), [716](#)
 - alfatable1, [715](#)
 - CodeFactors, [714](#)
 - CodeGenerator, [714](#)
 - CompileAlgebra, [713](#)
 - CompileStatement, [714](#)
 - CompileSubExpressions, [714](#)
 - CompleteTerm, [714](#)
 - findcommand, [712](#)
 - GenerateFactors, [715](#)
 - inictable, [712](#)
 - insubexpbuffers, [715](#)
 - IsIdStatement, [713](#)
 - IsRHS, [713](#)
 - OPTION0, [712](#)
 - OPTION1, [712](#)
 - OPTION2, [712](#)
 - ParenthesesTest, [712](#)
 - REDUCESUBEXPBUFFERS, [712](#)
 - SkipAName, [713](#)
 - subexpbuffers, [715](#)
 - TestTables, [714](#)
 - topsubexpbuffers, [715](#)
- COMPILERBUFFER
 - fsizes.h, [1343](#)
- CompileStatement
 - compiler.c, [714](#)
 - declare.h, [908](#)
- CompileSubExpressions
 - compiler.c, [714](#)
 - declare.h, [949](#)
- compiletype
 - C_const, [54](#)
- COMPINC
 - fsizes.h, [1345](#)
- CompleteTerm
 - compiler.c, [714](#)
 - declare.h, [949](#)
- complex
 - FuNcTiOn, [144](#)
 - SyMbOl, [430](#)
 - VeCtOr, [471](#)
- compMat
 - MGraph, [210](#)
- ComposeTableNames
 - declare.h, [976](#)
 - minos.c, [1723](#)
- ComPress
 - declare.h, [911](#)
 - sort.c, [2719](#)
- compress.c, [742](#), [743](#)
- COMPRESSBUFFER
 - fsizes.h, [1342](#)
- CompressBuffer
 - R_const, [377](#)
- compression
 - ExPrEsSiOn, [123](#)
- CompressPointer
 - R_const, [377](#)
- CompressSize
 - InDeXeNtRy, [153](#)
 - M_const, [173](#)
- compressSize
 - N_const, [257](#)
- compressSpace
 - N_const, [253](#)
- ComprTop
 - R_const, [377](#)
- COMPTREE
 - structs.h, [2895](#)
- comsym
 - T_const, [441](#)
- comtool.c, [751](#), [757](#)
 - AddLHS, [753](#)
 - AddNtoC, [755](#)
 - AddNtoL, [754](#)
 - AddRHS, [754](#)
 - clearcbuf, [753](#)
 - ClearTree, [756](#)
 - DoubleCbuffer, [753](#)
 - FindTree, [756](#)
 - finishcbuf, [752](#)
 - inicbufs, [752](#)
 - IniFbuffer, [756](#)
 - InsTree, [756](#)
 - numcommute, [757](#)
 - RedoTree, [756](#)
- comtool.h, [764](#), [765](#)
- CoMulti
 - comexpr.c, [571](#)
 - declare.h, [933](#)
- CoMultiBracket
 - compcomm.c, [612](#)
 - declare.h, [924](#)
- CoMultiply
 - comexpr.c, [572](#)
 - declare.h, [933](#)
- CoNFactorize
 - compcomm.c, [618](#)
 - declare.h, [928](#)
- CoNFunction
 - declare.h, [933](#)
 - names.c, [1824](#)
- conid
 - EEdge, [99](#)
- CONJUGATION
 - ftypes.h, [1394](#)
- connComp
 - EGraph, [106](#)
- connect
 - Assign, [19](#)
- connectClass
 - MGraph, [211](#)
- connectLeg
 - MGraph, [211](#)
- connectNode

- MGraph, 211
- connVisit
 - EGraph, 106
- CoNoDrop
 - compcomm.c, 603
 - declare.h, 934
- CoNoHide
 - compcomm.c, 604
 - declare.h, 938
- CoNoIntoHide
 - compcomm.c, 604
 - declare.h, 938
- CoNormalize
 - compcomm.c, 606
 - declare.h, 934
- CoNoSkip
 - compcomm.c, 604
 - declare.h, 934
- CoNotInParallel
 - compcomm.c, 610
 - declare.h, 923
- CoNoUnHide
 - compcomm.c, 604
 - declare.h, 938
- CoNPrint
 - compcomm.c, 602
 - declare.h, 933
- ConstructName
 - declare.h, 1016
 - pre.c, 2327
- CoNTable
 - declare.h, 940
 - names.c, 1825
- CoNTensor
 - declare.h, 933
 - names.c, 1824
- ContentMerge
 - declare.h, 967
 - execute.c, 1155
- contents
 - DoLoOp, 91
- CONTENTTERM
 - ftypes.h, 1398
- CONTRACT
 - ftypes.h, 1414
- CoUnFactorize
 - compcomm.c, 618
 - declare.h, 929
- ConvertArgument
 - declare.h, 1001
 - ratio.c, 2502
- convertblock
 - declare.h, 973
 - minos.c, 1721
- ConvertFromPoly
 - declare.h, 994
 - notation.c, 1933
- convertiniinfo
 - declare.h, 973
 - minos.c, 1721
- convertnamesblock
 - declare.h, 973
 - minos.c, 1721
- ConvertParticle
 - diawrap.cc, 1051
- ConvertToPoly
 - declare.h, 994
 - notation.c, 1933
- ConWord
 - declare.h, 890
 - tools.c, 3078
- CoNWrite
 - compcomm.c, 608
 - declare.h, 933
- CoOff
 - compcomm.c, 601
 - declare.h, 934
- CoOn
 - compcomm.c, 601
 - declare.h, 934
- CoOnce
 - comexpr.c, 572
 - declare.h, 935
- CoOnly
 - comexpr.c, 572
 - declare.h, 935
- CoOptimizeOption
 - compcomm.c, 618
 - declare.h, 935
- CoParticle
 - declare.h, 1017
 - model.c, 1749
- CoPolyFun
 - compcomm.c, 614
 - declare.h, 935
- CoPolyRatFun
 - compcomm.c, 614
 - declare.h, 935
- CoPopHide
 - compcomm.c, 603
 - declare.h, 938
- CoPrint
 - compcomm.c, 602
 - declare.h, 935
- CoPrintB
 - compcomm.c, 602
 - declare.h, 935
- CoPrintTable
 - comexpr.c, 573
 - declare.h, 955
- CoProcessBucket
 - compcomm.c, 615
 - declare.h, 937
- CoProperCount
 - compcomm.c, 601
 - declare.h, 936

- CoPushHide
 - compcomm.c, [602](#)
 - declare.h, [937](#)
- CoPutInside
 - compcomm.c, [618](#)
 - declare.h, [924](#)
- copy
 - EEdge, [97](#)
 - EGraph, [104](#)
 - ENode, [116](#)
 - MNodeClass, [223](#)
- COPY1ARG
 - declare.h, [797](#)
- COPYARG
 - declare.h, [795](#)
- CopyArg
 - declare.h, [795](#)
- CopyExpression
 - declare.h, [977](#)
 - store.c, [2828](#)
- CopyFile
 - declare.h, [884](#)
 - tools.c, [3074](#)
- COPYFUN
 - declare.h, [795](#)
- COPYFUN3
 - declare.h, [796](#)
- COPYLONG
 - reken.c, [2547](#)
- COPYSUB
 - declare.h, [796](#)
- CopyTree
 - declare.h, [879](#)
 - names.c, [1821](#)
- CoRatio
 - compcomm.c, [608](#)
 - declare.h, [936](#)
- CoRCycleSymmetrize
 - compcomm.c, [608](#)
 - declare.h, [936](#)
- CoRedefine
 - compcomm.c, [608](#)
 - declare.h, [936](#)
- CoRemoveSpectator
 - declare.h, [1013](#)
 - spectator.c, [2782](#)
- CoRenumber
 - compcomm.c, [609](#)
 - declare.h, [936](#)
- CoRepeat
 - compcomm.c, [611](#)
 - declare.h, [936](#)
- CoReplaceLoop
 - compcomm.c, [613](#)
 - declare.h, [953](#)
- CoSave
 - declare.h, [937](#)
 - store.c, [2824](#)
- CoSelect
 - comexpr.c, [572](#)
 - declare.h, [937](#)
- CoSet
 - declare.h, [937](#)
 - names.c, [1826](#)
- CoSetExitFlag
 - compcomm.c, [611](#)
 - declare.h, [937](#)
- CoSetUserFlag
 - compcomm.c, [620](#)
 - declare.h, [1018](#)
- COSFUNCTION
 - ftypes.h, [1396](#)
- COSHFUNCTION
 - ftypes.h, [1397](#)
- CoSkip
 - compcomm.c, [603](#)
 - declare.h, [937](#)
- CoSort
 - compcomm.c, [614](#)
 - declare.h, [939](#)
- CoSplitArg
 - compcomm.c, [607](#)
 - declare.h, [939](#)
- CoSplitFirstArg
 - compcomm.c, [607](#)
 - declare.h, [939](#)
- CoSplitLastArg
 - compcomm.c, [607](#)
 - declare.h, [939](#)
- COST, [75](#)
 - add, [76](#)
 - div, [76](#)
 - mul, [76](#)
 - pow, [76](#)
- CoStrictRounding
 - float.c, [1290](#)
- CoStuffle
 - compcomm.c, [615](#)
 - declare.h, [932](#)
- CoSum
 - compcomm.c, [609](#)
 - declare.h, [939](#)
- CoSwitch
 - compcomm.c, [619](#)
 - declare.h, [943](#)
- CoSymbol
 - declare.h, [939](#)
 - names.c, [1822](#)
- CoSymmetrize
 - compcomm.c, [607](#)
 - declare.h, [939](#)
- CoTable
 - declare.h, [940](#)
 - names.c, [1825](#)
- CoTableBase
 - declare.h, [925](#)

- tables.c, 2955
- CoTBaddto
 - declare.h, 943
 - tables.c, 2955
- CoTBaudit
 - declare.h, 943
 - tables.c, 2956
- CoTbcleanup
 - declare.h, 944
 - tables.c, 2957
- CoTBcreate
 - declare.h, 944
 - tables.c, 2955
- CoTBenter
 - declare.h, 944
 - tables.c, 2956
- CoTBhelp
 - declare.h, 944
 - tables.c, 2957
- CoTBload
 - declare.h, 944
 - tables.c, 2956
- CoTBoff
 - declare.h, 944
 - tables.c, 2957
- CoTBon
 - declare.h, 944
 - tables.c, 2956
- CoTBopen
 - declare.h, 945
 - tables.c, 2955
- CoTBreplace
 - declare.h, 945
 - tables.c, 2957
- CoTBuse
 - declare.h, 945
 - tables.c, 2957
- CoTerm
 - compcomm.c, 614
 - declare.h, 940
- CoTestUse
 - declare.h, 945
 - tables.c, 2956
- CoThreadBucket
 - compcomm.c, 615
 - declare.h, 945
- CoToFloat
 - float.c, 1289
- CoToPolynomial
 - compcomm.c, 616
 - declare.h, 995
- CoToRat
 - float.c, 1290
- CoToSpectator
 - declare.h, 1013
 - spectator.c, 2781
- CoToTensor
 - compcomm.c, 609
- declare.h, 940
- CoToVector
 - compcomm.c, 609
 - declare.h, 941
- CoTrace4
 - compcomm.c, 609
 - declare.h, 941
- CoTraceN
 - compcomm.c, 609
 - declare.h, 941
- CoTransform
 - declare.h, 941
 - transform.c, 3135
- CoTryReplace
 - compcomm.c, 611
 - declare.h, 942
- COULDCOMMUTE
 - ftypes.h, 1450
- CoUnFactorize
 - compcomm.c, 618
 - declare.h, 928
- CoUnHide
 - compcomm.c, 604
 - declare.h, 938
- CoUnitTrace
 - compcomm.c, 614
 - declare.h, 936
- count
 - PNodeClass, 342
- count_operators
 - optimize.cc, 1984, 1985
- count_operators_cse
 - optimize.cc, 1990
- count_operators_cse_topdown
 - optimize.cc, 1991
- CountComp
 - compcomm.c, 612
 - declare.h, 922
- CountDo
 - declare.h, 819
 - reshuf.c, 2606
- counter
 - ExPrEsSiOn, 122
- CountFun
 - declare.h, 819
 - reshuf.c, 2606
- COUNTFUNCTION
 - ftypes.h, 1393
- countfunnum
 - M_const, 188
- CountTerms1
 - declare.h, 972
 - execute.c, 1155
- coupl
 - FGInput, 127
- couple
 - PNodeClass, 341
- couplings

- MoDeL, [227](#)
- VeRtEx, [473](#)
- CoVector
 - declare.h, [942](#)
 - names.c, [1823](#)
- CoVertex
 - declare.h, [1018](#)
 - model.c, [1749](#)
- CoWhile
 - compcomm.c, [613](#)
 - declare.h, [942](#)
- CoWrite
 - compcomm.c, [608](#)
 - declare.h, [942](#)
- cple
 - NCInput, [271](#)
 - PGInput, [337](#)
- cpigcp
 - Model, [236](#)
- cpiglg
 - Model, [236](#)
- cpignvl
 - Model, [236](#)
- cpigvl
 - Model, [236](#)
- cpleft
 - Assign, [26](#)
- cprocedureExtension
 - P_const, [318](#)
- CRANGE
 - ftypes.h, [1376](#)
- Crash
 - declare.h, [972](#)
 - tools.c, [3087](#)
- CreateExpression
 - declare.h, [1000](#)
 - ratio.c, [2500](#)
- CreateFile
 - declare.h, [883](#)
 - tools.c, [3074](#)
- CreateHandle
 - declare.h, [884](#)
 - tools.c, [3075](#)
- CreateLogFile
 - declare.h, [883](#)
 - tools.c, [3074](#)
- CreateStream
 - declare.h, [916](#)
 - tools.c, [3073](#)
- CreateVertex
 - declare.h, [1017](#)
 - model.c, [1748](#)
- csav
 - SGroup, [394](#)
- CSEEq, [76](#)
 - operator(), [77](#)
- CSEHash, [77](#)
 - operator(), [77](#)
- CSET
 - ftypes.h, [1375](#)
- CSUBEXP
 - ftypes.h, [1376](#)
- csum
 - Interaction, [160](#)
- CSYMBOL
 - ftypes.h, [1375](#)
- cTable
 - FixedGlobals, [136](#)
- CTD
 - minos.c, [1720](#)
- cTerm
 - N_const, [248](#)
- ctloop
 - MCOpI, [205](#)
- ctotal
 - Process, [371](#)
- ctyp
 - PGInput, [337](#)
- CurBufWrt
 - O_const, [282](#)
- curcl
 - MGraph, [213](#)
- CurDictFunWithArgs
 - O_const, [286](#)
- CurDictInDollars
 - O_const, [287](#)
- CurDictNotInFunctions
 - O_const, [286](#)
- CurDictNumbers
 - O_const, [286](#)
- CurDictNumberWarning
 - O_const, [286](#)
- CurDictSpecials
 - O_const, [286](#)
- CurDictVariables
 - O_const, [286](#)
- curdirp
 - setfile.c, [2687](#)
- CurDum
 - R_const, [380](#)
- CurExpr
 - R_const, [382](#)
- current
 - TabIEbAsE, [446](#)
- CURRENTBRACKET
 - execute.c, [1152](#)
- CurrentDictionary
 - O_const, [285](#)
- currentExternalChannel
 - X_const, [475](#)
- currentMODEL
 - TERMINFO, [455](#)
- currentModel
 - TERMINFO, [455](#)
- currentPrompt
 - X_const, [475](#)

- CurrentStream
 - C_const, [47](#)
- currentTerm
 - N_const, [253](#)
- cursortdirp
 - setfile.c, [2687](#)
- cut
 - EEdge, [99](#)
- cvallist
 - lInput, [149](#)
- CVECTOR
 - ftypes.h, [1375](#)
- CVECTOR1
 - ftypes.h, [1376](#)
- cycle
 - DiStRiBuTe, [86](#)
 - PeRmUtE, [333](#)
 - PeRmUtEp, [334](#)
- CYCLE1
 - declare.h, [794](#)
- CYCLEARG
 - ftypes.h, [1445](#)
- CYCLESYMMETRIC
 - ftypes.h, [1431](#)
- CYCLI
 - ftypes.h, [1456](#)
- CYCLR
 - ftypes.h, [1456](#)
- d
 - flint::fmpz, [138](#)
 - flint::mpoly, [242](#)
 - flint::mpoly_ctx, [243](#)
 - flint::mpoly_factor, [244](#)
 - flint::poly, [343](#)
 - flint::poly_factor, [362](#)
- DAEMON
 - pre.c, [2310](#)
- daemonize
 - X_const, [475](#)
- data
 - node, [276](#)
- date
 - objects, [290](#)
- dbase, [78](#)
 - fullname, [80](#)
 - handle, [79](#)
 - iblocks, [79](#)
 - info, [78](#)
 - mode, [78](#)
 - name, [79](#)
 - nblocks, [79](#)
 - rwmode, [79](#)
 - tablenameefill, [79](#)
 - tablenames, [80](#)
 - tablenamessize, [79](#)
 - topnumber, [79](#)
- DBGOUT
 - parallel.c, [2059](#)
- DBGOUT_NINTERMS
 - parallel.c, [2059](#)
- dbufnum
 - M_const, [178](#)
- DeAllocFileHandle
 - declare.h, [962](#)
 - setfile.c, [2686](#)
- DEBUG
 - wildcard.c, [3211](#)
- DebugFlag
 - P_const, [322](#)
- debugFlags
 - C_const, [70](#)
- debugflags
 - OPTIMIZE, [297](#)
- DECLARATION
 - ftypes.h, [1372](#)
- declare.h, [766](#), [1021](#)
 - ABS, [788](#)
 - AccumGCD, [812](#)
 - Add2Com, [803](#)
 - Add2ComStrings, [917](#)
 - ADD2POS, [800](#)
 - Add3Com, [803](#)
 - Add4Com, [803](#)
 - Add5Com, [803](#)
 - AddArgs, [812](#)
 - AddArrayIndex, [811](#)
 - AddCoef, [813](#)
 - AddComString, [945](#)
 - AddDictionary, [1010](#)
 - AddDollar, [880](#)
 - AddDubious, [881](#)
 - AddExpression, [880](#)
 - AddFunction, [881](#)
 - AddIndex, [881](#)
 - AddLHS, [914](#)
 - AddLineFeed, [790](#)
 - AddLong, [813](#)
 - AddName, [877](#)
 - AddNtoC, [915](#)
 - AddNtoL, [915](#)
 - AddObject, [976](#)
 - AddPLon, [813](#)
 - AddPoly, [814](#)
 - ADDPOS, [799](#)
 - AddPotModdollar, [957](#)
 - AddRat, [815](#)
 - AddRHS, [914](#)
 - AddSet, [882](#)
 - AddSymbol, [880](#)
 - AddTableName, [976](#)
 - AddToCB, [794](#)
 - AddToCom, [917](#)
 - AddToDictionary, [1010](#)
 - AddToDollarBuffer, [951](#)
 - AddToIndex, [974](#)
 - AddToLine, [815](#)

- AddToPreTypes, [913](#)
- AddToScratch, [1014](#)
- AddToString, [905](#)
- AddVector, [881](#)
- AddWild, [815](#)
- AddWildcardName, [912](#)
- AdjustRenumScratch, [835](#)
- AllLoops, [1019](#)
- AllocFileHandle, [962](#)
- AllocSetups, [892](#)
- AllocSort, [892](#)
- AllocSortFileName, [892](#)
- AllPaths, [1020](#)
- Apply, [826](#)
- ApplyExec, [826](#)
- ApplyReset, [826](#)
- AreArgsEqual, [889](#)
- ArgDotproductMerge, [921](#)
- ArgFactorize, [918](#)
- ArgSymbolMerge, [920](#)
- ArgumentExplode, [971](#)
- ArgumentImplode, [971](#)
- AssignDollar, [951](#)
- BACKINOUT, [794](#)
- BASEPOSITION, [801](#)
- Bernoulli, [832](#)
- BigLong, [815](#)
- BinarySearch, [963](#)
- BinomGen, [816](#)
- BracketNormalize, [846](#)
- CacheNumberFree, [805](#)
- CacheNumberMalloc, [805](#)
- CacheNumberMallocAddMemory, [989](#)
- CatchDollar, [951](#)
- ChainIn, [971](#)
- ChainOut, [971](#)
- CharOut, [893](#)
- CheckRecoveryFile, [986](#)
- CheckTableDeclarations, [826](#)
- CheckWild, [816](#)
- Chisholm, [816](#)
- CleanDollarFactors, [955](#)
- CleanExpr, [816](#)
- CleanUp, [816](#)
- CleanupArgCache, [920](#)
- CleanupSort, [962](#)
- CleanupTerm, [966](#)
- ClearBracketIndex, [969](#)
- clearcbuf, [961](#)
- ClearMacro, [896](#)
- ClearOptimize, [959](#)
- ClearPushback, [892](#)
- ClearSpectators, [1013](#)
- ClearTableTree, [960](#)
- ClearTree, [950](#)
- ClearWild, [817](#)
- ClearWildcardNames, [912](#)
- closeAllExternalChannels, [979](#)
- CloseChannel, [886](#)
- closeExternalChannel, [979](#)
- CloseFile, [883](#)
- CloseStream, [876](#)
- CmpArray, [890](#)
- CoAntiBracket, [922](#)
- CoAntiPutInside, [924](#)
- CoAntiSymmetrize, [922](#)
- CoApply, [925](#)
- CoArgExplode, [922](#)
- CoArgImplode, [922](#)
- CoArgToExtraSymbol, [996](#)
- CoArgument, [923](#)
- CoAssign, [949](#)
- CoAuto, [942](#)
- CoBracket, [924](#)
- CoBreak, [943](#)
- CoCanonicalize, [823](#)
- CoCase, [943](#)
- CoCFunction, [924](#)
- CoChainin, [942](#)
- CoChainout, [942](#)
- CoChisholm, [941](#)
- CoClearTable, [941](#)
- CoClearUserFlag, [1018](#)
- CoCollect, [925](#)
- CoCommutelnSet, [882](#)
- CoCompress, [925](#)
- CoContract, [925](#)
- CoCopySpectator, [1013](#)
- CoCreateAll, [1019](#)
- CoCreateAllLoops, [1018](#)
- CoCreateAllPaths, [1018](#)
- CoCreateSpectator, [1012](#)
- CoCTable, [940](#)
- CoCTensor, [924](#)
- CoCycleSymmetrize, [925](#)
- CoDeallocateTable, [955](#)
- CodeFactors, [950](#)
- CoDefault, [943](#)
- CodeGenerator, [949](#)
- CoDelete, [925](#)
- CoDenominators, [926](#)
- CodeToLine, [810](#)
- CoDimension, [926](#)
- CoDiscard, [926](#)
- CoDisorder, [926](#)
- CoDo, [996](#)
- CoDrop, [926](#)
- CoDropCoefficient, [926](#)
- CoDropSymbols, [926](#)
- CoElse, [927](#)
- CoElseif, [927](#)
- CoEmptySpectator, [1013](#)
- CoEndArgument, [927](#)
- CoEndDo, [996](#)
- CoEndIf, [927](#)
- CoEndInExpression, [924](#)

- CoEndInside, [927](#)
- CoEndModel, [1018](#)
- CoEndRepeat, [927](#)
- CoEndSwitch, [943](#)
- CoEndTerm, [927](#)
- CoEndWhile, [928](#)
- CoExit, [928](#)
- CoExtraSymbols, [996](#)
- CoFactArg, [928](#)
- CoFactDollar, [928](#)
- CoFactorize, [928](#)
- CoFill, [929](#)
- CoFillExpression, [929](#)
- CoFindLoop, [953](#)
- CoFixIndex, [929](#)
- CoFlags, [934](#)
- CoFormat, [929](#)
- CoFromPolynomial, [995](#)
- CoFunction, [882](#)
- CoFunPowers, [953](#)
- CoGlobal, [929](#)
- CoGlobalFactorized, [930](#)
- CoGoTo, [930](#)
- CoHide, [938](#)
- Cold, [930](#)
- ColdExpression, [949](#)
- ColdNew, [930](#)
- ColdOld, [930](#)
- Colf, [930](#)
- ColfMatch, [930](#)
- ColfNoMatch, [931](#)
- ColIndex, [931](#)
- ColnExpression, [923](#)
- ColnParallel, [923](#)
- ColnInside, [923](#)
- ColnInsideFirst, [931](#)
- ColnIntoHide, [938](#)
- CoKeep, [931](#)
- CoLabel, [931](#)
- CoLoad, [931](#)
- CoLocal, [931](#)
- CoLocalFactorized, [932](#)
- CoMakeInteger, [934](#)
- CoMany, [932](#)
- CoMerge, [932](#)
- CoMetric, [932](#)
- Commute, [817](#)
- CoModel, [1017](#)
- CoModOption, [932](#)
- CoModuleOption, [932](#)
- CoModulus, [933](#)
- CompactifyTree, [879](#)
- Compare1, [818](#)
- CompareArgs, [889](#)
- CompareFunctions, [817](#)
- CompareHSymbols, [980](#)
- CompareSymbols, [980](#)
- CompareTerms, [808](#)
- CompArg, [817](#)
- CompCoef, [818](#)
- CompGroup, [818](#)
- CompileAlgebra, [945](#)
- CompileStatement, [908](#)
- CompileSubExpressions, [949](#)
- CompleteTerm, [949](#)
- ComposeTableNames, [976](#)
- ComPress, [911](#)
- CoMulti, [933](#)
- CoMultiBracket, [924](#)
- CoMultiply, [933](#)
- CoNFactorize, [928](#)
- CoNFunction, [933](#)
- CoNoDrop, [934](#)
- CoNoHide, [938](#)
- CoNoIntoHide, [938](#)
- CoNormalize, [934](#)
- CoNoSkip, [934](#)
- CoNotInParallel, [923](#)
- CoNoUnHide, [938](#)
- CoNPrint, [933](#)
- ConstructName, [1016](#)
- CoNTable, [940](#)
- CoNTensor, [933](#)
- ContentMerge, [967](#)
- CoNUnFactorize, [929](#)
- ConvertArgument, [1001](#)
- convertblock, [973](#)
- ConvertFromPoly, [994](#)
- convertiniinfo, [973](#)
- convertnamesblock, [973](#)
- ConvertToPoly, [994](#)
- ConWord, [890](#)
- CoNWrite, [933](#)
- CoOff, [934](#)
- CoOn, [934](#)
- CoOnce, [935](#)
- CoOnly, [935](#)
- CoOptimizeOption, [935](#)
- CoParticle, [1017](#)
- CoPolyFun, [935](#)
- CoPolyRatFun, [935](#)
- CoPopHide, [938](#)
- CoPrint, [935](#)
- CoPrintB, [935](#)
- CoPrintTable, [955](#)
- CoProcessBucket, [937](#)
- CoProperCount, [936](#)
- CoPushHide, [937](#)
- CoPutInside, [924](#)
- COPY1ARG, [797](#)
- COPYARG, [795](#)
- CopyArg, [795](#)
- CopyExpression, [977](#)
- CopyFile, [884](#)
- COPYFUN, [795](#)
- COPYFUN3, [796](#)

- COPYSUB, [796](#)
- CopyTree, [879](#)
- CoRatio, [936](#)
- CoRCycleSymmetrize, [936](#)
- CoRedefine, [936](#)
- CoRemoveSpectator, [1013](#)
- CoReNumber, [936](#)
- CoRepeat, [936](#)
- CoReplaceLoop, [953](#)
- CoSave, [937](#)
- CoSelect, [937](#)
- CoSet, [937](#)
- CoSetExitFlag, [937](#)
- CoSetUserFlag, [1018](#)
- CoSkip, [937](#)
- CoSort, [939](#)
- CoSplitArg, [939](#)
- CoSplitFirstArg, [939](#)
- CoSplitLastArg, [939](#)
- CoStuffle, [932](#)
- CoSum, [939](#)
- CoSwitch, [943](#)
- CoSymbol, [939](#)
- CoSymmetrize, [939](#)
- CoTable, [940](#)
- CoTableBase, [925](#)
- CoTBaddto, [943](#)
- CoTBaudit, [943](#)
- CoTbcleanup, [944](#)
- CoTBcreate, [944](#)
- CoTBenter, [944](#)
- CoTBhelp, [944](#)
- CoTBload, [944](#)
- CoTBoff, [944](#)
- CoTBon, [944](#)
- CoTBopen, [945](#)
- CoTBreplace, [945](#)
- CoTBuse, [945](#)
- CoTerm, [940](#)
- CoTestUse, [945](#)
- CoThreadBucket, [945](#)
- CoToPolynomial, [995](#)
- CoToSpectator, [1013](#)
- CoToTensor, [940](#)
- CoToVector, [941](#)
- CoTrace4, [941](#)
- CoTraceN, [941](#)
- CoTransform, [941](#)
- CoTryReplace, [942](#)
- CoUnFactorize, [928](#)
- CoUnHide, [938](#)
- CoUnitTrace, [936](#)
- CountComp, [922](#)
- CountDo, [819](#)
- CountFun, [819](#)
- CountTerms1, [972](#)
- CoVector, [942](#)
- CoVertex, [1018](#)
- CoWhile, [942](#)
- CoWrite, [942](#)
- Crash, [972](#)
- CreateExpression, [1000](#)
- CreateFile, [883](#)
- CreateHandle, [884](#)
- CreateLogFile, [883](#)
- CreateStream, [916](#)
- CreateVertex, [1017](#)
- CYCLE1, [794](#)
- DeAllocFileHandle, [962](#)
- Deferred, [820](#)
- defineChannel, [979](#)
- DeleteObject, [975](#)
- DeleteRecoveryFile, [987](#)
- DeleteStore, [821](#)
- DenToFunction, [972](#)
- DetCommu, [817](#)
- DetCurDum, [821](#)
- DetVars, [821](#)
- DictFromBytes, [1012](#)
- DictToBytes, [1012](#)
- DIFBASE, [800](#)
- DIFPOS, [800](#)
- DimensionExpression, [820](#)
- DimensionSubterm, [819](#)
- DimensionTerm, [819](#)
- Distribute, [821](#)
- DistrN, [1016](#)
- DIVfunction, [1000](#)
- DivLong, [822](#)
- DIVPOS, [800](#)
- DivRat, [822](#)
- Divvy, [822](#)
- DoArgPlode, [922](#)
- DoArgument, [917](#)
- DoBrackets, [921](#)
- DoBreakDo, [902](#)
- DoCall, [902](#)
- DoCanonicalize, [823](#)
- DoChain, [941](#)
- DoCheckpoint, [988](#)
- DoClearOptimize, [898](#)
- DoClearUserFlag, [1017](#)
- DoCommentChar, [904](#)
- DoContinueDo, [902](#)
- DoDebug, [902](#)
- DoDefine, [894](#)
- DoDelta, [822](#)
- DoDelta3, [822](#)
- DoDimension, [881](#)
- DoDistrib, [824](#)
- DoDo, [902](#)
- DoElements, [882](#)
- DoElse, [901](#)
- DoElseif, [901](#)
- DoEnddo, [902](#)
- DoEndif, [901](#)

- DoEndInside, [903](#)
- DoEndNamespace, [1016](#)
- DoEndprocedure, [903](#)
- DoEndSwitch, [1015](#)
- DoesCommu, [817](#)
- DoExecStatement, [908](#)
- DoExecute, [873](#)
- DoExpr, [949](#)
- DoExternal, [896](#)
- DoFactDollar, [897](#)
- DoFactorize, [929](#)
- DoFindLoop, [953](#)
- DoFromExternal, [897](#)
- Dolf, [901](#)
- Dolfdef, [900](#)
- Dolfndef, [901](#)
- DolfStatement, [827](#)
- Dolfydef, [901](#)
- DoInclude, [896](#)
- DoInParallel, [923](#)
- DoinParallel, [970](#)
- DoInside, [903](#)
- DollarFactorize, [955](#)
- DOLLARNAME, [798](#)
- DollarRaiseLow, [881](#)
- DoToFunction, [954](#)
- DoToIndex, [954](#)
- DoToLong, [955](#)
- DoToNumber, [954](#)
- DoToSymbol, [954](#)
- DoToTensor, [954](#)
- DoToTerms, [963](#)
- DoToVector, [954](#)
- DoMessage, [898](#)
- DoModDollar, [970](#)
- DoModLocal, [970](#)
- DoModMax, [970](#)
- DoModMin, [970](#)
- DoModSum, [970](#)
- DoNamespace, [1016](#)
- DoNoParallel, [969](#)
- DonotinParallel, [971](#)
- DoOnePow, [827](#)
- DoOptimize, [898](#)
- DoParallel, [969](#)
- DoPartitions, [823](#)
- DoPermutations, [824](#)
- DoPipe, [900](#)
- DoPipeStatement, [908](#)
- DoPolyfun, [908](#)
- DoPolyratfun, [908](#)
- DoPrcExtension, [904](#)
- DoPreAdd, [1009](#)
- DoPreAddSeparator, [978](#)
- DoPreAppend, [899](#)
- DoPreAppendPath, [1014](#)
- DoPreAssign, [899](#)
- DoPreBreak, [899](#)
- DoPreCase, [900](#)
- DoPreClose, [903](#)
- DoPreCloseDictionary, [1009](#)
- DoPreCreate, [899](#)
- DoPreDefault, [899](#)
- DoPreEndSwitch, [899](#)
- DoPreExchange, [900](#)
- DoPreOpenDictionary, [1009](#)
- DoPreOut, [898](#)
- DoPrePrependPath, [1014](#)
- DoPrePrintTimes, [903](#)
- DoPreRemove, [904](#)
- DoPreReset, [904](#)
- DoPreRmSeparator, [978](#)
- DoPreShow, [900](#)
- DoPreSortReallocate, [904](#)
- DoPreSwitch, [899](#)
- DoPreUseDictionary, [1009](#)
- DoPreWrite, [903](#)
- DoPrint, [917](#)
- DoProcedure, [903](#)
- DoProcessBucket, [970](#)
- DoPrompt, [897](#)
- DoPutInside, [921](#)
- DoRecovery, [987](#)
- DoRedefine, [895](#)
- DoReverseInclude, [896](#)
- DoRevert, [829](#)
- DoRmExternal, [897](#)
- DoSetExternal, [897](#)
- DoSetExternalAttr, [897](#)
- DoSetRandom, [898](#)
- DoSetups, [877](#)
- DoSetUserFlag, [1017](#)
- DoShattering, [824](#)
- DoShtuffle, [809](#)
- DoShuffle, [824](#)
- DoSkipExtraSymbols, [898](#)
- DoStuffle, [825](#)
- DoSumF1, [829](#)
- DoSumF2, [829](#)
- DoSwitch, [1015](#)
- DoSymmetrize, [917](#)
- DoSystem, [900](#)
- DoTable, [940](#)
- DoTableExpansion, [824](#)
- DoTail, [874](#)
- DoTempSet, [882](#)
- DoTerminate, [901](#)
- DoTheta, [829](#)
- DoTimeOutAfter, [898](#)
- DoToExternal, [896](#)
- DoTopologyCanonicalize, [823](#)
- DoubleBuffer, [888](#)
- DoubleCbuffer, [913](#)
- DoubleIfBuffers, [916](#)
- DoubleList, [888](#)
- DoubleLList, [888](#)

DoubleSwitchBuffers, 1015
DoUndefine, 896
DoUse, 1016
DropCoefficient, 846
DropSymbols, 847
DUMMYUSE, 799
DumpNode, 879
DumpTree, 879
EmptyTable, 940
EndOfToken, 907
EndSort, 830
EntVar, 831
EpfCon, 831
EpfFind, 831
EpfGen, 831
EqualArg, 831
Error0, 875
Error1, 875
Error2, 875
EvalDoLoopArg, 963
EvalPrelf, 906
EXCH, 790
ExchangeDollars, 965
ExchangeExpressions, 965
EXCHINOUT, 794
EXCHN, 790
execarg, 872
ExecInside, 923
ExecModule, 907
ExecStore, 908
execterm, 872
ExistsObject, 975
ExpandBuffer, 888
ExpandRat, 1001
ExpandTripleDots, 911
EXPRNAME, 798
ExprStatus, 989
EXTERNLOCK, 806
EXTERNRWLOCK, 807
ExtraSymbol, 846
ExtraSymFun, 997
Factorial, 831
FactorIn, 832
FactorInExpr, 832
FILLARG, 795
FILLEXP, 796
FILLFUN, 795
FILLFUN3, 796
FILLSUB, 796
FindAll, 832
FindArg, 919
FindBracket, 969
FindCase, 1015
findcommand, 886
FindDictionary, 1010
FindExtraSymbol, 1012
FindFunction, 1012
FindFunWithArgs, 1012
FindIndex, 1012
FindInIndex, 811
FindInKeyWord, 894
FindKeyWord, 894
FindLocalSubterm, 994
FindLus, 952
FindMulti, 832
FindOnce, 832
FindOnly, 833
FindRange, 991
FindRest, 833
FindrNumber, 833
FindSubexpression, 995
FindSubterm, 994
FindSymbol, 1011
FindTableNumber, 977
FindTableTree, 961
FindTB, 826
FindTree, 950
FindVar, 827
FindVector, 1011
FiniLine, 833
finishcbuf, 961
FinishShuffle, 825
FinishStuffle, 825
FiniShuffle, 808
FiniTerm, 833
FlipTable, 971
FLUSHCONSOLE, 793
FlushFile, 885
FlushOut, 834
FlushSpectators, 1014
FORM_INLINE, 804
FreeNameTree, 880
FreeTableBase, 976
From0List, 887
FromList, 887
FromPolyRatFun, 998
FromVarList, 888
FullCleanUp, 908
FullRenummer, 952
FullSymmetrize, 863
FunLevel, 834
FunMatchCy, 842
FunMatchSy, 843
GarbHand, 835
GCDclean, 998
GCDfunction, 997
GCDfunction3, 997
GcdLong, 835
GCDterms, 998
GenDiagrams, 823
GenerateFactors, 950
Generator, 835
GenLoops, 1019
GenPaths, 1020
GetAppendChannel, 886
GetAutoName, 878

GETBIDENTITY, 808
GetBinom, 837
GETCFROMEXTCHANNEL, 809
GetChannel, 886
GetChar, 893
GETCOEF, 789
GetContent, 966
getCurrentExternalChannel, 979
GetDbase, 974
GetDollar, 909
GetDollarNumber, 897
GetDolNum, 957
GetDoParam, 996
GetFirstBracket, 966
GetFirstTerm, 966
GetFromSpectator, 1013
GetFromStore, 838
GetFromStream, 875
GetFunction, 878
GETIDENTITY, 808
GetIfDollarFactor, 996
GetIfDollarNum, 827
GetInput, 892
GetLabel, 948
GetLastExprName, 878
GetLong, 838
GetLongModInverses, 856
GetModInverses, 855
GetMoreFromMem, 838
GetMoreTerms, 838
GetName, 877
GetNode, 877
GetNumber, 878
GetOName, 879
GetOneTerm, 838
GetPosFile, 885
GetPreVar, 891
GetRunningTime, 852
GetSetupPar, 891
GETSTOP, 789
GetStreamPosition, 913
GetTable, 839
GetTableName, 977
GETTERM, 809
GetTerm, 839
GetVar, 878
GetWildcardName, 912
Globalize, 913
Glue, 839
HighWarning, 873
HowMany, 972
iexp, 888
INCLENG, 789
Include, 896
InFunction, 839
inicbufs, 886
inictable, 886
IniBuffer, 997
IniLine, 840
INILOCK, 806
IniModule, 893
INIRWLOCK, 807
IniSpecialModule, 893
iniTools, 989
initPresetExternalChannels, 978
InitRecovery, 986
IniVars, 840
iniwranf, 981
InsertArg, 919
InsertTerm, 840
InsideDollar, 963
InsTableTree, 961
InsTree, 950
InvPoly, 1001
iranf, 981
ISEQUALPOS, 801
ISEQUALPOSINC, 799
IsExponentSign, 1010
ISGEPOS, 802
ISGEPOSINC, 800
IsIdStatement, 946
ISLESSPOS, 802
IsLikeVector, 889
ISMINPOS, 801
IsMultipleOf, 968
IsMultiplySign, 1011
ISNEGPOS, 802
ISNOTEQUALPOS, 801
ISNOTZEROPOS, 802
ISPOSPOS, 802
IsRHS, 946
IsSetMember, 968
ISZEROPOS, 802
LcmLong, 835
LinkTree, 879
LoadInputFile, 892
LoadInstruction, 894
LoadModel, 1018
LoadStatement, 894
LocalConvertToPoly, 994
LocateBase, 860
LocateFile, 876
LOCK, 807
LongCopy, 910
LongLongCopy, 910
LongToLine, 841
LookInStream, 875
LoopOutput, 1019
LowerSortLevel, 962
Lus, 952
M_alloc, 808
M_check1, 968
M_free, 912
M_print, 968
MakeDate, 887
MakeDirty, 841

MakeDollarInteger, 955
MakeDollarMod, 956
MakeDubious, 878
MakeGlobal, 907
MakeInteger, 918
MakeInverses, 855
MakeLongRational, 960
MakeMod, 919
MakeModTable, 842
MakeNameTree, 880
MakeRational, 960
MakeSetupAllocs, 965
Malloc1, 874
MarkDirty, 841
MarkPolyRatFunDirty, 806
MatchArgument, 843
MatchCy, 842
MatchE, 842
MatchFunction, 843
MatchIsPossible, 953
MaX, 788
MEMORYMACROS, 804
MergeDotproductLists, 1000
MergePatches, 843
MergeSymbolLists, 1000
MesCall, 909
MesCerr, 844
MesComp, 844
MesNesting, 805
MesPrint, 909
MessPreNesting, 913
MesWork, 909
MiN, 788
MLOCK, 808
ModuleInstruction, 893
Modulus, 844
Moebius, 981
MoveDummies, 845
MULfunc, 1000
MulLong, 845
Mully, 845
MULPOS, 800
MulRat, 845
MultDo, 845
MultiplyToLine, 1011
MultiplyWithTerm, 999
MUNLOCK, 808
NameConflict, 883
NCOPY, 791
NCOPYB, 792
NCOPYI, 791
NCOPYI32, 792
NeedNumber, 792
NestingChecksum, 805
NewDbase, 976
NewSort, 846
NEXTARG, 796
NextFileIndex, 811
NextPrime, 981
Normalize, 846
NormalModulus, 855
NormPolyTerm, 993
NOTSTARTPOS, 801
NumberFree, 805
NumberMalloc, 804
NumberMallocAddMemory, 988
numcommute, 952
NumCopy, 910
NumToStr, 905
OpenAddFile, 883
OpenBracketIndex, 969
OpenDbase, 975
openExternalChannel, 978
OpenFile, 883
OpenInput, 874
OpenStream, 876
OpenTemp, 847
Optimize, 957
optimize_print_code, 1009
Pack, 847
ParenthesesTest, 946
ParseNumber, 791
ParseSign, 791
ParseSignedNumber, 791
PasteFile, 847
PasteTerm, 811
PathOutput, 1020
Permute, 848
PermuteP, 848
poly_div, 1002
poly_factorize_argument, 1005
poly_factorize_dollar, 1006
poly_factorize_expression, 1006
poly_free_poly_vars, 1008
poly_gcd, 1001
poly_inverse, 1003
poly_mul, 1003
poly_ratfun_add, 1003
poly_ratfun_normalize, 1004
poly_rem, 1002
poly_unfactorize_expression, 1007
PolyDiv, 998
PolyFunClean, 842
PolyFunDirty, 842
PolyFunMul, 848
PolyRatFunSpecial, 962
POPPREASSIGNLEVEL, 806
PopPreVars, 893
PopVariables, 849
PositionStream, 876
pParseObject, 906
PreCalc, 905
PreCmp, 905
PreEq, 905
PreEval, 905
PreIfDollarEval, 967

- PrelfEval, [906](#)
- PreLoad, [906](#)
- PrepPoly, [849](#)
- PreProcessor, [887](#)
- PreProInstruction, [894](#)
- PreRandom, [982](#)
- PreSkip, [906](#)
- PREV, [798](#)
- PrintDeprecation, [852](#)
- PrintExtraSymbol, [995](#)
- PrintRunningTime, [852](#)
- PrintSeq, [911](#)
- PrintSubTerm, [911](#)
- PrintSubtermList, [995](#)
- PrintTerm, [910](#)
- PrintTermC, [910](#)
- PrintTime, [969](#)
- PrintWords, [911](#)
- ProcessOption, [876](#)
- Processor, [850](#)
- Product, [851](#)
- PrtLong, [851](#)
- PrtTerms, [851](#)
- PruneExtraSymbols, [997](#)
- PUSHPREASSIGNLEVEL, [806](#)
- PutArgInScratch, [991](#)
- PutBracket, [852](#)
- PutBracketInIndex, [969](#)
- PutExtraSymbols, [999](#)
- PutIn, [852](#)
- PutInside, [847](#)
- PutInSpectator, [1013](#)
- PutInStore, [853](#)
- PutInVflags, [865](#)
- PutOut, [853](#)
- PutPreVar, [874](#)
- PutTableNames, [976](#)
- PutTermInDollar, [951](#)
- PUTZERO, [801](#)
- Quotient, [854](#)
- RaisPow, [854](#)
- RaisPowCached, [854](#)
- RaisPowMod, [855](#)
- RatioFind, [856](#)
- RatioGen, [856](#)
- RatToLine, [856](#)
- ReadFile, [884](#)
- ReadFromScratch, [1014](#)
- ReadFromStream, [875](#)
- ReadijObject, [975](#)
- ReadIndex, [973](#)
- ReadIniInfo, [974](#)
- ReadObject, [975](#)
- ReadParticle, [1017](#)
- ReadPolyRatFun, [997](#)
- ReadPosFile, [884](#)
- ReadRange, [991](#)
- ReadSaveExpression, [983](#)
- ReadSaveHeader, [982](#)
- ReadSaveIndex, [982](#)
- ReadSaveTerm32, [984](#)
- ReadSaveVariables, [985](#)
- ReadSnum, [857](#)
- RecalcSetups, [891](#)
- RecoveryFilename, [987](#)
- REDLENG, [789](#)
- RedoTableTree, [961](#)
- RedoTree, [950](#)
- ReleaseTB, [980](#)
- Remain10, [857](#)
- Remain4, [857](#)
- RemoveDictionary, [1010](#)
- RemoveDollars, [972](#)
- ReNumber, [857](#)
- ReOpenFile, [883](#)
- ReplaceDollar, [880](#)
- ReserveTempFiles, [910](#)
- ResetScratch, [857](#)
- ResetVariables, [913](#)
- ResolveSet, [857](#)
- ReverseStatements, [876](#)
- RevertScratch, [858](#)
- ReWorkT, [827](#)
- RunAddArg, [992](#)
- RunCycle, [992](#)
- RunDecode, [990](#)
- RunDedup, [993](#)
- RunDropArg, [993](#)
- RunEncode, [990](#)
- RunExplode, [991](#)
- RunHtoZArg, [993](#)
- RunImplode, [990](#)
- RunIsLyndon, [992](#)
- RunMulArg, [992](#)
- RunPermute, [991](#)
- RunReplace, [990](#)
- RunReverse, [992](#)
- RunSelectArg, [993](#)
- RunToLyndon, [992](#)
- RunTransform, [990](#)
- RunZtoHArg, [993](#)
- RWLOCKR, [807](#)
- RWLOCKW, [807](#)
- ScanFunctions, [858](#)
- SeekFile, [885](#)
- SeekScratch, [858](#)
- selectExternalChannel, [979](#)
- set_del, [978](#)
- set_in, [977](#)
- set_set, [977](#)
- set_sub, [978](#)
- SETBASELENGTH, [799](#)
- SETBASEPOSITION, [799](#)
- SetDictionaryOptions, [1010](#)
- SetEndHScratch, [858](#)
- SetEndScratch, [858](#)

SETERROR, 799
SetExpr, 917
SetExprCases, 964
SetFileIndex, 858
SETKILLMODEFOREXTERNALCHANNEL, 809
SetMods, 872
setonoff, 916
SetPosFile, 885
SetScratch, 873
SetSpecialMode, 907
SETSTARTPOS, 801
SETTERMINATORFOREXTERNALCHANNEL, 809
Sflush, 859
SGN, 789
ShrinkDictionary, 1011
Shuffle, 825
simp1token, 946
simp2token, 947
simp3atoken, 947
simp3btoken, 947
simp4token, 947
simp5token, 947
simp6token, 948
SimpleSplitMerge, 963
SimpleSplitMergeRec, 963
Simplify, 859
simpwtoken, 947
SizeOfDollar, 967
SizeOfExpression, 967
SkipAName, 948
SKIPBLANKS, 792
SKIPBRA1, 793
SKIPBRA2, 793
SKIPBRA3, 793
SKIPBRA4, 793
SKIPBRA5, 794
SkipField, 889
SkipName, 1016
SortTheList, 953
SortWeights, 921
SortWild, 859
SpareTable, 873
SpecialCleanup, 872
SplitMerge, 860
StageSort, 912
StartFiles, 887
StartLoops, 1019
StartPrepro, 900
StartVariables, 810
StoreTerm, 861
str_dup, 973
StrCmp, 889
StrCopy, 812
strDup1, 873
StrHICmp, 890
StrICmp, 890
StrICont, 890
StrLen, 890
StudyPattern, 954
StuffAdd, 789
Stuffle, 825
StuffRootAdd, 825
SubPLon, 862
SubsInAll, 964
Substitute, 862
SwitchSplitMerge, 1015
SwitchSplitMergeRec, 1015
SymbolNormalize, 980
SymFind, 862
SymGen, 862
Symmetrize, 863
SynchFile, 885
TableReset, 827
TABLESIZE, 797
TakeArgContent, 918
TakeContent, 999
TakeDollarContent, 955
TakeExtraSymbols, 999
TakeIDfunction, 965
TakeLongRoot, 960
TakeModulus, 863
TakeNormalModulus, 863
TakeRatRoot, 960
TakeSymbolContent, 998
TalToLine, 863
TELLFILE, 810
TellFile, 885
TenVec, 864
TenVecFind, 864
TermAssign, 952
TermFree, 805
Terminate, 798
TerminatImpl, 877
TermMalloc, 804
TermMallocAddMemory, 989
TermRenummer, 864
TermsInBracket, 972
TermsInDollar, 967
TermsInExpression, 967
TestArgNum, 991
TestDoLoop, 1001
TestDrop, 864
TestEndDoLoop, 1001
TestFunFlag, 980
TestMatch, 865
TestName, 882
TestPartitions, 823
TestSelect, 964
TestSub, 866
TestTables, 948
TestTerm, 989
TestUse, 826
TheDefine, 895
TheUndefine, 895
TimeChildren, 866

- TimeCPU, [866](#)
- TimeWallClock, [867](#)
- ToFast, [891](#)
- ToGeneral, [891](#)
- tokenize, [946](#)
- TOKENTOLINE, [790](#)
- TokenToLine, [867](#)
- TOLONG, [803](#)
- ToPolyFunGeneral, [891](#)
- ToStorage, [867](#)
- ToToken, [909](#)
- Trace4, [867](#)
- Trace4Gen, [867](#)
- Trace4no, [868](#)
- TraceFind, [868](#)
- TraceN, [868](#)
- TraceNgen, [868](#)
- TraceNno, [868](#)
- Traces, [868](#)
- TransferBuffer, [965](#)
- TransformRational, [1011](#)
- TranslateExpression, [967](#)
- TreatPolyRatFun, [982](#)
- Trick, [869](#)
- TruncateFile, [886](#)
- TryDo, [869](#)
- TryEnvironment, [977](#)
- TryFileSetups, [965](#)
- TryRecover, [790](#)
- TwoExprCompare, [968](#)
- UngetChar, [791](#)
- UngetFromStream, [790](#)
- UNLOCK, [807](#)
- UnPack, [869](#)
- UNRWLOCK, [807](#)
- UnsetAllowDelay, [893](#)
- UnSetDictionary, [1010](#)
- UnSetMods, [872](#)
- UpdateMaxSize, [995](#)
- UpdatePositions, [968](#)
- VARNAME, [798](#)
- VarStore, [869](#)
- WantAddLongs, [804](#)
- WantAddPointers, [803](#)
- WantAddPositions, [804](#)
- Warning, [873](#)
- WCOPY, [792](#)
- WildDollars, [952](#)
- WildFill, [869](#)
- WORDDIF, [797](#)
- wranf, [981](#)
- WriteAll, [870](#)
- WriteArgument, [870](#)
- WRITEBUFTOEXTCHANNEL, [809](#)
- WriteDictionary, [1011](#)
- WriteDollarFactorToBuffer, [951](#)
- WriteDollarToBuffer, [951](#)
- WriteExpression, [870](#)
- WRITEFILE, [809](#)
- WriteFileToFile, [884](#)
- WriteIndex, [974](#)
- WriteIndexBlock, [973](#)
- WriteIniInfo, [974](#)
- WriteInnerTerm, [870](#)
- WriteLists, [870](#)
- WriteNamesBlock, [974](#)
- WriteObject, [975](#)
- WriteOne, [870](#)
- WriteSetup, [871](#)
- WriteStats, [871](#)
- WriteStoreHeader, [986](#)
- WriteString, [904](#)
- WriteSubTerm, [871](#)
- WriteTerm, [872](#)
- writeToChannel, [979](#)
- WriteTokens, [946](#)
- WriteUnfinString, [904](#)
- WrtPower, [812](#)
- wsizeof, [798](#)
- ZEROARG, [795](#)
- ZeroFillRange, [797](#)
- DeclareVector
 - vector.h, [3200](#)
- DECODEARG
 - ftypes.h, [1444](#)
- DEDUPARG
 - ftypes.h, [1446](#)
- defaultcase
 - SWITCH, [426](#)
- DEFAULTFNAMELENGTH
 - startup.c, [2793](#)
- DEFAULTPROCESSBUCKETSIZE
 - fsizes.h, [1345](#)
- defaulttempfilename
 - startup.c, [2796](#)
- DEFAULTTHREADBUCKETSIZE
 - fsizes.h, [1346](#)
- DEFAULTTHREADLOADBALANCING
 - fsizes.h, [1346](#)
- DEFAULTTHREADS
 - fsizes.h, [1346](#)
- defaultv
 - OptDef, [292](#)
- DeferFlag
 - R_const, [380](#)
- Deferred
 - declare.h, [820](#)
 - proces.c, [2428](#)
- deferskipped
 - N_const, [254](#)
- defineChannel
 - declare.h, [979](#)
 - pre.c, [2325](#)
- defined
 - TabIEs, [451](#)
- DEFINEVALUE

- setfile.c, [2684](#)
- DEFINITION
 - ftypes.h, [1372](#)
- defpart
 - MInput, [219](#)
 - Model, [237](#)
- DefPosition
 - R_const, [376](#)
- deg
 - ANode, [13](#)
 - ENode, [117](#)
 - MNode, [221](#)
 - NCand, [270](#)
 - NStack, [278](#)
 - PNodeClass, [341](#)
- degree
 - poly, [350](#)
- degree_vector
 - poly, [354](#)
- DelayPrevar
 - P_const, [320](#)
- deleted
 - EEdge, [98](#)
- DeleteObject
 - declare.h, [975](#)
 - minos.c, [1725](#)
- DeleteRecoveryFile
 - checkpoint.c, [530](#)
 - declare.h, [987](#)
- DeleteStore
 - declare.h, [821](#)
 - store.c, [2824](#)
- delGroup
 - SGroup, [392](#)
- DELTA
 - ftypes.h, [1389](#)
- DELTA2
 - ftypes.h, [1393](#)
- DELTA3
 - ftypes.h, [1391](#)
- deltaEuler
 - float.c, [1286](#)
- deltaEulerC
 - float.c, [1286](#)
- deltaMZV
 - float.c, [1286](#)
- DELTAP
 - ftypes.h, [1393](#)
- den
 - Fraction, [141](#)
- DENOMINATOR
 - ftypes.h, [1389](#)
- DENOMINATORSYMBOL
 - ftypes.h, [1403](#)
- denomnum
 - M_const, [187](#)
- DENSETABLE
 - ftypes.h, [1453](#)
- DenToFunction
 - declare.h, [972](#)
 - wildcard.c, [3212](#)
- derivative
 - poly, [351](#)
- description
 - fixedset, [137](#)
- det
 - ECand, [95](#)
 - EStack, [119](#)
- DetCommu
 - declare.h, [817](#)
 - normal.c, [1868](#)
- DetCurDum
 - declare.h, [821](#)
 - reshuf.c, [2605](#)
- detEdge
 - Assign, [21](#)
- DetVars
 - declare.h, [821](#)
 - store.c, [2825](#)
- DGraph, [80](#)
 - edges, [81](#)
 - nedges, [81](#)
 - nnodes, [81](#)
 - nodes, [81](#)
- DIAGRAMS
 - ftypes.h, [1400](#)
- diagrams.c, [1041](#), [1042](#)
 - CoCanonicalize, [1041](#)
 - DoCanonicalize, [1041](#)
 - DoShattering, [1042](#)
 - DoTopologyCanonicalize, [1041](#)
- diaoffset
 - TERMINFO, [455](#)
- diawrap.cc, [1050](#), [1052](#)
 - ConvertParticle, [1051](#)
 - fendMG, [1051](#)
 - GenDiagrams, [1052](#)
 - LoadModel, [1051](#)
 - MAXPOINTS, [1051](#)
 - numParticle, [1051](#)
 - ProcessTopology, [1052](#)
 - ReConvertParticle, [1051](#)
 - SetDualOpts, [1052](#)
- dict.c, [1065](#), [1069](#)
 - AddDictionary, [1067](#)
 - AddToDictionary, [1067](#)
 - DictFromBytes, [1069](#)
 - DictToBytes, [1069](#)
 - DoPreAdd, [1069](#)
 - DoPreCloseDictionary, [1068](#)
 - DoPreOpenDictionary, [1068](#)
 - DoPreUseDictionary, [1068](#)
 - FindDictionary, [1067](#)
 - FindExtraSymbol, [1067](#)
 - FindFunction, [1066](#)
 - FindFunWithArgs, [1067](#)

- FindIndex, [1066](#)
- FindSymbol, [1066](#)
- FindVector, [1066](#)
- IsExponentSign, [1066](#)
- IsMultiplySign, [1066](#)
- RemoveDictionary, [1068](#)
- SetDictionaryOptions, [1068](#)
- ShrinkDictionary, [1068](#)
- TransformRational, [1066](#)
- UnSetDictionary, [1068](#)
- UseDictionary, [1067](#)
- DICT_ALLNUMBERS
 - ftypes.h, [1451](#)
- DICT_DOFUNWITHARGS
 - ftypes.h, [1451](#)
- DICT_DOSPECIALS
 - ftypes.h, [1451](#)
- DICT_DOVARIABLES
 - ftypes.h, [1451](#)
- DICT_FUNCTION
 - ftypes.h, [1452](#)
- DICT_FUNCTION_WITH_ARGUMENTS
 - ftypes.h, [1452](#)
- DICT_INDEX
 - ftypes.h, [1452](#)
- DICT_INDOLLARS
 - ftypes.h, [1452](#)
- DICT_INTEGERNUMBER
 - ftypes.h, [1452](#)
- DICT_INTEGERONLY
 - ftypes.h, [1450](#)
- DICT_NOFUNWITHARGS
 - ftypes.h, [1451](#)
- DICT_NONUMBERS
 - ftypes.h, [1450](#)
- DICT_NOSPECIALS
 - ftypes.h, [1451](#)
- DICT_NOTINDOLLARS
 - ftypes.h, [1451](#)
- DICT_NOVARIABLES
 - ftypes.h, [1451](#)
- DICT_RANGE
 - ftypes.h, [1453](#)
- DICT_RATIONALNUMBER
 - ftypes.h, [1452](#)
- DICT_RATIONALONLY
 - ftypes.h, [1450](#)
- DICT_SPECIALCHARACTER
 - ftypes.h, [1453](#)
- DICT_SYMBOL
 - ftypes.h, [1452](#)
- DICT_VECTOR
 - ftypes.h, [1452](#)
- DictFromBytes
 - declare.h, [1012](#)
 - dict.c, [1069](#)
- Dictionaries
 - O_const, [284](#)
- DICTIONARY, [81](#)
 - characters, [83](#)
 - elements, [82](#)
 - funwith, [83](#)
 - gnumelements, [83](#)
 - name, [82](#)
 - numbers, [82](#)
 - numelements, [82](#)
 - ranges, [83](#)
 - sizeelements, [82](#)
 - variables, [82](#)
- DICTIONARY_ELEMENT, [83](#)
 - lhs, [84](#)
 - rhs, [84](#)
 - size, [84](#)
 - type, [84](#)
- DictToBytes
 - declare.h, [1012](#)
 - dict.c, [1069](#)
- DidClean
 - C_const, [67](#)
- DIFBASE
 - declare.h, [800](#)
- DIFPOS
 - declare.h, [800](#)
- dimension
 - CbUf, [73](#)
 - fixedset, [137](#)
 - FuNcTiOn, [146](#)
 - InDeX, [150](#)
 - SeTs, [389](#)
 - SyMbOl, [430](#)
 - VeCtOr, [472](#)
- DimensionExpression
 - declare.h, [820](#)
 - reshuf.c, [2606](#)
- DimensionSubterm
 - declare.h, [819](#)
 - reshuf.c, [2606](#)
- DIMENSIONSYPMBOL
 - ftypes.h, [1404](#)
- DimensionTerm
 - declare.h, [819](#)
 - reshuf.c, [2606](#)
- dir
 - EEdge, [100](#)
- dirEdge
 - EGraph, [107](#)
- DirtPow
 - C_const, [66](#)
- DIRTYFLAG
 - ftypes.h, [1418](#)
- dirtyflag
 - FiLeDaTa, [132](#)
- DIRTYSYMLFLAG
 - ftypes.h, [1418](#)
- discardDisc
 - MGraph, [215](#)

- discardIso
 - MGraph, [215](#)
- discardRefine
 - MGraph, [215](#)
- DisOrderFlag
 - N_const, [258](#)
- DISTRIBUTE
 - structs.h, [2898](#)
- DiStRiBuTe, [84](#)
 - cycle, [86](#)
 - n, [85](#)
 - n1, [85](#)
 - n2, [85](#)
 - obj1, [85](#)
 - obj2, [85](#)
 - out, [85](#)
 - sign, [85](#)
- Distribute
 - declare.h, [821](#)
 - symmetr.c, [2924](#)
- DISTRIBUTION
 - ftypes.h, [1391](#)
- DistrN
 - declare.h, [1016](#)
 - tools.c, [3087](#)
- div
 - COST, [76](#)
 - poly, [355](#)
- DivFloats
 - float.c, [1284](#)
- DIVFUNCTION
 - ftypes.h, [1395](#)
- DIVfunction
 - declare.h, [1000](#)
 - ratio.c, [2502](#)
- divides
 - poly, [356](#)
- DivLong
 - declare.h, [822](#)
 - reken.c, [2551](#)
- DivLong_fix_args
 - optimize.cc, [1987](#)
- divmod
 - poly, [356](#)
- divmod_heap
 - poly, [359](#)
- divmod_mpoly
 - flintinterface.h, [1266](#)
- divmod_one_term
 - poly, [358](#)
- divmod_poly
 - flintinterface.h, [1266](#)
- divmod_univar
 - poly, [358](#)
- DIVPOS
 - declare.h, [800](#)
- DivRat
 - declare.h, [822](#)
- reken.c, [2549](#)
- divrem
 - ratio.c, [2503](#)
- Divvy
 - declare.h, [822](#)
 - reken.c, [2548](#)
- DNode, [86](#)
 - extloop, [86](#)
 - intrct, [86](#)
- do_optimization
 - optimize.cc, [1993](#)
- DO_UFFLE
 - structs.h, [2897](#)
- do_uffle
 - SHvariables, [396](#)
- DOALL
 - ftypes.h, [1450](#)
- DoArgPlode
 - compcomm.c, [615](#)
 - declare.h, [922](#)
- DoArgument
 - compcomm.c, [606](#)
 - declare.h, [917](#)
- DoBrackets
 - compcomm.c, [612](#)
 - declare.h, [921](#)
- DoBreakDo
 - declare.h, [902](#)
 - pre.c, [2317](#)
- DoCall
 - declare.h, [902](#)
 - pre.c, [2316](#)
- DoCanonicalize
 - declare.h, [823](#)
 - diagrams.c, [1041](#)
- DoChain
 - compcomm.c, [610](#)
 - declare.h, [941](#)
- DoCheckpoint
 - checkpoint.c, [532](#)
 - declare.h, [988](#)
- DoClearOptimize
 - declare.h, [898](#)
 - pre.c, [2326](#)
- DoClearUserFlag
 - declare.h, [1017](#)
 - pre.c, [2328](#)
- DoCommentChar
 - declare.h, [904](#)
 - pre.c, [2315](#)
- DoContinueDo
 - declare.h, [902](#)
 - pre.c, [2317](#)
- DoDebug
 - declare.h, [902](#)
 - pre.c, [2317](#)
- DoDefine
 - declare.h, [894](#)

- pre.c, [2315](#)
- DoDelta
 - declare.h, [822](#)
 - normal.c, [1868](#)
- DoDelta3
 - declare.h, [822](#)
 - reshuf.c, [2607](#)
- DoDimension
 - declare.h, [881](#)
 - names.c, [1823](#)
- DoDistrib
 - declare.h, [824](#)
 - reshuf.c, [2607](#)
- DoDo
 - declare.h, [902](#)
 - pre.c, [2317](#)
- DODOUBLE
 - tools.c, [3071](#)
- DoElements
 - declare.h, [882](#)
 - names.c, [1826](#)
- DoElse
 - declare.h, [901](#)
 - pre.c, [2317](#)
- DoElseif
 - declare.h, [901](#)
 - pre.c, [2318](#)
- DoEnddo
 - declare.h, [902](#)
 - pre.c, [2318](#)
- DoEndif
 - declare.h, [901](#)
 - pre.c, [2318](#)
- DoEndInside
 - declare.h, [903](#)
 - pre.c, [2319](#)
- DoEndNamespace
 - declare.h, [1016](#)
 - pre.c, [2327](#)
- DoEndprocedure
 - declare.h, [903](#)
 - pre.c, [2318](#)
- DoEndSwitch
 - declare.h, [1015](#)
 - if.c, [1647](#)
- DoesCommu
 - declare.h, [817](#)
 - normal.c, [1868](#)
- DOESNOTCOMMUTE
 - ftypes.h, [1450](#)
- DoExecStatement
 - declare.h, [908](#)
 - module.c, [1758](#)
- DoExecute
 - declare.h, [873](#)
 - execute.c, [1153](#)
- DOEXPAND
 - tools.c, [3071](#)
- DoExpr
 - comexpr.c, [570](#)
 - declare.h, [949](#)
- DoExternal
 - declare.h, [896](#)
 - pre.c, [2324](#)
- DoFactDollar
 - declare.h, [897](#)
 - pre.c, [2325](#)
- DoFactorize
 - compcomm.c, [618](#)
 - declare.h, [929](#)
- DoFindLoop
 - compcomm.c, [613](#)
 - declare.h, [953](#)
- DoFromExternal
 - declare.h, [897](#)
 - pre.c, [2325](#)
- Dolf
 - declare.h, [901](#)
 - pre.c, [2318](#)
- Dolfdef
 - declare.h, [900](#)
 - pre.c, [2318](#)
- Dolfndef
 - declare.h, [901](#)
 - pre.c, [2319](#)
- DolfStatement
 - declare.h, [827](#)
 - if.c, [1647](#)
- Dolfydef
 - declare.h, [901](#)
 - pre.c, [2318](#)
- DolInclude
 - declare.h, [896](#)
 - pre.c, [2316](#)
- DolnParallel
 - compcomm.c, [610](#)
 - declare.h, [923](#)
- DoinParallel
 - declare.h, [970](#)
 - module.c, [1758](#)
- DolInside
 - declare.h, [903](#)
 - pre.c, [2319](#)
- DOLARGUMENT
 - ftypes.h, [1438](#)
- DOLINDEX
 - ftypes.h, [1438](#)
- dollar
 - ARGBUFFER, [14](#)
- dollar.c, [1083](#), [1093](#)
 - AddPotModdollar, [1092](#)
 - AddToDollarBuffer, [1085](#)
 - AssignDollar, [1084](#)
 - CatchDollar, [1084](#)
 - CleanDollarFactors, [1090](#)
 - DollarFactorize, [1089](#)

- DollarRaiseLow, [1088](#)
- DolToFunction, [1086](#)
- DolToIndex, [1086](#)
- DolToLong, [1086](#)
- DolToNumber, [1086](#)
- DolToSymbol, [1086](#)
- DolToTensor, [1085](#)
- DolToTerms, [1086](#)
- DolToVector, [1086](#)
- EvalDoLoopArg, [1088](#)
- ExchangeDollars, [1087](#)
- ExecInside, [1087](#)
- GetDolNum, [1092](#)
- InsideDollar, [1087](#)
- IsMultipleOf, [1088](#)
- IsSetMember, [1088](#)
- MakeDollarInteger, [1090](#)
- MakeDollarMod, [1091](#)
- PreflDollarEval, [1087](#)
- PutTermInDollar, [1085](#)
- SizeOfDollar, [1087](#)
- STEP2, [1084](#)
- TakeDollarContent, [1090](#)
- TermAssign, [1085](#)
- TermsInDollar, [1087](#)
- TestDoLoop, [1089](#)
- TestEndDoLoop, [1089](#)
- TranslateExpression, [1088](#)
- TwoExprCompare, [1088](#)
- WildDollars, [1085](#)
- WriteDollarFactorToBuffer, [1085](#)
- WriteDollarToBuffer, [1084](#)
- dollar_buf, [87](#)
- DOLLAREXP2
 - ftypes.h, [1388](#)
- DOLLAREXPRESSION
 - ftypes.h, [1387](#)
- DollarFactorize
 - declare.h, [955](#)
 - dollar.c, [1089](#)
- DOLLARFLAG
 - ftypes.h, [1433](#)
- DollarInOutBuffer
 - O_const, [285](#)
- DollarList
 - P_const, [316](#)
- DOLLARNAME
 - declare.h, [798](#)
- dollarname
 - DoLoOp, [91](#)
- DOLLARNAMES
 - ftypes.h, [1443](#)
- dollarname
 - C_const, [42](#)
- DollarOutBuffer
 - O_const, [282](#)
- DollarOutSizeBuffer
 - O_const, [285](#)
- DollarRaiseLow
 - declare.h, [881](#)
 - dollar.c, [1088](#)
- DoLIARs, [87](#)
 - factors, [88](#)
 - index, [88](#)
 - name, [88](#)
 - nfactors, [89](#)
 - node, [88](#)
 - numdummies, [89](#)
 - size, [88](#)
 - type, [88](#)
 - where, [88](#)
 - zero, [88](#)
- Dollars
 - variable.h, [3192](#)
- DOLLARSTREAM
 - ftypes.h, [1364](#)
- dollarzero
 - M_const, [189](#)
- DOLNUMBER
 - ftypes.h, [1438](#)
- DOLOOP
 - structs.h, [2896](#)
- DoLoOp, [89](#)
 - contents, [91](#)
 - dollarname, [91](#)
 - errorsinloop, [92](#)
 - firstdollar, [92](#)
 - firstloopcall, [92](#)
 - firstnum, [91](#)
 - incdollar, [92](#)
 - incnum, [91](#)
 - lastdollar, [92](#)
 - lastnum, [91](#)
 - name, [90](#)
 - NoShowInput, [92](#)
 - NumPreTypes, [92](#)
 - p, [90](#)
 - PreflLevel, [92](#)
 - PreSwitchLevel, [93](#)
 - startlinenumber, [91](#)
 - type, [91](#)
 - vars, [90](#)
- dolooplevel
 - C_const, [62](#)
- doloopnest
 - C_const, [51](#)
- DoLoops
 - variable.h, [3189](#)
- doloopstack
 - C_const, [51](#)
- doloopstacksize
 - C_const, [62](#)
- DOLSUBTERM
 - ftypes.h, [1438](#)
- DOLTERMS
 - ftypes.h, [1438](#)

- DolToFunction
 - declare.h, 954
 - dollar.c, 1086
- DolToIndex
 - declare.h, 954
 - dollar.c, 1086
- DolToLong
 - declare.h, 955
 - dollar.c, 1086
- DolToNumber
 - declare.h, 954
 - dollar.c, 1086
- DolToSymbol
 - declare.h, 954
 - dollar.c, 1086
- DolToTensor
 - declare.h, 954
 - dollar.c, 1085
- DolToTerms
 - declare.h, 963
 - dollar.c, 1086
- DolToVector
 - declare.h, 954
 - dollar.c, 1086
- DOLUNDEFINED
 - ftypes.h, 1438
- DOLWILDARGS
 - ftypes.h, 1438
- DOLZERO
 - ftypes.h, 1439
- DoMessage
 - declare.h, 898
 - pre.c, 2319
- DoModDollar
 - declare.h, 970
 - module.c, 1758
- DoModLocal
 - declare.h, 970
 - module.c, 1758
- DoModMax
 - declare.h, 970
 - module.c, 1757
- DoModMin
 - declare.h, 970
 - module.c, 1757
- DoModSum
 - declare.h, 970
 - module.c, 1757
- DoNamespace
 - declare.h, 1016
 - pre.c, 2327
- DONE
 - proces.c, 2420
- DoNoParallel
 - declare.h, 969
 - module.c, 1757
- DonotinParallel
 - declare.h, 971
- module.c, 1758
- DoOnePow
 - declare.h, 827
 - proces.c, 2427
- DoOptimize
 - declare.h, 898
 - pre.c, 2326
- DoParallel
 - declare.h, 969
 - module.c, 1757
- DoPartitions
 - declare.h, 823
 - reshuf.c, 2607
- DoPermutations
 - declare.h, 824
 - reshuf.c, 2608
- DoPipe
 - declare.h, 900
 - pre.c, 2319
- DoPipeStatement
 - declare.h, 908
 - module.c, 1758
- DoPolyfun
 - declare.h, 908
 - module.c, 1757
- DoPolyratfun
 - declare.h, 908
 - module.c, 1757
- DoPrcExtension
 - declare.h, 904
 - pre.c, 2319
- DoPreAdd
 - declare.h, 1009
 - dict.c, 1069
- DoPreAddSeparator
 - declare.h, 978
 - pre.c, 2324
- DoPreAppend
 - declare.h, 899
 - pre.c, 2320
- DoPreAppendPath
 - declare.h, 1014
 - pre.c, 2326
- DoPreAssign
 - declare.h, 899
 - pre.c, 2315
- DoPreBreak
 - declare.h, 899
 - pre.c, 2321
- DoPreCase
 - declare.h, 900
 - pre.c, 2321
- DoPreClose
 - declare.h, 903
 - pre.c, 2320
- DoPreCloseDictionary
 - declare.h, 1009
 - dict.c, 1068

- DoPreCreate
 - declare.h, [899](#)
 - pre.c, [2320](#)
- DoPreDefault
 - declare.h, [899](#)
 - pre.c, [2321](#)
- DoPreEndSwitch
 - declare.h, [899](#)
 - pre.c, [2321](#)
- DoPreExchange
 - declare.h, [900](#)
 - pre.c, [2316](#)
- DoPreOpenDictionary
 - declare.h, [1009](#)
 - dict.c, [1068](#)
- DoPreOut
 - declare.h, [898](#)
 - pre.c, [2319](#)
- DoPrePrependPath
 - declare.h, [1014](#)
 - pre.c, [2327](#)
- DoPrePrintTimes
 - declare.h, [903](#)
 - pre.c, [2320](#)
- DoPreRemove
 - declare.h, [904](#)
 - pre.c, [2320](#)
- DoPreReset
 - declare.h, [904](#)
 - pre.c, [2326](#)
- DoPreRmSeparator
 - declare.h, [978](#)
 - pre.c, [2324](#)
- DoPreShow
 - declare.h, [900](#)
 - pre.c, [2321](#)
- DoPreSortReallocate
 - declare.h, [904](#)
 - pre.c, [2320](#)
- DoPreSwitch
 - declare.h, [899](#)
 - pre.c, [2321](#)
- DoPreUseDictionary
 - declare.h, [1009](#)
 - dict.c, [1068](#)
- DoPreWrite
 - declare.h, [903](#)
 - pre.c, [2320](#)
- DoPrint
 - compcomm.c, [602](#)
 - declare.h, [917](#)
- DoProcedure
 - declare.h, [903](#)
 - pre.c, [2321](#)
- DoProcessBucket
 - declare.h, [970](#)
 - module.c, [1758](#)
- DoPrompt
 - declare.h, [897](#)
 - pre.c, [2324](#)
- DoPutInside
 - compcomm.c, [619](#)
 - declare.h, [921](#)
- DoRecovery
 - checkpoint.c, [531](#)
 - declare.h, [987](#)
- DoRedefine
 - declare.h, [895](#)
 - pre.c, [2315](#)
- DoReverseInclude
 - declare.h, [896](#)
 - pre.c, [2316](#)
- DoRevert
 - declare.h, [829](#)
 - normal.c, [1868](#)
- DoRmExternal
 - declare.h, [897](#)
 - pre.c, [2325](#)
- DoSetExternal
 - declare.h, [897](#)
 - pre.c, [2324](#)
- DoSetExternalAttr
 - declare.h, [897](#)
 - pre.c, [2325](#)
- DoSetRandom
 - declare.h, [898](#)
 - pre.c, [2326](#)
- DoSetups
 - declare.h, [877](#)
 - setfile.c, [2685](#)
- DoSetUserFlag
 - declare.h, [1017](#)
 - pre.c, [2328](#)
- DoShattering
 - declare.h, [824](#)
 - diagrams.c, [1042](#)
- DoShtuffle
 - declare.h, [809](#)
- DoShuffle
 - declare.h, [824](#)
 - reshuf.c, [2608](#)
- DoSkipExtraSymbols
 - declare.h, [898](#)
 - pre.c, [2326](#)
- DoStuffle
 - declare.h, [825](#)
 - reshuf.c, [2608](#)
- DoSumF1
 - declare.h, [829](#)
 - ratio.c, [2498](#)
- DoSumF2
 - declare.h, [829](#)
 - ratio.c, [2498](#)
- DoSwitch
 - declare.h, [1015](#)
 - if.c, [1647](#)

- DoSymmetrize
 - compcomm.c, [607](#)
 - declare.h, [917](#)
- DoSystem
 - declare.h, [900](#)
 - pre.c, [2322](#)
- DoTable
 - declare.h, [940](#)
 - names.c, [1825](#)
- DoTableExpansion
 - declare.h, [824](#)
 - tables.c, [2954](#)
- DoTail
 - declare.h, [874](#)
 - startup.c, [2793](#)
- dotchar
 - setfile.c, [2687](#)
- DoTempSet
 - declare.h, [882](#)
 - names.c, [1826](#)
- DoTerminate
 - declare.h, [901](#)
 - pre.c, [2317](#)
- DoTheta
 - declare.h, [829](#)
 - normal.c, [1868](#)
- DoTimeOutAfter
 - declare.h, [898](#)
 - pre.c, [2327](#)
- DoToExternal
 - declare.h, [896](#)
 - pre.c, [2325](#)
- DoTopologyCanonicalize
 - declare.h, [823](#)
 - diagrams.c, [1041](#)
- DOTPBIT
 - ftypes.h, [1437](#)
- DOTPRODUCT
 - ftypes.h, [1387](#)
- DoubleBuffer
 - declare.h, [888](#)
 - tools.c, [3084](#)
- DoubleCbuffer
 - comtool.c, [753](#)
 - declare.h, [913](#)
- DoubleFlag
 - O_const, [288](#)
- DOUBLEFORTRANMODE
 - ftypes.h, [1383](#)
- DoubleIfBuffers
 - declare.h, [916](#)
 - if.c, [1647](#)
- DoubleList
 - declare.h, [888](#)
 - tools.c, [3084](#)
- DoubleLList
 - declare.h, [888](#)
 - tools.c, [3084](#)
- DOUBLEPRECISIONFLAG
 - ftypes.h, [1384](#)
- DoubleSwitchBuffers
 - declare.h, [1015](#)
 - if.c, [1648](#)
- DoubleTable
 - float.c, [1285](#)
- DoUndefine
 - declare.h, [896](#)
 - pre.c, [2316](#)
- DoUse
 - declare.h, [1016](#)
 - pre.c, [2328](#)
- DROP
 - ftypes.h, [1420](#)
- DROPARG
 - ftypes.h, [1446](#)
- DropCoefficient
 - declare.h, [846](#)
 - normal.c, [1869](#)
- DROPGEXPRESSION
 - ftypes.h, [1421](#)
- DROPHGEXPRESSION
 - ftypes.h, [1422](#)
- DROPHLEXPRESSION
 - ftypes.h, [1422](#)
- DROPLEXPRESSION
 - ftypes.h, [1420](#)
- DROPPEDEXPRESSION
 - ftypes.h, [1421](#)
- DROPSPECTATOREXPRESSION
 - ftypes.h, [1423](#)
- DropSymbols
 - declare.h, [847](#)
 - normal.c, [1869](#)
- DuBiOuS, [93](#)
 - dummy, [93](#)
 - name, [93](#)
 - node, [93](#)
- Dubious
 - variable.h, [3193](#)
- DubiousList
 - C_const, [43](#)
- DumFound
 - N_const, [249](#)
- DUMFUN
 - ftypes.h, [1390](#)
- DumFunPlace
 - N_const, [250](#)
- DumInd
 - M_const, [184](#)
- DUMMY
 - ftypes.h, [1436](#)
- dummy
 - ARGBUFFER, [15](#)
 - BracketInfo, [36](#)
 - DuBiOuS, [93](#)
 - MoDeL, [229](#)

- DUMMYBUFFER
 - ftypes.h, [1443](#)
- DUMMYFLAG
 - ftypes.h, [1428](#)
- DUMMYFUN
 - ftypes.h, [1391](#)
- DUMMYINDEX_
 - ftypes.h, [1426](#)
- DUMMYPADDING
 - AEdge, [8](#)
 - EEdge, [100](#)
 - EGraph, [113](#)
 - ENode, [118](#)
 - MCBlock, [193](#)
 - MCEdge, [196](#)
 - MConn, [203](#)
 - MCOp, [205](#)
 - OptDef, [292](#)
 - Options, [307](#)
 - SProcess, [414](#)
- DUMMYPADDING1
 - EGraph, [114](#)
 - MGraph, [216](#)
- DUMMYPADDING2
 - MGraph, [217](#)
- dummyrenumlist
 - N_const, [251](#)
- dummysubexp
 - T_const, [441](#)
- DUMMYTEN
 - ftypes.h, [1391](#)
- DUMMYUSE
 - declare.h, [799](#)
- DumNum
 - C_const, [65](#)
- dumnumflag
 - C_const, [59](#)
- DUMPINTERMS
 - ftypes.h, [1370](#)
- DumPlace
 - N_const, [249](#)
- DumpNode
 - declare.h, [879](#)
 - names.c, [1821](#)
- DUMPOUTTERMS
 - ftypes.h, [1370](#)
- DUMPTOCOMPILER
 - ftypes.h, [1370](#)
- DUMPTOPARALLEL
 - ftypes.h, [1370](#)
- DUMPTOSORT
 - ftypes.h, [1370](#)
- DumpTree
 - declare.h, [879](#)
 - names.c, [1821](#)
- e
 - bufIPstruct, [37](#)
- e3
 - TrAcEs, [462](#)
- e4
 - TrAcEs, [463](#)
- eat
 - P_const, [321](#)
- EATTENSOR
 - ftypes.h, [1418](#)
- ebufnum
 - T_const, [438](#)
- ECand, [94](#)
 - ~ECand, [94](#)
 - det, [95](#)
 - ECand, [94](#)
 - nplist, [95](#)
 - plist, [95](#)
 - prECand, [95](#)
- eclass
 - SGroup, [393](#)
- econn
 - EGraph, [109](#)
- EDGE
 - ftypes.h, [1400](#)
- edgen
 - EStack, [119](#)
- edges
 - Assign, [26](#)
 - DGraph, [81](#)
 - EGraph, [110](#)
 - ENode, [117](#)
 - MCBlock, [193](#)
- edgeSym
 - Assign, [22](#)
- edtype
 - EEdge, [100](#)
- EEdge, [95](#)
 - ~EEdge, [96](#)
 - conid, [99](#)
 - copy, [97](#)
 - cut, [99](#)
 - deleted, [98](#)
 - dir, [100](#)
 - DUMMYPADDING, [100](#)
 - edtype, [100](#)
 - EEdge, [96](#)
 - egraph, [98](#)
 - emom, [99](#)
 - ext, [98](#)
 - extMom, [99](#)
 - id, [98](#)
 - lmom, [99](#)
 - momdir, [99](#)
 - momset, [99](#)
 - nlegs, [98](#)
 - nodes, [98](#)
 - opicom, [100](#)
 - print, [97](#)
 - ptcl, [98](#)
 - setEMom, [97](#)

- setId, 97
 - setLMom, 97
 - setType, 97
 - visited, 99
- eers
 - TrAcEs, 463
- EESYMBOL
 - ftypes.h, 1404
- EFLine, 100
 - EFLine, 101
 - elist, 101
 - fkind, 101
 - ftype, 101
 - nlist, 101
 - print, 101
- EGraph, 102
 - ~EGraph, 104
 - addFLine, 108
 - ald, 110
 - assigned, 110
 - bconn, 113
 - biconnE, 106
 - bicount, 114
 - bidef, 113
 - biinitE, 106
 - bilow, 113
 - bisearchE, 107
 - chkMomConsv, 108
 - cmpMom, 107
 - connComp, 106
 - connVisit, 106
 - copy, 104
 - dirEdge, 107
 - DUMMYPADDING, 113
 - DUMMYPADDING1, 114
 - econn, 109
 - edges, 110
 - EGraph, 104
 - endSetExtLoop, 106
 - esym, 111
 - extMom, 113
 - extMomConsv, 107
 - extperm, 111
 - findRoot, 107
 - flines, 114
 - fltrace, 108
 - fromDGraph, 105
 - fromMGraph, 105
 - fsign, 110
 - getFLines, 108
 - groupLMom, 107
 - gSubld, 110
 - isExternal, 105
 - isFermion, 106
 - isOptE, 105
 - legParticle, 108
 - loopm, 114
 - maxdeg, 112
 - mgraph, 109
 - mld, 110
 - model, 109
 - multp, 111
 - nadj2ptv, 113
 - nEdges, 112
 - nExtern, 112
 - nFLines, 114
 - nLoops, 112
 - nNodes, 112
 - nodes, 109
 - nopi2p, 113
 - nopicomp, 112
 - nsym, 110
 - nsym1, 111
 - opi2plp, 113
 - opiCount, 114
 - opt, 109
 - optQGrafA, 105
 - optQGrafM, 105
 - pld, 111
 - prFLines, 108
 - print, 104
 - printPy, 104
 - proc, 109
 - sEdges, 111
 - setExtern, 105
 - setExtLoop, 106
 - sld, 110
 - sLoops, 112
 - sMaxdeg, 111
 - sNodes, 111
 - sproc, 109
 - totalc, 112
- egraph
 - Assign, 23
 - EEdge, 98
 - ENode, 117
 - MGraph, 213
 - SProcess, 412
- elem
 - SGroup, 393
- element
 - objects, 291
- ELEMENTLOADED
 - ftypes.h, 1441
- elements
 - DICTIONARY, 82
- ELEMENTUSED
 - ftypes.h, 1441
- elist
 - EFLine, 101
- emom
 - EEdge, 99
- empty
 - FiLeInDeX, 133
- EMPTYINDEX
 - ftypes.h, 1428

EMPTYINDEX
 structs.h, 2893
emptystring
 startup.c, 2796
EmptyTable
 declare.h, 940
 names.c, 1825
EMSYMBOL
 ftypes.h, 1404
ENCODEARG
 ftypes.h, 1444
END
 ftypes.h, 1436
end
 Options, 304
endAGraph
 SProcess, 410
endianness
 STOREHEADER, 418
endmg
 Options, 306
endMGraph
 SProcess, 410
ENDMODULE
 ftypes.h, 1366
EndNest
 N_const, 248
ENDOFINPUT
 ftypes.h, 1365
ENDOFSTREAM
 ftypes.h, 1365
EndOfToken
 declare.h, 907
 tools.c, 3079
endoftokens
 C_const, 50
endProc
 Options, 304
endSetExtLoop
 EGraph, 106
EndSort
 declare.h, 830
 sort.c, 2712
endSubProc
 Options, 304
endswitch
 SWITCH, 427
EndTable
 float.c, 1286
ENode, 115
 ~ENode, 116
 copy, 116
 deg, 117
 DUMMYPADDING, 118
 edges, 117
 egraph, 117
 ENode, 116
 extloop, 118
 id, 117
 initAss, 116
 intrct, 118
 klow, 118
 maxdeg, 117
 ndtype, 118
 print, 117
 setExtern, 116
 setId, 116
 setType, 117
 visited, 118
entriesinindex
 iniinfo, 155
EntVar
 declare.h, 831
 names.c, 1820
EpfCon
 declare.h, 831
 opera.c, 1949
EpfFind
 declare.h, 831
 opera.c, 1949
EpfGen
 declare.h, 831
 opera.c, 1950
eqnidxs
 optimization, 294
eqnum
 Stream, 424
EQUAL
 ftypes.h, 1435
EqualArg
 declare.h, 831
 reshuf.c, 2607
error
 MoDeL, 229
 VerRtEx, 473
Error0
 declare.h, 875
 message.c, 1705
Error1
 declare.h, 875
 message.c, 1705
Error2
 declare.h, 875
 message.c, 1705
ErrorBlock
 O_const, 289
ErrorInDollar
 N_const, 255
ERROROUT
 ftypes.h, 1370
errorsinloop
 DoLoOp, 92
Eside
 R_const, 381
eSize
 AStack, 29

- EStack, 119
 - ~EStack, 119
 - det, 119
 - edgen, 119
 - EStack, 119
 - nplist, 120
 - plist, 120
 - print, 119
- eStack
 - AStack, 29
- eStackP
 - AStack, 29
- esym
 - EGraph, 111
 - MGraph, 214
- EULER
 - ftypes.h, 1401
- EvalDoLoopArg
 - declare.h, 963
 - dollar.c, 1088
- EvalPrelf
 - declare.h, 906
 - pre.c, 2322
- evaluate.c, 1138, 1142
 - auxr1, 1139
 - auxr2, 1139
 - auxr3, 1139
 - auxr4, 1139
 - auxr5, 1139
 - ClearFloat, 1141
 - EvaluateFun, 1141
 - EXTRAPRECISION, 1139
 - FormtoZ, 1140
 - GetFloatArgument, 1141
 - GetPiArgument, 1141
 - IntegerToFloat, 1140
 - ln2, 1142
 - PackFloat, 1140
 - RatToFloat, 1140
 - RatToFloat, 1140
 - RND, 1139
 - SetFloatPrecision, 1141
 - UnpackFloat, 1140
- EvaluateEuler
 - float.c, 1291
- EvaluateFun
 - evaluate.c, 1141
- EVEN_
 - ftypes.h, 1425
- EXCH
 - declare.h, 790
- ExchangeDollars
 - declare.h, 965
 - dollar.c, 1087
- ExchangeExpressions
 - declare.h, 965
 - execute.c, 1153
- EXCHINOUT
 - declare.h, 794
- EXCHN
 - declare.h, 790
- execarg
 - argument.c, 478
 - declare.h, 872
- ExecInside
 - declare.h, 923
 - dollar.c, 1087
- ExecMode
 - S_const, 387
- ExecModule
 - declare.h, 907
 - module.c, 1756
- ExecStore
 - declare.h, 908
 - module.c, 1756
- execterm
 - argument.c, 478
 - declare.h, 872
- execute.c, 1151, 1156
 - BRACKETCURRENTEXPR, 1152
 - BRACKETOTHEREXPR, 1152
 - CleanExpr, 1152
 - CleanupTerm, 1154
 - ContentMerge, 1155
 - CountTerms1, 1155
 - CURRENTBRACKET, 1152
 - DoExecute, 1153
 - ExchangeExpressions, 1153
 - GetContent, 1154
 - GetFirstBracket, 1154
 - GetFirstTerm, 1154
 - MakeGlobal, 1152
 - NOBRACKETACTIVE, 1152
 - PopVariables, 1152
 - PutBracket, 1153
 - PutInVflags, 1153
 - SetMods, 1153
 - SizeOfExpression, 1155
 - SpecialCleanup, 1153
 - TermsInBracket, 1155
 - TermsInExpression, 1155
 - TestDrop, 1152
 - UnSetMods, 1153
 - UpdatePositions, 1155
- EXECUTINGIF
 - ftypes.h, 1368
- EXECUTINGPRESWITCH
 - ftypes.h, 1369
- ExistsObject
 - declare.h, 975
 - minos.c, 1725
- exitflag
 - M_const, 189
- expand_memory
 - poly, 348
- ExpandBuffer

- declare.h, 888
- tools.c, 3084
- ExpandEuler
 - float.c, 1287
- ExpandMZV
 - float.c, 1287
- ExpandRat
 - declare.h, 1001
 - ratio.c, 2502
- ExpandTripleDots
 - declare.h, 911
 - pre.c, 2314
- expchanged
 - R_const, 381
- ExpectedSign
 - N_const, 260
- expflags
 - R_const, 382
- EXPFUNCTION
 - ftypes.h, 1402
- EXPLODEARG
 - ftypes.h, 1445
- expnum
 - M_const, 187
- EXPONENT
 - ftypes.h, 1389
- exprbufsize
 - ParallelVars, 324
- EXPRESSION
 - ftypes.h, 1387
- ExPrEsSiOn, 120
 - bracketinfo, 122
 - compression, 123
 - counter, 122
 - hidelevel, 122
 - inmem, 122
 - name, 122
 - namesize, 123
 - newbracketinfo, 122
 - node, 123
 - numdummies, 124
 - numfactors, 124
 - onfile, 121
 - printf flag, 123
 - prototype, 121
 - renum, 121
 - renumlists, 122
 - replace, 123
 - size, 121
 - sizeprototype, 124
 - status, 123
 - uflags, 124
 - vflags, 123
 - whichbuffer, 123
- expression
 - FiLeInDeX, 133
- ExpressionList
 - C_const, 43
- EXPRESSIONNUMBER
 - ftypes.h, 1448
- EXPRESSIONONLY
 - ftypes.h, 1374
- Expressions
 - variable.h, 3193
- exprfillwarning
 - C_const, 60
- EXPRHEAD
 - ftypes.h, 1419
- EXPRNAME
 - declare.h, 798
- EXPRNAMES
 - ftypes.h, 1443
- exprnames
 - C_const, 43
- exprnr
 - OPTIMIZERESULT, 298
- exprnumber
 - SpecTatoR, 407
- EXPRSOUT
 - ftypes.h, 1371
- ExprStat
 - FixedGlobals, 135
- ExprStatus
 - bugtool.c, 524
 - declare.h, 989
- exprtodo
 - ParallelVars, 324
- EXPTOEXP
 - ftypes.h, 1417
- ext
 - EEdge, 98
- extcmd.c, 1187, 1189
 - closeAllExternalChannels, 1189
 - closeExternalChannel, 1188
 - getCurrentExternalChannel, 1188
 - initPresetExternalChannels, 1188
 - openExternalChannel, 1188
 - selectExternalChannel, 1188
- EXTERNALCHANNELOUT
 - ftypes.h, 1371
- EXTERNALCHANNELSTREAM
 - ftypes.h, 1364
- externalset
 - TERMINFO, 456
- EXTERNLOCK
 - declare.h, 806
- externonly
 - VeRtEx, 474
- EXTERNRWLOCK
 - declare.h, 807
- EXTEUCLIDEAN
 - ftypes.h, 1399
- extloop
 - DNode, 86
 - ENode, 118
 - MNode, 221

- extMom
 - EEdge, [99](#)
 - EGraph, [113](#)
- extMomConsv
 - EGraph, [107](#)
- extonly
 - Particle, [331](#)
 - PInput, [339](#)
- extperm
 - EGraph, [111](#)
 - SProcess, [414](#)
- EXTRAPARAMETER
 - ftypes.h, [1437](#)
- EXTRAPRECISION
 - evaluate.c, [1139](#)
- extrasym
 - C_const, [51](#)
- EXTRASymbol
 - ftypes.h, [1448](#)
- ExtraSymbol
 - declare.h, [846](#)
 - normal.c, [1867](#)
- extrasymbols
 - C_const, [69](#)
- EXTRASymFUN
 - ftypes.h, [1398](#)
- ExtraSymFun
 - declare.h, [997](#)
 - notation.c, [1934](#)
- extself
 - MGraph, [216](#)
- f
 - sOrT, [402](#)
- FaCdOILaR, [124](#)
 - size, [125](#)
 - type, [125](#)
 - value, [125](#)
 - where, [125](#)
- facnum
 - M_const, [187](#)
- FACTARGFLAG
 - ftypes.h, [1447](#)
- factor
 - factorized_poly, [126](#)
 - TrAcEn, [460](#)
 - TrAcEs, [465](#)
- factor.c, [1208](#), [1209](#)
 - FactorIn, [1208](#)
 - FactorInExpr, [1208](#)
- FACTORIAL
 - ftypes.h, [1391](#)
- Factorial
 - declare.h, [831](#)
 - reken.c, [2556](#)
- factorials
 - T_const, [435](#)
- FACTORIN
 - ftypes.h, [1395](#)
- FactorIn
 - declare.h, [832](#)
 - factor.c, [1208](#)
- FactorInExpr
 - declare.h, [832](#)
 - factor.c, [1208](#)
- factorize_mpoly
 - flintinterface.h, [1266](#)
- factorize_poly
 - flintinterface.h, [1266](#)
- factorized_poly, [125](#)
 - add_factor, [126](#)
 - factor, [126](#)
 - power, [126](#)
 - tostring, [126](#)
- FactorMode
 - O_const, [288](#)
- FactorNum
 - O_const, [289](#)
- factors
 - DoLIaRS, [88](#)
- FACTORSYMBOL
 - ftypes.h, [1404](#)
- FALSE_EXPR
 - pre.c, [2311](#)
- FATAL_SIG_ERROR
 - portsignals.h, [2303](#)
- FBUFFERSIZE
 - fsizes.h, [1347](#)
- fbufferize
 - M_const, [182](#)
- fbufnum
 - T_const, [439](#)
- fendMG
 - diawrap.cc, [1051](#)
- ff
 - sOrT, [402](#)
- ffbufnum
 - C_const, [61](#)
- FG
 - inivar.h, [1673](#)
 - variable.h, [3196](#)
- FGInput, [126](#)
 - coupl, [127](#)
 - finalc, [127](#)
 - finaln, [127](#)
 - initlc, [127](#)
 - initln, [127](#)
 - nfinal, [127](#)
 - ninitl, [127](#)
- fh
 - SpecTatoR, [406](#)
- FiLe, [128](#)
 - active, [130](#)
 - filesize, [129](#)
 - handle, [130](#)
 - inbuffer, [130](#)
 - name, [129](#)

- numblocks, [130](#)
- PObuffer, [129](#)
- POfill, [129](#)
- POfull, [129](#)
- POposition, [129](#)
- POsize, [130](#)
- POstop, [129](#)
- file
 - sOrT, [399](#)
- FiLeDaTa, [131](#)
 - dirtyflag, [132](#)
 - Fill, [131](#)
 - Handle, [132](#)
 - Index, [131](#)
 - Position, [132](#)
- FILEHANDLE
 - structs.h, [2895](#)
- FILEINDEX
 - structs.h, [2894](#)
- FiLeInDeX, [132](#)
 - empty, [133](#)
 - expression, [133](#)
 - next, [133](#)
 - number, [133](#)
- filelist
 - tools.c, [3088](#)
- filelistsize
 - tools.c, [3088](#)
- filenum
 - N_const, [257](#)
- FileOnlyFlag
 - M_const, [176](#)
- fileposition
 - StreaM, [421](#)
- FILES
 - form3.h, [1329](#)
- filesize
 - FiLe, [129](#)
- FILESTREAM
 - ftypes.h, [1364](#)
- Fill
 - FiLeDaTa, [131](#)
- fill
 - PF_BUFFER, [334](#)
- FILLARG
 - declare.h, [795](#)
- fillEGraph
 - Assign, [19](#)
- FILLEXPRESS
 - declare.h, [796](#)
- FILLFUN
 - declare.h, [795](#)
- FILLFUN3
 - declare.h, [796](#)
- fillpoint
 - TaBIEbAsE, [446](#)
- FILLSUB
 - declare.h, [796](#)
- FILLVALUE
 - tools.c, [3070](#)
- finalc
 - FGInput, [127](#)
- finaln
 - FGInput, [127](#)
- finalPart
 - Process, [370](#)
- FinalStats
 - C_const, [58](#)
- finalstep
 - TrAcEs, [466](#)
- find_Horner_MCTS
 - optimize.cc, [1991](#)
- find_Horner_MCTS_expand_tree
 - optimize.cc, [1991](#)
- find_optimizations
 - optimize.cc, [1993](#)
- FindAll
 - declare.h, [832](#)
 - pattern.c, [2167](#)
- FindArg
 - argument.c, [479](#)
 - declare.h, [919](#)
- FindBracket
 - declare.h, [969](#)
 - index.c, [1663](#)
- FindCase
 - declare.h, [1015](#)
 - if.c, [1648](#)
- findcommand
 - compiler.c, [712](#)
 - declare.h, [886](#)
- FindDictionary
 - declare.h, [1010](#)
 - dict.c, [1067](#)
- FindExtraSymbol
 - declare.h, [1012](#)
 - dict.c, [1067](#)
- FindFunction
 - declare.h, [1012](#)
 - dict.c, [1066](#)
- FindFunWithArgs
 - declare.h, [1012](#)
 - dict.c, [1067](#)
- FindIndex
 - declare.h, [1012](#)
 - dict.c, [1066](#)
- FindInIndex
 - declare.h, [811](#)
 - store.c, [2827](#)
- FindInKeyWord
 - declare.h, [894](#)
 - pre.c, [2314](#)
- findInteractionCode
 - Model, [232](#)
- findInteractionName
 - Model, [232](#)

- FindKeyWord
 - declare.h, [894](#)
 - pre.c, [2314](#)
- FindLocalSubterm
 - declare.h, [994](#)
 - notation.c, [1934](#)
- FINDLOOP
 - ftypes.h, [1439](#)
- FindLus
 - declare.h, [952](#)
 - lus.c, [1678](#)
- findMClass
 - Model, [233](#)
- FindMulti
 - declare.h, [832](#)
 - findpat.c, [1220](#)
- FindOnce
 - declare.h, [832](#)
 - findpat.c, [1220](#)
- FindOnly
 - declare.h, [833](#)
 - findpat.c, [1220](#)
- findParticleCode
 - Model, [232](#)
- findParticleName
 - Model, [232](#)
- findpat.c, [1219](#), [1220](#)
 - FindMulti, [1220](#)
 - FindOnce, [1220](#)
 - FindOnly, [1220](#)
 - FindRest, [1220](#)
- findPattern
 - N_const, [251](#)
- FindRange
 - declare.h, [991](#)
 - transform.c, [3138](#)
- FindRest
 - declare.h, [833](#)
 - findpat.c, [1220](#)
- FindrNumber
 - declare.h, [833](#)
 - store.c, [2827](#)
- findRoot
 - EGraph, [107](#)
- FindSubexpression
 - declare.h, [995](#)
 - notation.c, [1934](#)
- FindSubterm
 - declare.h, [994](#)
 - notation.c, [1934](#)
- FindSymbol
 - declare.h, [1011](#)
 - dict.c, [1066](#)
- FindTableNumber
 - declare.h, [977](#)
 - minos.c, [1724](#)
- FindTableTree
 - declare.h, [961](#)
- tables.c, [2954](#)
- FindTB
 - declare.h, [826](#)
 - tables.c, [2955](#)
- findTerm
 - N_const, [250](#)
- FindTree
 - comtool.c, [756](#)
 - declare.h, [950](#)
- FindVar
 - declare.h, [827](#)
 - if.c, [1647](#)
- FindVector
 - declare.h, [1011](#)
 - dict.c, [1066](#)
- FiniLine
 - declare.h, [833](#)
 - sch.c, [2646](#)
- FINISH
 - ftypes.h, [1440](#)
- finishcbuf
 - comtool.c, [752](#)
 - declare.h, [961](#)
- finished
 - tree_node, [469](#)
- FinishShuffle
 - declare.h, [825](#)
 - reshuf.c, [2608](#)
- FinishStuffle
 - declare.h, [825](#)
 - reshuf.c, [2609](#)
- finishuf
 - SHvariables, [396](#)
- FINISHUFFLE
 - structs.h, [2897](#)
- FiniShuffle
 - declare.h, [808](#)
- FiniTarm
 - declare.h, [833](#)
 - proces.c, [2425](#)
- first
 - SeTs, [388](#)
- first_variable
 - poly, [350](#)
- FIRSTBRACKET
 - ftypes.h, [1394](#)
- firstconstindex
 - C_const, [54](#)
- firstctypemessage
 - C_const, [56](#)
- firstdollar
 - DoLoOp, [92](#)
- firstindexblock
 - iniinfo, [155](#)
- firstloopcall
 - DoLoOp, [92](#)
- FIRSTMODULE
 - ftypes.h, [1366](#)

- firstnameblock
 - iniinfo, [156](#)
- firstnamespace
 - P_const, [317](#)
- firstnum
 - DoLoOp, [91](#)
- FIRSTTERM
 - ftypes.h, [1398](#)
- FIRSTUSERFUNCTION
 - ftypes.h, [1402](#)
- FIRSTUSERSYMBOL
 - ftypes.h, [1404](#)
- fix_sign_fmpz_mpoly_ratfun
 - flintinterface.h, [1271](#)
- fix_sign_fmpz_poly_ratfun
 - flintinterface.h, [1271](#)
- fixarg
 - optimize.cc, [1986](#)
- FIXED_
 - ftypes.h, [1426](#)
- FIXEDGLOBALS
 - structs.h, [2899](#)
- FixedGlobals, [134](#)
 - cTable, [136](#)
 - ExprStat, [135](#)
 - fname, [135](#)
 - fname2, [135](#)
 - fname2base, [136](#)
 - fname2size, [136](#)
 - fnamebase, [135](#)
 - fnamesize, [136](#)
 - FunNam, [135](#)
 - OperaFind, [134](#)
 - Operation, [134](#)
 - s_one, [135](#)
 - swmes, [135](#)
 - VarType, [135](#)
- fixedset, [136](#)
 - description, [137](#)
 - dimension, [137](#)
 - name, [137](#)
 - type, [137](#)
- fixedsets
 - inivar.h, [1673](#)
 - variable.h, [3196](#)
- FixIndices
 - C_const, [49](#)
- fkind
 - EFLine, [101](#)
- flags
 - FuNcTiOn, [145](#)
 - InDeX, [150](#)
 - indexblock, [151](#)
 - KEYWORD, [162](#)
 - KEYWORDV, [163](#)
 - SeTs, [389](#)
 - SETUPPARAMETERS, [390](#)
 - SpecTatoR, [407](#)
 - SyMbOl, [430](#)
 - TabIEs, [449](#)
 - TERMINFO, [456](#)
 - VeCtOr, [472](#)
- flg0
 - MNodeClass, [226](#)
- flg1
 - MNodeClass, [226](#)
- flg2
 - MNodeClass, [226](#)
- flines
 - EGraph, [114](#)
- flint::fmpz, [137](#)
 - ~fmpz, [138](#)
 - d, [138](#)
 - fmpz, [138](#)
 - print, [138](#)
- flint::mpoly, [242](#)
 - ~mpoly, [242](#)
 - d, [242](#)
 - mpoly, [242](#)
 - print, [242](#)
- flint::mpoly_ctx, [243](#)
 - ~mpoly_ctx, [243](#)
 - d, [243](#)
 - mpoly_ctx, [243](#)
- flint::mpoly_factor, [243](#)
 - ~mpoly_factor, [244](#)
 - d, [244](#)
 - mpoly_factor, [244](#)
- flint::poly, [343](#)
 - ~poly, [343](#)
 - d, [343](#)
 - poly, [343](#)
 - print, [343](#)
- flint::poly_factor, [361](#)
 - ~poly_factor, [362](#)
 - d, [362](#)
 - poly_factor, [362](#)
- flint_check_version
 - flintwrap.cc, [1276](#)
- flint_div
 - flintwrap.cc, [1275](#)
- flint_factorize_argument
 - flintwrap.cc, [1275](#)
- flint_factorize_dollar
 - flintwrap.cc, [1275](#)
- flint_final_cleanup_master
 - flintwrap.cc, [1275](#)
- flint_final_cleanup_thread
 - flintwrap.cc, [1275](#)
- flint_gcd
 - flintwrap.cc, [1275](#)
- flint_inverse
 - flintwrap.cc, [1276](#)
- flint_mul
 - flintwrap.cc, [1276](#)
- flint_ratfun_add

- flintwrap.cc, 1276
- flint_ratfun_normalize
 - flintwrap.cc, 1276
- flint_rem
 - flintwrap.cc, 1276
- flintinterface.cc, 1237, 1238
 - IFW, 1237
 - UNIVARIATE_DENSITY_THR, 1237
- flintinterface.h, 1263, 1272
 - cleanup, 1266
 - cleanup_master, 1266
 - divmod_mpoly, 1266
 - divmod_poly, 1266
 - factorize_mpoly, 1266
 - factorize_poly, 1266
 - fix_sign_fmpz_mpoly_ratfun, 1271
 - fix_sign_fmpz_poly_ratfun, 1271
 - fmpz_get_form, 1267
 - fmpz_set_form, 1267
 - form_sort, 1267
 - from_argument_mpoly, 1267
 - from_argument_poly, 1267
 - gcd_mpoly, 1268
 - gcd_poly, 1268
 - get_variables, 1268
 - inverse_poly, 1268
 - mul_mpoly, 1268
 - mul_poly, 1269
 - ratfun_add_mpoly, 1269
 - ratfun_add_poly, 1269
 - ratfun_normalize_mpoly, 1269
 - ratfun_normalize_poly, 1269
 - ratfun_read_mpoly, 1270
 - ratfun_read_poly, 1270
 - simplify_fmpz, 1271
 - simplify_fmpz_poly, 1271
 - to_argument_mpoly, 1270
 - to_argument_poly, 1270, 1271
 - var_map_t, 1265
- FlintPolyFlag
 - C_const, 62
- flintwrap.cc, 1274, 1277
 - flint_check_version, 1276
 - flint_div, 1275
 - flint_factorize_argument, 1275
 - flint_factorize_dollar, 1275
 - flint_final_cleanup_master, 1275
 - flint_final_cleanup_thread, 1275
 - flint_gcd, 1275
 - flint_inverse, 1276
 - flint_mul, 1276
 - flint_ratfun_add, 1276
 - flint_ratfun_normalize, 1276
 - flint_rem, 1276
- FlipLONG
 - O_const, 283
- FlipPOINTER
 - O_const, 283
- FlipPOS
 - O_const, 283
- FlipTable
 - declare.h, 971
 - tables.c, 2955
- FlipWORD
 - O_const, 283
- flist
 - MNodeClass, 225
 - MOrbits, 241
- float.c, 1281, 1292
 - AddFloats, 1284
 - AddRatToFloat, 1284
 - AddWithFloat, 1291
 - CalculateEuler, 1287
 - CalculateMZV, 1287
 - CalculateMZVhalf, 1286
 - CheckFloat, 1289
 - Chop, 1291
 - ClearMZVTables, 1289
 - CoChop, 1290
 - CoEvaluate, 1291
 - CoStrictRounding, 1290
 - CoToFloat, 1289
 - CoToRat, 1290
 - deltaEuler, 1286
 - deltaEulerC, 1286
 - deltaMZV, 1286
 - DivFloats, 1284
 - DoubleTable, 1285
 - EndTable, 1286
 - EvaluateEuler, 1291
 - ExpandEuler, 1287
 - ExpandMZV, 1287
 - FloatFunToRat, 1288
 - FloatToInteger, 1283
 - FloatToRat, 1284
 - Form_mpf_clear, 1283
 - Form_mpf_empty, 1288
 - Form_mpf_init, 1283
 - Form_mpf_set_prec_raw, 1283
 - FormtoZ, 1283
 - GMPSREAD, 1282
 - IntegerToFloat, 1284
 - MergeWithFloat, 1291
 - MulFloats, 1284
 - MulRatToFloat, 1285
 - PackFloat, 1287
 - PrintFloat, 1289
 - RatToFloat, 1288
 - ReadFloat, 1288
 - SetFloatPrecision, 1289
 - SetupMPFTables, 1289
 - SetupMZVTables, 1289
 - SimpleDelta, 1285
 - SimpleDeltaC, 1285
 - SingleTable, 1285
 - StrictRounding, 1290

- TestFloat, [1288](#)
- ToFloat, [1290](#)
- ToRat, [1290](#)
- UnpackFloat, [1287](#)
- WITHCUTOFF, [1282](#)
- ZeroTable, [1288](#)
- ZtoForm, [1283](#)
- FLOATFUN
 - ftypes.h, [1401](#)
- FloatFunToRat
 - float.c, [1288](#)
- FLOATMODE
 - ftypes.h, [1429](#)
- FloatToInteger
 - float.c, [1283](#)
- FloatToRat
 - float.c, [1284](#)
- FLOOP
 - ftypes.h, [1456](#)
- fltrace
 - EGraph, [108](#)
- flush_cache
 - checkpoint.c, [531](#)
- FLUSHCONSOLE
 - declare.h, [793](#)
- FlushFile
 - declare.h, [885](#)
 - tools.c, [3076](#)
- FlushOut
 - declare.h, [834](#)
 - sort.c, [2715](#)
- FlushSpectators
 - declare.h, [1014](#)
 - spectator.c, [2782](#)
- fmprz
 - flint::fmprz, [138](#)
- fmprz_get_form
 - flintinterface.h, [1267](#)
- fmprz_set_form
 - flintinterface.h, [1267](#)
- fname
 - FixedGlobals, [135](#)
- fname2
 - FixedGlobals, [135](#)
- fname2base
 - FixedGlobals, [136](#)
- fname2size
 - FixedGlobals, [136](#)
- fnamebase
 - FixedGlobals, [135](#)
- fnamesize
 - FixedGlobals, [136](#)
- FoldName
 - StreaM, [422](#)
- ForFindOnly
 - N_const, [250](#)
- form3.h, [1323](#), [1334](#)
 - AWORDMASK, [1328](#)
- BETAVERSION, [1326](#)
- BITSINLONG, [1327](#)
- BITSINWORD, [1327](#)
- BOOL, [1333](#)
- bzero, [1332](#)
- FILES, [1329](#)
- form_alignof, [1329](#)
- FULLMAX, [1328](#)
- GetPID, [1332](#)
- inline, [1326](#)
- LONG, [1332](#)
- LONG_MAX_VALUE, [1328](#)
- LONG_MIN_VALUE, [1327](#)
- MAJORVERSION, [1326](#)
- MAX_OPEN_FILES, [1332](#)
- MAXLONG, [1329](#)
- MAXPOSITIVE, [1329](#)
- MAXPOSITIVE2, [1329](#)
- MAXPOSITIVE4, [1329](#)
- MINORVERSION, [1326](#)
- MLONG, [1333](#)
- NDEBUG, [1326](#)
- PRODUCTIONDATE, [1326](#)
- RLONG, [1333](#)
- SBYTE, [1333](#)
- SPECMASK, [1328](#)
- STATIC_ASSERT, [1326](#)
- STATIC_ASSERT__1, [1326](#)
- STATIC_ASSERT__2, [1327](#)
- STATIC_ASSERT__3, [1327](#)
- TOPBITONLY, [1328](#)
- TOPLONGBITONLY, [1328](#)
- UBYTE, [1333](#)
- Uclose, [1330](#)
- Uflush, [1330](#)
- Ugetpos, [1331](#)
- UINT, [1333](#)
- ULONG, [1333](#)
- Uopen, [1330](#)
- Uread, [1330](#)
- Useek, [1331](#)
- Usetbuf, [1330](#)
- Usetpos, [1331](#)
- Ustdout, [1332](#)
- Usync, [1331](#)
- Utell, [1331](#)
- Utruncate, [1331](#)
- UWORD, [1332](#)
- Uwrite, [1330](#)
- WILDMASK, [1328](#)
- WITHSORTBOTS, [1329](#)
- WORD, [1332](#)
- WORD_MAX_VALUE, [1327](#)
- WORD_MIN_VALUE, [1327](#)
- WORDMASK, [1328](#)
- form_alignof
 - form3.h, [1329](#)
- FORM_INLINE

- declare.h, 804
- Form_mpf_clear
 - float.c, 1283
- Form_mpf_empty
 - float.c, 1288
- Form_mpf_init
 - float.c, 1283
- Form_mpf_set_prec_raw
 - float.c, 1283
- form_sort
 - flintinterface.h, 1267
- FORMNAME
 - startup.c, 2792
- FormtoZ
 - evaluate.c, 1140
 - float.c, 1283
- FortDotChar
 - O_const, 289
- FortFirst
 - O_const, 288
- Fortran90Kind
 - C_const, 51
- FortranCont
 - M_const, 187
- FORTRANCONTINUATIONLINES
 - fsizes.h, 1342
- FORTRANMODE
 - ftypes.h, 1383
- FoStage4
 - R_const, 376
- fPatches
 - sOrT, 401
- fPatchesStop
 - sOrT, 401
- fPatchN
 - sOrT, 405
- fPatchN1
 - sOrT, 404
- Fraction, 138
 - add, 140
 - den, 141
 - Fraction, 139
 - gcd, 140
 - isEq, 140
 - normal, 140
 - num, 141
 - print, 139
 - ratio, 141
 - setValue, 139
 - sub, 140
- freelg
 - MNode, 222
- FreeNameTree
 - declare.h, 880
 - names.c, 1822
- FreeTableBase
 - declare.h, 976
 - minos.c, 1723
- from
 - PF_BUFFER, 336
- From0List
 - declare.h, 887
 - tools.c, 3083
- from_argument_mpoly
 - flintinterface.h, 1267
- from_argument_poly
 - flintinterface.h, 1267
- from_coefficient_list
 - poly, 353
- FROMBRAC
 - ftypes.h, 1417
- fromDGraph
 - EGraph, 105
- FROMDISK
 - minos.h, 1744
- FROMFUNCTION
 - ftypes.h, 1443
- fromindex
 - T_const, 444
- FromList
 - declare.h, 887
 - tools.c, 3083
- fromMGraph
 - Assign, 18
 - EGraph, 105
- FROMMODULEOPTION
 - ftypes.h, 1447
- fromPerm
 - MOrbits, 240
- FROMPOINTINSTRUCTION
 - ftypes.h, 1448
- FromPolyRatFun
 - declare.h, 998
 - ratio.c, 2501
- FROMSET
 - ftypes.h, 1417
- FromStdin
 - M_const, 191
- FromVarList
 - declare.h, 888
 - tools.c, 3083
- Frozen
 - N_const, 248
- Fscr
 - R_const, 376
- fsign
 - EGraph, 110
- fsizes.h, 1338, 1348
 - COMMERCIALSIZE, 1342
 - COMPILERBUFFER, 1343
 - COMPINC, 1345
 - COMPRESSBUFFER, 1342
 - DEFAULTPROCESSBUCKETSIZE, 1345
 - DEFAULTTHREADBUCKETSIZE, 1346
 - DEFAULTTHREADLOADBALANCING, 1346
 - DEFAULTTHREADS, 1346

- FBUFFERSIZE, [1347](#)
- FORTTRANCONTINUATIONLINES, [1342](#)
- GZIPDEFAULT, [1345](#)
- INDENTSPACE, [1346](#)
- INITNAMESIZE, [1341](#)
- INITNODESIZE, [1341](#)
- JUMPRATIO, [1347](#)
- MAXBRACKETBUFFERSIZE, [1345](#)
- MAXCOUPLINGS, [1348](#)
- MAXENAME, [1340](#)
- MAXFLAGS, [1342](#)
- MAXLEVELS, [1344](#)
- MAXFPATCHES, [1343](#)
- MAXIF, [1341](#)
- MAXLABELS, [1342](#)
- MAXLEGS, [1348](#)
- MAXLEVELS, [1342](#)
- MAXLHS, [1343](#)
- MAXLINELENGTH, [1347](#)
- MAXLOOPS, [1342](#)
- MAXMATCH, [1341](#)
- MAXMULTIBRACKETLEVELS, [1347](#)
- MAXNEST, [1341](#)
- MAXNUMBERSIZE, [1340](#)
- MAXNUMSIZE, [1345](#)
- MAXPARLEVEL, [1340](#)
- MAXPARTICLES, [1348](#)
- MAXPATCHES, [1343](#)
- MAXPRENAMESIZE, [1340](#)
- MAXREPEAT, [1340](#)
- MAXSAVEFUNCTION, [1340](#)
- MAXWILDC, [1343](#)
- MINALLOC, [1347](#)
- MULTIINDENTSPACE, [1347](#)
- NORMSIZE, [1341](#)
- NPAIR1, [1347](#)
- NPAIR2, [1347](#)
- NUMFACS, [1342](#)
- NUMFIXED, [1341](#)
- NUMOPTIONS, [1348](#)
- NUMSTORECACHES, [1346](#)
- NUMTABLEENTRIES, [1343](#)
- SFHSIZE, [1345](#)
- SHMWINSIZE, [1345](#)
- SIZEFACS, [1341](#)
- SIZESTORECACHE, [1346](#)
- SLARGEBUFFER, [1344](#)
- SMAFXPATCHES, [1344](#)
- SMAXPATCHES, [1344](#)
- SORTIOSIZE, [1343](#)
- SPECTATORSIZE, [1344](#)
- SSMALLBUFFER, [1343](#)
- SSMALLOVERFLOW, [1344](#)
- SSORTIOSIZE, [1344](#)
- STERMSSMALL, [1344](#)
- TABLEEXTENSION, [1345](#)
- THREADSCRATCHOUTSIZE, [1346](#)
- THREADSCRATCHSIZE, [1346](#)
- ftype
 - EFLine, [101](#)
- ftypes.h, [1351](#), [1458](#)
 - ABSFUNCTION, [1392](#)
 - ACOSFUNCTION, [1396](#)
 - ACOSHFUNCTION, [1397](#)
 - ADDARG, [1446](#)
 - AGMFUNCTION, [1402](#)
 - ALLARGS, [1444](#)
 - ALLDIRTY, [1419](#)
 - ALLFUNCTIONS, [1403](#)
 - ALLFUNPOWERS, [1439](#)
 - ALLINTEGERDOUBLE, [1384](#)
 - ALLMZVFUNCTIONS, [1403](#)
 - ALLVARIABLES, [1374](#)
 - ALSODOLLARS, [1442](#)
 - ALSOFUNARGS, [1442](#)
 - ALSOPOWERS, [1442](#)
 - ALSOVERSE, [1385](#)
 - ANDCOND, [1436](#)
 - ANTISYMMETRIC, [1431](#)
 - ANYTYPE, [1375](#)
 - ARGFIELD, [1388](#)
 - ARGHEAD, [1419](#)
 - ARGRANGE, [1444](#)
 - ARGTOARG, [1416](#)
 - ARGWILD, [1387](#)
 - ARLTOARL, [1416](#)
 - ASINFUNCTION, [1396](#)
 - ASINHFUNCTION, [1397](#)
 - ATAN2FUNCTION, [1397](#)
 - ATANFUNCTION, [1397](#)
 - ATANHFUNCTION, [1397](#)
 - ATENDOFMODULE, [1373](#)
 - AUTONAMES, [1443](#)
 - BASECODE, [1447](#)
 - BERNOULLIFUNCTION, [1393](#)
 - BINOMIAL, [1391](#)
 - BIPART, [1456](#)
 - BLOCK, [1401](#)
 - BUILTINDOLLARS, [1405](#)
 - BUILTININDICES, [1404](#)
 - BUILTINSYMBOLS, [1404](#)
 - BUILTINVECTORS, [1405](#)
 - CDELETE, [1374](#)
 - CDELTA, [1376](#)
 - CDOLLAR, [1376](#)
 - CDOTPRODUCT, [1375](#)
 - CDOUBLEDOT, [1377](#)
 - CDUBIOUS, [1376](#)
 - CEXPRESSION, [1375](#)
 - CFUN, [1457](#)
 - CFUNCTION, [1375](#)
 - CHECKEXTERN, [1454](#)
 - CHECKLOGTYPE, [1386](#)
 - CHISHOLM, [1385](#)
 - CINDEX, [1375](#)
 - CLEANFLAG, [1418](#)

- CLEAR, [1437](#)
- CLEARMODULE, [1366](#)
- CMODE, [1383](#)
- CMODEL, [1376](#)
- CNUMBER, [1376](#)
- COEFFI, [1434](#)
- COEFFICIENTSONLY, [1442](#)
- COEFFSYMBOL, [1403](#)
- COMFUNPOWERS, [1439](#)
- CONJUGATION, [1394](#)
- CONTENTTERM, [1398](#)
- CONTRACT, [1414](#)
- COSFUNCTION, [1396](#)
- COSHFUNCTION, [1397](#)
- COULDCOMMUTE, [1450](#)
- COUNTFUNCTION, [1393](#)
- CRANGE, [1376](#)
- CSET, [1375](#)
- CSUBEXP, [1376](#)
- CSYMBOL, [1375](#)
- CVECTOR, [1375](#)
- CVECTOR1, [1376](#)
- CYCLEARG, [1445](#)
- CYCLESYMMETRIC, [1431](#)
- CYCLI, [1456](#)
- CYCLR, [1456](#)
- DECLARATION, [1372](#)
- DECODEARG, [1444](#)
- DEDUPARG, [1446](#)
- DEFINITION, [1372](#)
- DELTA, [1389](#)
- DELTA2, [1393](#)
- DELTA3, [1391](#)
- DELTAP, [1393](#)
- DENOMINATOR, [1389](#)
- DENOMINATORSYMBOL, [1403](#)
- DENSETABLE, [1453](#)
- DIAGRAMS, [1400](#)
- DICT_ALLNUMBERS, [1451](#)
- DICT_DOFUNWITHARGS, [1451](#)
- DICT_DOSPECIALS, [1451](#)
- DICT_DOVARIABLES, [1451](#)
- DICT_FUNCTION, [1452](#)
- DICT_FUNCTION_WITH_ARGUMENTS, [1452](#)
- DICT_INDEX, [1452](#)
- DICT_INDOLLARS, [1452](#)
- DICT_INTEGERNUMBER, [1452](#)
- DICT_INTEGERONLY, [1450](#)
- DICT_NOFUNWITHARGS, [1451](#)
- DICT_NONUMBERS, [1450](#)
- DICT_NOSPECIALS, [1451](#)
- DICT_NOTINDOLLARS, [1451](#)
- DICT_NOVARIABLES, [1451](#)
- DICT_RANGE, [1453](#)
- DICT_RATIONALNUMBER, [1452](#)
- DICT_RATIONALONLY, [1450](#)
- DICT_SPECIALCHARACTER, [1453](#)
- DICT_SYMBOL, [1452](#)
- DICT_VECTOR, [1452](#)
- DIMENSIONSYPHON, [1404](#)
- DIRTYFLAG, [1418](#)
- DIRTYSYMPHON, [1418](#)
- DISTRIBUTION, [1391](#)
- DIVFUNCTION, [1395](#)
- DOALL, [1450](#)
- DOESNOTCOMMUTE, [1450](#)
- DOLARGUMENT, [1438](#)
- DOLINDEX, [1438](#)
- DOLLAREXP2, [1388](#)
- DOLLAREXPRESSION, [1387](#)
- DOLLARFLAG, [1433](#)
- DOLLARNAMES, [1443](#)
- DOLLARSTREAM, [1364](#)
- DOLNUMBER, [1438](#)
- DOLSUBTERM, [1438](#)
- DOLTERMS, [1438](#)
- DOLUNDEFINED, [1438](#)
- DOLWILDARGS, [1438](#)
- DOLZERO, [1439](#)
- DOTPHON, [1437](#)
- DOTPRODUCT, [1387](#)
- DOUBLEFORTRANMODE, [1383](#)
- DOUBLEPRECISIONFLAG, [1384](#)
- DROP, [1420](#)
- DROPARG, [1446](#)
- DROPGEXPRESSION, [1421](#)
- DROPHGEXPRESSION, [1422](#)
- DROPHLEXPRESSION, [1422](#)
- DROPLEXPRESSION, [1420](#)
- DROPPEDEXPRESSION, [1421](#)
- DROPSPECTATOREXPRESSION, [1423](#)
- DUMFUN, [1390](#)
- DUMMY, [1436](#)
- DUMMYBUFFER, [1443](#)
- DUMMYFLAG, [1428](#)
- DUMMYFUN, [1391](#)
- DUMMYINDEX, [1426](#)
- DUMMYTEN, [1391](#)
- DUMPINTERMS, [1370](#)
- DUMPOUTTERMS, [1370](#)
- DUMPTOCOMPIER, [1370](#)
- DUMPTOPARALLEL, [1370](#)
- DUMPTOSORT, [1370](#)
- EATTENSOR, [1418](#)
- EDGE, [1400](#)
- EESYPHON, [1404](#)
- ELEMENTLOADED, [1441](#)
- ELEMENTUSED, [1441](#)
- EMPTYINDEX, [1428](#)
- EMSYPHON, [1404](#)
- ENCODEARG, [1444](#)
- END, [1436](#)
- ENDMODULE, [1366](#)
- ENDOFINPUT, [1365](#)
- ENDOFSTREAM, [1365](#)
- EQUAL, [1435](#)

ERROROUT, 1370
EULER, 1401
EVEN_, 1425
EXECUTINGIF, 1368
EXECUTINGPRESWITCH, 1369
EXPFUNCTION, 1402
EXPLODEARG, 1445
EXPONENT, 1389
EXPRESSION, 1387
EXPRESSIONNUMBER, 1448
EXPRESSIONONLY, 1374
EXPRHEAD, 1419
EXPRNAMES, 1443
EXPRSOUT, 1371
EXPTOEXP, 1417
EXTERNALCHANNELOUT, 1371
EXTERNALCHANNELSTREAM, 1364
EXTEUCLIDEAN, 1399
EXTRAPARAMETER, 1437
EXTRASymbol, 1448
EXTRASymFUN, 1398
FACTARGFLAG, 1447
FACTORIAL, 1391
FACTORIN, 1395
FACTORSYMBOL, 1404
FILESTREAM, 1364
FINDLOOP, 1439
FINISH, 1440
FIRSTBRACKET, 1394
FIRSTMODULE, 1366
FIRSTTERM, 1398
FIRSTUSERFUNCTION, 1402
FIRSTUSERSYMBOL, 1404
FIXED_, 1426
FLOATFUN, 1401
FLOATMODE, 1429
FLOOP, 1456
FORTRANMODE, 1383
FROMBRAC, 1417
FROMFUNCTION, 1443
FROMMODULEOPTION, 1447
FROMPOINTINSTRUCTION, 1448
FROMSET, 1417
FUNBIT, 1437
FUNCTION, 1388
FUNCTIONONLY, 1374
FUNCTIONSORT, 1428
FUNHEAD, 1419
FUNNYDOLLAR, 1428
FUNNYVEC, 1427
FUNNYWILD, 1427
FUNTOFUN, 1416
GAMMA, 1389
GAMMA1, 1427
GAMMA5, 1427
GAMMA6, 1427
GAMMA7, 1427
GAMMAFIVE, 1390
GAMMAFUN, 1402
GAMMAFUNCTION, 1425
GAMMAI, 1389
GAMMASEVEN, 1390
GAMMASIX, 1390
GCDFUNCTION, 1395
GENCOMMUTE, 1437
GENERALFUNCTION, 1425
GENNONCOMMUTE, 1437
GFIVE, 1433
GIDENT, 1433
GLOBAL, 1436
GLOBALEXPRESSION, 1421
GLOBALMODULE, 1366
GLOBALSPECTATORFLAG, 1453
GMINUS, 1433
GPLUS, 1433
GREATER, 1435
GREATEREQUAL, 1435
HAAKJE, 1389
HAAKJE0, 1388
HEADERFILE, 1365
HIDDENGEXPRESSION, 1421
HIDDENLEXPRESSION, 1421
HIDE, 1420
HIDEGEXPRESSION, 1422
HIDELEXPRESSION, 1422
HPLFUNCTION, 1402
HTOZARG, 1446
IDFUNCTION, 1399
IDHEAD, 1433
IFDOLLAR, 1434
IFDOLLAREXTRA, 1434
IFEXPRESSION, 1434
IFFLOATNUMBER, 1435
IFISFACTORIZED, 1434
IFOCCURS, 1435
IFUSERFLAG, 1435
IMPLODEARG, 1445
INCEXPRESSION, 1421
INDEX, 1387
INDEX_, 1426
INDEXONLY, 1374
INDTOIND, 1416
INDTOSUB, 1416
INPUTOUT, 1370
INPUTSTREAM, 1365
INTFUNCTION, 1393
INTOHIDE, 1420
INTOHIDEGEXPRESSION, 1422
INTOHIDELEXPRESSION, 1422
INUSE, 1450
INVERSEFACTORIAL, 1391
INVERSEFUNCTION, 1399
INVERSETABLE, 1442
ISCOMPRESSED, 1429
ISFACTORIZED, 1429
ISFORTRAN90, 1385

- ISINRHS, [1429](#)
- ISLYNDON, [1445](#)
- ISLYNDONR, [1445](#)
- ISNOTFORTRAN90, [1385](#)
- ISUNMODIFIED, [1429](#)
- ISYMBOL, [1403](#)
- ISZERO, [1429](#)
- KEEPZERO, [1430](#)
- LBRACE, [1379](#)
- LESS, [1435](#)
- LESSEQUAL, [1435](#)
- LEVICIVITA, [1391](#)
- LHSIDE, [1382](#)
- LHSIDEX, [1382](#)
- LI2FUNCTION, [1398](#)
- LINFUNCTION, [1398](#)
- LISTEDLOOP, [1371](#)
- LNFUNCTION, [1396](#)
- LNUMBER, [1389](#)
- LOADDOLLAR, [1417](#)
- LOCALEXPRESSION, [1420](#)
- LONGNUMBER, [1433](#)
- LOOKINGFORELSE, [1369](#)
- LOOKINGFORENDIF, [1369](#)
- LPARENTHESIS, [1378](#)
- LRPARENTHESSES, [1381](#)
- MAINSORT, [1428](#)
- MAKEARGS, [1444](#)
- MAKERATIONAL, [1399](#)
- MAPLEMODE, [1383](#)
- MATCH, [1434](#)
- MATCHFUNCTION, [1393](#)
- MATHEMATICAMODE, [1383](#)
- MAXBUILTINFUNCTION, [1402](#)
- MAXFUNCTION, [1392](#)
- MAXPOWEROF, [1395](#)
- MAXRANGEINDICATOR, [1444](#)
- MINFUNCTION, [1392](#)
- MINPOWEROF, [1395](#)
- MINSPEC, [1428](#)
- MINVECTOR, [1388](#)
- MIXED, [1373](#)
- MIXED2, [1373](#)
- MOD2FUNCTION, [1392](#)
- MODFUNCTION, [1392](#)
- MODLOCAL, [1441](#)
- MODMAX, [1441](#)
- MODMIN, [1441](#)
- MODNONE, [1441](#)
- MODSUM, [1441](#)
- MODULEINSTACK, [1368](#)
- MOEBIUS, [1400](#)
- MPLFUNCTION, [1402](#)
- MULFUNCTION, [1400](#)
- MULTIPLEOF, [1434](#)
- MULTIPLYARG, [1446](#)
- MUSTCLEANPRF, [1418](#)
- MZV, [1401](#)
- MZVHALF, [1402](#)
- NAMENOTFOUND, [1438](#)
- NEG0_, [1425](#)
- NEG_, [1425](#)
- NEWFACTARG, [1447](#)
- NEWLYDEFINEDEXPRESSION, [1424](#)
- NEWSTATEMENT, [1369](#)
- NMIN4SHIFT, [1386](#)
- NOAUTO, [1373](#)
- NOCHECKLOGTYPE, [1386](#)
- NODEFUNCTION, [1400](#)
- NODOUBLEMASK, [1384](#)
- NOFUNPOWERS, [1439](#)
- NOINDEX, [1427](#)
- NOINVERSES, [1442](#)
- NOLYNDON, [1447](#)
- NONBIPART, [1456](#)
- NOPARALLEL_CONVPOLY, [1367](#)
- NOPARALLEL_DOLLAR, [1367](#)
- NOPARALLEL_NPROC, [1367](#)
- NOPARALLEL_RHS, [1367](#)
- NOPARALLEL_SPECTATOR, [1367](#)
- NOPARALLEL_TBLDOLLAR, [1367](#)
- NOPARALLEL_USER, [1367](#)
- NOQUADMASK, [1384](#)
- NORMALFORMAT, [1384](#)
- NORMALIZEFLAG, [1433](#)
- NOSIGMA, [1455](#)
- NOSNAIL, [1455](#)
- NOSPACEFORMAT, [1384](#)
- NOTADPOLE, [1455](#)
- NOTEQUAL, [1436](#)
- NOTFLOOP, [1457](#)
- NOTRICK, [1385](#)
- NOTSIMPLE, [1456](#)
- NOUNPACK, [1443](#)
- NUMARG, [1444](#)
- NUMARGSFUN, [1392](#)
- NUMERATORSYMBOL, [1403](#)
- NUMERICLOOP, [1371](#)
- NUMFACTORS, [1398](#)
- NUMSPECSETS, [1429](#)
- NUMTERMSFUN, [1394](#)
- NUMTOIND, [1418](#)
- NUMTONUM, [1417](#)
- NUMTOSUB, [1418](#)
- NUMTOSYM, [1418](#)
- O_BACKWARD, [1449](#)
- O_CSE, [1448](#)
- O_CSEGREEDY, [1448](#)
- O_FORWARD, [1449](#)
- O_FORWARDANDBACKWARD, [1449](#)
- O_FORWARDORBACKWARD, [1449](#)
- O_GREEDY, [1448](#)
- O_MCTS, [1449](#)
- O_NONE, [1448](#)
- O_OCCURRENCE, [1449](#)
- O_SIMULATED_ANNEALING, [1449](#)

ODD_, 1426
OFFSHELL, 1455
OLDFACTARG, 1447
OLDSTATEMENT, 1369
ONEEXPRESSION, 1371
ONEPARTI, 1454
ONEPARTR, 1455
ONEPI, 1401
ONLYFUNCTIONS, 1450
ONSHELL, 1455
OPTHEAD, 1449
ORCOND, 1436
ORDEREDSET, 1453
PARALLELFLAG, 1368
PARTEST, 1373
PARTITIONS, 1399
PATTERNFUNCTION, 1394
PERMUTATIONS, 1399
PERMUTEARG, 1445
PFORTRANMODE, 1383
PHI, 1401
PIPESTREAM, 1364
PISYMBOL, 1403
POLYADD, 1440
POLYDIV, 1440
POLYFUN, 1367
POLYGCD, 1440
POLYINTFAC, 1440
POLYMUL, 1440
POLYNORM, 1441
POLYREM, 1440
POLYSUB, 1440
POS0_, 1425
POS_, 1425
POSITIVEONLY, 1442
POSNEG, 1442
PRECALCSTREAM, 1364
PRELOWERAFTER, 1368
PRENOACTION, 1368
PREPROONLY, 1369
PRERAISEAFTER, 1368
PREREADSTREAM, 1364
PREREADSTREAM2, 1364
PREREADSTREAM3, 1365
PRETYPEDO, 1372
PRETYPEIF, 1372
PRETYPEINSIDE, 1372
PRETYPENONE, 1371
PRETYPEPROCEDURE, 1372
PRETYPESWITCH, 1372
PREVARSTREAM, 1364
PRIMENUMBER, 1399
PRINTALL, 1424
PRINTCONTENT, 1423
PRINTCONTENTS, 1423
PRINTLFILE, 1424
PRINTOFF, 1423
PRINTON, 1423
PRINTONEFUNCTION, 1424
PRINTONETERM, 1424
PROCEDUREFILE, 1365
PROPERORDERFLAG, 1439
PUTFIRST, 1399
PVFUNV, 1457
PVFUNWP, 1457
Q_, 1426
QUADRUPLEFORTRANMODE, 1384
QUADRUPLEPRECISIONFLAG, 1384
RANDOMFUNCTION, 1398
RANPERM, 1398
RATIO, 1414
RATIONALMODE, 1429
RBRACE, 1379
RCYCLESYMMETRIC, 1431
READSPECTATORFLAG, 1453
REDEFINEDEXPRESSION, 1424
REDUCEMODE, 1383
REGULAR, 1439
REGULAREXPRESSION, 1424
REGULARSYMBOL, 1448
REMFUNCTION, 1395
REPLACEARG, 1444
REPLACELOOP, 1439
REPLACEMENT, 1390
REVERSEARG, 1445
REVERSEFILESTREAM, 1365
REVERSEFUNCTION, 1390
REVERSEORDER, 1431
RH SIDE, 1382
ROOTFUNCTION, 1394
RPARENTHESIS, 1378
SEARCHINGPRECASE, 1369
SEARCHINGPREENDSWITCH, 1369
SELECTARG, 1446
SEPARATESYMBOL, 1404
SETBIT, 1437
SETEXP, 1388
SETFUNCTION, 1389
SETNUMBER, 1417
SETONLY, 1374
SETSET, 1387
SETTONUM, 1417
SETUPFILE, 1366
SIGFUNCTION, 1392
SIGMA, 1455
SIGNFUN, 1392
SIMPLE, 1456
SINFUNCTION, 1396
SINHFUNCTION, 1397
SIZEOFFUNCTION, 1400
SKIP, 1420
SKIPGEXPRESSION, 1421
SKIPLEXPRESSION, 1420
SKIPUNHIDEGEXPRESSION, 1423
SKIPUNHIDELEXPRESSION, 1423
SNAIL, 1455

SNUMBER, 1388
SORT, 1436
SORTANTIPOWER, 1386
SORTHIGHFIRST, 1385
SORTLOWFIRST, 1385
SORTMODULE, 1366
SORTPOWERFIRST, 1385
SPARSETABLE, 1453
SPECIFICATION, 1372
SPECTATOREXPRESSION, 1423
SQRTFUNCTION, 1396
STATEMENT, 1373
STATSMERGETOFILE, 1386
STATSOUT, 1371
STATSPOSTSORT, 1386
STATSSPLITMERGE, 1386
STORE, 1436
STOREDEXPRESSION, 1421
STOREMODULE, 1366
SUBAFTER, 1432
SUBAFTERNOT, 1432
SUBALL, 1432
SUBDISORDER, 1432
SUBEXPR, 1434
SUBEXPRESSION, 1387
SUBEXPSIZE, 1419
SUBMANY, 1432
SUBMASK, 1432
SUBMULTI, 1431
SUBONCE, 1431
SUBONLY, 1431
SUBROUTINEFILE, 1365
SUBSELECT, 1432
SUBSORT, 1428
SUBTERMUSED1, 1419
SUBTERMUSED2, 1419
SUBVECTOR, 1432
SUMF1, 1390
SUMF2, 1390
SUMMEDIND, 1427
SUMNUM1, 1415
SUMNUM2, 1415
SYMBOL, 1386
SYMBOL_, 1426
SYMBOLONLY, 1374
SYMMETRIC, 1431
SYMMETRIZE, 1415
SYMTONUM, 1415
SYMTOSUB, 1415
SYMTOSYM, 1415
TABLEBASEFILE, 1366
TABLEFUNCTION, 1394
TABLESTUB, 1395
TADPOLE, 1456
TAKETRACE, 1414
TANFUNCTION, 1396
TANHFUNCTION, 1397
TCOMMA, 1379
TCONJUGATE, 1381
TDELTA, 1378
TDIVIDE, 1380
TDOLLAR, 1378
TDOT, 1379
TDOTPRODUCT, 1377
TDUBIOUS, 1378
EMPTY, 1381
TENDOFIT, 1380
TENSORFUNCTION, 1425
TENVEC, 1415
TERMFUNCTION, 1394
TERMSINBRACKET, 1395
TERMSINEXPR, 1394
TEXPRESSION, 1377
TFLOAT, 1382
TFUN, 1457
TFUN1, 1457
TFUNCLOSE, 1379
TFUNCTION, 1377
TFUNOPEN, 1379
TGENINDEX, 1381
THETA, 1393
THETA2, 1393
THREADSDEBUG, 1370
TINDEX, 1377
TMINUS, 1380
TMPPOLYFUN, 1388
TMULTIPLY, 1379
TNOT, 1380
TNUMBER, 1378
TNUMBER1, 1381
TOBEFACTORED, 1430
TOBEUNFACTORED, 1430
TOFLOAT, 1401
TOLYNDON, 1445
TOLYNDONR, 1446
TOOUTPUT, 1373
TOPO, 1400
TOPOLOGIES, 1400
TOPOLOGIESONLY, 1453
TOPOLYNOMIALFLAG, 1447
TORAT, 1401
TPLUS, 1380
TPOWER, 1380
TPOWER1, 1381
TSET, 1377
TSETCLOSE, 1380
TSETDOL, 1382
TSETNUM, 1381
TSETOPEN, 1380
TSGAMMA, 1381
TSUBEXP, 1378
TSYMBOL, 1377
TVECTOR, 1377
TWILDARG, 1379
TWILDCARD, 1378
TYPEADJUSTBOUNDS, 1410

TYPEALLLOOPS, 1414
TYPEALLPATHS, 1414
TYPEAPPLY, 1411
TYPEAPPLYRESET, 1411
TYPEARG, 1407
TYPEARGEXPLODE, 1412
TYPEARGHEADSIZE, 1419
TYPEARGIMPLODE, 1412
TYPEARGTOEXTRASymbol, 1413
TYPEASSIGN, 1409
TYPECANONICALIZE, 1413
TYPECHAININ, 1411
TYPECHAINOUT, 1411
TYPECHISHOLM, 1407
TYPECLEARUSERFLAG, 1414
TYPECOUNT, 1406
TYPEDENOMINATORS, 1412
TYPEDETCURDUM, 1410
TYPEDISCARD, 1406
TYPEDOLOOP, 1413
TYPEDROPCOEFFICIENT, 1412
TYPEDROPSYMBOLS, 1413
TYPEELIF, 1407
TYPEELSE, 1406
TYPEENDDOLOOP, 1413
TYPEENDIF, 1407
TYPEENDREPEAT, 1406
TYPEENDSWITCH, 1414
TYPEEXIT, 1408
TYPEEXPRESSION, 1405
TYPEFACTARG, 1409
TYPEFACTARG2, 1409
TYPEFACTOR, 1411
TYPEFINDLOOP, 1409
TYPEFPRINT, 1408
TYPEFROMPOLYNOMIAL, 1412
TYPEGOTO, 1406
TYPEIDNEW, 1405
TYPEIDOLD, 1405
TYPEIF, 1406
TYPEINEXPRESSION, 1410
TYPEINSIDE, 1410
TYPEISFUN, 1382
TYPEISMYSTERY, 1382
TYPEISSUB, 1382
TYPEMERGE, 1411
TYPEMULT, 1406
TYPENORM, 1407
TYPENORM2, 1407
TYPENORM3, 1408
TYPENORM4, 1411
TYPEOPERATION, 1405
TYPEPRINT, 1408
TYPEPUTINSIDE, 1413
TYPEREDEFPRE, 1408
TYPERENUMBER, 1409
TYPEREPEAT, 1406
TYPEREVERSE, 1407
TYPESETEXIT, 1408
TYPESETUSERFLAG, 1414
TYPESORT, 1410
TYPESPLITARG, 1408
TYPESPLITARG2, 1408
TYPESPLITFIRSTARG, 1410
TYPESPLITLASTARG, 1410
TYPESTUFFLE, 1412
TYPESUM, 1407
TYPESUMFIX, 1409
TYPESWITCH, 1413
TYPETERM, 1410
TYPETESTUSE, 1411
TYPETOPOLYNOMIAL, 1412
TYPETOSPECTATOR, 1413
TYPETRANSFORM, 1412
TYPETRY, 1409
TYPEUNRAVEL, 1409
UNHIDE, 1420
UNHIDEGEXPRESSION, 1422
UNHIDELEXPRESSION, 1422
UNPACK, 1443
USEDFLAG, 1428
VARNAMES, 1443
VARTYPECOMPLEX, 1430
VARTYPEIMAGINARY, 1430
VARTYPEMINUS, 1430
VARTYPENONE, 1430
VARTYPEROOTOFUNITY, 1430
VECTBIT, 1437
VECTOMIN, 1416
VECTOR, 1387
VECTOR_, 1426
VECTORONLY, 1374
VECTOSUB, 1416
VECTOVEC, 1416
VERTEXFUNCTION, 1424
VORTRANMODE, 1383
WILDARGFUN, 1396
WILDARGINDEX, 1405
WILDARGSYMBOL, 1403
WILDARGVECTOR, 1405
WILDCARDS, 1417
WILDDUMMY, 1415
WITHAUTO, 1373
WITHBLOCKS, 1454
WITHEDGES, 1454
WITHERROR, 1363
WITHONEPISETS, 1454
WITHOUTERROR, 1363
WITHOUTNODES, 1454
WITHOUTSEMICOLON, 1368
WITHSEMICOLON, 1368
WITHSYMMETRIZEF, 1454
WITHSYMMETRIZEI, 1454
WRITEOUT, 1371
YESLYNDON, 1447
Z_, 1426

- ZTOHARG, 1446
- full
 - PF_BUFFER, 335
- FullCleanUp
 - declare.h, 908
 - module.c, 1756
- FULLMAX
 - form3.h, 1328
- fullname
 - dbase, 80
 - P_const, 317
- fullnamesize
 - P_const, 321
- FullProto
 - N_const, 248
- FullRenumbrer
 - declare.h, 952
 - reshuf.c, 2605
- FullSymmetrize
 - declare.h, 863
 - function.c, 1472
- FUN_INFO, 141
 - commute, 142
 - location, 142
 - numargs, 142
 - numfunnies, 142
 - numwildcards, 142
 - symmet, 142
 - tensor, 142
- FunArg
 - T_const, 442
- funargs
 - N_const, 251
- FUNBIT
 - ftypes.h, 1437
- func
 - KEYWORD, 162
 - ReNuMbEr, 384
- FUNCTION
 - ftypes.h, 1388
- FuNcTiOn, 143
 - commute, 144
 - complex, 144
 - dimension, 146
 - flags, 145
 - maxnumargs, 146
 - minnumargs, 146
 - name, 144
 - namesize, 145
 - node, 145
 - number, 144
 - spec, 145
 - symmetric, 145
 - symminfo, 144
 - tabl, 144
- function.c, 1470, 1473
 - ChainIn, 1472
 - ChainOut, 1473
 - CompGroup, 1472
 - FullSymmetrize, 1472
 - MakeDirty, 1471
 - MarkDirty, 1471
 - MatchFunction, 1473
 - PolyFunClean, 1471
 - PolyFunDirty, 1471
 - ScanFunctions, 1473
 - SymFind, 1472
 - SymGen, 1472
 - Symmetrize, 1471
- FunctionList
 - C_const, 43
- FUNCTIONONLY
 - ftypes.h, 1374
- FUNCTIONS
 - structs.h, 2895
- Functions
 - C_const, 46
- functions
 - variable.h, 3190
- FUNCTIONSORT
 - ftypes.h, 1428
- FUNHEAD
 - ftypes.h, 1419
- funinds
 - N_const, 251
- FunInfo
 - N_const, 252
- funisize
 - N_const, 256
- FunLevel
 - declare.h, 834
 - reshuf.c, 2605
- funlocs
 - N_const, 251
- FunMatchCy
 - declare.h, 842
 - symmetr.c, 2925
- FunMatchSy
 - declare.h, 843
 - symmetr.c, 2925
- FunNam
 - FixedGlobals, 135
- funnum
 - ReNuMbEr, 385
- FUNNYDOLLAR
 - ftypes.h, 1428
- FUNNYVEC
 - ftypes.h, 1427
- FUNNYWILD
 - ftypes.h, 1427
- funoffset
 - R_const, 382
- funpowers
 - C_const, 57
- FunScratch
 - N_const, 252

- funsized
 - NeStInG, [272](#)
- FunSorts
 - N_const, [253](#)
- FUNTOFUN
 - ftypes.h, [1416](#)
- funwith
 - DICTIONARY, [83](#)
- fval
 - OPTIMIZE, [295](#)
- fwin.h, [1496](#), [1498](#)
 - ALTSEPARATOR, [1497](#)
 - CARRIAGEReturn, [1496](#)
 - LINEFEED, [1496](#)
 - P_term, [1497](#)
 - PATHSEPARATOR, [1497](#)
 - SEPARATOR, [1497](#)
 - WITH_ENV, [1497](#)
 - WITHRETURN, [1497](#)
 - WITHSYSTEM, [1497](#)
- fwrite_cached
 - checkpoint.c, [531](#)
- gamm
 - TrAcEs, [464](#)
- GAMMA
 - ftypes.h, [1389](#)
- GAMMA1
 - ftypes.h, [1427](#)
- GAMMA5
 - ftypes.h, [1427](#)
- gamma5
 - TrAcEs, [465](#)
- GAMMA6
 - ftypes.h, [1427](#)
- GAMMA7
 - ftypes.h, [1427](#)
- GAMMAFIVE
 - ftypes.h, [1390](#)
- GAMMAFUN
 - ftypes.h, [1402](#)
- GAMMAFUNCTION
 - ftypes.h, [1425](#)
- GAMMAI
 - ftypes.h, [1389](#)
- GAMMASEVEN
 - ftypes.h, [1390](#)
- GAMMASIX
 - ftypes.h, [1390](#)
- GarbHand
 - declare.h, [835](#)
 - sort.c, [2720](#)
- gcd
 - Fraction, [140](#)
- gcd_heuristic_failed, [146](#)
- gcd_heuristic_possible
 - polygcd.cc, [2250](#)
- gcd_linear_helper
 - polygcd.cc, [2251](#)
- gcd_mpoly
 - flintinterface.h, [1268](#)
- gcd_poly
 - flintinterface.h, [1268](#)
- GCDclean
 - declare.h, [998](#)
 - ratio.c, [2501](#)
- GCDFUNCTION
 - ftypes.h, [1395](#)
- GCDfunction
 - declare.h, [997](#)
 - ratio.c, [2498](#)
- GCDfunction3
 - declare.h, [997](#)
 - ratio.c, [2498](#)
- GcdLong
 - declare.h, [835](#)
 - reken.c, [2554](#)
- GcdLong_fix_args
 - optimize.cc, [1987](#)
- GCDMAX
 - reken.c, [2547](#)
- GCDterms
 - declare.h, [998](#)
 - ratio.c, [2500](#)
- gcmmod
 - M_const, [171](#)
- gcNumDollars
 - M_const, [182](#)
- gCnumpows
 - M_const, [185](#)
- gCodesFlag
 - M_const, [179](#)
- gDefDim
 - M_const, [177](#)
- gDefDim4
 - M_const, [177](#)
- GENCOMMUTE
 - ftypes.h, [1437](#)
- GenDiagrams
 - declare.h, [823](#)
 - diawrap.cc, [1052](#)
- GENERALFUNCTION
 - ftypes.h, [1425](#)
- generate
 - MGraph, [208](#)
 - SProcess, [409](#)
- generate_expression
 - optimize.cc, [1997](#)
- generate_instructions
 - optimize.cc, [1989](#)
- generate_output
 - optimize.cc, [1996](#)
- GenerateFactors
 - compiler.c, [715](#)
 - declare.h, [950](#)
- Generator
 - declare.h, [835](#)

- proces.c, [2425](#)
- GenLoops
 - declare.h, [1019](#)
 - lus.c, [1679](#)
- genNext
 - SGroup, [392](#)
- GENNONCOMMUTE
 - ftypes.h, [1437](#)
- GenPaths
 - declare.h, [1020](#)
 - lus.c, [1680](#)
- GenSpace
 - sOrT, [403](#)
- GenTerms
 - sOrT, [403](#)
- get_brackets
 - optimize.cc, [1984](#)
- get_expression
 - optimize.cc, [1983](#)
- get_variables
 - flintinterface.h, [1268](#)
 - poly, [352](#)
- GetAppendChannel
 - declare.h, [886](#)
 - tools.c, [3077](#)
- GetAutoName
 - declare.h, [878](#)
 - names.c, [1820](#)
- GETBIDENTITY
 - declare.h, [808](#)
- GetBinom
 - declare.h, [837](#)
 - reken.c, [2554](#)
- GETCFROMEXTCHANNEL
 - declare.h, [809](#)
- GetChannel
 - declare.h, [886](#)
 - tools.c, [3077](#)
- GetChar
 - declare.h, [893](#)
 - pre.c, [2311](#)
- GETCOEF
 - declare.h, [789](#)
- GetContent
 - declare.h, [966](#)
 - execute.c, [1154](#)
- getCurrentExternalChannel
 - declare.h, [979](#)
 - extcmd.c, [1188](#)
- GetDbase
 - declare.h, [974](#)
 - minos.c, [1722](#)
- getDef
 - Options, [303](#)
- GetDollar
 - declare.h, [909](#)
 - names.c, [1820](#)
- GetDollarNumber
 - declare.h, [897](#)
 - pre.c, [2326](#)
- GetDolNum
 - declare.h, [957](#)
 - dollar.c, [1092](#)
- GetDoParam
 - compcomm.c, [617](#)
 - declare.h, [996](#)
- GetFile
 - R_const, [379](#)
- GetFirstBracket
 - declare.h, [966](#)
 - execute.c, [1154](#)
- GetFirstTerm
 - declare.h, [966](#)
 - execute.c, [1154](#)
- getFLines
 - EGraph, [108](#)
- GetFloatArgument
 - evaluate.c, [1141](#)
- GetFromSpectator
 - declare.h, [1013](#)
 - spectator.c, [2782](#)
- GetFromStore
 - declare.h, [838](#)
 - store.c, [2825](#)
- GetFromStream
 - declare.h, [875](#)
 - tools.c, [3072](#)
- GetFunction
 - declare.h, [878](#)
 - names.c, [1819](#)
- GETIDENTITY
 - declare.h, [808](#)
- GetIfDollarFactor
 - compcomm.c, [617](#)
 - declare.h, [996](#)
- GetIfDollarNum
 - declare.h, [827](#)
 - if.c, [1647](#)
- GetInput
 - declare.h, [892](#)
 - pre.c, [2311](#)
- GetLabel
 - declare.h, [948](#)
 - names.c, [1827](#)
- GetLastExprName
 - declare.h, [878](#)
 - names.c, [1819](#)
- getLegParticle
 - Assign, [19](#)
- GetLong
 - declare.h, [838](#)
 - reken.c, [2554](#)
- GetLongModInverses
 - declare.h, [856](#)
 - reken.c, [2553](#)
- GetModInverses

- declare.h, [855](#)
- reken.c, [2552](#)
- GetMoreFromMem
 - declare.h, [838](#)
 - store.c, [2825](#)
- GetMoreTerms
 - declare.h, [838](#)
 - store.c, [2825](#)
- GetName
 - declare.h, [877](#)
 - names.c, [1819](#)
- GetNode
 - declare.h, [877](#)
 - names.c, [1819](#)
- GetNumber
 - declare.h, [878](#)
 - names.c, [1819](#)
- getOldDef
 - Options, [303](#)
- GetOName
 - declare.h, [879](#)
 - names.c, [1820](#)
- GetOneFile
 - R_const, [380](#)
- GetOneTerm
 - declare.h, [838](#)
 - store.c, [2825](#)
- GetPiArgument
 - evaluate.c, [1141](#)
- GetPID
 - form3.h, [1332](#)
- GetPosFile
 - declare.h, [885](#)
 - tools.c, [3076](#)
- getpower
 - optimize.cc, [1985](#)
- GetPreVar
 - declare.h, [891](#)
 - pre.c, [2312](#)
- getQGDef
 - Options, [303](#)
- GetRunningTime
 - declare.h, [852](#)
 - startup.c, [2795](#)
- GetSetupPar
 - declare.h, [891](#)
 - setfile.c, [2685](#)
- GETSTOP
 - declare.h, [789](#)
- GetStreamPosition
 - declare.h, [913](#)
 - tools.c, [3073](#)
- GetTable
 - declare.h, [839](#)
 - store.c, [2827](#)
- GetTableName
 - declare.h, [977](#)
 - minos.c, [1723](#)
- GETTERM
 - declare.h, [809](#)
- GetTerm
 - declare.h, [839](#)
 - store.c, [2824](#)
- getValue
 - Options, [301](#)
- GetVar
 - declare.h, [878](#)
 - names.c, [1820](#)
- GetWildcardName
 - declare.h, [912](#)
 - names.c, [1822](#)
- gextrasym
 - M_const, [173](#)
- gextrasymbols
 - M_const, [190](#)
- gFinalStats
 - M_const, [180](#)
- GFIVE
 - ftypes.h, [1433](#)
- gFortran90Kind
 - M_const, [172](#)
- gfunpowers
 - M_const, [178](#)
- ggextrasym
 - M_const, [173](#)
- ggextrasymbols
 - M_const, [190](#)
- ggFinalStats
 - M_const, [180](#)
- ggIndentSpace
 - M_const, [190](#)
- ggNoSpacesInNumbers
 - M_const, [182](#)
- ggnumextrasym
 - M_const, [183](#)
- ggOldFactArgFlag
 - M_const, [183](#)
- ggOldGCDflag
 - M_const, [183](#)
- ggOldParallelStats
 - M_const, [181](#)
- ggProcessStats
 - M_const, [181](#)
- ggShortStatsMax
 - M_const, [190](#)
- ggStatsFlag
 - M_const, [186](#)
- ggThreadBalancing
 - M_const, [180](#)
- ggThreadBucketSize
 - M_const, [176](#)
- ggThreadsFlag
 - M_const, [180](#)
- ggThreadSortFileSynch
 - M_const, [181](#)
- ggThreadStats

- M_const, 180
- ggWTimeStatsFlag
 - M_const, 184
- GIDENT
 - ftypes.h, 1433
- gIndentSpace
 - M_const, 190
- ginsidefirst
 - M_const, 177
- glsFortran90
 - M_const, 182
- gLineLength
 - M_const, 186
- GLOBAL
 - ftypes.h, 1436
- GLOBALEXPRESSION
 - ftypes.h, 1421
- GlobalFunctions
 - variable.h, 3191
- GlobalIndices
 - variable.h, 3191
- Globalize
 - declare.h, 913
 - names.c, 1828
- GLOBALMODULE
 - ftypes.h, 1366
- globalnamefill
 - NaMeTree, 268
- globalnodefill
 - NaMeTree, 268
- GlobalSetElements
 - variable.h, 3192
- GlobalSets
 - variable.h, 3192
- GLOBALSPECTATORFLAG
 - ftypes.h, 1453
- GlobalSymbols
 - variable.h, 3191
- GlobalVectors
 - variable.h, 3192
- Glue
 - declare.h, 839
 - ratio.c, 2498
- GMINUS
 - ftypes.h, 1433
- gmodmode
 - M_const, 185
- GMPSPREAD
 - float.c, 1282
- gNamesFlag
 - M_const, 179
- gncmod
 - M_const, 184
- gNoSpacesInNumbers
 - M_const, 182
- gnpowmod
 - M_const, 184
- GNUC_PREREQ
 - parallel.h, 2133
- gNumDictionaries
 - O_const, 287
- gnumelements
 - DICTIONARY, 83
- gnumextrasym
 - M_const, 183
- gNumPre
 - P_const, 321
- gOldFactArgFlag
 - M_const, 182
- gOldGCDflag
 - M_const, 183
- gOldParallelStats
 - M_const, 181
- gOutNumberType
 - M_const, 185
- gOutputMode
 - M_const, 185
- gOutputSpaces
 - M_const, 185
- gparallelflag
 - M_const, 179
- GPLUS
 - ftypes.h, 1433
- gPolyFun
 - M_const, 188
- gPolyFunExp
 - M_const, 189
- gPolyFunInv
 - M_const, 189
- gPolyFunPow
 - M_const, 189
- gPolyFunType
 - M_const, 189
- gPolyFunVar
 - M_const, 189
- gpowmod
 - M_const, 171
- gProcessBucketSize
 - M_const, 175
- gProcessStats
 - M_const, 181
- gproperorderflag
 - M_const, 179
- grcc.cc, 1498
- grcc.h, 1626
- grccmodel
 - MoDeL, 228
- grccparam.h, 1643
- GREATER
 - ftypes.h, 1435
- GREATEREQUAL
 - ftypes.h, 1435
- greedymaxperc
 - OPTIMIZE, 297
- greedyminnum
 - OPTIMIZE, 296

- greedytimelimit
 - OPTIMIZE, 296
- group
 - MGraph, 211
- groupLMom
 - EGraph, 107
- gShortStatsMax
 - M_const, 190
- gSizeCommuteInSet
 - M_const, 179
- gSortType
 - M_const, 179
- gStatsFlag
 - M_const, 178
- gSubId
 - EGraph, 110
- gThreadBalancing
 - M_const, 180
- gThreadBucketSize
 - M_const, 175
- gThreadsFlag
 - M_const, 180
- gThreadSortFileSynch
 - M_const, 181
- gThreadStats
 - M_const, 180
- gTokensWriteFlag
 - M_const, 178
- gUniTrace
 - M_const, 185
- gUnitTrace
 - M_const, 185
- gWTimeStatsFlag
 - M_const, 183
- gzipCompress
 - R_const, 378
- GZIPDEFAULT
 - fsizes.h, 1345
- HAAKJE
 - ftypes.h, 1389
- HAAKJE0
 - ftypes.h, 1388
- halfmod
 - C_const, 48
- Handle
 - FiLeDaTa, 132
- handle
 - ChAnNeL, 75
 - dbase, 79
 - FiLe, 130
 - StreaM, 423
- HANDLERS, 146
 - newhandle, 147
 - newlogonly, 147
 - oldhandle, 147
 - oldlogonly, 147
 - oldprinttype, 147
 - oldsilent, 147
- hash
 - node, 276
- hash_range
 - optimize.cc, 1989
- havesortdir
 - M_const, 191
- HEADERFILE
 - ftypes.h, 1365
- headermark
 - STOREHEADER, 418
- headnode
 - NaMeTree, 268
- helptb
 - tables.c, 2959
- hi
 - VaRrEnUm, 470
- HIDDENGEXPRESSION
 - ftypes.h, 1421
- HIDDENLEXPRESSION
 - ftypes.h, 1421
- HIDE
 - ftypes.h, 1420
- hidefile
 - R_const, 376
- HIDEGEXPRESSION
 - ftypes.h, 1422
- HideLevel
 - C_const, 68
- hidelevel
 - ExPrEsSiOn, 122
- HIDELEXPRESSION
 - ftypes.h, 1422
- HideSize
 - M_const, 173
- highfirst
 - setfile.c, 2687
- HighWarning
 - declare.h, 873
 - message.c, 1706
- HoldFlag
 - M_const, 187
- horner
 - OPTIMIZE, 295
- Horner_tree
 - optimize.cc, 1988
- hornerdirection
 - OPTIMIZE, 296
- HowMany
 - declare.h, 972
 - if.c, 1647
- hparallelflag
 - M_const, 179
- HPLFUNCTION
 - ftypes.h, 1402
- hProcessBucketSize
 - M_const, 175
- HTOZARG
 - ftypes.h, 1446

- humanBytesSuffix
 - sort.c, [2725](#)
- HumanStatsFlag
 - C_const, [62](#)
- HumanString
 - sort.c, [2711](#)
- HUMANSTRLEN
 - sort.c, [2710](#)
- HUMANSUFFLEN
 - sort.c, [2710](#)
- humanTermsSuffix
 - sort.c, [2725](#)
- i
 - bufIPstruct, [37](#)
- iblocks
 - dbase, [79](#)
- iBufError
 - P_const, [319](#)
- iBuffer
 - C_const, [49](#)
- iBufferSize
 - C_const, [53](#)
- icode
 - lInput, [148](#)
 - Interaction, [161](#)
- id
 - EEdge, [98](#)
 - ENode, [117](#)
 - Interaction, [160](#)
 - MNode, [221](#)
 - Particle, [330](#)
 - Process, [370](#)
 - SProcess, [413](#)
- idallflag
 - T_const, [439](#)
- idallmaxnum
 - T_const, [439](#)
- idallnum
 - T_const, [439](#)
- IDFUNCTION
 - ftypes.h, [1399](#)
- idfunctionflag
 - N_const, [261](#)
- IDHEAD
 - ftypes.h, [1433](#)
- idoption
 - C_const, [56](#)
- iexp
 - declare.h, [888](#)
 - tools.c, [3084](#)
- if.c, [1646](#), [1649](#)
 - DoEndSwitch, [1647](#)
 - DofStatement, [1647](#)
 - DoSwitch, [1647](#)
 - DoubleIfBuffers, [1647](#)
 - DoubleSwitchBuffers, [1648](#)
 - FindCase, [1648](#)
 - FindVar, [1647](#)
 - GetIfDollarNum, [1647](#)
 - HowMany, [1647](#)
 - SwitchSplitMerge, [1648](#)
 - SwitchSplitMergeRec, [1648](#)
- IfCount
 - C_const, [48](#)
- IFDOLLAR
 - ftypes.h, [1434](#)
- IFDOLLAREXTRA
 - ftypes.h, [1434](#)
- IFEXPRESSION
 - ftypes.h, [1434](#)
- IFFLOATNUMBER
 - ftypes.h, [1435](#)
- IfHeap
 - C_const, [48](#)
- IFISFACTORIZED
 - ftypes.h, [1434](#)
- IfLevel
 - C_const, [67](#)
- iflevel
 - SWITCH, [427](#)
- IFOCCURS
 - ftypes.h, [1435](#)
- IfStack
 - C_const, [49](#)
- IfSumCheck
 - C_const, [52](#)
- IFUSERFLAG
 - ftypes.h, [1435](#)
- IFW
 - flintinterface.cc, [1237](#)
- IgnoreDeprecation
 - M_const, [191](#)
- lInput, [148](#)
 - cvallist, [149](#)
 - icode, [148](#)
 - name, [148](#)
 - nplistn, [148](#)
 - plistc, [149](#)
 - plistn, [148](#)
- ilist
 - NCand, [270](#)
 - NStack, [279](#)
- IMPLODEARG
 - ftypes.h, [1445](#)
- improve
 - optimization, [294](#)
- INANDOUT
 - minos.h, [1745](#)
- inbuffer
 - FiLe, [130](#)
 - Stream, [423](#)
- IncDir
 - M_const, [172](#)
- incdollar
 - DoLoOp, [92](#)
- INCEXPRESSION

- ftypes.h, [1421](#)
- INCLENG
 - declare.h, [789](#)
- Include
 - declare.h, [896](#)
 - pre.c, [2316](#)
- incMat
 - MNodeClass, [224](#)
- incnum
 - DoLoOp, [91](#)
- incoef
 - SHvariables, [395](#)
- IndDum
 - M_const, [184](#)
 - N_const, [261](#)
- INDENTSPACE
 - fsizes.h, [1346](#)
- IndentSpace
 - O_const, [287](#)
- INDEX
 - ftypes.h, [1387](#)
- InDeX, [149](#)
 - dimension, [150](#)
 - flags, [150](#)
 - name, [149](#)
 - namesize, [150](#)
 - nmin4, [150](#)
 - node, [150](#)
 - number, [150](#)
 - type, [149](#)
- Index
 - FiLeDaTa, [131](#)
- index
 - DoLIaRS, [88](#)
 - OptQGRef, [308](#)
 - PF_BUFFER, [335](#)
- index.c, [1663](#), [1664](#)
 - ClearBracketIndex, [1664](#)
 - FindBracket, [1663](#)
 - OpenBracketIndex, [1664](#)
 - PutBracketInIndex, [1663](#)
 - PutInside, [1664](#)
- INDEX_
 - ftypes.h, [1426](#)
- indexblock, [151](#)
 - flags, [151](#)
 - objects, [151](#)
 - position, [151](#)
 - previousblock, [151](#)
- indexbuffer
 - BrAcKeTiNfO, [33](#)
- indexbufferize
 - BrAcKeTiNfO, [34](#)
- INDEXENTRY
 - structs.h, [2894](#)
- InDeXeNtRy, [152](#)
 - CompressSize, [153](#)
 - length, [153](#)
 - name, [154](#)
 - nfunctions, [154](#)
 - nindices, [153](#)
 - nsymbols, [153](#)
 - nvectors, [154](#)
 - position, [153](#)
 - size, [154](#)
 - variables, [153](#)
- indexfill
 - BrAcKeTiNfO, [34](#)
- IndexList
 - C_const, [44](#)
- INDEXONLY
 - ftypes.h, [1374](#)
- indi
 - ReNuMbEr, [384](#)
- Indices
 - C_const, [46](#)
- indices
 - parallel.h, [2134](#)
 - variable.h, [3190](#)
- indnum
 - ReNuMbEr, [385](#)
- INDTOIND
 - ftypes.h, [1416](#)
- INDTOSUB
 - ftypes.h, [1416](#)
- inexprlevel
 - C_const, [65](#)
- inexprstack
 - C_const, [53](#)
- inexprsumcheck
 - C_const, [64](#)
- InFbrack
 - O_const, [288](#)
- infile
 - R_const, [376](#)
- INFILEINDEX
 - structs.h, [2893](#)
- info
 - dbase, [78](#)
- InFunction
 - declare.h, [839](#)
 - proces.c, [2421](#)
- InHiBuf
 - R_const, [378](#)
- inicbufs
 - comtool.c, [752](#)
 - declare.h, [886](#)
- inictable
 - compiler.c, [712](#)
 - declare.h, [886](#)
- IniFbuffer
 - comtool.c, [756](#)
 - declare.h, [997](#)
- iniinfo, [155](#)
 - entriesinindex, [155](#)
 - firstindexblock, [155](#)

- firstnameblock, 156
 - lastindexblock, 155
 - lastnameblock, 156
 - numberofindexblocks, 155
 - numberofnamesblocks, 156
 - numeroftables, 155
- IniLine
 - declare.h, 840
 - sch.c, 2646
- INILOCK
 - declare.h, 806
- IniModule
 - declare.h, 893
 - pre.c, 2313
- InInBuf
 - R_const, 377
- INIRWLOCK
 - declare.h, 807
- IniSpecialModule
 - declare.h, 893
 - pre.c, 2313
- init
 - MCBlock, 193
 - MConn, 198
 - MCOpI, 204
 - MGraph, 208
 - MNodeClass, 223
- initAss
 - ENode, 116
- initCEdges
 - MConn, 198
- initlc
 - FGInput, 127
- initln
 - FGInput, 127
- initlPart
 - Process, 370
- INITNAMESIZE
 - fsizes.h, 1341
- INITNODESIZE
 - fsizes.h, 1341
- iniTools
 - declare.h, 989
 - tools.c, 3082
- initPerm
 - MOrbits, 240
- initPresetExternalChannels
 - declare.h, 978
 - extcmd.c, 1188
- InitRecovery
 - checkpoint.c, 530
 - declare.h, 986
- inivar.h, 1673, 1674
 - A, 1673
 - BufferForOutput, 1674
 - FG, 1673
 - fixedsets, 1673
 - setupfilename, 1674
- IniVars
 - declare.h, 840
 - startup.c, 2794
- iniwranf
 - declare.h, 981
 - reken.c, 2557
- inline
 - form3.h, 1326
- inlist
 - TrAcEn, 459
 - TrAcEs, 462
- inmem
 - ExPrEsSiOn, 122
- InnerTest
 - C_const, 64
- inNum
 - sOrT, 405
- InNumMem
 - T_const, 438
- InOutBuf
 - P_const, 319
- inparallelflag
 - C_const, 60
- inPatches
 - sOrT, 401
- inprimelist
 - T_const, 444
- InputFileName
 - M_const, 172
- INPUTONLY
 - minos.h, 1745
- INPUTOUT
 - ftypes.h, 1370
- INPUTSTREAM
 - ftypes.h, 1365
- inscbuf
 - INSIDEINFO, 158
- inscheme
 - O_const, 284
- InScratch
 - N_const, 254
- InScratch1
 - N_const, 254
- InsertArg
 - argument.c, 479
 - declare.h, 919
- InsertTerm
 - declare.h, 840
 - proces.c, 2422
- inside
 - P_const, 317
- InsideDollar
 - declare.h, 963
 - dollar.c, 1087
- insidefirst
 - C_const, 55
- INSIDEINFO, 156
 - buffer, 157

- inscbuf, [158](#)
- numdollars, [157](#)
- oldcbuf, [157](#)
- oldcnumlhs, [158](#)
- oldcompiletype, [157](#)
- oldnumpotmoddollars, [157](#)
- oldparallelflag, [157](#)
- oldrbuf, [158](#)
- size, [157](#)
- insidelevel
 - C_const, [65](#)
- insidestack
 - C_const, [52](#)
- insidesumcheck
 - C_const, [64](#)
- INSLENGTH
 - sort.c, [2710](#)
- InsTableTree
 - declare.h, [961](#)
 - tables.c, [2954](#)
- InsTree
 - comtool.c, [756](#)
 - declare.h, [950](#)
- insubexpbuffers
 - compiler.c, [715](#)
- integer_lcoeff
 - poly, [350](#)
- IntegerToFloat
 - float.c, [1284](#)
- IntegerToFloatr
 - evaluate.c, [1140](#)
- Interact
 - M_const, [176](#)
- Interaction, [158](#)
 - ~Interaction, [159](#)
 - clist, [160](#)
 - csum, [160](#)
 - icode, [161](#)
 - id, [160](#)
 - Interaction, [159](#)
 - loop, [161](#)
 - mdl, [160](#)
 - name, [160](#)
 - nclist, [161](#)
 - nlegs, [161](#)
 - nplist, [161](#)
 - nslist, [161](#)
 - plist, [160](#)
 - prInteraction, [160](#)
 - slist, [160](#)
- interacts
 - Model, [235](#)
- internalset
 - TERMINFO, [456](#)
- INTFUNCTION
 - ftypes.h, [1393](#)
- INTOHIDE
 - ftypes.h, [1420](#)
- INTOHIDEGEXPRESSION
 - ftypes.h, [1422](#)
- INTOHIDELEXPRESSION
 - ftypes.h, [1422](#)
- intPart
 - Model, [235](#)
- intracestack
 - N_const, [256](#)
- intrct
 - DNode, [86](#)
 - ENode, [118](#)
- INUSE
 - ftypes.h, [1450](#)
- inverse_poly
 - flintinterface.h, [1268](#)
- INVERSEFACTORIAL
 - ftypes.h, [1391](#)
- INVERSEFUNCTION
 - ftypes.h, [1399](#)
- INVERSETABLE
 - ftypes.h, [1442](#)
- invertices
 - MoDeL, [228](#)
- invfacnum
 - M_const, [188](#)
- InvPoly
 - declare.h, [1001](#)
 - ratio.c, [2502](#)
- iPatches
 - sOrT, [402](#)
- iPointer
 - C_const, [49](#)
- iranf
 - declare.h, [981](#)
 - reken.c, [2558](#)
- is_dense_univariate
 - poly, [349](#)
- is_integer
 - poly, [349](#)
- is_monomial
 - poly, [349](#)
- is_one
 - poly, [349](#)
- is_zero
 - poly, [349](#)
- IsBracket
 - O_const, [288](#)
- ISCOMPRESSED
 - ftypes.h, [1429](#)
- isConnected
 - MGraph, [209](#)
- isEq
 - Fraction, [140](#)
- ISEQUALPOS
 - declare.h, [801](#)
- ISEQUALPOSINC
 - declare.h, [799](#)
- IsExponentSign

- declare.h, 1010
- dict.c, 1066
- isExternal
 - EGraph, 105
 - MGraph, 208
- ISFACTORIZED
 - ftypes.h, 1429
- isFermion
 - EGraph, 106
- ISFORTRAN90
 - ftypes.h, 1385
- IsFortran90
 - C_const, 61
- ISGEPOS
 - declare.h, 802
- ISGEPOSINC
 - declare.h, 800
- IsIdStatement
 - compiler.c, 713
 - declare.h, 946
- ISINRHS
 - ftypes.h, 1429
- isIsomorphic
 - Assign, 22
 - MGraph, 209
- ISLESSPOS
 - declare.h, 802
- IsLikeVector
 - declare.h, 889
 - tools.c, 3085
- ISLYNDON
 - ftypes.h, 1445
- ISLYNDONR
 - ftypes.h, 1445
- ISMINPOS
 - declare.h, 801
- IsMultipleOf
 - declare.h, 968
 - dollar.c, 1088
- IsMultiplySign
 - declare.h, 1011
 - dict.c, 1066
- ISNEGPOS
 - declare.h, 802
- isNeutral
 - Particle, 330
- isnextchar
 - StreaM, 424
- ISNOTEQUALPOS
 - declare.h, 801
- ISNOTFORTRAN90
 - ftypes.h, 1385
- ISNOTZEROPOS
 - declare.h, 802
- isOptE
 - EGraph, 105
- isOptM
 - MGraph, 210
- isOrdLegs
 - Assign, 22
- isOrdPLeg
 - Assign, 21
- ISPOSPOS
 - declare.h, 802
- IsRHS
 - compiler.c, 713
 - declare.h, 946
- IsSetMember
 - declare.h, 968
 - dollar.c, 1088
- iStop
 - C_const, 49
- ISUNMODIFIED
 - ftypes.h, 1429
- ISYMBOL
 - ftypes.h, 1403
- ISZERO
 - ftypes.h, 1429
- ISZEROPOS
 - declare.h, 802
- ival
 - OPTIMIZE, 295
- j3
 - TrAcEs, 462
- j4
 - TrAcEs, 462
- JUMPRATIO
 - fsizes.h, 1347
- jumpratio
 - M_const, 184
- KEEPZERO
 - ftypes.h, 1430
- KeptInHold
 - R_const, 379
- KEYWORD, 162
 - flags, 162
 - func, 162
 - name, 162
 - type, 162
- KEYWORDV, 163
 - flags, 163
 - name, 163
 - type, 163
 - var, 163
- KILL
 - pre.c, 2310
- KILLALL
 - pre.c, 2310
- killSignal
 - X_const, 475
- killWholeGroup
 - X_const, 475
- klow
 - ENode, 118
- kstep

- TrAcEs, [464](#)
- ktoi
 - sOrT, [401](#)
- l
 - node, [276](#)
- LabelNames
 - C_const, [49](#)
- Labels
 - C_const, [50](#)
- LargeSize
 - sOrT, [402](#)
- last
 - SeTs, [389](#)
- last_monomial
 - poly, [354](#)
- last_monomial_index
 - poly, [354](#)
- lastdollar
 - DoLoOp, [92](#)
- lastindexblock
 - iniinfo, [155](#)
- lastLen
 - longMultiStruct, [167](#)
- lastnameblock
 - iniinfo, [156](#)
- lastnamespace
 - P_const, [317](#)
- lastnum
 - DoLoOp, [91](#)
- LBRACE
 - ftypes.h, [1379](#)
- lBuffer
 - sOrT, [399](#)
- lc3
 - TrAcEs, [464](#)
- lc4
 - TrAcEs, [465](#)
- LcmLong
 - declare.h, [835](#)
 - reken.c, [2555](#)
- lcoeff_multivar
 - poly, [351](#)
- lcoeff_univar
 - poly, [351](#)
- lDefDim
 - C_const, [55](#)
- lDefDim4
 - C_const, [55](#)
- LeaveNegative
 - T_const, [441](#)
- left
 - NaMeNode, [263](#)
 - NoDe, [273](#)
 - tree, [466](#)
- legcouple
 - MoDeL, [228](#)
 - TERMINFO, [455](#)
- legEdgeParticle
 - Assign, [19](#)
- legPart
 - Assign, [20](#)
- legParticle
 - EGraph, [108](#)
- length
 - InDeXeNtRy, [153](#)
- lenLONG
 - STOREHEADER, [418](#)
- lenPOINTER
 - STOREHEADER, [418](#)
- lenPOS
 - STOREHEADER, [418](#)
- lenWORD
 - STOREHEADER, [418](#)
- LESS
 - ftypes.h, [1435](#)
- LESSEQUAL
 - ftypes.h, [1435](#)
- level
 - R_const, [381](#)
 - SHvariables, [397](#)
 - TERMINFO, [455](#)
 - TrAcEn, [460](#)
 - TrAcEs, [465](#)
- LEVICIVITA
 - ftypes.h, [1391](#)
- IFill
 - sOrT, [399](#)
- lhdollarerror
 - P_const, [321](#)
- lhdollarflag
 - C_const, [61](#)
- lhs
 - CbUf, [72](#)
 - DICTIONARY_ELEMENT, [84](#)
- LHSIDE
 - ftypes.h, [1382](#)
- LHSIDEX
 - ftypes.h, [1382](#)
- LI2FUNCTION
 - ftypes.h, [1398](#)
- lijst
 - LIST, [164](#)
- LINEFEED
 - fwin.h, [1496](#)
 - unix.h, [3181](#)
- LineLength
 - C_const, [68](#)
- linenumber
 - StreaM, [421](#)
- LINFUNCTION
 - ftypes.h, [1398](#)
- LinkTree
 - declare.h, [879](#)
 - names.c, [1821](#)
- LIST, [164](#)
 - lijst, [164](#)

- maxnum, [164](#)
- message, [164](#)
- num, [164](#)
- numclear, [165](#)
- numglobal, [165](#)
- numtemp, [165](#)
- size, [165](#)
- LISTEDLOOP
 - ftypes.h, [1371](#)
- listinprint
 - N_const, [253](#)
- ListPoly
 - T_const, [437](#)
- ListSymbols
 - T_const, [437](#)
- ListSymbolsSize
 - T_const, [440](#)
- lloser
 - NoDe, [273](#)
- lmom
 - EEdge, [99](#)
- ln2
 - evaluate.c, [1142](#)
- LNFUNCTION
 - ftypes.h, [1396](#)
- LNUMBER
 - ftypes.h, [1389](#)
- lo
 - VaRrEnUm, [470](#)
- LOADDOLLAR
 - ftypes.h, [1417](#)
- LoadInputFile
 - declare.h, [892](#)
 - tools.c, [3072](#)
- LoadInstruction
 - declare.h, [894](#)
 - pre.c, [2313](#)
- loadmode
 - PROCEDURE, [367](#)
- LoadModel
 - declare.h, [1018](#)
 - diawrap.cc, [1051](#)
- LoadStatement
 - declare.h, [894](#)
 - pre.c, [2313](#)
- LocalConvertToPoly
 - declare.h, [994](#)
 - notation.c, [1933](#)
- LOCALEXPRESSION
 - ftypes.h, [1420](#)
- LocateBase
 - declare.h, [860](#)
 - minos.c, [1721](#)
- LocateFile
 - declare.h, [876](#)
 - tools.c, [3072](#)
- location
 - FUN_INFO, [142](#)
- LOCK
 - declare.h, [807](#)
- locwildvalue
 - T_const, [442](#)
- log
 - ParallelVars, [325](#)
- LogFileName
 - M_const, [172](#)
- LogHandle
 - C_const, [68](#)
- LogType
 - M_const, [186](#)
- LONG
 - form3.h, [1332](#)
- LONG_MAX_VALUE
 - form3.h, [1328](#)
- LONG_MIN_VALUE
 - form3.h, [1327](#)
- LongCopy
 - declare.h, [910](#)
 - tools.c, [3081](#)
- LongLongCopy
 - declare.h, [910](#)
 - tools.c, [3081](#)
- longMultiStruct, [166](#)
 - buffer, [166](#)
 - bufpos, [166](#)
 - lastLen, [167](#)
 - next, [167](#)
 - nPacks, [166](#)
 - packpos, [166](#)
- LONGNUMBER
 - ftypes.h, [1433](#)
- LongToLine
 - declare.h, [841](#)
 - sch.c, [2647](#)
- LOOKINGFORELSE
 - ftypes.h, [1369](#)
- LOOKINGFORENDIF
 - ftypes.h, [1369](#)
- LookInStream
 - declare.h, [875](#)
 - tools.c, [3072](#)
- loop
 - Interaction, [161](#)
 - MCBlock, [193](#)
 - MCOpI, [205](#)
 - Process, [371](#)
 - SProcess, [413](#)
- LoopList
 - P_const, [316](#)
- loopm
 - EGraph, [114](#)
- LoopOutput
 - declare.h, [1019](#)
 - lus.c, [1679](#)
- LowerSortLevel
 - declare.h, [962](#)

- sort.c, [2724](#)
- lowfirst
 - setfile.c, [2688](#)
- LPARENTHESIS
 - ftypes.h, [1378](#)
- IPatch
 - sOrT, [404](#)
- IPolyFun
 - C_const, [68](#)
- IPolyFunExp
 - C_const, [69](#)
- IPolyFunInv
 - C_const, [68](#)
- IPolyFunPow
 - C_const, [69](#)
- IPolyFunType
 - C_const, [69](#)
- IPolyFunVar
 - C_const, [69](#)
- LRPARENTHESSES
 - ftypes.h, [1381](#)
- ISortType
 - C_const, [58](#)
- lsrc
 - NoDe, [274](#)
- ITop
 - sOrT, [399](#)
- IUniTrace
 - C_const, [64](#)
- IUnitTrace
 - C_const, [66](#)
- Lus
 - declare.h, [952](#)
 - lus.c, [1678](#)
- lus.c, [1678](#), [1681](#)
 - AllLoops, [1679](#)
 - AllOnePI, [1680](#)
 - AllPaths, [1680](#)
 - FindLus, [1678](#)
 - GenLoops, [1679](#)
 - GenPaths, [1680](#)
 - LoopOutput, [1679](#)
 - Lus, [1678](#)
 - OutputOnePI, [1681](#)
 - PathOutput, [1680](#)
 - RemoveBridges, [1680](#)
 - SortTheList, [1679](#)
 - StartLoops, [1679](#)
 - TakeOneLine, [1681](#)
- IWorkPointer
 - T_const, [438](#)
- IWorkSize
 - R_const, [378](#)
- IWorkSpace
 - T_const, [434](#)
- M
 - AllGlobals, [10](#)
- m
 - MODNUM, [237](#)
- M_alloc
 - declare.h, [808](#)
- M_check1
 - declare.h, [968](#)
 - tools.c, [3082](#)
- M_const, [167](#)
 - atstartup, [189](#)
 - BracketFactors, [191](#)
 - ClearStore, [191](#)
 - CompressSize, [173](#)
 - countfunnum, [188](#)
 - dbufnum, [178](#)
 - denomnum, [187](#)
 - dollarzero, [189](#)
 - DumInd, [184](#)
 - exitflag, [189](#)
 - expnum, [187](#)
 - facnum, [187](#)
 - fbuffersize, [182](#)
 - FileOnlyFlag, [176](#)
 - FortranCont, [187](#)
 - FromStdin, [191](#)
 - gcmmod, [171](#)
 - gcNumDollars, [182](#)
 - gCnumpows, [185](#)
 - gCodesFlag, [179](#)
 - gDefDim, [177](#)
 - gDefDim4, [177](#)
 - gextrasym, [173](#)
 - gextrasymbols, [190](#)
 - gFinalStats, [180](#)
 - gFortran90Kind, [172](#)
 - gfunpowers, [178](#)
 - ggextrasym, [173](#)
 - ggextrasymbols, [190](#)
 - ggFinalStats, [180](#)
 - ggIndentSpace, [190](#)
 - ggNoSpacesInNumbers, [182](#)
 - ggnumextrasym, [183](#)
 - ggOldFactArgFlag, [183](#)
 - ggOldGCDflag, [183](#)
 - ggOldParallelStats, [181](#)
 - ggProcessStats, [181](#)
 - ggShortStatsMax, [190](#)
 - ggStatsFlag, [186](#)
 - ggThreadBalancing, [180](#)
 - ggThreadBucketSize, [176](#)
 - ggThreadsFlag, [180](#)
 - ggThreadSortFileSynch, [181](#)
 - ggThreadStats, [180](#)
 - ggWTimeStatsFlag, [184](#)
 - gIndentSpace, [190](#)
 - ginsidefirst, [177](#)
 - gIsFortran90, [182](#)
 - gLineLength, [186](#)
 - gmodmode, [185](#)
 - gNamesFlag, [179](#)

- gncmod, 184
- gNoSpacesInNumbers, 182
- gnpowmod, 184
- gnumextrasym, 183
- gOldFactArgFlag, 182
- gOldGCDflag, 183
- gOldParallelStats, 181
- gOutNumberType, 185
- gOutputMode, 185
- gOutputSpaces, 185
- gparallelflag, 179
- gPolyFun, 188
- gPolyFunExp, 189
- gPolyFunInv, 189
- gPolyFunPow, 189
- gPolyFunType, 189
- gPolyFunVar, 189
- gpowmod, 171
- gProcessBucketSize, 175
- gProcessStats, 181
- gproperorderflag, 179
- gShortStatsMax, 190
- gSizeCommutelnSet, 179
- gSortType, 179
- gStatsFlag, 178
- gThreadBalancing, 180
- gThreadBucketSize, 175
- gThreadsFlag, 180
- gThreadSortFileSynch, 181
- gThreadStats, 180
- gTokensWriteFlag, 178
- gUniTrace, 185
- gUnitTrace, 185
- gWTimeStatsFlag, 183
- havesortdir, 191
- HideSize, 173
- HoldFlag, 187
- hparallelflag, 179
- hProcessBucketSize, 175
- IgnoreDeprecation, 191
- IncDir, 172
- IndDum, 184
- InputFileName, 172
- Interact, 176
- invfacnum, 188
- jumpratio, 184
- LogFileName, 172
- LogType, 186
- matchfunnum, 188
- MaxBracketBufferSize, 174
- maxFlevels, 181
- MaxParLevel, 176
- MaxStreamSize, 174
- MaxTal, 184
- MaxTer, 173
- MaxWildcards, 185
- mTraceDum, 186
- MultiRun, 182
- NumFixedFunctions, 177
- NumFixedSets, 177
- NumSpectatorFiles, 183
- NumStoreCaches, 190
- OffsetIndex, 186
- OffsetVector, 186
- OldChildTime, 175
- OldMilliTime, 175
- oldnumextrasymbols, 173
- OldOrderFlag, 188
- OldSecTime, 175
- onerhs, 191
- Ordering, 187
- OutBuffer, 172
- OutBufSize, 176
- Path, 172
- PrintTotalSize, 182
- qError, 186
- rbufnum, 178
- RepMax, 186
- resetTimeOnClear, 181
- S0, 171
- sbufnum, 178
- ScratSize, 173
- SetupDir, 172
- SetupFile, 172
- shmWinSize, 175
- silent, 187
- SIOsize, 174
- SizeForSpectatorFiles, 183
- SizeStoreCache, 174
- SkipClears, 178
- SLargeSize, 174
- SMaxFpatches, 177
- SMaxPatches, 177
- SpectatorFiles, 173
- SpectatorSize, 176
- SSmallEsize, 174
- SSmallSize, 174
- StdOut, 177
- STermsInSmall, 174
- sumnum, 188
- sumpnum, 188
- SumTime, 176
- TempDir, 171
- TempSortDir, 171
- termfunnum, 188
- TimeLimit, 176
- totalnumberofthreads, 179
- tracebackflag, 187
- vectorzero, 191
- Willnd, 184
- WorkSize, 175
- zbufnum, 178
- zeropos, 171
- zerorhs, 191
- M_free
 - declare.h, 912

- tools.c, [3082](#)
- M_print
 - declare.h, [968](#)
 - tools.c, [3083](#)
- main
 - startup.c, [2794](#)
- MAINSORT
 - ftypes.h, [1428](#)
- MAJORVERSION
 - form3.h, [1326](#)
- MAKEARGS
 - ftypes.h, [1444](#)
- MakeDate
 - declare.h, [887](#)
 - tools.c, [3081](#)
- MakeDirty
 - declare.h, [841](#)
 - function.c, [1471](#)
- MakeDollarInteger
 - declare.h, [955](#)
 - dollar.c, [1090](#)
- MakeDollarMod
 - declare.h, [956](#)
 - dollar.c, [1091](#)
- MakeDubious
 - declare.h, [878](#)
 - names.c, [1827](#)
- MakeGlobal
 - declare.h, [907](#)
 - execute.c, [1152](#)
- MakeInteger
 - argument.c, [481](#)
 - declare.h, [918](#)
- MakeInverses
 - declare.h, [855](#)
 - reken.c, [2552](#)
- MakeLongRational
 - declare.h, [960](#)
 - reken.c, [2555](#)
- MakeMod
 - argument.c, [481](#)
 - declare.h, [919](#)
- MakeModTable
 - declare.h, [842](#)
 - reken.c, [2556](#)
- MakeNameTree
 - declare.h, [880](#)
 - names.c, [1821](#)
- MAKERATIONAL
 - ftypes.h, [1399](#)
- MakeRational
 - declare.h, [960](#)
 - reken.c, [2555](#)
- MakeSetupAllocs
 - declare.h, [965](#)
 - setfile.c, [2686](#)
- Malloc1
 - declare.h, [874](#)
- tools.c, [3082](#)
- mallocprotect.h, [1700](#)
- MAPLEMODE
 - ftypes.h, [1383](#)
- MarkDirty
 - declare.h, [841](#)
 - function.c, [1471](#)
- MarkPolyRatFunDirty
 - declare.h, [806](#)
- MaskPointer
 - N_const, [250](#)
- mass
 - PaRtlcLe, [327](#)
- MASTER
 - parallel.h, [2131](#)
- MATCH
 - ftypes.h, [1434](#)
- match
 - SProcess, [410](#)
- MatchArgument
 - declare.h, [843](#)
 - symmetr.c, [2925](#)
- MatchCy
 - declare.h, [842](#)
 - symmetr.c, [2925](#)
- MatchE
 - declare.h, [842](#)
 - symmetr.c, [2924](#)
- MATCHFUNCTION
 - ftypes.h, [1393](#)
- MatchFunction
 - declare.h, [843](#)
 - function.c, [1473](#)
- matchfunnum
 - M_const, [188](#)
- MatchIsPossible
 - declare.h, [953](#)
 - smart.c, [2703](#)
- MATHEMATICAMODE
 - ftypes.h, [1383](#)
- MaX
 - declare.h, [788](#)
- MAX_OPEN_FILES
 - form3.h, [1332](#)
- MaxBracket
 - R_const, [379](#)
- MAXBRACKETBUFFERSIZE
 - fsizes.h, [1345](#)
- MaxBracketBufferSize
 - M_const, [174](#)
- MAXBUILTINFUNCTION
 - ftypes.h, [1402](#)
- maxcase
 - SWITCH, [427](#)
- MAXCOUPLINGS
 - fsizes.h, [1348](#)
- maxcpl
 - Model, [236](#)

- maxdeg
 - EGraph, [112](#)
 - ENode, [117](#)
 - MGraph, [213](#)
 - MNodeClass, [226](#)
- MaxDoLoops
 - variable.h, [3189](#)
- MaxDum
 - R_const, [381](#)
- MAXENAME
 - fsizes.h, [1340](#)
- MaxExprSize
 - S_const, [386](#)
- MAXFLAGS
 - fsizes.h, [1342](#)
- MAXFLEVELS
 - fsizes.h, [1344](#)
- maxFlevels
 - M_const, [181](#)
- MAXFPATCHES
 - fsizes.h, [1343](#)
- MaxFpatches
 - sOrT, [404](#)
- MAXFUNCTION
 - ftypes.h, [1392](#)
- MaxFunSorts
 - N_const, [257](#)
- maxi
 - MiNmAx, [218](#)
- MAXIF
 - fsizes.h, [1341](#)
- MaxIf
 - C_const, [56](#)
- MAXLABELS
 - fsizes.h, [1342](#)
- MaxLabels
 - C_const, [55](#)
- MAXLEGS
 - fsizes.h, [1348](#)
- MAXLEVELS
 - fsizes.h, [1342](#)
- MAXLHS
 - fsizes.h, [1343](#)
- maxlhs
 - CbUf, [73](#)
- MAXLINELENGTH
 - fsizes.h, [1347](#)
- MAXLONG
 - form3.h, [1329](#)
- maxloop
 - Model, [236](#)
- MAXLOOPS
 - fsizes.h, [1342](#)
- MAXMATCH
 - fsizes.h, [1341](#)
- MAXMULTIBRACKETLEVELS
 - fsizes.h, [1347](#)
- MAXNEST
 - fsizes.h, [1341](#)
- maxnlegs
 - Model, [236](#)
 - Process, [371](#)
- maxnum
 - LIST, [164](#)
- maxnumargs
 - FuNcTiOn, [146](#)
- MAXNUMBEROFNONCOMTERMS
 - normal.c, [1867](#)
- MAXNUMBERSIZE
 - fsizes.h, [1340](#)
- MaxNumPre
 - variable.h, [3189](#)
- MAXNUMSIZE
 - fsizes.h, [1345](#)
- MaxNumStreams
 - C_const, [56](#)
- MAXPARLEVEL
 - fsizes.h, [1340](#)
- MaxParLevel
 - M_const, [176](#)
- MAXPARTICLES
 - fsizes.h, [1348](#)
- MAXPATCHES
 - fsizes.h, [1343](#)
- MaxPatches
 - sOrT, [403](#)
- MAXPOINTS
 - diawrap.cc, [1051](#)
- MAXPOSITIVE
 - form3.h, [1329](#)
- MAXPOSITIVE2
 - form3.h, [1329](#)
- MAXPOSITIVE4
 - form3.h, [1329](#)
- maxpower
 - STOREHEADER, [419](#)
 - SyMbOl, [429](#)
- MAXPOWEROF
 - ftypes.h, [1395](#)
- MaxPreAssignLevel
 - P_const, [321](#)
- MaxPreIfLevel
 - P_const, [320](#)
- MAXPRENAMESIZE
 - fsizes.h, [1340](#)
- MaxPreTypes
 - P_const, [320](#)
- MaxProcedures
 - variable.h, [3189](#)
- MAXRANGEINDICATOR
 - ftypes.h, [1444](#)
- MaxRenumScratch
 - N_const, [258](#)
- MAXREPEAT
 - fsizes.h, [1340](#)
- maxrhs

- CbUf, [74](#)
- MAXSAVEFUNCTION
 - fsizes.h, [1340](#)
- MaxStreamSize
 - M_const, [174](#)
- MaxSwitch
 - C_const, [68](#)
- MaxTal
 - M_const, [184](#)
- MaxTer
 - M_const, [173](#)
- maxtermlevel
 - C_const, [59](#)
- maxtermsize
 - sOrT, [404](#)
- MaxTreeSize
 - CbUf, [74](#)
 - TaBIEs, [453](#)
- maxvar
 - OPTIMIZERESULT, [299](#)
- MAXWILDC
 - fsizes.h, [1343](#)
- MaxWildcards
 - M_const, [185](#)
- mBer
 - T_const, [443](#)
- MCBlock, [192](#)
 - ~MCBlock, [192](#)
 - DUMMYPADDING, [193](#)
 - edges, [193](#)
 - init, [193](#)
 - loop, [193](#)
 - MCBlock, [192](#)
 - nartps, [193](#)
 - nmedges, [193](#)
- MCBridge, [194](#)
 - ~MCBridge, [194](#)
 - MCBridge, [194](#)
 - next, [194](#)
 - nodes, [194](#)
- MCEdge, [195](#)
 - ~MCEdge, [195](#)
 - DUMMYPADDING, [196](#)
 - MCEdge, [195](#)
 - momdir, [195](#)
 - momset, [195](#)
 - nodes, [195](#)
- MCLEANFLAG
 - minos.h, [1744](#)
- MConn, [196](#)
 - ~MConn, [197](#)
 - addArtic, [199](#)
 - addBlock, [199](#)
 - addBlockSelf, [199](#)
 - addBridge, [198](#)
 - addCEdge, [198](#)
 - addOPIC, [198](#)
 - articuls, [200](#)
 - blksp, [200](#)
 - blkstk, [201](#)
 - blkstkptr, [203](#)
 - blocks, [200](#)
 - bridges, [200](#)
 - cedges, [200](#)
 - DUMMYPADDING, [203](#)
 - init, [198](#)
 - initCEdges, [198](#)
 - MConn, [197](#)
 - na1blocks, [202](#)
 - narticuls, [202](#)
 - nbacked, [201](#)
 - nblksp, [203](#)
 - nblocks, [202](#)
 - nbridges, [201](#)
 - nctopic, [201](#)
 - ne0bridges, [202](#)
 - ne1bridges, [202](#)
 - neblocks, [202](#)
 - nlpopic, [201](#)
 - nmultiedges, [202](#)
 - nopic, [201](#)
 - nosp, [203](#)
 - nselfloops, [202](#)
 - opics, [200](#)
 - opisp, [200](#)
 - opistk, [200](#)
 - opistkptr, [203](#)
 - prEdges, [199](#)
 - print, [199](#)
 - pushEdge, [198](#)
 - pushNode, [198](#)
 - sedges, [201](#)
 - snodes, [201](#)
- mconn
 - MGraph, [217](#)
- MCOpI, [203](#)
 - ~MCOpI, [204](#)
 - ctloop, [205](#)
 - DUMMYPADDING, [205](#)
 - init, [204](#)
 - loop, [205](#)
 - MCOpI, [204](#)
 - mom0lg, [205](#)
 - nedges, [205](#)
 - next, [205](#)
 - nlegs, [205](#)
 - nnodes, [204](#)
 - nodes, [204](#)
- mcts_best_schemes
 - optimize.cc, [2003](#)
- mcts_expr_score
 - optimize.cc, [2002](#)
- mcts_factorized
 - optimize.cc, [2002](#)
- mcts_root
 - optimize.cc, [2002](#)

- mcts_separated
 - optimize.cc, 2002
- mcts_vars
 - optimize.cc, 2002
- mctsdecaymode
 - OPTIMIZE, 297
- mctsnumexpand
 - OPTIMIZE, 296
- mctsnumkeep
 - OPTIMIZE, 296
- mctsnumrepeat
 - OPTIMIZE, 296
- mctstimelimit
 - OPTIMIZE, 296
- mdefined
 - TaBIEs, 451
- mdel
 - TrAcEs, 463
- MDIRTYFLAG
 - minos.h, 1744
- mdl
 - Interaction, 160
 - Particle, 330
- mdum
 - TrAcEs, 464
- me
 - ParallelVars, 324
- mean
 - OptDef, 291
 - OptQGDef, 308
- MemDebugFlag
 - C_const, 61
- MEMORYMACROS
 - declare.h, 804
- mepf
 - TrAcEs, 463
- merge_operators
 - optimize.cc, 1992
- MergeDotproductLists
 - declare.h, 1000
 - ratio.c, 2500
- MergePatches
 - declare.h, 843
 - sort.c, 2721
- MergeSymbolLists
 - declare.h, 1000
 - ratio.c, 2500
- MergeWithFloat
 - float.c, 1291
- MesCall
 - declare.h, 909
 - message.c, 1706
- MesCerr
 - declare.h, 844
 - message.c, 1706
- MesComp
 - declare.h, 844
 - message.c, 1706
- MesNesting
 - declare.h, 805
- MesPrint
 - declare.h, 909
 - message.c, 1706
- message
 - LIST, 164
- message.c, 1704, 1708
 - Error0, 1705
 - Error1, 1705
 - Error2, 1705
 - HighWarning, 1706
 - MesCall, 1706
 - MesCerr, 1706
 - MesComp, 1706
 - MesPrint, 1706
 - MesWork, 1705
 - PrintSeq, 1707
 - PrintSubTerm, 1707
 - PrintTerm, 1707
 - PrintTermC, 1707
 - PrintWords, 1707
 - Warning, 1706
- MessPreNesting
 - declare.h, 913
 - pre.c, 2324
- MesWork
 - declare.h, 909
 - message.c, 1705
- method
 - OPTIMIZE, 296
- mfac
 - T_const, 438
- MGraph, 206
 - ~MGraph, 208
 - adjMat, 213
 - bicol, 217
 - biconnME, 210
 - bicount, 217
 - bidef, 217
 - bilow, 217
 - bipart, 216
 - bisearchME, 210
 - block, 216
 - c1PI, 214
 - c1PINoTadBlock, 214
 - cDiag, 214
 - clist, 212
 - cNoTadBlock, 214
 - cNoTadpole, 214
 - compMat, 210
 - connectClass, 211
 - connectLeg, 211
 - connectNode, 211
 - curcl, 213
 - discardDisc, 215
 - discardIso, 215
 - discardRefine, 215

- DUMMYPADDING1, [216](#)
- DUMMYPADDING2, [217](#)
- egraph, [213](#)
- esym, [214](#)
- extself, [216](#)
- generate, [208](#)
- group, [211](#)
- init, [208](#)
- isConnected, [209](#)
- isExternal, [208](#)
- isIsomorphic, [209](#)
- isOptM, [210](#)
- maxdeg, [213](#)
- mconn, [217](#)
- MGraph, [208](#)
- mId, [212](#)
- mindeg, [213](#)
- modmat, [217](#)
- multiedge, [216](#)
- nCallRefine, [215](#)
- nClasses, [213](#)
- nEdges, [212](#)
- newGraph, [211](#)
- nExtern, [213](#)
- ngconn, [215](#)
- ngen, [215](#)
- nLoops, [212](#)
- nNodes, [212](#)
- nodes, [212](#)
- nsym, [213](#)
- opi, [215](#)
- opiloop, [215](#)
- opt, [211](#)
- orbits, [212](#)
- permMat, [209](#)
- pId, [212](#)
- print, [209](#)
- printAdjMat, [209](#)
- printPy, [209](#)
- refineClass, [210](#)
- selfloop, [216](#)
- tadblock, [216](#)
- tadpole, [216](#)
- visit, [209](#)
- wscon, [214](#)
- wsopi, [214](#)
- mgraph
 - Assign, [23](#)
 - EGraph, [109](#)
 - SProcess, [412](#)
- mgrcount
 - Process, [370](#)
 - SProcess, [414](#)
- mId
 - EGraph, [110](#)
 - MGraph, [212](#)
- MIN
 - declare.h, [788](#)
- MINALLOC
 - fsizes.h, [1347](#)
- mincase
 - SWITCH, [427](#)
- mindeg
 - MGraph, [213](#)
- MINFUNCTION
 - ftypes.h, [1392](#)
- mini
 - MiNmAx, [218](#)
- MiNmAx, [218](#)
 - maxi, [218](#)
 - mini, [218](#)
 - size, [218](#)
- minnumargs
 - FuNcTiOn, [146](#)
- MINORVERSION
 - form3.h, [1326](#)
- minos.c, [1719](#), [1726](#)
 - AddObject, [1724](#)
 - AddTableName, [1723](#)
 - AddToIndex, [1724](#)
 - CFD, [1720](#)
 - ComposeTableNames, [1723](#)
 - convertblock, [1721](#)
 - convertiniinfo, [1721](#)
 - convertnamesblock, [1721](#)
 - CTD, [1720](#)
 - DeleteObject, [1725](#)
 - ExistsObject, [1725](#)
 - FindTableNumber, [1724](#)
 - FreeTableBase, [1723](#)
 - GetDbase, [1722](#)
 - GetTableName, [1723](#)
 - LocateBase, [1721](#)
 - minosread, [1720](#)
 - minoswrite, [1720](#)
 - NewDbase, [1723](#)
 - OpenDbase, [1723](#)
 - PutTableNames, [1724](#)
 - ReadijObject, [1725](#)
 - ReadIndex, [1721](#)
 - ReadIniInfo, [1722](#)
 - ReadObject, [1724](#)
 - str_dup, [1721](#)
 - withoutflush, [1725](#)
 - WriteIndex, [1722](#)
 - WriteIndexBlock, [1722](#)
 - WriteIniInfo, [1722](#)
 - WriteNamesBlock, [1722](#)
 - WriteObject, [1724](#)
- minos.h, [1743](#), [1746](#)
 - FROMDISK, [1744](#)
 - INANDOUT, [1745](#)
 - INPUTONLY, [1745](#)
 - MCLEANFLAG, [1744](#)
 - MDIRTYFLAG, [1744](#)
 - minosread, [1745](#)

- minoswrite, 1745
 - NOCOMPRESS, 1745
 - OUTPUTONLY, 1745
 - TODISK, 1744
 - withoutflush, 1746
- minosread
 - minos.c, 1720
 - minos.h, 1745
- minoswrite
 - minos.c, 1720
 - minos.h, 1745
- minpower
 - SyMbOI, 429
- MINPOWEROF
 - ftypes.h, 1395
- MInput, 219
 - cnamlist, 219
 - defpart, 219
 - name, 219
 - ncouple, 219
- minsidefirst
 - C_const, 55
- MINSPEC
 - ftypes.h, 1428
- minvar
 - OPTIMIZERESULT, 299
- MinVecArg
 - T_const, 442
- MINVECTOR
 - ftypes.h, 1388
- MIXED
 - ftypes.h, 1373
- MIXED2
 - ftypes.h, 1373
- mkCIMat
 - MNodeClass, 224
- mkFlist
 - MNodeClass, 224
- mkNdCI
 - MNodeClass, 224
- mkSlaveInfile
 - ParallelVars, 324
- mkSPProcess
 - Process, 369
- MLOCK
 - declare.h, 808
- MLONG
 - form3.h, 1333
- mm
 - TaBIEs, 449
- MNode, 220
 - class, 221
 - cmaxdeg, 221
 - cmindeg, 221
 - deg, 221
 - extloop, 221
 - freelg, 222
 - id, 221
 - MNode, 220
 - visited, 222
- MNodeClass, 222
 - ~MNodeClass, 223
 - clCmp, 223
 - clist, 225
 - clmat, 225
 - clord, 225
 - cmaxdeg, 225
 - cmindeg, 225
 - cmpMNCArray, 224
 - copy, 223
 - flg0, 226
 - flg1, 226
 - flg2, 226
 - flist, 225
 - incMat, 224
 - init, 223
 - maxdeg, 226
 - mkCIMat, 224
 - mkFlist, 224
 - mkNdCI, 224
 - MNodeClass, 223
 - nClasses, 226
 - ndcl, 225
 - nNodes, 226
 - printMat, 223
 - reorder, 224
- mnumlhs
 - CbUf, 74
- mnumrhs
 - CbUf, 74
- mod
 - poly, 356
- MOD2FUNCTION
 - ftypes.h, 1392
- mode
 - dbase, 78
 - TaBIEs, 454
- MoDeL, 227
 - couplings, 227
 - dummy, 229
 - error, 229
 - grccmodel, 228
 - invertices, 228
 - legcouple, 228
 - name, 228
 - ncouplings, 229
 - nparticles, 228
 - nvertices, 228
 - sizecouplings, 228
 - sizevertices, 228
 - vertices, 227
- Model, 229
 - ~Model, 231
 - addInteraction, 231
 - addInteractionEnd, 232
 - addParticle, 231

- addParticleEnd, [231](#)
- allPart, [235](#)
- allParticles, [233](#)
- antiParticle, [233](#)
- cnlist, [234](#)
- cplgcp, [236](#)
- cplglg, [236](#)
- cplgnvl, [236](#)
- cplgvl, [236](#)
- defpart, [237](#)
- findInteractionCode, [232](#)
- findInteractionName, [232](#)
- findMClass, [233](#)
- findParticleCode, [232](#)
- findParticleName, [232](#)
- interacts, [235](#)
- intPart, [235](#)
- maxcpl, [236](#)
- maxloop, [236](#)
- maxnlegs, [236](#)
- Model, [231](#)
- nallPart, [235](#)
- name, [234](#)
- ncouple, [234](#)
- ncplgcp, [236](#)
- nInteracts, [235](#)
- nintPart, [235](#)
- normalParticle, [233](#)
- nParticles, [234](#)
- particleCode, [232](#)
- particleName, [232](#)
- particles, [234](#)
- pdef, [235](#)
- printIInput, [234](#)
- printMInput, [233](#)
- printPInput, [234](#)
- prModel, [231](#)
- prParticleArray, [233](#)
- vdef, [235](#)
- model
 - Assign, [24](#)
 - EGraph, [109](#)
 - Options, [305](#)
 - Output, [313](#)
 - Process, [370](#)
 - SProcess, [411](#)
- model.c, [1748](#), [1749](#)
 - CoEndModel, [1749](#)
 - CoModel, [1748](#)
 - CoParticle, [1749](#)
 - CoVertex, [1749](#)
 - CreateVertex, [1748](#)
 - ReadParticle, [1748](#)
- ModelLevel
 - C_const, [63](#)
- modeloptions
 - R_const, [382](#)
- models
 - C_const, [47](#)
- modelspace
 - C_const, [63](#)
- MODFUNCTION
 - ftypes.h, [1392](#)
- modinverses
 - C_const, [51](#)
- MODLOCAL
 - ftypes.h, [1441](#)
- modmat
 - MGraph, [217](#)
- MODMAX
 - ftypes.h, [1441](#)
- MODMIN
 - ftypes.h, [1441](#)
- modmode
 - C_const, [66](#)
- modn
 - poly, [361](#)
- MODNONE
 - ftypes.h, [1441](#)
- MODNUM, [237](#)
 - a, [237](#)
 - m, [237](#)
 - na, [237](#)
 - nm, [238](#)
- modnum
 - POLYMOD, [363](#)
- MoDoPtDoLIaRS, [238](#)
 - number, [238](#)
 - type, [238](#)
- ModOptdollars
 - variable.h, [3194](#)
- ModOptDolList
 - C_const, [45](#)
- modp
 - poly, [361](#)
- modpowers
 - C_const, [48](#)
- MODSUM
 - ftypes.h, [1441](#)
- module.c, [1755](#), [1759](#)
 - CoModOption, [1756](#)
 - CoModuleOption, [1756](#)
 - DoExecStatement, [1758](#)
 - DoinParallel, [1758](#)
 - DoModDollar, [1758](#)
 - DoModLocal, [1758](#)
 - DoModMax, [1757](#)
 - DoModMin, [1757](#)
 - DoModSum, [1757](#)
 - DoNoParallel, [1757](#)
 - DonotinParallel, [1758](#)
 - DoParallel, [1757](#)
 - DoPipeStatement, [1758](#)
 - DoPolyfun, [1757](#)
 - DoPolyratfun, [1757](#)
 - DoProcessBucket, [1758](#)

- ExecModule, [1756](#)
- ExecStore, [1756](#)
- FullCleanup, [1756](#)
- ModuleInstruction, [1756](#)
- SetSpecialMode, [1756](#)
- MODULEINSTACK
 - ftypes.h, [1368](#)
- ModuleInstruction
 - declare.h, [893](#)
 - module.c, [1756](#)
- Modulus
 - declare.h, [844](#)
 - reken.c, [2556](#)
- MOEBIUS
 - ftypes.h, [1400](#)
- Moebius
 - declare.h, [981](#)
 - reken.c, [2557](#)
- moebiustable
 - R_const, [377](#)
- moebiustablesizesize
 - R_const, [382](#)
- mom0lg
 - MCOpi, [205](#)
- momdir
 - EEdge, [99](#)
 - MCEdge, [195](#)
- momset
 - EEdge, [99](#)
 - MCEdge, [195](#)
- monomial_compare
 - poly, [354](#)
- monomial_larger, [239](#)
 - monomial_larger, [239](#)
 - operator(), [239](#)
- MOrbits, [239](#)
 - ~MOrbits, [240](#)
 - flist, [241](#)
 - fromPerm, [240](#)
 - initPerm, [240](#)
 - MOrbits, [240](#)
 - nd2or, [241](#)
 - nNodes, [241](#)
 - nOrbits, [241](#)
 - or2nd, [241](#)
 - print, [240](#)
 - toOrbits, [240](#)
- MoveDummies
 - declare.h, [845](#)
 - reshuf.c, [2605](#)
- mparallelflag
 - C_const, [60](#)
- mpi.c, [1768](#), [1785](#)
 - MPI_ERRCODE_CHECK, [1770](#)
 - PF_Bcast, [1773](#)
 - PF_Broadcast, [1779](#)
 - PF_IRecvRbuf, [1773](#)
 - PF_ISendSbuf, [1772](#)
 - PF_LibInit, [1770](#)
 - PF_LibTerminate, [1771](#)
 - PF_LongMultiBroadcast, [1784](#)
 - PF_LongMultiPackImpl, [1782](#)
 - PF_LongMultiUnpackImpl, [1783](#)
 - PF_LongSinglePack, [1780](#)
 - PF_LongSingleReceive, [1781](#)
 - PF_LongSingleSend, [1781](#)
 - PF_LongSingleUnpack, [1780](#)
 - PF_maxDollarChunkSize, [1784](#)
 - PF_Pack, [1776](#)
 - PF_PACKSIZE, [1770](#)
 - PF_PackString, [1777](#)
 - PF_PrepareLongMultiPack, [1782](#)
 - PF_PrepareLongSinglePack, [1779](#)
 - PF_PreparePack, [1776](#)
 - PF_PrintPackBuf, [1775](#)
 - PF_Probe, [1771](#)
 - PF_RawProbe, [1775](#)
 - PF_RawRecv, [1774](#)
 - PF_RawSend, [1774](#)
 - PF_RealTime, [1770](#)
 - PF_Receive, [1778](#)
 - PF_RecvWbuf, [1772](#)
 - PF_Send, [1778](#)
 - PF_Unpack, [1776](#)
 - PF_UnpackString, [1777](#)
 - PF_WaitRbuf, [1773](#)
- MPI_Barrier
 - mpidbg.h, [1807](#)
- MPI_Bcast
 - mpidbg.h, [1807](#)
- MPI_Bsend
 - mpidbg.h, [1804](#)
- MPI_Cancel
 - mpidbg.h, [1807](#)
- MPI_ERRCODE_CHECK
 - mpi.c, [1770](#)
- MPI_lbsend
 - mpidbg.h, [1805](#)
- MPI_lprobe
 - mpidbg.h, [1806](#)
- MPI_irecv
 - mpidbg.h, [1805](#)
- MPI_irsend
 - mpidbg.h, [1805](#)
- MPI_lsend
 - mpidbg.h, [1805](#)
- MPI_issend
 - mpidbg.h, [1805](#)
- MPI_Probe
 - mpidbg.h, [1807](#)
- MPI_Recv
 - mpidbg.h, [1804](#)
- MPI_Rsend
 - mpidbg.h, [1804](#)
- MPI_Send
 - mpidbg.h, [1804](#)

MPI_Ssend
 mpidbg.h, 1804
 MPI_Test
 mpidbg.h, 1805
 MPI_Test_cancelled
 mpidbg.h, 1807
 MPI_Testall
 mpidbg.h, 1806
 MPI_Testany
 mpidbg.h, 1806
 MPI_Testsome
 mpidbg.h, 1806
 MPI_Wait
 mpidbg.h, 1805
 MPI_Waitall
 mpidbg.h, 1806
 MPI_Waitany
 mpidbg.h, 1806
 MPI_Waitsome
 mpidbg.h, 1806
 mpidbg.h, 1802, 1808
 MPI_Barrier, 1807
 MPI_Bcast, 1807
 MPI_Bsend, 1804
 MPI_Cancel, 1807
 MPI_Ibsend, 1805
 MPI_lprobe, 1806
 MPI_lrecv, 1805
 MPI_lrsend, 1805
 MPI_lsend, 1805
 MPI_lssend, 1805
 MPI_Probe, 1807
 MPI_Recv, 1804
 MPI_Rsend, 1804
 MPI_Send, 1804
 MPI_Ssend, 1804
 MPI_Test, 1805
 MPI_Test_cancelled, 1807
 MPI_Testall, 1806
 MPI_Testany, 1806
 MPI_Testsome, 1806
 MPI_Wait, 1805
 MPI_Waitall, 1806
 MPI_Waitany, 1806
 MPI_Waitsome, 1806
 MPIDBG_EXTARG, 1804
 MPIDBG_Out, 1804
 MPIDBG_RANK, 1804
 MPIDBG_EXTARG
 mpidbg.h, 1804
 MPIDBG_Out
 mpidbg.h, 1804
 MPIDBG_RANK
 mpidbg.h, 1804
 MPLFUNCTION
 ftypes.h, 1402
 mpoly
 flint::mpoly, 242
 mpoly_ctx
 flint::mpoly_ctx, 243
 mpoly_factor
 flint::mpoly_factor, 244
 mpower
 O_const, 287
 mProcessBucketSize
 C_const, 53
 mTraceDum
 M_const, 186
 mul
 COST, 76
 poly, 355
 mul_heap
 poly, 357
 mul_mpoly
 flintinterface.h, 1268
 mul_one_term
 poly, 357
 mul_poly
 flintinterface.h, 1269
 mul_univar
 poly, 357
 MulFloats
 float.c, 1284
 MULfunc
 declare.h, 1000
 ratio.c, 2502
 MULFUNCTION
 ftypes.h, 1400
 MulLong
 declare.h, 845
 reken.c, 2550
 Mully
 declare.h, 845
 reken.c, 2548
 mulpat
 T_const, 442
 MULPOS
 declare.h, 800
 MulRat
 declare.h, 845
 reken.c, 2549
 MulRatToFloat
 float.c, 1285
 MultDo
 declare.h, 845
 reshuf.c, 2606
 MultiBracketBuf
 C_const, 51
 MultiBracketLevels
 C_const, 61
 multiedge
 MGraph, 216
 MULTIINDENTSPACE
 fsizes.h, 1347
 MULTIPLEOF
 ftypes.h, 1434

- MULTIPLYARG
 - ftypes.h, 1446
- MultiplyToLine
 - declare.h, 1011
 - sch.c, 2647
- MultiplyWithTerm
 - declare.h, 999
 - ratio.c, 2499
- MultiRun
 - M_const, 182
- MultiThreaded
 - S_const, 387
- multp
 - EGraph, 111
- MUNLOCK
 - declare.h, 808
- MUSTCLEANPRF
 - ftypes.h, 1418
- mustfree
 - PROCEDURE, 367
- MustTestTable
 - C_const, 65
- my_random_shuffle
 - optimize.cc, 1983
- mytime.cc, 1816
- mytime.h, 1816
- MZV
 - ftypes.h, 1401
- MZVHALF
 - ftypes.h, 1402
- N
 - AllGlobals, 10
- n
 - DiStRiBuTe, 85
 - PeRmUtE, 332
 - PeRmUtEp, 333
- n1
 - DiStRiBuTe, 85
- n2
 - DiStRiBuTe, 85
- N_const, 245
 - argaddress, 249
 - arglist, 253
 - arglistsize, 257
 - cmod, 254
 - compressSize, 257
 - compressSpace, 253
 - cTerm, 248
 - currentTerm, 253
 - deferskipped, 254
 - DisOrderFlag, 258
 - DumFound, 249
 - DumFunPlace, 250
 - dummyrenumlist, 251
 - DumPlace, 249
 - EndNest, 248
 - ErrorInDollar, 255
 - ExpectedSign, 260
 - filenum, 257
 - findPattern, 251
 - findTerm, 250
 - ForFindOnly, 250
 - Frozen, 248
 - FullProto, 248
 - funargs, 251
 - funinds, 251
 - FunInfo, 252
 - funisize, 256
 - funlocs, 251
 - FunScratch, 252
 - FunSorts, 253
 - idfunctionflag, 261
 - IndDum, 261
 - InScratch, 254
 - InScratch1, 254
 - intracestack, 256
 - listinprint, 253
 - MaskPointer, 250
 - MaxFunSorts, 257
 - MaxRenumScratch, 258
 - nargs, 256
 - ncmod, 260
 - ninterms, 255
 - nogroundlevel, 258
 - NoScrat2, 251
 - numfargs, 255
 - numflocs, 256
 - NumFound, 259
 - numfuninfo, 257
 - NumFunSorts, 257
 - numlistinprint, 260
 - numoffuns, 256
 - NumTotWildArgs, 255
 - numtracesctack, 256
 - NumWild, 258
 - OldPosIn, 247
 - OldPosOut, 247
 - oldtype, 258
 - oldvalue, 258
 - patstop, 249
 - patternbuffer, 252
 - patternbuffersize, 260
 - PoinScratch, 252
 - poly_num_vars, 261
 - poly_vars, 254
 - poly_vars_type, 261
 - PolyFunTodo, 260
 - PolyNormFlag, 259
 - polysortflag, 257
 - RenumScratch, 252
 - RepFunList, 249
 - RepFunNum, 258
 - ReplaceScrat, 251
 - RepPoint, 248
 - RSSize, 256
 - selecttermundo, 252

- SHcombi, [254](#)
- SHcombisize, [255](#)
- SHvar, [254](#)
- SignCheck, [260](#)
- sizeselecttermundo, [260](#)
- SoScratC, [253](#)
- SplitScratch, [252](#)
- SplitScratch1, [253](#)
- SplitScratchSize, [254](#)
- SplitScratchSize1, [255](#)
- subsubveto, [258](#)
- TeInFun, [259](#)
- terfirstcomm, [249](#)
- termbuffer, [252](#)
- terstart, [249](#)
- terstop, [249](#)
- TeSuOut, [259](#)
- theposition, [248](#)
- tlisbuf, [253](#)
- tlisize, [257](#)
- tohunt, [256](#)
- tracestack, [251](#)
- tryterm, [261](#)
- UsedFunction, [250](#)
- UsedIndex, [250](#)
- UsedOtherFind, [255](#)
- UsedSymbol, [250](#)
- UsedVector, [250](#)
- UseFindOnly, [255](#)
- WildArgs, [259](#)
- WildDirt, [259](#)
- WildEat, [259](#)
- WildReserve, [259](#)
- WildStop, [248](#)
- WildValue, [248](#)
- na
 - MODNUM, [237](#)
- na1blocks
 - MConn, [202](#)
- nadj2ptv
 - EGraph, [113](#)
- nAGraphs
 - Assign, [25](#)
 - Process, [373](#)
 - SProcess, [415](#)
- nallPart
 - Model, [235](#)
- name
 - ChAnNeL, [75](#)
 - dbase, [79](#)
 - DICTIONARY, [82](#)
 - DoLIaRS, [88](#)
 - DoLoOp, [90](#)
 - DuBiOuS, [93](#)
 - ExPrEsSiOn, [122](#)
 - FiLe, [129](#)
 - fixedset, [137](#)
 - FuNcTiOn, [144](#)
 - IInput, [148](#)
 - InDeX, [149](#)
 - InDeXeNtRy, [154](#)
 - Interaction, [160](#)
 - KEYWORD, [162](#)
 - KEYWORDV, [163](#)
 - MInput, [219](#)
 - MoDeL, [228](#)
 - Model, [234](#)
 - NaMeNode, [263](#)
 - NaMeSpAcE, [265](#)
 - OptDef, [291](#)
 - OptQGDef, [308](#)
 - OptQGRef, [308](#)
 - Particle, [330](#)
 - PInput, [338](#)
 - pReVaR, [365](#)
 - PROCEDURE, [367](#)
 - SeTs, [388](#)
 - SpecTatoR, [407](#)
 - StreaM, [422](#)
 - SyMbOl, [429](#)
 - TaBIEbAsE, [446](#)
 - VeCtOr, [471](#)
- nameblock, [262](#)
 - names, [262](#)
 - position, [262](#)
 - previousblock, [262](#)
- namebuffer
 - NaMeTree, [266](#)
- NameConflict
 - declare.h, [883](#)
 - names.c, [1827](#)
- namefill
 - NaMeTree, [267](#)
- NAMENODE
 - structs.h, [2894](#)
- NaMeNode, [262](#)
 - balance, [263](#)
 - left, [263](#)
 - name, [263](#)
 - number, [264](#)
 - parent, [263](#)
 - right, [263](#)
 - type, [264](#)
- namenode
 - NaMeTree, [266](#)
- NAMENOTFOUND
 - ftypes.h, [1438](#)
- nameofexpr
 - OPTIMIZERESULT, [298](#)
- names
 - nameblock, [262](#)
- names.c, [1817](#), [1828](#)
 - AddDollar, [1826](#)
 - AddDubious, [1827](#)
 - AddExpression, [1827](#)
 - AddFunction, [1824](#)

- AddIndex, [1823](#)
- AddName, [1819](#)
- AddSet, [1825](#)
- AddSymbol, [1822](#)
- AddVector, [1823](#)
- AddWildcardName, [1822](#)
- ClearWildcardNames, [1822](#)
- CoAuto, [1826](#)
- CoCFunction, [1824](#)
- CoCommutInSet, [1824](#)
- CoCTable, [1825](#)
- CoCTensor, [1825](#)
- CoDimension, [1823](#)
- CoFunction, [1824](#)
- CoIndex, [1823](#)
- CompactifyTree, [1821](#)
- CoNFunction, [1824](#)
- CoNTable, [1825](#)
- CoNTensor, [1824](#)
- CopyTree, [1821](#)
- CoSet, [1826](#)
- CoSymbol, [1822](#)
- CoTable, [1825](#)
- CoVector, [1823](#)
- DoDimension, [1823](#)
- DoElements, [1826](#)
- DoTable, [1825](#)
- DoTempSet, [1826](#)
- DumpNode, [1821](#)
- DumpTree, [1821](#)
- EmptyTable, [1825](#)
- EntVar, [1820](#)
- FreeNameTree, [1822](#)
- GetAutoName, [1820](#)
- GetDollar, [1820](#)
- GetFunction, [1819](#)
- GetLabel, [1827](#)
- GetLastExprName, [1819](#)
- GetName, [1819](#)
- GetNode, [1819](#)
- GetNumber, [1819](#)
- GetOName, [1820](#)
- GetVar, [1820](#)
- GetWildcardName, [1822](#)
- Globalize, [1828](#)
- LinkTree, [1821](#)
- MakeDubious, [1827](#)
- MakeNameTree, [1821](#)
- NameConflict, [1827](#)
- RemoveDollars, [1828](#)
- ReplaceDollar, [1826](#)
- ResetVariables, [1827](#)
- TestName, [1828](#)
- NamesFlag
 - C_const, [57](#)
- namesize
 - ExPrEsSiOn, [123](#)
 - FuNcTiOn, [145](#)
- InDeX, [150](#)
- NaMeTree, [267](#)
- SeTs, [389](#)
- SyMbOl, [430](#)
- VeCtOr, [472](#)
- NaMeSpAcE, [264](#)
 - name, [265](#)
 - next, [265](#)
 - previous, [265](#)
 - usenames, [265](#)
- NAMETREE
 - structs.h, [2895](#)
- NaMeTree, [266](#)
 - clearnamefill, [268](#)
 - clearnodefill, [268](#)
 - globalnamefill, [268](#)
 - globalnodefill, [268](#)
 - headnode, [268](#)
 - namebuffer, [266](#)
 - namefill, [267](#)
 - namenode, [266](#)
 - namesize, [267](#)
 - nodefill, [267](#)
 - nodesize, [267](#)
 - oldnamefill, [267](#)
 - oldnodefill, [267](#)
- nAOPI
 - Assign, [25](#)
 - Process, [373](#)
 - SProcess, [415](#)
- nargs
 - N_const, [256](#)
 - PaRtl, [326](#)
 - pReVaR, [365](#)
- narticuls
 - MConn, [202](#)
- nartps
 - MCBlock, [193](#)
- nbacked
 - MConn, [201](#)
- nBer
 - T_const, [443](#)
- nblksp
 - MConn, [203](#)
- nblocks
 - dbase, [79](#)
 - MConn, [202](#)
- nbridges
 - MConn, [201](#)
- nCallRefine
 - MGraph, [215](#)
- NCand, [269](#)
 - ~NCand, [269](#)
 - deg, [270](#)
 - ilist, [270](#)
 - NCand, [269](#)
 - nilist, [270](#)
 - prNCand, [269](#)

- st, [270](#)
- ncase
 - SWITCHTABLE, [428](#)
- NCInput, [270](#)
 - cldeg, [271](#)
 - clnum, [271](#)
 - cltyp, [271](#)
 - cmaxd, [271](#)
 - cmind, [271](#)
 - cple, [271](#)
 - ptcl, [271](#)
- ncl
 - ToPoTyPe, [458](#)
- nclass
 - PNodeClass, [342](#)
 - SProcess, [412](#)
- nClasses
 - MGraph, [213](#)
 - MNodeClass, [226](#)
- nclist
 - Interaction, [161](#)
- ncmod
 - C_const, [65](#)
 - N_const, [260](#)
- NCOPY
 - declare.h, [791](#)
- NCOPYB
 - declare.h, [792](#)
- NCOPYI
 - declare.h, [791](#)
- NCOPYI32
 - declare.h, [792](#)
- ncouple
 - MInput, [219](#)
 - Model, [234](#)
 - SProcess, [413](#)
- ncouplings
 - MoDeL, [229](#)
 - VeRtEx, [473](#)
- ncplgcp
 - Model, [236](#)
- nctopic
 - MConn, [201](#)
- nd2cl
 - PNodeClass, [342](#)
 - SProcess, [412](#)
- nd2or
 - MOrbits, [241](#)
- ndcl
 - MNodeClass, [225](#)
- NDEBUG
 - form3.h, [1326](#)
- ndtype
 - ENode, [118](#)
- ne0bridges
 - MConn, [202](#)
- ne1bridges
 - MConn, [202](#)
- neblocks
 - MConn, [202](#)
- neclass
 - SGroup, [393](#)
- nEdges
 - Assign, [25](#)
 - EGraph, [112](#)
 - MGraph, [212](#)
 - SProcess, [413](#)
- nedges
 - DGraph, [81](#)
 - MCOpI, [205](#)
- NeedNumber
 - declare.h, [792](#)
- NEG0_
 - ftypes.h, [1425](#)
- NEG_
 - ftypes.h, [1425](#)
- nElem
 - SGroup, [392](#)
- nelem
 - SGroup, [393](#)
- Nest
 - T_const, [433](#)
- NESTING
 - structs.h, [2897](#)
- NeStInG, [272](#)
 - argsize, [272](#)
 - funsize, [272](#)
 - termsize, [272](#)
- NestingChecksum
 - declare.h, [805](#)
- nestingsum
 - SWITCH, [428](#)
- NestPoin
 - T_const, [433](#)
- NestStop
 - T_const, [433](#)
- nETotal
 - Assign, [25](#)
- neutral
 - Particle, [331](#)
- newAGraph
 - Options, [305](#)
- newbracketinfo
 - ExPrEsSiOn, [122](#)
- NEWCODE
 - reshuf.c, [2605](#)
- NewDbase
 - declare.h, [976](#)
 - minos.c, [1723](#)
- NEWFACTARG
 - ftypes.h, [1447](#)
- newGraph
 - MGraph, [211](#)
- newGroup
 - SGroup, [392](#)
- newhandle

- HANDLERS, [147](#)
- newleg
 - ANode, [12](#)
- newlogonly
 - HANDLERS, [147](#)
- NEWLYDEFINEDEXPRESSION
 - ftypes.h, [1424](#)
- newmaxtermsize
 - sOrT, [405](#)
- newMGraph
 - Options, [304](#)
- NEWORDER
 - argument.c, [478](#)
- NewSort
 - declare.h, [846](#)
 - sort.c, [2712](#)
- NEWSPLITMERGE
 - sort.c, [2710](#)
- NEWSPLITMERGETIMSORT
 - sort.c, [2710](#)
- NEWSTATEMENT
 - ftypes.h, [1369](#)
- NEWTRICK
 - reken.c, [2547](#)
- next
 - BrAcKeTiNdEx, [32](#)
 - FiLeInDeX, [133](#)
 - longMultiStruct, [167](#)
 - MCBridge, [194](#)
 - MCOpI, [205](#)
 - NaMeSpAcE, [265](#)
 - StOrEcAcHe, [416](#)
- NEXTARG
 - declare.h, [796](#)
- nextchar
 - StreaM, [424](#)
- nextElem
 - SGroup, [393](#)
- nExtern
 - Assign, [25](#)
 - EGraph, [112](#)
 - MGraph, [213](#)
 - Process, [371](#)
 - SProcess, [413](#)
- NextFileIndex
 - declare.h, [811](#)
 - store.c, [2826](#)
- NextPrime
 - declare.h, [981](#)
 - reken.c, [2557](#)
- nfac
 - T_const, [443](#)
- nfactors
 - DoLIARs, [89](#)
- nfial
 - FGInput, [127](#)
 - Process, [371](#)
 - SProcess, [412](#)
- nFlines
 - EGraph, [114](#)
- nfun
 - PaRtl, [326](#)
- nfunctions
 - InDeXeNtRy, [154](#)
- ngconn
 - MGraph, [215](#)
- ngen
 - MGraph, [215](#)
- ngraphs
 - Process, [372](#)
- nhalfmod
 - C_const, [66](#)
- nlist
 - NCand, [270](#)
 - NStack, [279](#)
- nincoef
 - SHvariables, [396](#)
- nindices
 - InDeXeNtRy, [153](#)
- ninitl
 - FGInput, [127](#)
 - Process, [371](#)
 - SProcess, [412](#)
- nInteracts
 - Model, [235](#)
- ninterms
 - N_const, [255](#)
 - sOrT, [403](#)
- nintPart
 - Model, [235](#)
- nlegs
 - AEdge, [8](#)
 - ANode, [13](#)
 - EEdge, [98](#)
 - Interaction, [161](#)
 - MCOpI, [205](#)
- nlist
 - EFLine, [101](#)
- nLoops
 - EGraph, [112](#)
 - MGraph, [212](#)
- nlpopic
 - MConn, [201](#)
- nm
 - MODNUM, [238](#)
- nmedges
 - MCBlock, [193](#)
- nMGraphs
 - Process, [372](#)
 - SProcess, [414](#)
- nmin4
 - InDeX, [150](#)
- NMIN4SHIFT
 - ftypes.h, [1386](#)
- nMOPI
 - Process, [373](#)

- SProcess, [414](#)
- nmultiedges
 - MConn, [202](#)
- nNodes
 - Assign, [24](#)
 - EGraph, [112](#)
 - MGraph, [212](#)
 - MNodeClass, [226](#)
 - MOrbits, [241](#)
 - SProcess, [413](#)
- nnodes
 - DGraph, [81](#)
 - MCOp, [204](#)
 - PNodeClass, [342](#)
 - SGroup, [393](#)
- NOAUTO
 - ftypes.h, [1373](#)
- NOBRACKETACTIVE
 - execute.c, [1152](#)
- NOCHECKLOGTYPE
 - ftypes.h, [1386](#)
- NOCOMPRESS
 - minos.h, [1745](#)
- NoCompress
 - C_const, [61](#)
 - R_const, [378](#)
- NODE
 - parallel.c, [2060](#)
- NoDe, [273](#)
 - left, [273](#)
 - lloser, [273](#)
 - lsrc, [274](#)
 - right, [273](#)
 - rloser, [273](#)
 - rsrc, [274](#)
- node, [274](#)
 - calcHash, [275](#)
 - cmp, [275](#)
 - data, [276](#)
 - DoLIaRS, [88](#)
 - DuBiOuS, [93](#)
 - ExPrEsSiOn, [123](#)
 - FuNcTiOn, [145](#)
 - hash, [276](#)
 - InDeX, [150](#)
 - l, [276](#)
 - node, [275](#)
 - operator(), [275](#)
 - r, [276](#)
 - SeTs, [389](#)
 - sign, [276](#)
 - SyMbOl, [430](#)
 - VeCtOr, [472](#)
- NodeEq, [276](#)
 - operator(), [277](#)
- nodefill
 - NaMeTree, [267](#)
- NODEFUNCTION
 - ftypes.h, [1400](#)
- NodeHash, [277](#)
 - operator(), [277](#)
- noden
 - NStack, [278](#)
- nodes
 - AEdge, [8](#)
 - Assign, [25](#)
 - DGraph, [81](#)
 - EEdge, [98](#)
 - EGraph, [109](#)
 - MCBridge, [194](#)
 - MCEdge, [195](#)
 - MCOp, [204](#)
 - MGraph, [212](#)
- nodesize
 - NaMeTree, [267](#)
- NODOUBLEMASK
 - ftypes.h, [1384](#)
- NOFUNPOWERS
 - ftypes.h, [1439](#)
- nogroundlevel
 - N_const, [258](#)
- NOINDEX
 - ftypes.h, [1427](#)
- NOINVERSES
 - ftypes.h, [1442](#)
- NOLYNDON
 - ftypes.h, [1447](#)
- NONBIPART
 - ftypes.h, [1456](#)
- NOPARALLEL_CONVPOLY
 - ftypes.h, [1367](#)
- NOPARALLEL_DOLLAR
 - ftypes.h, [1367](#)
- NOPARALLEL_NPROC
 - ftypes.h, [1367](#)
- NOPARALLEL_RHS
 - ftypes.h, [1367](#)
- NOPARALLEL_SPECTATOR
 - ftypes.h, [1367](#)
- NOPARALLEL_TBLDOLLAR
 - ftypes.h, [1367](#)
- NOPARALLEL_USER
 - ftypes.h, [1367](#)
- nopi
 - Process, [372](#)
- nopi2p
 - EGraph, [113](#)
- nopic
 - MConn, [201](#)
- nopicomp
 - EGraph, [112](#)
- nopisp
 - MConn, [203](#)
- NOQUADMASK
 - ftypes.h, [1384](#)
- nOrbits

- MOrbits, [241](#)
- normal
 - Fraction, [140](#)
- normal.c, [1866](#), [1870](#)
 - BracketNormalize, [1869](#)
 - Commute, [1867](#)
 - CompareFunctions, [1867](#)
 - DetCommu, [1868](#)
 - DoDelta, [1868](#)
 - DoesCommu, [1868](#)
 - DoRevert, [1868](#)
 - DoTheta, [1868](#)
 - DropCoefficient, [1869](#)
 - DropSymbols, [1869](#)
 - ExtraSymbol, [1867](#)
 - MAXNUMBEROFNONCOMTERMS, [1867](#)
 - Normalize, [1867](#)
 - SymbolNormalize, [1869](#)
 - TestFunFlag, [1869](#)
 - TreatPolyRatFun, [1868](#)
- NORMALFORMAT
 - ftypes.h, [1384](#)
- Normalize
 - declare.h, [846](#)
 - normal.c, [1867](#)
- normalize
 - poly, [353](#)
- NORMALIZEFLAG
 - ftypes.h, [1433](#)
- NormalModulus
 - declare.h, [855](#)
 - reken.c, [2552](#)
- normalParticle
 - Model, [233](#)
- NormPolyTerm
 - declare.h, [993](#)
 - notation.c, [1933](#)
- NORMSIZE
 - fsizes.h, [1341](#)
- NoScrat2
 - N_const, [251](#)
- NOSHELL
 - pre.c, [2311](#)
- NoShowInput
 - C_const, [54](#)
 - DoLoOp, [92](#)
- NOSIGMA
 - ftypes.h, [1455](#)
- NOSNAIL
 - ftypes.h, [1455](#)
- NOSPACEFORMAT
 - ftypes.h, [1384](#)
- NoSpacesInNumbers
 - O_const, [285](#)
- NOTADPOLE
 - ftypes.h, [1455](#)
- notation.c, [1932](#), [1935](#)
 - ConvertFromPoly, [1933](#)
 - ConvertToPoly, [1933](#)
 - ExtraSymFun, [1934](#)
 - FindLocalSubterm, [1934](#)
 - FindSubexpression, [1934](#)
 - FindSubterm, [1934](#)
 - LocalConvertToPoly, [1933](#)
 - NormPolyTerm, [1933](#)
 - PrintExtraSymbol, [1934](#)
 - PrintSubtermList, [1934](#)
 - PruneExtraSymbols, [1935](#)
- NOTEQUAL
 - ftypes.h, [1436](#)
- NOTFLOOP
 - ftypes.h, [1457](#)
- NOTRICK
 - ftypes.h, [1385](#)
- NOTSIMPLE
 - ftypes.h, [1456](#)
- NOTSTARTPOS
 - declare.h, [801](#)
- NOUNPACK
 - ftypes.h, [1443](#)
- nPacks
 - longMultiStruct, [166](#)
- NPAIR1
 - fsizes.h, [1347](#)
- NPAIR2
 - fsizes.h, [1347](#)
- nParticles
 - Model, [234](#)
- nparticles
 - MoDeL, [228](#)
 - VerRtEx, [473](#)
- nplist
 - ECand, [95](#)
 - EStack, [120](#)
 - Interaction, [161](#)
- nplistn
 - lInput, [148](#)
- npowmod
 - C_const, [65](#)
- nqgopt
 - Options, [307](#)
- nselfloops
 - MConn, [202](#)
- NSIG
 - portsignals.h, [2303](#)
- nSize
 - AStack, [29](#)
- nslist
 - Interaction, [161](#)
- NStack, [277](#)
 - ~NStack, [278](#)
 - deg, [278](#)
 - ilist, [279](#)
 - nilist, [279](#)
 - noden, [278](#)
 - NStack, [278](#)

- print, [278](#)
- st, [279](#)
- nStack
 - AStack, [28](#)
- nStackP
 - AStack, [29](#)
- nSubproc
 - Process, [371](#)
- nsym
 - EGraph, [110](#)
 - MGraph, [213](#)
- nsym1
 - EGraph, [111](#)
- nsymbols
 - InDeXeNtRy, [153](#)
- nt3
 - TrAcEs, [462](#)
- nt4
 - TrAcEs, [462](#)
- num
 - Fraction, [141](#)
 - LIST, [164](#)
 - TrAcEn, [459](#)
 - TrAcEs, [465](#)
- num_terms
 - BracketInfo, [36](#)
- num_visits
 - tree_node, [469](#)
- NUMARG
 - ftypes.h, [1444](#)
- numargs
 - FUN_INFO, [142](#)
 - PaRtl, [326](#)
- NUMARGSFUN
 - ftypes.h, [1392](#)
- number
 - FiLeInDeX, [133](#)
 - FuNcTiOn, [144](#)
 - InDeX, [150](#)
 - MoDoPtDoLIaRS, [238](#)
 - NaMeNode, [264](#)
 - PaRtlcLe, [327](#)
 - SyMbOl, [430](#)
 - VeCtOr, [471](#)
- number_of_terms
 - poly, [350](#)
- NUMBEREXTRAWORDS
 - tools.c, [3071](#)
- NumberFree
 - declare.h, [805](#)
- NumberMalloc
 - declare.h, [804](#)
- NumberMallocAddMemory
 - declare.h, [988](#)
 - tools.c, [3083](#)
- NumberMemHeap
 - T_const, [436](#)
- NumberMemMax
 - T_const, [440](#)
- NUMBERMEMSTARTNUM
 - tools.c, [3071](#)
- NumberMemTop
 - T_const, [440](#)
- numberofindexblocks
 - iniinfo, [155](#)
- numberofnamesblocks
 - iniinfo, [156](#)
- numberoftables
 - iniinfo, [155](#)
- numbers
 - DICTIONARY, [82](#)
- numblocks
 - FiLe, [130](#)
- numbufs
 - PF_BUFFER, [336](#)
- numcases
 - SWITCH, [427](#)
- numclear
 - LIST, [165](#)
- numcommute
 - comtool.c, [757](#)
 - declare.h, [952](#)
- NumCopy
 - declare.h, [910](#)
 - tools.c, [3081](#)
- numdia
 - TERMINFO, [455](#)
- NumDictionaries
 - O_const, [286](#)
- NumDollars
 - variable.h, [3192](#)
- numdollars
 - INSIDEINFO, [157](#)
- NumDoLoops
 - variable.h, [3189](#)
- NumDubious
 - variable.h, [3193](#)
- numdum
 - CbUf, [72](#)
- numdummies
 - DoLIaRS, [89](#)
 - ExPrEsSiOn, [124](#)
 - TabIEs, [454](#)
- numelements
 - DICTIONARY, [82](#)
- NUMERATORSYMBOL
 - ftypes.h, [1403](#)
- NUMERICALLOOP
 - ftypes.h, [1371](#)
- NUMERICALVALUE
 - setfile.c, [2684](#)
- NumExpressions
 - variable.h, [3193](#)
- numextern
 - TERMINFO, [456](#)
- NUMFACS

- fsizes.h, [1342](#)
- NUMFACTORS
 - ftypes.h, [1398](#)
- numfactors
 - ExPrEsSiOn, [124](#)
- numfargs
 - N_const, [255](#)
- NUMFIXED
 - fsizes.h, [1341](#)
- NumFixedFunctions
 - M_const, [177](#)
- NumFixedSets
 - M_const, [177](#)
- numflocs
 - N_const, [256](#)
- NumFound
 - N_const, [259](#)
- NumFunctions
 - variable.h, [3190](#)
- numfuninfo
 - N_const, [257](#)
- numfunnies
 - FUN_INFO, [142](#)
- NumFunSorts
 - N_const, [257](#)
- numglobal
 - LIST, [165](#)
- NumInBrack
 - O_const, [284](#)
- numind
 - TaBIEs, [451](#)
- NumIndices
 - variable.h, [3190](#)
- numinfilelist
 - tools.c, [3088](#)
- NumLabels
 - C_const, [55](#)
- numlhs
 - CbUf, [73](#)
- numlistinprint
 - N_const, [260](#)
- NumListSymbols
 - T_const, [441](#)
- NumMem
 - T_const, [437](#)
- nummodels
 - C_const, [63](#)
- NumModOptdollars
 - variable.h, [3194](#)
- numoffuns
 - N_const, [256](#)
- NumOldNumFactors
 - S_const, [387](#)
- NumOldOnFile
 - S_const, [387](#)
- NUMOPTIONS
 - fsizes.h, [1348](#)
- NumOutputChannels
 - variable.h, [3192](#)
- numpart
 - PaRtl, [326](#)
- numParticle
 - diawrap.cc, [1051](#)
- numpoly
 - T_const, [441](#)
- NumPotModdollars
 - variable.h, [3194](#)
- NumPre
 - variable.h, [3189](#)
- NumPreSwitchStrings
 - P_const, [320](#)
- NumPreTypes
 - DoLoOp, [92](#)
 - P_const, [320](#)
- NumProcedures
 - variable.h, [3188](#)
- numrbufs
 - ParallelVars, [325](#)
- numrhs
 - CbUf, [73](#)
- numsbufs
 - ParallelVars, [325](#)
- NumSetElements
 - variable.h, [3191](#)
- NumSets
 - variable.h, [3191](#)
- NUMSPECSETS
 - ftypes.h, [1429](#)
- NumSpectatorFiles
 - M_const, [183](#)
- NUMSTORECACHES
 - fsizes.h, [1346](#)
- NumStoreCaches
 - M_const, [190](#)
- NumStreams
 - C_const, [56](#)
- numsym
 - POLYMOD, [363](#)
- NumSymbols
 - variable.h, [3191](#)
- NumTableBases
 - variable.h, [3191](#)
- NUMTABLEENTRIES
 - fsizes.h, [1343](#)
- numtables
 - TaBIEbAsE, [446](#)
- numtasks
 - ParallelVars, [324](#)
- numtemp
 - LIST, [165](#)
- NumTerms
 - CbUf, [72](#)
- NUMTERMSFUN
 - ftypes.h, [1394](#)
- NUMTOIND
 - ftypes.h, [1418](#)

- NUMTONUM
 - ftypes.h, [1417](#)
- numtopo
 - TERMINFO, [455](#)
- NumToStr
 - declare.h, [905](#)
 - tools.c, [3078](#)
- NUMTOSUB
 - ftypes.h, [1418](#)
- NUMTOSYM
 - ftypes.h, [1418](#)
- NumTotWildArgs
 - N_const, [255](#)
- numtracesctack
 - N_const, [256](#)
- numtree
 - CbUf, [74](#)
 - TaBIes, [452](#)
- NumVectors
 - variable.h, [3191](#)
- NumWild
 - N_const, [258](#)
- NumWildcardNames
 - C_const, [55](#)
- numwildcards
 - FUN_INFO, [142](#)
- numxsymbol
 - variable.h, [3194](#)
- nvectors
 - InDeXeNtRy, [154](#)
- nvert
 - SProcess, [413](#)
 - ToPoTyPe, [458](#)
- nvertices
 - MoDeL, [228](#)
- NwildC
 - C_const, [66](#)
- O
 - AllGlobals, [11](#)
- O_BACKWARD
 - ftypes.h, [1449](#)
- O_const, [279](#)
 - BlockSpaces, [285](#)
 - bracket, [282](#)
 - bufferedInd, [288](#)
 - CheckPower, [284](#)
 - CurBufWrt, [282](#)
 - CurDictFunWithArgs, [286](#)
 - CurDictInDollars, [287](#)
 - CurDictNotInFunctions, [286](#)
 - CurDictNumbers, [286](#)
 - CurDictNumberWarning, [286](#)
 - CurDictSpecials, [286](#)
 - CurDictVariables, [286](#)
 - CurrentDictionary, [285](#)
 - Dictionaries, [284](#)
 - DollarInOutBuffer, [285](#)
 - DollarOutBuffer, [282](#)
 - DollarOutSizeBuffer, [285](#)
 - DoubleFlag, [288](#)
 - ErrorBlock, [289](#)
 - FactorMode, [288](#)
 - FactorNum, [289](#)
 - FlipLONG, [283](#)
 - FlipPOINTER, [283](#)
 - FlipPOS, [283](#)
 - FlipWORD, [283](#)
 - FortDotChar, [289](#)
 - FortFirst, [288](#)
 - gNumDictionaries, [287](#)
 - IndentSpace, [287](#)
 - InFbrack, [288](#)
 - inscheme, [284](#)
 - IsBracket, [288](#)
 - mpower, [287](#)
 - NoSpacesInNumbers, [285](#)
 - NumDictionaries, [286](#)
 - NumInBrack, [284](#)
 - OptimizationLevel, [289](#)
 - Optimize, [285](#)
 - OptimizeResult, [281](#)
 - OutFill, [282](#)
 - OutInBuffer, [285](#)
 - OutputLine, [281](#)
 - OutSkip, [288](#)
 - OutStop, [281](#)
 - powerFlag, [287](#)
 - PrintType, [288](#)
 - ReNUMBERVec, [284](#)
 - ResizeData, [283](#)
 - resizeFlag, [287](#)
 - ResizeLONG, [283](#)
 - ResizeNCWORD, [283](#)
 - ResizePOINTER, [284](#)
 - ResizePOS, [284](#)
 - ResizeWORD, [283](#)
 - SaveData, [281](#)
 - SaveHeader, [281](#)
 - schemenum, [287](#)
 - SizeDictionaries, [286](#)
 - tabstring, [282](#)
 - tensorList, [284](#)
 - termbuf, [282](#)
 - transFlag, [287](#)
 - wlen, [285](#)
 - wpoin, [282](#)
 - wpos, [282](#)
- O_CSE
 - ftypes.h, [1448](#)
- O_CSEGREEDY
 - ftypes.h, [1448](#)
- O_FORWARD
 - ftypes.h, [1449](#)
- O_FORWARDANDBACKWARD
 - ftypes.h, [1449](#)
- O_FORWARDORBACKWARD

- ftypes.h, 1449
- O_GREEDY
 - ftypes.h, 1448
- O_MCTS
 - ftypes.h, 1449
- O_NONE
 - ftypes.h, 1448
- O_OCCURRENCE
 - ftypes.h, 1449
- O_SIMULATED_ANNEALING
 - ftypes.h, 1449
- obj1
 - DiStRiBuTe, 85
- obj2
 - DiStRiBuTe, 85
- objects, 289
 - date, 290
 - element, 291
 - indexblock, 151
 - PeRmUtE, 332
 - PeRmUtEp, 333
 - position, 290
 - size, 290
 - spare1, 290
 - spare2, 290
 - spare3, 291
 - tablenumber, 290
 - uncompressed, 290
- occurrence_order
 - optimize.cc, 1985
- ODD_
 - ftypes.h, 1426
- OffsetIndex
 - M_const, 186
- OffsetVector
 - M_const, 186
- OFFSHELL
 - ftypes.h, 1455
- oldcbuf
 - INSIDEINFO, 157
- OldChildTime
 - M_const, 175
- oldcnumlhs
 - INSIDEINFO, 158
- oldcompiletype
 - INSIDEINFO, 157
- olddelay
 - StreaM, 424
- OLDFACTARG
 - ftypes.h, 1447
- OldFactArgFlag
 - C_const, 61
- OldGCDflag
 - C_const, 62
- oldhandle
 - HANDLERS, 147
- oldlogonly
 - HANDLERS, 147
- OldMilliTime
 - M_const, 175
- oldnamefill
 - NaMeTree, 267
- oldnodefill
 - NaMeTree, 267
- oldnoshowinput
 - StreaM, 424
- oldnumextrasymbols
 - M_const, 173
- OldNumFactors
 - S_const, 387
- oldnumpotmoddollars
 - INSIDEINFO, 157
- OldOnFile
 - S_const, 386
- OldOrderFlag
 - M_const, 188
- oldparallellflag
 - INSIDEINFO, 157
- OldParallelStats
 - C_const, 58
- OldPosIn
 - N_const, 247
- OldPosOut
 - N_const, 247
- oldprinttype
 - HANDLERS, 147
- oldrbuf
 - INSIDEINFO, 158
- OldSecTime
 - M_const, 175
- oldsilent
 - HANDLERS, 147
- OLDSTATEMENT
 - ftypes.h, 1369
- OldTime
 - R_const, 377
- oldtype
 - N_const, 258
- Olduflags
 - S_const, 387
- oldvalue
 - N_const, 258
- Oldvflags
 - S_const, 387
- OMPI_SKIP_MPICXX
 - parallel.h, 2134
- one_byte
 - structs.h, 2896
- ONEEXPRESSION
 - ftypes.h, 1371
- ONEPARTI
 - ftypes.h, 1454
- ONEPARTR
 - ftypes.h, 1455
- ONEPI
 - ftypes.h, 1401

- onerhs
 - M_const, [191](#)
- onfile
 - ExPrEsSiOn, [121](#)
- ONLYFUNCTIONS
 - ftypes.h, [1450](#)
- ONOFFVALUE
 - setfile.c, [2684](#)
- ONSHELL
 - ftypes.h, [1455](#)
- OpenAddFile
 - declare.h, [883](#)
 - tools.c, [3074](#)
- OpenBracketIndex
 - declare.h, [969](#)
 - index.c, [1664](#)
- OpenDbase
 - declare.h, [975](#)
 - minos.c, [1723](#)
- OpenDictionary
 - P_const, [321](#)
- openExternalChannel
 - declare.h, [978](#)
 - extcmd.c, [1188](#)
- OpenFile
 - declare.h, [883](#)
 - tools.c, [3073](#)
- OpenInput
 - declare.h, [874](#)
 - startup.c, [2793](#)
- OpenStream
 - declare.h, [876](#)
 - tools.c, [3072](#)
- OpenTemp
 - declare.h, [847](#)
 - store.c, [2822](#)
- OPER_ADD
 - optimize.cc, [2001](#)
- OPER_COMMA
 - optimize.cc, [2001](#)
- OPER_MUL
 - optimize.cc, [2001](#)
- opera.c, [1949](#), [1952](#)
 - Chisholm, [1951](#)
 - EpfCon, [1949](#)
 - EpfFind, [1949](#)
 - EpfGen, [1950](#)
 - TenVec, [1952](#)
 - TenVecFind, [1952](#)
 - Trace4, [1950](#)
 - Trace4Gen, [1950](#)
 - Trace4no, [1950](#)
 - TraceFind, [1951](#)
 - TraceN, [1951](#)
 - TraceNgen, [1951](#)
 - TraceNno, [1951](#)
 - Traces, [1951](#)
 - Trick, [1950](#)
- OperaFind
 - FixedGlobals, [134](#)
- Operation
 - FixedGlobals, [134](#)
- operator!=
 - poly, [348](#)
- operator<
 - BracketInfo, [36](#)
 - optimization, [293](#)
- operator<<
 - polyfact.cc, [2232](#)
- operator()
 - CSEEq, [77](#)
 - CSEHash, [77](#)
 - monomial_larger, [239](#)
 - node, [275](#)
 - NodeEq, [277](#)
 - NodeHash, [277](#)
- operator+
 - poly, [347](#)
- operator+=
 - poly, [346](#)
- operator-
 - poly, [347](#)
- operator=
 - poly, [346](#)
- operator/
 - poly, [347](#)
- operator/=
 - poly, [346](#)
- operator=
 - poly, [348](#)
 - tree_node, [469](#)
- operator==
 - poly, [347](#)
- operator%
 - poly, [347](#)
- operator%=
 - poly, [347](#)
- operator[]
 - poly, [348](#)
- operator*
 - poly, [347](#)
- operator*=
 - poly, [346](#)
- opi
 - MGraph, [215](#)
- opi2plp
 - EGraph, [113](#)
- opicomp
 - EEdge, [100](#)
- opiCount
 - EGraph, [114](#)
- opics
 - MConn, [200](#)
- opiloop
 - MGraph, [215](#)
- opisp

- MConn, 200
- opistk
 - MConn, 200
- opistkptr
 - MConn, 203
- opt
 - Assign, 24
 - EGraph, 109
 - MGraph, 211
 - Output, 313
 - Process, 370
 - SProcess, 411
 - ToPoTyPe, 457
- OptDef, 291
 - defaultv, 292
 - DUMMYPADDING, 292
 - mean, 291
 - name, 291
- OPTHEAD
 - ftypes.h, 1449
- optimization, 292
 - arg1, 293
 - arg2, 294
 - coeff, 294
 - eqnidxs, 294
 - improve, 294
 - operator<, 293
 - type, 293
- OptimizationLevel
 - O_const, 289
- OPTIMIZE, 295
 - debugflags, 297
 - fval, 295
 - greedymaxperc, 297
 - greedyminnum, 296
 - greedytimelimit, 296
 - horner, 295
 - hornerdirection, 296
 - ival, 295
 - mctsdecaymode, 297
 - mctsnumexpand, 296
 - mctsnumkeep, 296
 - mctsnumrepeat, 296
 - mctstimelimit, 296
 - method, 296
 - printstats, 297
 - salter, 297
 - schemeflags, 297
 - spare, 297
- Optimize
 - declare.h, 957
 - O_const, 285
 - optimize.cc, 1998
- optimize.cc, 1981, 2003
 - build_Horner_tree, 1987
 - buildTree, 1991
 - ClearOptimize, 2000
 - count_operators, 1984, 1985
 - count_operators_cse, 1990
 - count_operators_cse_topdown, 1991
 - DivLong_fix_args, 1987
 - do_optimization, 1993
 - find_Horner_MCTS, 1991
 - find_Horner_MCTS_expand_tree, 1991
 - find_optimizations, 1993
 - fixarg, 1986
 - GcdLong_fix_args, 1987
 - generate_expression, 1997
 - generate_instructions, 1989
 - generate_output, 1996
 - get_brackets, 1984
 - get_expression, 1983
 - getpower, 1985
 - hash_range, 1989
 - Horner_tree, 1988
 - mcts_best_schemes, 2003
 - mcts_expr_score, 2002
 - mcts_factorized, 2002
 - mcts_root, 2002
 - mcts_separated, 2002
 - mcts_vars, 2002
 - merge_operators, 1992
 - my_random_shuffle, 1983
 - occurrence_order, 1985
 - OPER_ADD, 2001
 - OPER_COMMA, 2001
 - OPER_MUL, 2001
 - Optimize, 1998
 - optimize_best_Horner_schemes, 2001
 - optimize_best_instr, 2002
 - optimize_best_num_oper, 2002
 - optimize_best_vars, 2002
 - optimize_expr, 2001
 - optimize_expression_given_Horner, 1996
 - optimize_greedy, 1994
 - optimize_num_vars, 2001
 - optimize_print_code, 1998
 - partial_factorize, 1994
 - print_instr, 1983
 - print_tree, 1989
 - recycle_variables, 1995
 - simulated_annealing, 1991
 - term_compare, 1988
- optimize_best_Horner_schemes
 - optimize.cc, 2001
- optimize_best_instr
 - optimize.cc, 2002
- optimize_best_num_oper
 - optimize.cc, 2002
- optimize_best_vars
 - optimize.cc, 2002
- optimize_expr
 - optimize.cc, 2001
- optimize_expression_given_Horner
 - optimize.cc, 1996
- optimize_greedy

- optimize.cc, [1994](#)
- optimize_num_vars
 - optimize.cc, [2001](#)
- optimize_print_code
 - declare.h, [1009](#)
 - optimize.cc, [1998](#)
- OPTIMIZERESULT, [298](#)
 - code, [298](#)
 - codesize, [298](#)
 - exprnr, [298](#)
 - maxvar, [299](#)
 - minvar, [299](#)
 - nameofexpr, [298](#)
- OptimizeResult
 - O_const, [281](#)
- optintimes
 - T_const, [440](#)
- option
 - SHvariables, [397](#)
- OPTION0
 - compiler.c, [712](#)
- OPTION1
 - compiler.c, [712](#)
- OPTION2
 - compiler.c, [712](#)
- Options, [299](#)
 - ~Options, [301](#)
 - argag, [306](#)
 - argemg, [306](#)
 - argmg, [306](#)
 - begin, [304](#)
 - beginProc, [304](#)
 - beginSubProc, [304](#)
 - DUMMYPPADDING, [307](#)
 - end, [304](#)
 - endmg, [306](#)
 - endProc, [304](#)
 - endSubProc, [304](#)
 - getDef, [303](#)
 - getOldDef, [303](#)
 - getQGDef, [303](#)
 - getValue, [301](#)
 - model, [305](#)
 - newAGraph, [305](#)
 - newMGraph, [304](#)
 - nqgopt, [307](#)
 - Options, [301](#)
 - out, [305](#)
 - outag, [306](#)
 - outmg, [305](#)
 - outModel, [305](#)
 - print, [302](#)
 - printLevel, [302](#)
 - printModel, [302](#)
 - proc, [305](#)
 - qgopt, [306](#)
 - qgref, [306](#)
 - setDefaultValues, [301](#)
 - setEndMG, [303](#)
 - setErExit, [303](#)
 - setOldDefaultValues, [301](#)
 - setOutAG, [303](#)
 - setOutMG, [302](#)
 - setOutputF, [302](#)
 - setOutputP, [302](#)
 - setQGrafOpt, [301](#)
 - setValue, [301](#)
 - sproc, [305](#)
 - time0, [307](#)
 - time1, [307](#)
 - values, [306](#)
- OptQGDef, [307](#)
 - cname, [308](#)
 - mean, [308](#)
 - name, [308](#)
- optQGrafA
 - EGraph, [105](#)
- optQGrafM
 - EGraph, [105](#)
- OptQGRef, [308](#)
 - index, [308](#)
 - name, [308](#)
 - sign, [309](#)
- or2nd
 - MOrbits, [241](#)
- orbits
 - Assign, [24](#)
 - MGraph, [212](#)
- ORCOND
 - ftypes.h, [1436](#)
- ORDEREDSET
 - ftypes.h, [1453](#)
- Ordering
 - M_const, [187](#)
- origin
 - C_const, [63](#)
- out
 - DiStRiBuTe, [85](#)
 - Options, [305](#)
- outag
 - Options, [306](#)
- outBeginF
 - Output, [310](#)
- outBeginP
 - Output, [311](#)
- OutBuffer
 - M_const, [172](#)
- OutBufSize
 - M_const, [176](#)
- outEGraphF
 - Output, [312](#)
- outEGraphP
 - Output, [312](#)
- outEndF
 - Output, [311](#)
- outEndP

- Output, 311
- outfile
 - R_const, 376
- OutFill
 - O_const, 282
- outfun
 - SHvariables, 395
- outgrf
 - Output, 313
- outgrfp
 - Output, 313
- outgrp
 - Output, 314
- outgrpp
 - Output, 314
- OutInBuffer
 - O_const, 285
- outmg
 - Options, 305
- outModel
 - Options, 305
- outModelF
 - Output, 312
- outModelP
 - Output, 313
- OutNumberType
 - C_const, 67
- outproc
 - Output, 314
- outProcBegin0
 - Output, 311
- outProcBeginF
 - Output, 311
- outProcBeginP
 - Output, 311
- outProcEndF
 - Output, 312
- outProcEndP
 - Output, 312
- outProcP
 - Process, 369
- Output, 309
 - ~Output, 310
 - model, 313
 - opt, 313
 - outBeginF, 310
 - outBeginP, 311
 - outEGraphF, 312
 - outEGraphP, 312
 - outEndF, 311
 - outEndP, 311
 - outgrf, 313
 - outgrfp, 313
 - outgrp, 314
 - outgrpp, 314
 - outModelF, 312
 - outModelP, 313
 - outproc, 314
 - outProcBegin0, 311
 - outProcBeginF, 311
 - outProcBeginP, 311
 - outProcEndF, 312
 - outProcEndP, 312
 - Output, 310
 - outSProcBeginF, 312
 - outSProcBeginP, 312
 - proc, 313
 - proclD, 314
 - setOutgrf, 310
 - setOutgrp, 310
 - sproc, 313
- OutputLine
 - O_const, 281
- OutputMode
 - C_const, 67
- outputmode
 - sOrT, 405
- OutputOnePI
 - lus.c, 1681
- OUTPUTONLY
 - minos.h, 1745
- OutputSpaces
 - C_const, 67
- outsidedefun
 - C_const, 57
- OutSkip
 - O_const, 288
- outSProcBeginF
 - Output, 312
- outSProcBeginP
 - Output, 312
- OutStop
 - O_const, 281
- outterm
 - SHvariables, 395
- outtohide
 - R_const, 379
- P
 - AllGlobals, 11
- p
 - BracketInfo, 36
 - DoLoOp, 90
 - PROCEDURE, 367
- p1
 - PoSITiOn, 364
- P_const, 315
 - AllowDelay, 320
 - cComChar, 322
 - ComChar, 322
 - cprocedureExtension, 318
 - DebugFlag, 322
 - DelayPrevar, 320
 - DollarList, 316
 - eat, 321
 - firstnamespace, 317
 - fullname, 317

- fullnamesize, [321](#)
- gNumPre, [321](#)
- iBufError, [319](#)
- InOutBuf, [319](#)
- inside, [317](#)
- lastnamespace, [317](#)
- lhdollarerror, [321](#)
- LoopList, [316](#)
- MaxPreAssignLevel, [321](#)
- MaxPreIfLevel, [320](#)
- MaxPreTypes, [320](#)
- NumPreSwitchStrings, [320](#)
- NumPreTypes, [320](#)
- OpenDictionary, [321](#)
- PreAssignFlag, [319](#)
- PreAssignLevel, [321](#)
- PreAssignStack, [318](#)
- PreContinuation, [319](#)
- PreDebug, [321](#)
- preError, [322](#)
- preFill, [318](#)
- PreIfLevel, [320](#)
- PreIfStack, [318](#)
- PreInsideLevel, [320](#)
- PreOut, [319](#)
- PreproFlag, [319](#)
- preStart, [317](#)
- preStop, [317](#)
- PreSwitchLevel, [319](#)
- PreSwitchModes, [318](#)
- PreSwitchStrings, [317](#)
- PreTypes, [318](#)
- PreVarList, [316](#)
- procedureExtension, [318](#)
- ProcList, [317](#)
- pSize, [319](#)
- StopWatchZero, [318](#)
- P_term
 - fwin.h, [1497](#)
 - unix.h, [3181](#)
- Pack
 - declare.h, [847](#)
 - reken.c, [2548](#)
- PACK_LONG
 - parallel.c, [2058](#)
- PackFloat
 - evaluate.c, [1140](#)
 - float.c, [1287](#)
- packpos
 - longMultiStruct, [166](#)
- padding
 - SWITCH, [428](#)
- parallel
 - ParallelVars, [324](#)
- parallel.c, [2055](#), [2081](#)
 - _CHECK, [2058](#)
 - _CHECK, [2059](#)
 - CHECK, [2058](#)
- DBGOUT, [2059](#)
- DBGOUT_NINTERMS, [2059](#)
- NODE, [2060](#)
- PACK_LONG, [2058](#)
- PF, [2080](#)
- PF_BroadcastBuffer, [2070](#)
- PF_BroadcastCBuf, [2075](#)
- PF_BroadcastExpFlags, [2075](#)
- PF_BroadcastExpr, [2076](#)
- PF_BroadcastModifiedDollars, [2073](#)
- PF_BroadcastNumber, [2069](#)
- PF_BroadcastPreDollar, [2071](#)
- PF_BroadcastRedefinedPreVars, [2074](#)
- PF_BroadcastRHS, [2076](#)
- PF_BroadcastString, [2070](#)
- PF_CollectModifiedDollars, [2072](#)
- PF_Deferred, [2065](#)
- PF_EndSort, [2064](#)
- PF_FlushStdOutBuffer, [2080](#)
- PF_FreeErrorMessageBuffers, [2080](#)
- PF_GetSlaveTimes, [2069](#)
- PF_Init, [2067](#)
- PF_InParallelProcessor, [2077](#)
- PF_IRecvRbuf, [2062](#)
- PF_LibInit, [2060](#)
- PF_LibTerminate, [2061](#)
- PF_MLock, [2079](#)
- PF_MUnlock, [2079](#)
- PF_Probe, [2061](#)
- PF_Processor, [2066](#)
- PF_RawProbe, [2064](#)
- PF_RawRecv, [2063](#)
- PF_RawSend, [2063](#)
- PF_RealTime, [2060](#)
- PF_RecvFile, [2078](#)
- PF_RecvWbuf, [2061](#)
- PF_RestoreInsideInfo, [2075](#)
- PF_SendFile, [2077](#)
- PF_SNDFILEBUFSIZE, [2059](#)
- PF_STATS_SIZE, [2059](#)
- PF_StoreInsideInfo, [2074](#)
- PF_Terminate, [2068](#)
- PF_WaitRbuf, [2062](#)
- PF_WriteFileToFile, [2079](#)
- PRINTFBUF, [2057](#)
- recvBuffer, [2060](#)
- SWAP, [2057](#)
- UNPACK_LONG, [2058](#)
- parallel.h, [2129](#), [2162](#)
 - GNUPREREQ, [2133](#)
 - indices, [2134](#)
 - MASTER, [2131](#)
 - OMPI_SKIP_MPICXX, [2134](#)
 - PF, [2161](#)
 - PF_ANY_MSGTAG, [2134](#)
 - PF_ANY_SOURCE, [2134](#)
 - PF_Bcast, [2135](#)
 - PF_Broadcast, [2140](#)

- PF_BroadcastBuffer, 2151
- PF_BroadcastCBuf, 2155
- PF_BroadcastExpFlags, 2156
- PF_BroadcastExpr, 2157
- PF_BroadcastModifiedDollars, 2154
- PF_BroadcastNumber, 2150
- PF_BroadcastPreDollar, 2152
- PF_BroadcastRedefinedPreVars, 2155
- PF_BroadcastRHS, 2158
- PF_BroadcastString, 2151
- PF_BUFFER_MSGTAG, 2132
- PF_BYTE, 2134
- PF_CollectModifiedDollars, 2153
- PF_COMM, 2134
- PF_DATA_MSGTAG, 2132
- PF_Deferred, 2146
- PF_DOLLAR_MSGTAG, 2132
- PF_EMPTY_MSGTAG, 2133
- PF_ENDBUFFER_MSGTAG, 2132
- PF_EndSort, 2145
- PF_ENDSORT_MSGTAG, 2132
- PF_FlushStdOutBuffer, 2161
- PF_GetSlaveTimes, 2150
- PF_Init, 2148
- PF_InParallelProcessor, 2158
- PF_INT, 2134
- PF_ISendSbuf, 2135
- PF_LOG_MSGTAG, 2133
- PF_LongMultiBroadcast, 2145
- PF_LongMultiPack, 2135
- PF_LongMultiPackImpl, 2143
- PF_LongMultiUnpack, 2135
- PF_LongMultiUnpackImpl, 2144
- PF_LongSinglePack, 2141
- PF_LongSingleReceive, 2142
- PF_LongSingleSend, 2142
- PF_LongSingleUnpack, 2141
- PF_maxDollarChunkSize, 2161
- PF_MISC_MSGTAG, 2133
- PF_MLock, 2160
- PF_MUnlock, 2160
- PF_OPT_COLLECT_MSGTAG, 2133
- PF_OPT_HORNER_MSGTAG, 2133
- PF_OPT_MCTS_MSGTAG, 2133
- PF_Pack, 2137
- PF_PackString, 2138
- PF_PrepareLongMultiPack, 2143
- PF_PrepareLongSinglePack, 2140
- PF_PreparePack, 2137
- PF_Processor, 2147
- PF_RawRecv, 2136
- PF_RawSend, 2136
- PF_READY_MSGTAG, 2132
- PF_Receive, 2140
- PF_RecvFile, 2159
- PF_RESET, 2131
- PF_RestoreInsideInfo, 2157
- PF_Send, 2139
- PF_SendFile, 2158
- PF_STDOUT_MSGTAG, 2133
- PF_StoreInsideInfo, 2157
- PF_TERM_MSGTAG, 2132
- PF_Terminate, 2149
- PF_TIME, 2131
- PF_Unpack, 2138
- PF_UnpackString, 2139
- PF_WriteFileToFile, 2160
- PARALLELFLAG
 - ftypes.h, 1368
- parallelflag
 - C_const, 59
- ParallelVars, 323
 - exprbufsize, 324
 - exprtodo, 324
 - log, 325
 - me, 324
 - mkSlaveInfile, 324
 - numrbufs, 325
 - numsbufs, 325
 - numtasks, 324
 - parallel, 324
 - rbufs, 324
 - rhsInParallel, 324
 - sbuf, 323
 - slavebuf, 323
- parameter
 - SETUPPARAMETERS, 390
- parent
 - NaMeNode, 263
 - tree, 466
- ParenthesesTest
 - compiler.c, 712
 - declare.h, 946
- ParseNumber
 - declare.h, 791
- ParseSign
 - declare.h, 791
- ParseSignedNumber
 - declare.h, 791
- PARTEST
 - ftypes.h, 1373
- PARTI
 - structs.h, 2898
- PaRtl, 325
 - args, 326
 - nargs, 326
 - nfun, 326
 - numargs, 326
 - numpart, 326
 - psize, 326
 - where, 326
- partial_factorize
 - optimize.cc, 1994
- PaRtlcLe, 327
 - mass, 327
 - number, 327

- spin, [327](#)
- type, [327](#)
- Particle, [328](#)
 - ~Particle, [329](#)
 - acode, [331](#)
 - aname, [330](#)
 - aparticle, [329](#)
 - cmaxdeg, [331](#)
 - cmindeg, [331](#)
 - extonly, [331](#)
 - id, [330](#)
 - isNeutral, [330](#)
 - mdl, [330](#)
 - name, [330](#)
 - neutral, [331](#)
 - Particle, [329](#)
 - particleCode, [329](#)
 - particleName, [329](#)
 - pcode, [331](#)
 - prParticle, [329](#)
 - ptype, [331](#)
 - typeGName, [330](#)
 - typeName, [330](#)
- particle
 - PNodeClass, [341](#)
- particleCode
 - Model, [232](#)
 - Particle, [329](#)
- particleName
 - Model, [232](#)
 - Particle, [329](#)
- particles
 - Model, [234](#)
 - VerTex, [473](#)
- PARTITIONS
 - ftypes.h, [1399](#)
- partitions
 - T_const, [437](#)
- partodoflag
 - C_const, [60](#)
- PasteFile
 - declare.h, [847](#)
 - proces.c, [2424](#)
- PasteTerm
 - declare.h, [811](#)
 - proces.c, [2424](#)
- Patches
 - sOrT, [400](#)
- Path
 - M_const, [172](#)
- PathOutput
 - declare.h, [1020](#)
 - lus.c, [1680](#)
- PATHSEPARATOR
 - fwin.h, [1497](#)
 - unix.h, [3182](#)
- PATHVALUE
 - setfile.c, [2684](#)
- patstop
 - N_const, [249](#)
- pattern
 - BracketInfo, [36](#)
 - TaBIEs, [449](#)
- pattern.c, [2165](#), [2168](#)
 - FindAll, [2167](#)
 - PutInBuffers, [2166](#)
 - SubsInAll, [2167](#)
 - Substitute, [2167](#)
 - TakeIDfunction, [2168](#)
 - TestMatch, [2166](#)
 - TestSelect, [2167](#)
 - TransferBuffer, [2168](#)
- patternbuffer
 - N_const, [252](#)
- patternbuffersize
 - N_const, [260](#)
- PATTERNFUNCTION
 - ftypes.h, [1394](#)
- pBer
 - T_const, [436](#)
- pcode
 - Particle, [331](#)
 - PGInput, [337](#)
 - PInput, [338](#)
- pdef
 - Model, [235](#)
- pdel
 - TrAcEs, [463](#)
- pepf
 - TrAcEs, [463](#)
- PERM
 - structs.h, [2898](#)
- perm
 - TrAcEn, [459](#)
 - TrAcEs, [462](#)
- permg
 - SGroup, [394](#)
- permMat
 - MGraph, [209](#)
- PERMP
 - structs.h, [2898](#)
- perms
 - SGroup, [394](#)
- PERMUTATIONS
 - ftypes.h, [1399](#)
- PeRmUtE, [332](#)
 - cycle, [333](#)
 - n, [332](#)
 - objects, [332](#)
 - sign, [332](#)
- Permute
 - declare.h, [848](#)
 - symmetr.c, [2924](#)
- PERMUTEARG
 - ftypes.h, [1445](#)
- PeRmUtEp, [333](#)

- cycle, [334](#)
- n, [333](#)
- objects, [333](#)
- sign, [333](#)
- PermuteP
 - declare.h, [848](#)
 - symmetr.c, [2924](#)
- PF
 - parallel.c, [2080](#)
 - parallel.h, [2161](#)
- PF_ANY_MSGTAG
 - parallel.h, [2134](#)
- PF_ANY_SOURCE
 - parallel.h, [2134](#)
- PF_Bcast
 - mpi.c, [1773](#)
 - parallel.h, [2135](#)
- PF_Broadcast
 - mpi.c, [1779](#)
 - parallel.h, [2140](#)
- PF_BroadcastBuffer
 - parallel.c, [2070](#)
 - parallel.h, [2151](#)
- PF_BroadcastCBuf
 - parallel.c, [2075](#)
 - parallel.h, [2155](#)
- PF_BroadcastExpFlags
 - parallel.c, [2075](#)
 - parallel.h, [2156](#)
- PF_BroadcastExpr
 - parallel.c, [2076](#)
 - parallel.h, [2157](#)
- PF_BroadcastModifiedDollars
 - parallel.c, [2073](#)
 - parallel.h, [2154](#)
- PF_BroadcastNumber
 - parallel.c, [2069](#)
 - parallel.h, [2150](#)
- PF_BroadcastPreDollar
 - parallel.c, [2071](#)
 - parallel.h, [2152](#)
- PF_BroadcastRedefinedPreVars
 - parallel.c, [2074](#)
 - parallel.h, [2155](#)
- PF_BroadcastRHS
 - parallel.c, [2076](#)
 - parallel.h, [2158](#)
- PF_BroadcastString
 - parallel.c, [2070](#)
 - parallel.h, [2151](#)
- PF_BUFFER, [334](#)
 - active, [336](#)
 - buff, [334](#)
 - fill, [334](#)
 - from, [336](#)
 - full, [335](#)
 - index, [335](#)
 - numbufs, [336](#)
 - request, [335](#)
 - retstat, [335](#)
 - status, [335](#)
 - stop, [335](#)
 - tag, [335](#)
 - type, [335](#)
- PF_BUFFER_MSGTAG
 - parallel.h, [2132](#)
- PF_BYTE
 - parallel.h, [2134](#)
- PF_CollectModifiedDollars
 - parallel.c, [2072](#)
 - parallel.h, [2153](#)
- PF_COMM
 - parallel.h, [2134](#)
- PF_DATA_MSGTAG
 - parallel.h, [2132](#)
- PF_Deferred
 - parallel.c, [2065](#)
 - parallel.h, [2146](#)
- PF_DOLLAR_MSGTAG
 - parallel.h, [2132](#)
- PF_EMPTY_MSGTAG
 - parallel.h, [2133](#)
- PF_ENDBUFFER_MSGTAG
 - parallel.h, [2132](#)
- PF_EndSort
 - parallel.c, [2064](#)
 - parallel.h, [2145](#)
- PF_ENDSORT_MSGTAG
 - parallel.h, [2132](#)
- PF_FlushStdOutBuffer
 - parallel.c, [2080](#)
 - parallel.h, [2161](#)
- PF_FreeErrorMessageBuffers
 - parallel.c, [2080](#)
- PF_GetSlaveTimes
 - parallel.c, [2069](#)
 - parallel.h, [2150](#)
- PF_Init
 - parallel.c, [2067](#)
 - parallel.h, [2148](#)
- PF_InParallelProcessor
 - parallel.c, [2077](#)
 - parallel.h, [2158](#)
- PF_INT
 - parallel.h, [2134](#)
- PF_IRecvRbuf
 - mpi.c, [1773](#)
 - parallel.c, [2062](#)
- PF_ISendSbuf
 - mpi.c, [1772](#)
 - parallel.h, [2135](#)
- PF_LibInit
 - mpi.c, [1770](#)
 - parallel.c, [2060](#)
- PF_LibTerminate
 - mpi.c, [1771](#)

- parallel.c, [2061](#)
- PF_LOG_MSGTAG
 - parallel.h, [2133](#)
- PF_LongMultiBroadcast
 - mpi.c, [1784](#)
 - parallel.h, [2145](#)
- PF_LongMultiPack
 - parallel.h, [2135](#)
- PF_LongMultiPackImpl
 - mpi.c, [1782](#)
 - parallel.h, [2143](#)
- PF_LongMultiUnpack
 - parallel.h, [2135](#)
- PF_LongMultiUnpackImpl
 - mpi.c, [1783](#)
 - parallel.h, [2144](#)
- PF_LongSinglePack
 - mpi.c, [1780](#)
 - parallel.h, [2141](#)
- PF_LongSingleReceive
 - mpi.c, [1781](#)
 - parallel.h, [2142](#)
- PF_LongSingleSend
 - mpi.c, [1781](#)
 - parallel.h, [2142](#)
- PF_LongSingleUnpack
 - mpi.c, [1780](#)
 - parallel.h, [2141](#)
- PF_maxDollarChunkSize
 - mpi.c, [1784](#)
 - parallel.h, [2161](#)
- PF_MISC_MSGTAG
 - parallel.h, [2133](#)
- PF_MLock
 - parallel.c, [2079](#)
 - parallel.h, [2160](#)
- PF_MUnlock
 - parallel.c, [2079](#)
 - parallel.h, [2160](#)
- PF_OPT_COLLECT_MSGTAG
 - parallel.h, [2133](#)
- PF_OPT_HORNER_MSGTAG
 - parallel.h, [2133](#)
- PF_OPT_MCTS_MSGTAG
 - parallel.h, [2133](#)
- PF_Pack
 - mpi.c, [1776](#)
 - parallel.h, [2137](#)
- PF_PACKSIZE
 - mpi.c, [1770](#)
- PF_PackString
 - mpi.c, [1777](#)
 - parallel.h, [2138](#)
- PF_PrepareLongMultiPack
 - mpi.c, [1782](#)
 - parallel.h, [2143](#)
- PF_PrepareLongSinglePack
 - mpi.c, [1779](#)
- parallel.h, [2140](#)
- PF_PreparePack
 - mpi.c, [1776](#)
 - parallel.h, [2137](#)
- PF_PrintPackBuf
 - mpi.c, [1775](#)
- PF_Probe
 - mpi.c, [1771](#)
 - parallel.c, [2061](#)
- PF_Processor
 - parallel.c, [2066](#)
 - parallel.h, [2147](#)
- PF_RawProbe
 - mpi.c, [1775](#)
 - parallel.c, [2064](#)
- PF_RawRecv
 - mpi.c, [1774](#)
 - parallel.c, [2063](#)
 - parallel.h, [2136](#)
- PF_RawSend
 - mpi.c, [1774](#)
 - parallel.c, [2063](#)
 - parallel.h, [2136](#)
- PF_READY_MSGTAG
 - parallel.h, [2132](#)
- PF_RealTime
 - mpi.c, [1770](#)
 - parallel.c, [2060](#)
- PF_Receive
 - mpi.c, [1778](#)
 - parallel.h, [2140](#)
- PF_RecvFile
 - parallel.c, [2078](#)
 - parallel.h, [2159](#)
- PF_RecvWbuf
 - mpi.c, [1772](#)
 - parallel.c, [2061](#)
- PF_RESET
 - parallel.h, [2131](#)
- PF_RestoreInsideInfo
 - parallel.c, [2075](#)
 - parallel.h, [2157](#)
- PF_Send
 - mpi.c, [1778](#)
 - parallel.h, [2139](#)
- PF_SendFile
 - parallel.c, [2077](#)
 - parallel.h, [2158](#)
- PF_SNDFILEBUFSIZE
 - parallel.c, [2059](#)
- PF_STATS_SIZE
 - parallel.c, [2059](#)
- PF_STDOUT_MSGTAG
 - parallel.h, [2133](#)
- PF_StoreInsideInfo
 - parallel.c, [2074](#)
 - parallel.h, [2157](#)
- PF_TERM_MSGTAG

- parallel.h, 2132
- PF_Terminate
 - parallel.c, 2068
 - parallel.h, 2149
- PF_TIME
 - parallel.h, 2131
- PF_Unpack
 - mpi.c, 1776
 - parallel.h, 2138
- PF_UnpackString
 - mpi.c, 1777
 - parallel.h, 2139
- PF_WaitRbuf
 - mpi.c, 1773
 - parallel.c, 2062
- PF_WriteFileToFile
 - parallel.c, 2079
 - parallel.h, 2160
- pfac
 - T_const, 436
- PFORTTRANMODE
 - ftypes.h, 1383
- PGInput, 336
 - cdeg, 337
 - cnum, 337
 - cple, 337
 - ctyp, 337
 - pcode, 337
 - pname, 337
- pgq
 - SGroup, 394
- pgr
 - SGroup, 394
- PHEAD
 - structs.h, 2893
- PHEAD0
 - structs.h, 2893
- PHI
 - ftypes.h, 1401
- pId
 - EGraph, 111
 - MGraph, 212
- PInput, 338
 - acode, 338
 - aname, 338
 - extonly, 339
 - name, 338
 - pcode, 338
 - pypec, 338
 - ptypen, 338
- PIPESTREAM
 - ftypes.h, 1364
- PISYMBOL
 - ftypes.h, 1403
- plist
 - ECand, 95
 - EStack, 120
 - Interaction, 160
- plistic
 - lInput, 149
- plisn
 - lInput, 148
- pname
 - PGInput, 337
 - Stream, 422
- pnclass
 - Assign, 24
 - SProcess, 411
- PNodeClass, 339
 - ~PNodeClass, 340
 - cl2mcl, 342
 - cl2nd, 342
 - cmaxdeg, 341
 - cmindeg, 341
 - count, 342
 - couple, 341
 - deg, 341
 - nclass, 342
 - nd2cl, 342
 - nnodes, 342
 - particle, 341
 - PNodeClass, 340
 - prElem, 340
 - prPNodeClass, 340
 - sproc, 341
 - type, 341
- PObuffer
 - File, 129
- POfill
 - File, 129
- POfull
 - File, 129
- poin2a
 - sOrT, 401
- poina
 - sOrT, 401
- PoinFill
 - sOrT, 400
- PoinScratch
 - N_const, 252
- Pointer
 - CbUf, 71
- pointer
 - Stream, 422
- poly, 344
 - ~poly, 346
 - add, 354
 - all_variables, 350
 - argument_to_poly, 352
 - check_memory, 348
 - coefficient, 351
 - coefficient_list_divmod, 353
 - coefficients_modulo, 351
 - compare_degree_vector, 354
 - degree, 350
 - degree_vector, 354

- derivative, [351](#)
- div, [355](#)
- divides, [356](#)
- divmod, [356](#)
- divmod_heap, [359](#)
- divmod_one_term, [358](#)
- divmod_univar, [358](#)
- expand_memory, [348](#)
- first_variable, [350](#)
- flint::poly, [343](#)
- from_coefficient_list, [353](#)
- get_variables, [352](#)
- integer_lcoeff, [350](#)
- is_dense_univariate, [349](#)
- is_integer, [349](#)
- is_monomial, [349](#)
- is_one, [349](#)
- is_zero, [349](#)
- last_monomial, [354](#)
- last_monomial_index, [354](#)
- lcoeff_multivar, [351](#)
- lcoeff_univar, [351](#)
- mod, [356](#)
- modn, [361](#)
- modp, [361](#)
- monomial_compare, [354](#)
- mul, [355](#)
- mul_heap, [357](#)
- mul_one_term, [357](#)
- mul_univar, [357](#)
- normalize, [353](#)
- number_of_terms, [350](#)
- operator!=, [348](#)
- operator+, [347](#)
- operator+=, [346](#)
- operator-, [347](#)
- operator-=, [346](#)
- operator/, [347](#)
- operator/=, [346](#)
- operator=, [348](#)
- operator==, [347](#)
- operator%, [347](#)
- operator%=: [347](#)
- operator[], [348](#)
- operator*, [347](#)
- operator*=: [346](#)
- poly, [345](#), [346](#)
- poly_to_argument, [352](#)
- poly_to_argument_with_den, [352](#)
- pop_heap, [360](#)
- push_heap, [360](#)
- setmod, [351](#)
- sign, [350](#)
- simple_poly, [351](#), [352](#)
- size_of_form_notation, [353](#)
- size_of_form_notation_with_den, [353](#)
- size_of_terms, [361](#)
- sub, [355](#)
- terms, [361](#)
- termscopy, [348](#)
- to_coefficient_list, [353](#)
- to_string, [354](#)
- total_degree, [350](#)
- poly.cc, [2195](#)
- poly.h, [2228](#)
- poly_determine_modulus
 - polywrap.cc, [2270](#)
- poly_div
 - declare.h, [1002](#)
 - polywrap.cc, [2271](#)
- poly_divmod
 - polywrap.cc, [2271](#)
- poly_factor
 - flint::poly_factor, [362](#)
- poly_factorize
 - polywrap.cc, [2275](#)
- poly_factorize_argument
 - declare.h, [1005](#)
 - polywrap.cc, [2276](#)
- poly_factorize_dollar
 - declare.h, [1006](#)
 - polywrap.cc, [2277](#)
- poly_factorize_expression
 - declare.h, [1006](#)
 - polywrap.cc, [2277](#)
- poly_fix_minus_signs
 - polywrap.cc, [2275](#)
- poly_free_poly_vars
 - declare.h, [1008](#)
 - polywrap.cc, [2280](#)
- poly_gcd
 - declare.h, [1001](#)
 - polywrap.cc, [2270](#)
- poly_inverse
 - declare.h, [1003](#)
 - polywrap.cc, [2279](#)
- poly_mul
 - declare.h, [1003](#)
 - polywrap.cc, [2280](#)
- poly_num_vars
 - N_const, [261](#)
- poly_ratfun_add
 - declare.h, [1003](#)
 - polywrap.cc, [2273](#)
- poly_ratfun_normalize
 - declare.h, [1004](#)
 - polywrap.cc, [2274](#)
- poly_ratfun_read
 - polywrap.cc, [2271](#)
- poly_rem
 - declare.h, [1002](#)
 - polywrap.cc, [2271](#)
- poly_sort
 - polywrap.cc, [2272](#)
- poly_to_argument
 - poly, [352](#)

- poly_to_argument_with_den
 - poly, [352](#)
- poly_unfactorize_expression
 - declare.h, [1007](#)
 - polywrap.cc, [2278](#)
- poly_vars
 - N_const, [254](#)
- poly_vars_type
 - N_const, [261](#)
- PolyAct
 - T_const, [443](#)
- POLYADD
 - ftypes.h, [1440](#)
- POLYDIV
 - ftypes.h, [1440](#)
- PolyDiv
 - declare.h, [998](#)
 - ratio.c, [2501](#)
- polyfact.cc, [2231](#), [2232](#)
 - operator<<, [2232](#)
- polyfact.h, [2248](#)
- PolyFlag
 - sOrT, [404](#)
- POLYFUN
 - ftypes.h, [1367](#)
- PolyFun
 - R_const, [380](#)
- PolyFunClean
 - declare.h, [842](#)
 - function.c, [1471](#)
- PolyFunDirty
 - declare.h, [842](#)
 - function.c, [1471](#)
- PolyFunExp
 - R_const, [381](#)
- PolyFunInv
 - R_const, [380](#)
- PolyFunMul
 - declare.h, [848](#)
 - proces.c, [2430](#)
- PolyFunPow
 - R_const, [381](#)
- PolyFunTodo
 - N_const, [260](#)
- PolyFunType
 - R_const, [381](#)
- PolyFunVar
 - R_const, [381](#)
- POLYGCD
 - ftypes.h, [1440](#)
- polygcd.cc, [2249](#), [2251](#)
 - gcd_heuristic_possible, [2250](#)
 - gcd_linear_helper, [2251](#)
- polygcd.h, [2267](#)
- POLYINTFAC
 - ftypes.h, [1440](#)
- POLYMOD, [362](#)
 - arraysize, [363](#)
 - coefs, [363](#)
 - modnum, [363](#)
 - numsym, [363](#)
 - polysize, [363](#)
- POLYMUL
 - ftypes.h, [1440](#)
- POLYNORM
 - ftypes.h, [1441](#)
- PolyNormFlag
 - N_const, [259](#)
- PolyRatFunChanged
 - C_const, [69](#)
- PolyRatFunSpecial
 - declare.h, [962](#)
 - sort.c, [2724](#)
- POLYREM
 - ftypes.h, [1440](#)
- polysize
 - POLYMOD, [363](#)
- polysortflag
 - N_const, [257](#)
- POLYSUB
 - ftypes.h, [1440](#)
- PolyWise
 - sOrT, [404](#)
- polywrap.cc, [2268](#), [2280](#)
 - poly_determine_modulus, [2270](#)
 - poly_div, [2271](#)
 - poly_divmod, [2271](#)
 - poly_factorize, [2275](#)
 - poly_factorize_argument, [2276](#)
 - poly_factorize_dollar, [2277](#)
 - poly_factorize_expression, [2277](#)
 - poly_fix_minus_signs, [2275](#)
 - poly_free_poly_vars, [2280](#)
 - poly_gcd, [2270](#)
 - poly_inverse, [2279](#)
 - poly_mul, [2280](#)
 - poly_ratfun_add, [2273](#)
 - poly_ratfun_normalize, [2274](#)
 - poly_ratfun_read, [2271](#)
 - poly_rem, [2271](#)
 - poly_sort, [2272](#)
 - poly_unfactorize_expression, [2278](#)
 - POLYWRAP_DENOMPOWER_INCREASE_FACTOR, [2280](#)
 - POLYWRAP_DENOMPOWER_INCREASE_FACTOR
 - polywrap.cc, [2280](#)
- pop_heap
 - poly, [360](#)
- POposition
 - FiLe, [129](#)
- POPPREASSIGNLEVEL
 - declare.h, [806](#)
- PopPreVars
 - declare.h, [893](#)
 - pre.c, [2313](#)
- PopVariables

- declare.h, [849](#)
- execute.c, [1152](#)
- portsignals.h, [2302](#), [2306](#)
 - FATAL_SIG_ERROR, [2303](#)
 - NSIG, [2303](#)
 - SIGALRM, [2305](#)
 - SIGEMT, [2304](#)
 - SIGFPE, [2303](#)
 - SIGHUP, [2305](#)
 - SIGILL, [2304](#)
 - SIGINT, [2305](#)
 - SIGLOST, [2304](#)
 - SIGPIPE, [2304](#)
 - SIGPROF, [2305](#)
 - SIGQUIT, [2305](#)
 - SIGSEGV, [2303](#)
 - SIGSYS, [2304](#)
 - SIGTERM, [2304](#)
 - SIGVTALRM, [2305](#)
 - SIGXCPU, [2304](#)
 - SIGXFSZ, [2304](#)
- POS0_
 - ftypes.h, [1425](#)
- POS_
 - ftypes.h, [1425](#)
- PoSiTiOn, [363](#)
 - p1, [364](#)
- Position
 - FiLeDaTa, [132](#)
- position
 - indexblock, [151](#)
 - InDeXeNtRy, [153](#)
 - nameblock, [262](#)
 - objects, [290](#)
 - SpecTatoR, [406](#)
 - StOrEcAcHe, [416](#)
- PositionStream
 - declare.h, [876](#)
 - tools.c, [3073](#)
- POSITIVEONLY
 - ftypes.h, [1442](#)
- POsize
 - FiLe, [130](#)
- POSNEG
 - ftypes.h, [1442](#)
- POstop
 - FiLe, [129](#)
- posWorkPointer
 - T_const, [438](#)
- posWorkSize
 - R_const, [378](#)
- posWorkSpace
 - T_const, [434](#)
- PotModdollars
 - variable.h, [3194](#)
- PotModDoIList
 - C_const, [44](#)
- pow
 - COST, [76](#)
- power
 - factorized_poly, [126](#)
- powerFlag
 - O_const, [287](#)
- powmod
 - C_const, [48](#)
- pParseObject
 - declare.h, [906](#)
 - pre.c, [2323](#)
- prCand
 - Assign, [17](#)
- pre.c, [2307](#), [2329](#)
 - AddToPreTypes, [2324](#)
 - CharOut, [2312](#)
 - ClearMacro, [2315](#)
 - ClearPushback, [2311](#)
 - ConstructName, [2327](#)
 - DAEMON, [2310](#)
 - defineChannel, [2325](#)
 - DoBreakDo, [2317](#)
 - DoCall, [2316](#)
 - DoClearOptimize, [2326](#)
 - DoClearUserFlag, [2328](#)
 - DoCommentChar, [2315](#)
 - DoContinueDo, [2317](#)
 - DoDebug, [2317](#)
 - DoDefine, [2315](#)
 - DoDo, [2317](#)
 - DoElse, [2317](#)
 - DoElseif, [2318](#)
 - DoEnddo, [2318](#)
 - DoEndif, [2318](#)
 - DoEndInside, [2319](#)
 - DoEndNamespace, [2327](#)
 - DoEndprocedure, [2318](#)
 - DoExternal, [2324](#)
 - DoFactDollar, [2325](#)
 - DoFromExternal, [2325](#)
 - Dolf, [2318](#)
 - Dolfdef, [2318](#)
 - Dolfndef, [2319](#)
 - Dolfydef, [2318](#)
 - DoInclude, [2316](#)
 - DoInside, [2319](#)
 - DoMessage, [2319](#)
 - DoNamespace, [2327](#)
 - DoOptimize, [2326](#)
 - DoPipe, [2319](#)
 - DoPrcExtension, [2319](#)
 - DoPreAddSeparator, [2324](#)
 - DoPreAppend, [2320](#)
 - DoPreAppendPath, [2326](#)
 - DoPreAssign, [2315](#)
 - DoPreBreak, [2321](#)
 - DoPreCase, [2321](#)
 - DoPreClose, [2320](#)
 - DoPreCreate, [2320](#)

- DoPreDefault, [2321](#)
- DoPreEndSwitch, [2321](#)
- DoPreExchange, [2316](#)
- DoPreOut, [2319](#)
- DoPrePrependPath, [2327](#)
- DoPrePrintTimes, [2320](#)
- DoPreRemove, [2320](#)
- DoPreReset, [2326](#)
- DoPreRmSeparator, [2324](#)
- DoPreShow, [2321](#)
- DoPreSortReallocate, [2320](#)
- DoPreSwitch, [2321](#)
- DoPreWrite, [2320](#)
- DoProcedure, [2321](#)
- DoPrompt, [2324](#)
- DoRedefine, [2315](#)
- DoReverseInclude, [2316](#)
- DoRmExternal, [2325](#)
- DoSetExternal, [2324](#)
- DoSetExternalAttr, [2325](#)
- DoSetRandom, [2326](#)
- DoSetUserFlag, [2328](#)
- DoSkipExtraSymbols, [2326](#)
- DoSystem, [2322](#)
- DoTerminate, [2317](#)
- DoTimeOutAfter, [2327](#)
- DoToExternal, [2325](#)
- DoUndefine, [2316](#)
- DoUse, [2328](#)
- EvalPrelf, [2322](#)
- ExpandTripleDots, [2314](#)
- FALSE_EXPR, [2311](#)
- FindInKeyWord, [2314](#)
- FindKeyWord, [2314](#)
- GetChar, [2311](#)
- GetDollarNumber, [2326](#)
- GetInput, [2311](#)
- GetPreVar, [2312](#)
- Include, [2316](#)
- IniModule, [2313](#)
- IniSpecialModule, [2313](#)
- KILL, [2310](#)
- KILLALL, [2310](#)
- LoadInstruction, [2313](#)
- LoadStatement, [2313](#)
- MessPreNesting, [2324](#)
- NOSHELL, [2311](#)
- PopPreVars, [2313](#)
- pParseObject, [2323](#)
- PreCalc, [2323](#)
- PreCmp, [2323](#)
- PreEq, [2323](#)
- PreEval, [2323](#)
- PrelfEval, [2322](#)
- PreLoad, [2322](#)
- PreProcessor, [2313](#)
- PreProInstruction, [2313](#)
- PreSkip, [2322](#)
- PutPreVar, [2312](#)
- SHELL, [2310](#)
- SKIPBUFSIZE, [2310](#)
- SkipName, [2327](#)
- StartPrepro, [2322](#)
- STDERR, [2310](#)
- STRINGIFY, [2310](#)
- STRINGIFY__, [2310](#)
- TERMINAL, [2311](#)
- TheDefine, [2314](#)
- TheUndefine, [2316](#)
- TRUE_EXPR, [2311](#)
- UnsetAllowDelay, [2312](#)
- UserFlags, [2328](#)
- writeToChannel, [2325](#)
- PreAssignFlag
 - P_const, [319](#)
- PreAssignLevel
 - P_const, [321](#)
- PreAssignStack
 - P_const, [318](#)
- PreCalc
 - declare.h, [905](#)
 - pre.c, [2323](#)
- PRECALCSTREAM
 - ftypes.h, [1364](#)
- prECand
 - ECand, [95](#)
- PreCmp
 - declare.h, [905](#)
 - pre.c, [2323](#)
- PreContinuation
 - P_const, [319](#)
- PreDebug
 - P_const, [321](#)
- prEdges
 - MConn, [199](#)
- PreEq
 - declare.h, [905](#)
 - pre.c, [2323](#)
- preError
 - P_const, [322](#)
- PreEval
 - declare.h, [905](#)
 - pre.c, [2323](#)
- preFill
 - P_const, [318](#)
- PrelfDollarEval
 - declare.h, [967](#)
 - dollar.c, [1087](#)
- PrelfEval
 - declare.h, [906](#)
 - pre.c, [2322](#)
- PrelfLevel
 - DoLoOp, [92](#)
 - P_const, [320](#)
- PrelfStack
 - P_const, [318](#)

- PreInsideLevel
 - P_const, [320](#)
- prElem
 - PNodeClass, [340](#)
- PRELOAD, [364](#)
 - buffer, [364](#)
 - size, [364](#)
- PreLoad
 - declare.h, [906](#)
 - pre.c, [2322](#)
- PRELOWERAFTER
 - ftypes.h, [1368](#)
- PRENOACTION
 - ftypes.h, [1368](#)
- PreOut
 - P_const, [319](#)
- PrepPoly
 - declare.h, [849](#)
 - proces.c, [2429](#)
- PreProcessor
 - declare.h, [887](#)
 - pre.c, [2313](#)
- PreproFlag
 - P_const, [319](#)
- PreProInstruction
 - declare.h, [894](#)
 - pre.c, [2313](#)
- PREPROONLY
 - ftypes.h, [1369](#)
- PRERAISEAFTER
 - ftypes.h, [1368](#)
- PreRandom
 - declare.h, [982](#)
 - reken.c, [2558](#)
- PREREADSTREAM
 - ftypes.h, [1364](#)
- PREREADSTREAM2
 - ftypes.h, [1364](#)
- PREREADSTREAM3
 - ftypes.h, [1365](#)
- PreSkip
 - declare.h, [906](#)
 - pre.c, [2322](#)
- preStart
 - P_const, [317](#)
- preStop
 - P_const, [317](#)
- PreSwitchLevel
 - DoLoOp, [93](#)
 - P_const, [319](#)
- PreSwitchModes
 - P_const, [318](#)
- PreSwitchStrings
 - P_const, [317](#)
- PRETYPEDO
 - ftypes.h, [1372](#)
- PRETYPEIF
 - ftypes.h, [1372](#)
- PRETYPEINSIDE
 - ftypes.h, [1372](#)
- PRETYPENONE
 - ftypes.h, [1371](#)
- PRETYPEPROCEDURE
 - ftypes.h, [1372](#)
- PreTypes
 - P_const, [318](#)
- PRETYPESWITCH
 - ftypes.h, [1372](#)
- PREV
 - declare.h, [798](#)
- PREVAR
 - structs.h, [2896](#)
- PreVar
 - variable.h, [3189](#)
- pReVaR, [365](#)
 - argnames, [365](#)
 - name, [365](#)
 - nargs, [365](#)
 - value, [365](#)
 - wildarg, [366](#)
- PreVarList
 - P_const, [316](#)
- prevars
 - StreaM, [423](#)
- PREVARSTREAM
 - ftypes.h, [1364](#)
- previous
 - NaMeSpAcE, [265](#)
 - StreaM, [423](#)
- previousblock
 - indexblock, [151](#)
 - nameblock, [262](#)
- previousEfactor
 - T_const, [436](#)
- previousNoShowInput
 - StreaM, [424](#)
- prevline
 - StreaM, [421](#)
- prFLines
 - EGraph, [108](#)
- primelist
 - T_const, [436](#)
- PRIMENUMBER
 - ftypes.h, [1399](#)
- print
 - EEdge, [97](#)
 - EFLine, [101](#)
 - EGraph, [104](#)
 - ENode, [117](#)
 - EStack, [119](#)
 - flint::fmpz, [138](#)
 - flint::mpoly, [242](#)
 - flint::poly, [343](#)
 - Fraction, [139](#)
 - MConn, [199](#)
 - MGraph, [209](#)

- MOrbits, [240](#)
 - NStack, [278](#)
 - Options, [302](#)
 - SGroup, [392](#)
- print_instr
 - optimize.cc, [1983](#)
- print_tree
 - optimize.cc, [1989](#)
- printAdjMat
 - MGraph, [209](#)
- PRINTALL
 - ftypes.h, [1424](#)
- PrintBacktraceFlag
 - C_const, [62](#)
- PRINTCONTENT
 - ftypes.h, [1423](#)
- PRINTCONTENTS
 - ftypes.h, [1423](#)
- PrintDeprecation
 - declare.h, [852](#)
 - startup.c, [2795](#)
- prInteraction
 - Interaction, [160](#)
- PrintExtraSymbol
 - declare.h, [995](#)
 - notation.c, [1934](#)
- PRINTFBUF
 - parallel.c, [2057](#)
- printflag
 - ExPrEsSiOn, [123](#)
- PrintFloat
 - float.c, [1289](#)
- printInput
 - Model, [234](#)
- printLevel
 - Options, [302](#)
- PRINTLFILE
 - ftypes.h, [1424](#)
- printMat
 - MNodeClass, [223](#)
- printMInput
 - Model, [233](#)
- printModel
 - Options, [302](#)
- PRINTOFF
 - ftypes.h, [1423](#)
- PRINTON
 - ftypes.h, [1423](#)
- PRINTONEFUNCTION
 - ftypes.h, [1424](#)
- PRINTONETERM
 - ftypes.h, [1424](#)
- printPInput
 - Model, [234](#)
- printPy
 - EGraph, [104](#)
 - MGraph, [209](#)
- PrintRunningTime
 - declare.h, [852](#)
 - startup.c, [2795](#)
- printscratch
 - proces.c, [2431](#)
- PrintSeq
 - declare.h, [911](#)
 - message.c, [1707](#)
- printstats
 - OPTIMIZE, [297](#)
- PrintSubTerm
 - declare.h, [911](#)
 - message.c, [1707](#)
- PrintSubtermList
 - declare.h, [995](#)
 - notation.c, [1934](#)
- PrintTerm
 - declare.h, [910](#)
 - message.c, [1707](#)
- PrintTermC
 - declare.h, [910](#)
 - message.c, [1707](#)
- PrintTime
 - declare.h, [969](#)
 - sch.c, [2648](#)
- PrintTotalSize
 - M_const, [182](#)
- PrintType
 - O_const, [288](#)
- PrintWords
 - declare.h, [911](#)
 - message.c, [1707](#)
- prModel
 - Model, [231](#)
- prNCand
 - NCand, [269](#)
- proc
 - Assign, [24](#)
 - EGraph, [109](#)
 - Options, [305](#)
 - Output, [313](#)
 - SProcess, [411](#)
- PROCEDURE, [366](#)
 - loadmode, [367](#)
 - mustfree, [367](#)
 - name, [367](#)
 - p, [367](#)
- procedureExtension
 - P_const, [318](#)
- procedureextension
 - setfile.c, [2688](#)
- PROCEDUREFILE
 - ftypes.h, [1365](#)
- Procedures
 - variable.h, [3188](#)
- proces.c, [2419](#), [2431](#)
 - Deferred, [2428](#)
 - DONE, [2420](#)
 - DoOnePow, [2427](#)

- FinTerm, [2425](#)
 - Generator, [2425](#)
 - InFunction, [2421](#)
 - InsertTerm, [2422](#)
 - PasteFile, [2424](#)
 - PasteTerm, [2424](#)
 - PolyFunMul, [2430](#)
 - PrepPoly, [2429](#)
 - printscratch, [2431](#)
 - Processor, [2420](#)
 - TestSub, [2421](#)
- Process, [367](#)
 - ~Process, [369](#)
 - agrcount, [370](#)
 - astack, [372](#)
 - clist, [371](#)
 - ctotal, [371](#)
 - finalPart, [370](#)
 - id, [370](#)
 - initlPart, [370](#)
 - loop, [371](#)
 - maxnlegs, [371](#)
 - mgrcount, [370](#)
 - mkSProcess, [369](#)
 - model, [370](#)
 - nAGraphs, [373](#)
 - nAOPI, [373](#)
 - nExtern, [371](#)
 - nfinal, [371](#)
 - ngraphs, [372](#)
 - ninitl, [371](#)
 - nMGraphs, [372](#)
 - nMOPI, [373](#)
 - nopi, [372](#)
 - nSubproc, [371](#)
 - opt, [370](#)
 - outProcP, [369](#)
 - Process, [369](#)
 - prProcess, [369](#)
 - prProcessP, [369](#)
 - sec, [373](#)
 - sproc, [372](#)
 - sptbl, [372](#)
 - wAGraphs, [373](#)
 - wAOPI, [373](#)
 - wgraphs, [372](#)
 - wMGraphs, [373](#)
 - wMOPI, [373](#)
 - wopi, [372](#)
- ProcessBucketSize
 - C_const, [53](#)
- ProcessOption
 - declare.h, [876](#)
 - setfile.c, [2685](#)
- Processor
 - declare.h, [850](#)
 - proces.c, [2420](#)
- ProcessStats
 - C_const, [59](#)
- ProcessTopology
 - diawrap.cc, [1052](#)
- proclid
 - Output, [314](#)
- ProcList
 - P_const, [317](#)
- Product
 - declare.h, [851](#)
 - reken.c, [2553](#)
- PRODUCTIONDATE
 - form3.h, [1326](#)
- proexp
 - T_const, [442](#)
- PROPERORDERFLAG
 - ftypes.h, [1439](#)
- properorderflag
 - C_const, [60](#)
- ProtoType
 - C_const, [48](#)
- prototype
 - ExPrEsSiOn, [121](#)
 - TaBIEs, [449](#)
- prototypeSize
 - TaBIEs, [451](#)
- prParticle
 - Particle, [329](#)
- prParticleArray
 - Model, [233](#)
- prPNodeClass
 - PNodeClass, [340](#)
- prProcess
 - Process, [369](#)
- prProcessP
 - Process, [369](#)
- prSProcess
 - SProcess, [409](#)
- prStack
 - AStack, [28](#)
- PrtLong
 - declare.h, [851](#)
 - reken.c, [2554](#)
- PrtTerms
 - declare.h, [851](#)
 - sch.c, [2648](#)
- PruneExtraSymbols
 - declare.h, [997](#)
 - notation.c, [1935](#)
- pSize
 - P_const, [319](#)
- psize
 - PaRtl, [326](#)
- psq
 - SGroup, [394](#)
- psr
 - SGroup, [394](#)
- pStop
 - sOrT, [401](#)

- ptcl
 - AEdge, [8](#)
 - EEdge, [98](#)
 - NCInput, [271](#)
- ptype
 - Particle, [331](#)
- ptypespec
 - PInput, [338](#)
- ptypen
 - PInput, [338](#)
- push_heap
 - poly, [360](#)
- pushEdge
 - AStack, [28](#)
 - MConn, [198](#)
- pushNode
 - AStack, [28](#)
 - MConn, [198](#)
- PUSHPREASSIGNLEVEL
 - declare.h, [806](#)
- PutArgInScratch
 - declare.h, [991](#)
 - transform.c, [3138](#)
- PutBracket
 - declare.h, [852](#)
 - execute.c, [1153](#)
- PutBracketInIndex
 - declare.h, [969](#)
 - index.c, [1663](#)
- PutExtraSymbols
 - declare.h, [999](#)
 - ratio.c, [2499](#)
- PUTFIRST
 - ftypes.h, [1399](#)
- PutIn
 - declare.h, [852](#)
 - sort.c, [2713](#)
- PutInBuffers
 - pattern.c, [2166](#)
- PutInside
 - declare.h, [847](#)
 - index.c, [1664](#)
- putsizes
 - sOrT, [403](#)
- PutInSpectator
 - declare.h, [1013](#)
 - spectator.c, [2782](#)
- PutInStore
 - declare.h, [853](#)
 - store.c, [2824](#)
- PutInVflags
 - declare.h, [865](#)
 - execute.c, [1153](#)
- PutOut
 - declare.h, [853](#)
 - sort.c, [2714](#)
- PutPreVar
 - declare.h, [874](#)
- pre.c, [2312](#)
- PutTableNames
 - declare.h, [976](#)
 - minos.c, [1724](#)
- PutTermInDollar
 - declare.h, [951](#)
 - dollar.c, [1085](#)
- PUTZERO
 - declare.h, [801](#)
- PVFUNV
 - ftypes.h, [1457](#)
- PVFUNWP
 - ftypes.h, [1457](#)
- pWorkPointer
 - T_const, [438](#)
- pWorkSize
 - R_const, [378](#)
- pWorkSpace
 - T_const, [434](#)
- Q_
 - ftypes.h, [1426](#)
- qError
 - M_const, [186](#)
- qqopt
 - Options, [306](#)
- qqref
 - Options, [306](#)
- QUADRUPLEFORTRANMODE
 - ftypes.h, [1384](#)
- QUADRUPLEPRECISIONFLAG
 - ftypes.h, [1384](#)
- Quotient
 - declare.h, [854](#)
 - reken.c, [2553](#)
- R
 - AllGlobals, [10](#)
- r
 - node, [276](#)
- R_const, [374](#)
 - BracketOn, [379](#)
 - Cnumlhs, [378](#)
 - CompareRoutine, [377](#)
 - CompressBuffer, [377](#)
 - CompressPointer, [377](#)
 - ComprTop, [377](#)
 - CurDum, [380](#)
 - CurExpr, [382](#)
 - DeferFlag, [380](#)
 - DefPosition, [376](#)
 - Eside, [381](#)
 - expchanged, [381](#)
 - expflags, [382](#)
 - FoStage4, [376](#)
 - Fscr, [376](#)
 - funoffset, [382](#)
 - GetFile, [379](#)
 - GetOneFile, [380](#)

- gzipCompress, [378](#)
- hidefile, [376](#)
- infile, [376](#)
- InHiBuf, [378](#)
- InLnBuf, [377](#)
- KeptInHold, [379](#)
- level, [381](#)
- IWorkSize, [378](#)
- MaxBracket, [379](#)
- MaxDum, [381](#)
- modeloptions, [382](#)
- moebiustable, [377](#)
- moebiustablesize, [382](#)
- NoCompress, [378](#)
- OldTime, [377](#)
- outfile, [376](#)
- outtohide, [379](#)
- PolyFun, [380](#)
- PolyFunExp, [381](#)
- PolyFunInv, [380](#)
- PolyFunPow, [381](#)
- PolyFunType, [381](#)
- PolyFunVar, [381](#)
- posWorkSize, [378](#)
- pWorkSize, [378](#)
- ShortSortCount, [382](#)
- sLevel, [380](#)
- SortType, [382](#)
- Stage4Name, [380](#)
- StoreData, [376](#)
- TePos, [380](#)
- wranfcall, [379](#)
- wranfia, [377](#)
- wranfnpair1, [379](#)
- wranfnpair2, [379](#)
- wranfseed, [378](#)
- R_COPY_B
 - checkpoint.c, [527](#)
- R_COPY_LIST
 - checkpoint.c, [528](#)
- R_COPY_NAMETREE
 - checkpoint.c, [529](#)
- R_COPY_S
 - checkpoint.c, [528](#)
- R_FREE
 - checkpoint.c, [526](#)
- R_FREE_NAMETREE
 - checkpoint.c, [527](#)
- R_FREE_STREAM
 - checkpoint.c, [527](#)
- R_SET
 - checkpoint.c, [527](#)
- RaisPow
 - declare.h, [854](#)
 - reken.c, [2551](#)
- RaisPowCached
 - declare.h, [854](#)
 - reken.c, [2551](#)
- RaisPowMod
 - declare.h, [855](#)
 - reken.c, [2552](#)
- RANDOMFUNCTION
 - ftypes.h, [1398](#)
- ranges
 - DICTIONARY, [83](#)
- RANPERM
 - ftypes.h, [1398](#)
- ratfun_add_mpoly
 - flintinterface.h, [1269](#)
- ratfun_add_poly
 - flintinterface.h, [1269](#)
- ratfun_normalize_mpoly
 - flintinterface.h, [1269](#)
- ratfun_normalize_poly
 - flintinterface.h, [1269](#)
- ratfun_read_mpoly
 - flintinterface.h, [1270](#)
- ratfun_read_poly
 - flintinterface.h, [1270](#)
- RATIO
 - ftypes.h, [1414](#)
- ratio
 - Fraction, [141](#)
- ratio.c, [2496](#), [2503](#)
 - BinomGen, [2497](#)
 - ConvertArgument, [2502](#)
 - CreateExpression, [2500](#)
 - DIVfunction, [2502](#)
 - divrem, [2503](#)
 - DoSumF1, [2498](#)
 - DoSumF2, [2498](#)
 - ExpandRat, [2502](#)
 - FromPolyRatFun, [2501](#)
 - GCDclean, [2501](#)
 - GCDfunction, [2498](#)
 - GCDfunction3, [2498](#)
 - GCDterms, [2500](#)
 - Glue, [2498](#)
 - InvPoly, [2502](#)
 - MergeDotproductLists, [2500](#)
 - MergeSymbolLists, [2500](#)
 - MULfunc, [2502](#)
 - MultiplyWithTerm, [2499](#)
 - PolyDiv, [2501](#)
 - PutExtraSymbols, [2499](#)
 - RatioFind, [2497](#)
 - RatioGen, [2497](#)
 - ReadPolyRatFun, [2500](#)
 - TakeContent, [2499](#)
 - TakeExtraSymbols, [2499](#)
 - TakeSymbolContent, [2501](#)
 - TheErrorMessage, [2503](#)
- RatioFind
 - declare.h, [856](#)
 - ratio.c, [2497](#)
- RatioGen

- declare.h, [856](#)
- ratio.c, [2497](#)
- RATIONALMODE
 - ftypes.h, [1429](#)
- RatToFloat
 - evaluate.c, [1140](#)
 - float.c, [1288](#)
- RatToFloatr
 - evaluate.c, [1140](#)
- RatToLine
 - declare.h, [856](#)
 - sch.c, [2647](#)
- RBRACE
 - ftypes.h, [1379](#)
- rbufnum
 - M_const, [178](#)
- rbufs
 - ParallelVars, [324](#)
- RCYCLESYMMETRIC
 - ftypes.h, [1431](#)
- ReadFile
 - declare.h, [884](#)
 - tools.c, [3075](#)
- ReadFloat
 - float.c, [1288](#)
- ReadFromScratch
 - declare.h, [1014](#)
 - store.c, [2823](#)
- ReadFromStream
 - declare.h, [875](#)
 - tools.c, [3072](#)
- ReadijObject
 - declare.h, [975](#)
 - minos.c, [1725](#)
- ReadIndex
 - declare.h, [973](#)
 - minos.c, [1721](#)
- ReadIniInfo
 - declare.h, [974](#)
 - minos.c, [1722](#)
- ReadObject
 - declare.h, [975](#)
 - minos.c, [1724](#)
- ReadParticle
 - declare.h, [1017](#)
 - model.c, [1748](#)
- ReadPolyRatFun
 - declare.h, [997](#)
 - ratio.c, [2500](#)
- readpos
 - SpecTatoR, [406](#)
- ReadPosFile
 - declare.h, [884](#)
 - tools.c, [3075](#)
- ReadRange
 - declare.h, [991](#)
 - transform.c, [3138](#)
- ReadSaveExpression
 - declare.h, [983](#)
 - store.c, [2831](#)
- ReadSaveHeader
 - declare.h, [982](#)
 - store.c, [2828](#)
- ReadSaveIndex
 - declare.h, [982](#)
 - store.c, [2829](#)
- ReadSaveTerm32
 - declare.h, [984](#)
 - store.c, [2830](#)
- ReadSaveVariables
 - declare.h, [985](#)
 - store.c, [2829](#)
- ReadSnum
 - declare.h, [857](#)
 - tools.c, [3081](#)
- READSPECTATORFLAG
 - ftypes.h, [1453](#)
- RecalcSetups
 - declare.h, [891](#)
 - setfile.c, [2685](#)
- RecFlag
 - T_const, [443](#)
- ReConvertParticle
 - diawrap.cc, [1051](#)
- RecoveryFilename
 - checkpoint.c, [530](#)
 - declare.h, [987](#)
- recvBuffer
 - parallel.c, [2060](#)
- recycle_variables
 - optimize.cc, [1995](#)
- REDEFINEDEXPRESSION
 - ftypes.h, [1424](#)
- REDLENG
 - declare.h, [789](#)
- RedoTableTree
 - declare.h, [961](#)
 - tables.c, [2954](#)
- RedoTree
 - comtool.c, [756](#)
 - declare.h, [950](#)
- REDUCEMODE
 - ftypes.h, [1383](#)
- REDUCESUBEXPBUFFERS
 - compiler.c, [712](#)
- refineClass
 - MGraph, [210](#)
- REGULAR
 - ftypes.h, [1439](#)
- REGULAREXPRESSION
 - ftypes.h, [1424](#)
- REGULARSYMBOL
 - ftypes.h, [1448](#)
- reken.c, [2545](#), [2558](#)
 - AccumGCD, [2549](#)
 - AddLong, [2550](#)

- AddPLon, [2550](#)
- AddRat, [2548](#)
- Bernoulli, [2557](#)
- BigLong, [2550](#)
- CompCoef, [2555](#)
- COPYLONG, [2547](#)
- DivLong, [2551](#)
- DivRat, [2549](#)
- Divvy, [2548](#)
- Factorial, [2556](#)
- GcdLong, [2554](#)
- GCDMAX, [2547](#)
- GetBinom, [2554](#)
- GetLong, [2554](#)
- GetLongModInverses, [2553](#)
- GetModInverses, [2552](#)
- iniwranf, [2557](#)
- iranf, [2558](#)
- LcmLong, [2555](#)
- MakeInverses, [2552](#)
- MakeLongRational, [2555](#)
- MakeModTable, [2556](#)
- MakeRational, [2555](#)
- Modulus, [2556](#)
- Moebius, [2557](#)
- MulLong, [2550](#)
- Mully, [2548](#)
- MulRat, [2549](#)
- NEWTRICK, [2547](#)
- NextPrime, [2557](#)
- NormalModulus, [2552](#)
- Pack, [2548](#)
- PreRandom, [2558](#)
- Product, [2553](#)
- PrtLong, [2554](#)
- Quotient, [2553](#)
- RaisPow, [2551](#)
- RaisPowCached, [2551](#)
- RaisPowMod, [2552](#)
- Remain10, [2553](#)
- Remain4, [2554](#)
- Simplify, [2549](#)
- SubPLon, [2550](#)
- TakeLongRoot, [2555](#)
- TakeModulus, [2556](#)
- TakeNormalModulus, [2556](#)
- TakeRatRoot, [2549](#)
- UnPack, [2548](#)
- WARMUP, [2547](#)
- wranf, [2557](#)
- ReleaseTB
 - declare.h, [980](#)
 - tables.c, [2958](#)
- Remain10
 - declare.h, [857](#)
 - reken.c, [2553](#)
- Remain4
 - declare.h, [857](#)
- reken.c, [2554](#)
- REMFUNCTION
 - ftypes.h, [1395](#)
- RemoveBridges
 - lus.c, [1680](#)
- RemoveDictionary
 - declare.h, [1010](#)
 - dict.c, [1068](#)
- RemoveDollars
 - declare.h, [972](#)
 - names.c, [1828](#)
- renum
 - ExPrEsSiOn, [121](#)
- RENUMBER
 - structs.h, [2894](#)
- ReNuMbEr, [383](#)
 - func, [384](#)
 - funnum, [385](#)
 - indi, [384](#)
 - indnum, [385](#)
 - startposition, [384](#)
 - symp, [384](#)
 - symnum, [384](#)
 - vecnum, [385](#)
 - vect, [384](#)
- ReNumber
 - declare.h, [857](#)
 - reshuf.c, [2605](#)
- ReNUMBERVec
 - O_const, [284](#)
- renumlists
 - ExPrEsSiOn, [122](#)
- ReNumScratch
 - N_const, [252](#)
- ReOpenFile
 - declare.h, [883](#)
 - tools.c, [3074](#)
- reorder
 - MNodeClass, [224](#)
- reordLeg
 - Assign, [19](#)
- RepCount
 - T_const, [435](#)
- RepFunList
 - N_const, [249](#)
- RepFunNum
 - N_const, [258](#)
- replace
 - ExPrEsSiOn, [123](#)
- REPLACEARG
 - ftypes.h, [1444](#)
- ReplaceDollar
 - declare.h, [880](#)
 - names.c, [1826](#)
- REPLACELOOP
 - ftypes.h, [1439](#)
- REPLACEMENT
 - ftypes.h, [1390](#)

- ReplaceScrat
 - N_const, 251
- RepLevel
 - C_const, 64
- RepMax
 - M_const, 186
- RepPoint
 - N_const, 248
- RepSumCheck
 - C_const, 64
- RepTop
 - T_const, 435
- request
 - PF_BUFFER, 335
- reserved
 - STOREHEADER, 420
 - StreaM, 424
 - TaBIEs, 451
- ReserveTempFiles
 - declare.h, 910
 - startup.c, 2793
- ResetScratch
 - declare.h, 857
 - store.c, 2823
- resetTimeOnClear
 - M_const, 181
- ResetVariables
 - declare.h, 913
 - names.c, 1827
- reshuf.c, 2604, 2609
 - AdjustRenumScratch, 2605
 - CountDo, 2606
 - CountFun, 2606
 - DetCurDum, 2605
 - DimensionExpression, 2606
 - DimensionSubterm, 2606
 - DimensionTerm, 2606
 - DoDelta3, 2607
 - DoDistrib, 2607
 - DoPartitions, 2607
 - DoPermutations, 2608
 - DoShuffle, 2608
 - DoStuffle, 2608
 - EqualArg, 2607
 - FinishShuffle, 2608
 - FinishStuffle, 2609
 - FullRenum, 2605
 - FunLevel, 2605
 - MoveDummies, 2605
 - MultDo, 2606
 - NEWCODE, 2605
 - ReNumber, 2605
 - Shuffle, 2608
 - Stuffle, 2608
 - StuffRootAdd, 2609
 - TestPartitions, 2607
 - TryDo, 2607
- ResizeData
 - O_const, 283
- resizeFlag
 - O_const, 287
- ResizeLONG
 - O_const, 283
- ResizeNCWORD
 - O_const, 283
- ResizePOINTER
 - O_const, 284
- ResizePOS
 - O_const, 284
- ResizeWORD
 - O_const, 283
- ResolveSet
 - declare.h, 857
 - wildcard.c, 3212
- restore
 - AStack, 28
- restoreCouple
 - Assign, 23
- restoreMsg
 - AStack, 28
- resultAGraph
 - SProcess, 411
- resultMGraph
 - SProcess, 410
- retstat
 - PF_BUFFER, 335
- REVERSEARG
 - ftypes.h, 1445
- REVERSEFILESTREAM
 - ftypes.h, 1365
- REVERSEFUNCTION
 - ftypes.h, 1390
- REVERSEORDER
 - ftypes.h, 1431
- ReverseStatements
 - declare.h, 876
 - tools.c, 3073
- RevertScratch
 - declare.h, 858
 - store.c, 2823
- revision
 - STOREHEADER, 420
- ReWorkT
 - declare.h, 827
 - tables.c, 2957
- rght
 - NoDe, 273
- rhs
 - CbUf, 72
 - DICTIONARY_ELEMENT, 84
- RHSIDE
 - ftypes.h, 1382
- rhsInParallel
 - ParallelVars, 324
- right
 - NaMeNode, 263

- tree, [467](#)
- RLONG
 - form3.h, [1333](#)
- rloser
 - NoDe, [273](#)
- RND
 - evaluate.c, [1139](#)
- ROOTFUNCTION
 - ftypes.h, [1394](#)
- rootnum
 - CbUf, [74](#)
 - TaBIEs, [452](#)
- RPARENTHESIS
 - ftypes.h, [1378](#)
- rsrc
 - NoDe, [274](#)
- RSSize
 - N_const, [256](#)
- RunAddArg
 - declare.h, [992](#)
 - transform.c, [3136](#)
- RunCycle
 - declare.h, [992](#)
 - transform.c, [3136](#)
- RunDecode
 - declare.h, [990](#)
 - transform.c, [3135](#)
- RunDedup
 - declare.h, [993](#)
 - transform.c, [3136](#)
- RunDropArg
 - declare.h, [993](#)
 - transform.c, [3137](#)
- RunEncode
 - declare.h, [990](#)
 - transform.c, [3135](#)
- RunExplode
 - declare.h, [991](#)
 - transform.c, [3136](#)
- RunHtoZArg
 - declare.h, [993](#)
 - transform.c, [3138](#)
- RunImplode
 - declare.h, [990](#)
 - transform.c, [3135](#)
- RunIsLyndon
 - declare.h, [992](#)
 - transform.c, [3137](#)
- RunMulArg
 - declare.h, [992](#)
 - transform.c, [3137](#)
- RunPermute
 - declare.h, [991](#)
 - transform.c, [3136](#)
- RunReplace
 - declare.h, [990](#)
 - transform.c, [3135](#)
- RunReverse
 - declare.h, [992](#)
 - transform.c, [3136](#)
- RunSelectArg
 - declare.h, [993](#)
 - transform.c, [3137](#)
- RunToLyndon
 - declare.h, [992](#)
 - transform.c, [3137](#)
- RunTransform
 - declare.h, [990](#)
 - transform.c, [3135](#)
- RunZtoHArg
 - declare.h, [993](#)
 - transform.c, [3137](#)
- RWLOCKR
 - declare.h, [807](#)
- RWLOCKW
 - declare.h, [807](#)
- rwmode
 - dbase, [79](#)
- S
 - AllGlobals, [10](#)
- S0
 - M_const, [171](#)
 - T_const, [433](#)
- S_const, [386](#)
 - Balancing, [387](#)
 - CollectOverFlag, [388](#)
 - ExecMode, [387](#)
 - MaxExprSize, [386](#)
 - MultiThreaded, [387](#)
 - NumOldNumFactors, [387](#)
 - NumOldOnFile, [387](#)
 - OldNumFactors, [387](#)
 - OldOnFile, [386](#)
 - Olduflags, [387](#)
 - Oldvflags, [387](#)
- S_FLUSH_B
 - checkpoint.c, [528](#)
- s_one
 - FixedGlobals, [135](#)
- S_WRITE_B
 - checkpoint.c, [527](#)
- S_WRITE_DOLLAR
 - checkpoint.c, [529](#)
- S_WRITE_LIST
 - checkpoint.c, [528](#)
- S_WRITE_NAMETREE
 - checkpoint.c, [529](#)
- S_WRITE_S
 - checkpoint.c, [528](#)
- salter
 - OPTIMIZE, [297](#)
- saveCouple
 - Assign, [22](#)
- SaveData
 - O_const, [281](#)
- SaveHeader

- O_const, [281](#)
- SAVEREVISION
 - store.c, [2822](#)
- sBer
 - T_const, [438](#)
- sbuf
 - ParallelVars, [323](#)
- sBuffer
 - sOrT, [399](#)
- sbufnum
 - M_const, [178](#)
- SBYTE
 - form3.h, [1333](#)
- ScanFunctions
 - declare.h, [858](#)
 - function.c, [1473](#)
- sch.c, [2645](#), [2650](#)
 - AddArrayIndex, [2648](#)
 - AddToLine, [2646](#)
 - CodeToLine, [2647](#)
 - FiniLine, [2646](#)
 - IniLine, [2646](#)
 - LongToLine, [2647](#)
 - MultiplyToLine, [2647](#)
 - PrintTime, [2648](#)
 - PrtTerms, [2648](#)
 - RatToLine, [2647](#)
 - StrCopy, [2646](#)
 - TalToLine, [2647](#)
 - TokenToLine, [2647](#)
 - WriteAll, [2649](#)
 - WriteArgument, [2648](#)
 - WriteDictionary, [2648](#)
 - WriteExpression, [2649](#)
 - WriteInnerTerm, [2649](#)
 - WriteLists, [2648](#)
 - WriteOne, [2649](#)
 - WriteSubTerm, [2649](#)
 - WriteTerm, [2649](#)
 - WrtPower, [2648](#)
- schemeFlags
 - OPTIMIZE, [297](#)
- schemenum
 - O_const, [287](#)
- ScratSize
 - M_const, [173](#)
- SEARCHINGPRECASE
 - ftypes.h, [1369](#)
- SEARCHINGPREENDSWITCH
 - ftypes.h, [1369](#)
- sec
 - Process, [373](#)
- sEdges
 - EGraph, [111](#)
- sedges
 - MConn, [201](#)
- SeekFile
 - declare.h, [885](#)
 - tools.c, [3075](#)
- SeekScratch
 - declare.h, [858](#)
 - store.c, [2822](#)
- SELECTARG
 - ftypes.h, [1446](#)
- selectExternalChannel
 - declare.h, [979](#)
 - extcmd.c, [1188](#)
- selectLeg
 - Assign, [18](#)
- selecttermundo
 - N_const, [252](#)
- selectVertex
 - Assign, [18](#)
- selectVertexSimp
 - Assign, [18](#)
- selfloop
 - MGraph, [216](#)
- selUnAssLeg
 - Assign, [20](#)
- selUnAssVertex
 - Assign, [20](#)
- selUnAssVertexSimp
 - Assign, [20](#)
- SEPARATESYMBOL
 - ftypes.h, [1404](#)
- SEPARATOR
 - fwin.h, [1497](#)
 - unix.h, [3182](#)
- separators
 - C_const, [42](#)
- set_del
 - declare.h, [978](#)
 - tools.c, [3082](#)
- set_in
 - declare.h, [977](#)
 - tools.c, [3081](#)
- set_of_char
 - structs.h, [2896](#)
- set_set
 - declare.h, [977](#)
 - tools.c, [3081](#)
- set_sub
 - declare.h, [978](#)
 - tools.c, [3082](#)
- setAGraph
 - AStack, [27](#)
- SETBASELENGTH
 - declare.h, [799](#)
- SETBASEPOSITION
 - declare.h, [799](#)
- SETBIT
 - ftypes.h, [1437](#)
- SETBUFSIZE
 - setfile.c, [2685](#)
- setDefaultValues
 - Options, [301](#)

- SetDictionaryOptions
 - declare.h, 1010
 - dict.c, 1068
- SetDualOpts
 - diawrap.cc, 1052
- SetElementList
 - C_const, 44
- SetElements
 - variable.h, 3189
- setEMom
 - EEdge, 97
- SetEndHScratch
 - declare.h, 858
 - store.c, 2823
- setEndMG
 - Options, 303
- SetEndScratch
 - declare.h, 858
 - store.c, 2823
- setErExit
 - Options, 303
- SETERROR
 - declare.h, 799
- SETEXP
 - ftypes.h, 1388
- SetExpr
 - compcomm.c, 603
 - declare.h, 917
- SetExprCases
 - compcomm.c, 603
 - declare.h, 964
- setExtern
 - EGraph, 105
 - ENode, 116
- setexterntopo
 - T_const, 444
- setExtLoop
 - EGraph, 106
- SetfFloatPrecision
 - evaluate.c, 1141
- setfile.c, 2683, 2688
 - AllocFileHandle, 2686
 - AllocSetups, 2685
 - AllocSort, 2686
 - AllocSortFileName, 2686
 - commentchar, 2687
 - curdirp, 2687
 - cursortdirp, 2687
 - DeAllocFileHandle, 2686
 - DEFINEVALUE, 2684
 - DoSetups, 2685
 - dotchar, 2687
 - GetSetupPar, 2685
 - highfirst, 2687
 - lowfirst, 2688
 - MakeSetupAllocs, 2686
 - NUMERICALVALUE, 2684
 - ONOFFVALUE, 2684
 - PATHVALUE, 2684
 - procedureextension, 2688
 - ProcessOption, 2685
 - RecalcSetups, 2685
 - SETBUFSIZE, 2685
 - setupparameters, 2688
 - STRINGVALUE, 2684
 - TryEnvironment, 2687
 - TryFileSetups, 2687
 - WriteSetup, 2686
- SetFileIndex
 - declare.h, 858
 - store.c, 2826
- SetFloatPrecision
 - float.c, 1289
- SETFUNCTION
 - ftypes.h, 1389
- setId
 - EEdge, 97
 - ENode, 116
- setinterntopo
 - T_const, 444
- SETKILLMODEFOREXTERNALCHANNEL
 - declare.h, 809
- SetList
 - C_const, 44
- setLMom
 - EEdge, 97
- setmod
 - poly, 351
- SetMods
 - declare.h, 872
 - execute.c, 1153
- SETNUMBER
 - ftypes.h, 1417
- setOldDefaultValues
 - Options, 301
- SETONLY
 - ftypes.h, 1374
- setonoff
 - compcomm.c, 600
 - declare.h, 916
- setOutAG
 - Options, 303
- setOutgrf
 - Output, 310
- setOutgrp
 - Output, 310
- setOutMG
 - Options, 302
- setOutputF
 - Options, 302
- setOutputP
 - Options, 302
- SetPosFile
 - declare.h, 885
 - tools.c, 3076
- setQGrafOpt

- Options, 301
- SeTs, 388
 - dimension, 389
 - first, 388
 - flags, 389
 - last, 389
 - name, 388
 - namesize, 389
 - node, 389
 - type, 388
- Sets
 - variable.h, 3190
- SetScratch
 - declare.h, 873
 - store.c, 2823
- SETSET
 - ftypes.h, 1387
- SetSpecialMode
 - declare.h, 907
 - module.c, 1756
- SETSTARTPOS
 - declare.h, 801
- SETTERMINATORFOREXTERNALCHANNEL
 - declare.h, 809
- SETTONUM
 - ftypes.h, 1417
- setType
 - EEdge, 97
 - ENode, 117
- SetupDir
 - M_const, 172
- SETUPFILE
 - ftypes.h, 1366
- SetupFile
 - M_const, 172
- setupfilename
 - inivar.h, 1674
 - variable.h, 3196
- SetupFlag
 - C_const, 58
- SetupMPFTables
 - float.c, 1289
- SetupMZVTables
 - float.c, 1289
- SETUPPARAMETERS, 389
 - flags, 390
 - parameter, 390
 - type, 390
 - value, 390
- setupparameters
 - setfile.c, 2688
- setValue
 - Fraction, 139
 - Options, 301
- sfact
 - T_const, 438
- SFHSIZE
 - fsizes.h, 1345
- sFill
 - sOrT, 400
- Sflush
 - declare.h, 859
 - sort.c, 2714
- sFun
 - STOREHEADER, 419
- SGN
 - declare.h, 789
- sgn
 - TrAcEn, 459
 - TrAcEs, 463
- SGroup, 391
 - ~SGroup, 391
 - addGroup, 392
 - cgen, 394
 - clearGroup, 392
 - csav, 394
 - delGroup, 392
 - eclass, 393
 - elem, 393
 - genNext, 392
 - neclass, 393
 - nElem, 392
 - nelem, 393
 - newGroup, 392
 - nextElem, 393
 - nnodes, 393
 - permg, 394
 - perms, 394
 - pgq, 394
 - pgr, 394
 - print, 392
 - psq, 394
 - psr, 394
 - SGroup, 391
 - size, 393
- sHalf
 - sOrT, 400
- SHcombi
 - N_const, 254
- SHcombisize
 - N_const, 255
- SHELL
 - pre.c, 2310
- shellname
 - X_const, 475
- SHMWINSIZE
 - fsizes.h, 1345
- shmWinSize
 - M_const, 175
- ShortSortCount
 - R_const, 382
- ShortStats
 - C_const, 54
- ShortStatsMax
 - C_const, 64
- ShrinkDictionary

- declare.h, [1011](#)
- dict.c, [1068](#)
- Shuffle
 - declare.h, [825](#)
 - reshuf.c, [2608](#)
- SHvar
 - N_const, [254](#)
- SHvariables, [395](#)
 - combilast, [396](#)
 - do_uffle, [396](#)
 - finishuf, [396](#)
 - incoef, [395](#)
 - level, [397](#)
 - nincoef, [396](#)
 - option, [397](#)
 - outfun, [395](#)
 - outterm, [395](#)
 - stop1, [396](#)
 - stop2, [396](#)
 - ststop1, [396](#)
 - ststop2, [396](#)
 - thefunction, [397](#)
- sld
 - EGraph, [110](#)
- SIGALRM
 - portsignals.h, [2305](#)
- SIGEMT
 - portsignals.h, [2304](#)
- SIGFPE
 - portsignals.h, [2303](#)
- SIGFUNCTION
 - ftypes.h, [1392](#)
- SIGHUP
 - portsignals.h, [2305](#)
- SIGILL
 - portsignals.h, [2304](#)
- SIGINT
 - portsignals.h, [2305](#)
- SIGLOST
 - portsignals.h, [2304](#)
- SIGMA
 - ftypes.h, [1455](#)
- sign
 - DiStRiBuTe, [85](#)
 - node, [276](#)
 - OptQGRef, [309](#)
 - PeRmUtE, [332](#)
 - PeRmUtEp, [333](#)
 - poly, [350](#)
- sign1
 - TrAcEs, [465](#)
- sign2
 - TrAcEs, [465](#)
- SignCheck
 - N_const, [260](#)
- SIGNFUN
 - ftypes.h, [1392](#)
- SIGPIPE
 - portsignals.h, [2304](#)
- SIGPROF
 - portsignals.h, [2305](#)
- SIGQUIT
 - portsignals.h, [2305](#)
- SIGSEGV
 - portsignals.h, [2303](#)
- SIGSYS
 - portsignals.h, [2304](#)
- SIGTERM
 - portsignals.h, [2304](#)
- SIGVTALRM
 - portsignals.h, [2305](#)
- SIGXCPU
 - portsignals.h, [2304](#)
- SIGXFSZ
 - portsignals.h, [2304](#)
- silent
 - M_const, [187](#)
- simp1token
 - declare.h, [946](#)
 - token.c, [3041](#)
- simp2token
 - declare.h, [947](#)
 - token.c, [3042](#)
- simp3atoken
 - declare.h, [947](#)
 - token.c, [3042](#)
- simp3btoken
 - declare.h, [947](#)
 - token.c, [3042](#)
- simp4token
 - declare.h, [947](#)
 - token.c, [3042](#)
- simp5token
 - declare.h, [947](#)
 - token.c, [3042](#)
- simp6token
 - declare.h, [948](#)
 - token.c, [3042](#)
- SIMPLE
 - ftypes.h, [1456](#)
- simple_poly
 - poly, [351](#), [352](#)
- SimpleDelta
 - float.c, [1285](#)
- SimpleDeltaC
 - float.c, [1285](#)
- SimpleSplitMerge
 - declare.h, [963](#)
 - sort.c, [2724](#)
- SimpleSplitMergeRec
 - declare.h, [963](#)
 - sort.c, [2724](#)
- Simplify
 - declare.h, [859](#)
 - reken.c, [2549](#)
- simplify_fmpz

- flintinterface.h, [1271](#)
- simplify_fmpz_poly
 - flintinterface.h, [1271](#)
- simpwtoken
 - declare.h, [947](#)
 - token.c, [3041](#)
- simulated_annealing
 - optimize.cc, [1991](#)
- sInd
 - STOREHEADER, [419](#)
- SINFUNCTION
 - ftypes.h, [1396](#)
- SingleTable
 - float.c, [1285](#)
- SINHFUNCTION
 - ftypes.h, [1397](#)
- SIOsize
 - M_const, [174](#)
- size
 - ARGBUFFER, [14](#)
 - DICTIONARY_ELEMENT, [84](#)
 - DoLIaRS, [88](#)
 - ExPrEsSiOn, [121](#)
 - FaCdOILaR, [125](#)
 - InDeXeNtRy, [154](#)
 - INSIDEINFO, [157](#)
 - LIST, [165](#)
 - MiNmAx, [218](#)
 - objects, [290](#)
 - PRELOAD, [364](#)
 - SGroup, [393](#)
 - TaBIEbAsEsUbInDeX, [447](#)
- size_of_form_notation
 - poly, [353](#)
- size_of_form_notation_with_den
 - poly, [353](#)
- size_of_terms
 - poly, [361](#)
- SizeCommutInSet
 - C_const, [63](#)
- sizecouplings
 - MoDeL, [228](#)
- SizeDictionaries
 - O_const, [286](#)
- sizeelements
 - DICTIONARY, [82](#)
- SIZEFACS
 - fsizes.h, [1341](#)
- SizeForSpectatorFiles
 - M_const, [183](#)
- SizeInFile
 - sOrT, [399](#)
- SizeOfDollar
 - declare.h, [967](#)
 - dollar.c, [1087](#)
- SizeOfExpression
 - declare.h, [967](#)
 - execute.c, [1155](#)
- SIZEOFFUNCTION
 - ftypes.h, [1400](#)
- sizeprimelist
 - T_const, [444](#)
- sizeprototype
 - ExPrEsSiOn, [124](#)
- sizeselecttermundo
 - N_const, [260](#)
- SIZESTORECACHE
 - fsizes.h, [1346](#)
- SizeStoreCache
 - M_const, [174](#)
- sizevertices
 - MoDeL, [228](#)
- SKIP
 - ftypes.h, [1420](#)
- SkipAName
 - compiler.c, [713](#)
 - declare.h, [948](#)
- SKIPBLANKS
 - declare.h, [792](#)
- SKIPBRA1
 - declare.h, [793](#)
- SKIPBRA2
 - declare.h, [793](#)
- SKIPBRA3
 - declare.h, [793](#)
- SKIPBRA4
 - declare.h, [793](#)
- SKIPBRA5
 - declare.h, [794](#)
- SKIPBUFSIZE
 - pre.c, [2310](#)
- SkipClears
 - M_const, [178](#)
- SkipField
 - declare.h, [889](#)
 - tools.c, [3080](#)
- SKIPGEXPRESSION
 - ftypes.h, [1421](#)
- SKIPLEXPRESSION
 - ftypes.h, [1420](#)
- SkipName
 - declare.h, [1016](#)
 - pre.c, [2327](#)
- SKIPUNHIDEGEXPRESSION
 - ftypes.h, [1423](#)
- SKIPUNHIDELEXPRESSION
 - ftypes.h, [1423](#)
- SLARGEBUFFER
 - fsizes.h, [1344](#)
- SLargeSize
 - M_const, [174](#)
- slavebuf
 - ParallelVars, [323](#)
- sLevel
 - R_const, [380](#)
- slist

- Interaction, 160
- sLoops
 - EGraph, 112
- small_power
 - T_const, 435
- small_power_maxn
 - T_const, 441
- small_power_maxx
 - T_const, 441
- small_power_n
 - T_const, 435
- SmallEsize
 - sOrT, 402
- SmallSize
 - sOrT, 402
- smart.c, 2703, 2704
 - MatchIsPossible, 2703
 - StudyPattern, 2703
- sMaxdeg
 - EGraph, 111
- SMA XF PATCHES
 - fsizes.h, 1344
- SMaxFpatches
 - M_const, 177
- SMA XPATCHES
 - fsizes.h, 1344
- SMaxPatches
 - M_const, 177
- SNAIL
 - ftypes.h, 1455
- sNodes
 - EGraph, 111
- snodes
 - MConn, 201
- SNUMBER
 - ftypes.h, 1388
- SORT
 - ftypes.h, 1436
- sOrT, 397
 - cBuffer, 400
 - cBufferSize, 404
 - f, 402
 - ff, 402
 - file, 399
 - fPatches, 401
 - fPatchesStop, 401
 - fPatchN, 405
 - fPatchN1, 404
 - GenSpace, 403
 - GenTerms, 403
 - inNum, 405
 - inPatches, 401
 - iPatches, 402
 - ktoi, 401
 - LargeSize, 402
 - IBuffer, 399
 - IFill, 399
 - IPatch, 404
 - ITop, 399
 - MaxFpatches, 404
 - MaxPatches, 403
 - maxtermsize, 404
 - newmaxtermsize, 405
 - ninterms, 403
 - outputmode, 405
 - Patches, 400
 - poin2a, 401
 - poina, 401
 - PoinFill, 400
 - PolyFlag, 404
 - PolyWise, 404
 - pStop, 401
 - putinsize, 403
 - sBuffer, 399
 - sFill, 400
 - sHalf, 400
 - SizeInFile, 399
 - SmallEsize, 402
 - SmallSize, 402
 - SpaceLeft, 403
 - sPointer, 400
 - stage4, 405
 - stagelevel, 405
 - sTerms, 402
 - sTop, 400
 - sTop2, 400
 - Terms2InSmall, 403
 - TermsInSmall, 402
 - TermsLeft, 403
 - tree, 401
 - type, 404
 - used, 399
- sort.c, 2709, 2725
 - AddArgs, 2717
 - AddCoef, 2715
 - AddPoly, 2716
 - BinarySearch, 2724
 - CleanUpSort, 2724
 - Compare1, 2718
 - CompareHSymbols, 2719
 - CompareSymbols, 2718
 - ComPress, 2719
 - EndSort, 2712
 - FlushOut, 2715
 - GarbHand, 2720
 - humanBytesSuffix, 2725
 - HumanString, 2711
 - HUMANSTRLEN, 2710
 - HUMANSUFFLEN, 2710
 - humanTermsSuffix, 2725
 - INSLLENGTH, 2710
 - LowerSortLevel, 2724
 - MergePatches, 2721
 - NewSort, 2712
 - NEWSPLITMERGE, 2710
 - NEWSPLITMERGETIMSORT, 2710

- PolyRatFunSpecial, [2724](#)
- PutIn, [2713](#)
- PutOut, [2714](#)
- Sflush, [2714](#)
- SimpleSplitMerge, [2724](#)
- SimpleSplitMergeRec, [2724](#)
- SortWild, [2723](#)
- SplitMerge, [2720](#)
- StageSort, [2723](#)
- StoreTerm, [2722](#)
- totterms, [2725](#)
- WriteStats, [2711](#)
- SORTANTIPOWER
 - ftypes.h, [1386](#)
- SORTHIGHFIRST
 - ftypes.h, [1385](#)
- SORTING
 - structs.h, [2898](#)
- SORTIOSIZE
 - fsizes.h, [1343](#)
- SORTLOWFIRST
 - ftypes.h, [1385](#)
- SORTMODULE
 - ftypes.h, [1366](#)
- SORTPOWERFIRST
 - ftypes.h, [1385](#)
- SortReallocateFlag
 - C_const, [62](#)
- SortTheList
 - declare.h, [953](#)
 - lus.c, [1679](#)
- SortType
 - BrAcKeTiNfO, [34](#)
 - C_const, [58](#)
 - R_const, [382](#)
- SortWeights
 - argument.c, [482](#)
 - declare.h, [921](#)
- SortWild
 - declare.h, [859](#)
 - sort.c, [2723](#)
- SoScratC
 - N_const, [253](#)
- SpaceLeft
 - sOrT, [403](#)
- sparse
 - OPTIMIZE, [297](#)
 - TaBIes, [450](#)
 - VeRtEx, [474](#)
- spare1
 - objects, [290](#)
- spare2
 - objects, [290](#)
- spare3
 - objects, [291](#)
- SpareTable
 - declare.h, [873](#)
 - tables.c, [2955](#)
- sparse
 - TaBIes, [452](#)
- SPARSETABLE
 - ftypes.h, [1453](#)
- spec
 - FuNcTiOn, [145](#)
- SpecialCleanup
 - declare.h, [872](#)
 - execute.c, [1153](#)
- SPECIFICATION
 - ftypes.h, [1372](#)
- SPECMASK
 - form3.h, [1328](#)
- SpecTatoR, [406](#)
 - exprnumber, [407](#)
 - fh, [406](#)
 - flags, [407](#)
 - name, [407](#)
 - position, [406](#)
 - readpos, [406](#)
- spectator.c, [2781](#), [2783](#)
 - ClearSpectators, [2782](#)
 - CoCopySpectator, [2782](#)
 - CoCreateSpectator, [2781](#)
 - CoEmptySpectator, [2782](#)
 - CoRemoveSpectator, [2782](#)
 - CoToSpectator, [2781](#)
 - FlushSpectators, [2782](#)
 - GetFromSpectator, [2782](#)
 - PutInSpectator, [2782](#)
- SPECTATOREXPRESSION
 - ftypes.h, [1423](#)
- SpectatorFiles
 - M_const, [173](#)
- SPECTATORSIZE
 - fsizes.h, [1344](#)
- SpectatorSize
 - M_const, [176](#)
- spin
 - PaRtlcLe, [327](#)
- SplitMerge
 - declare.h, [860](#)
 - sort.c, [2720](#)
- SplitScratch
 - N_const, [252](#)
- SplitScratch1
 - N_const, [253](#)
- SplitScratchSize
 - N_const, [254](#)
- SplitScratchSize1
 - N_const, [255](#)
- sPointer
 - sOrT, [400](#)
- sproc
 - Assign, [24](#)
 - EGraph, [109](#)
 - Options, [305](#)
 - Output, [313](#)

- PNodeClass, [341](#)
- Process, [372](#)
- SProcess, [407](#)
 - ~SProcess, [409](#)
 - agraph, [412](#)
 - agrcount, [414](#)
 - assign, [410](#)
 - astack, [411](#)
 - cl2nd, [412](#)
 - clist, [413](#)
 - DUMMY_PADDING, [414](#)
 - egraph, [412](#)
 - endAGraph, [410](#)
 - endMGraph, [410](#)
 - extperm, [414](#)
 - generate, [409](#)
 - id, [413](#)
 - loop, [413](#)
 - match, [410](#)
 - mgraph, [412](#)
 - mgrcount, [414](#)
 - model, [411](#)
 - nAGraphs, [415](#)
 - nAOPI, [415](#)
 - nclass, [412](#)
 - ncouple, [413](#)
 - nd2cl, [412](#)
 - nEdges, [413](#)
 - nExtern, [413](#)
 - nfinal, [412](#)
 - ninitl, [412](#)
 - nMGraphs, [414](#)
 - nMOPI, [414](#)
 - nNodes, [413](#)
 - nvert, [413](#)
 - opt, [411](#)
 - pnclass, [411](#)
 - proc, [411](#)
 - prSProcess, [409](#)
 - resultAGraph, [411](#)
 - resultMGraph, [410](#)
 - SProcess, [409](#)
 - tCouple, [414](#)
 - toMNodeClass, [410](#)
 - wAGraphs, [415](#)
 - wAOPI, [415](#)
 - wMGraphs, [414](#)
 - wMOPI, [415](#)
- sptbl
 - Process, [372](#)
- SQRTFUNCTION
 - ftypes.h, [1396](#)
- SS
 - T_const, [433](#)
- SSMALLBUFFER
 - fsizes.h, [1343](#)
- SSmallEsize
 - M_const, [174](#)
- SSMALLOC_OVERFLOW
 - fsizes.h, [1344](#)
- SSmallSize
 - M_const, [174](#)
- SSORTIOSIZE
 - fsizes.h, [1344](#)
- sSym
 - STOREHEADER, [419](#)
- st
 - NCand, [270](#)
 - NStack, [279](#)
- stage4
 - sOrT, [405](#)
- Stage4Name
 - R_const, [380](#)
- stagelevel
 - sOrT, [405](#)
- StageSort
 - declare.h, [912](#)
 - sort.c, [2723](#)
- stap
 - TrAcEs, [463](#)
- start
 - BrAcKeTiNdEx, [32](#)
 - VaRrEnUm, [470](#)
- StartFiles
 - declare.h, [887](#)
 - tools.c, [3073](#)
- startlinenumber
 - DoLoOp, [91](#)
- StartLoops
 - declare.h, [1019](#)
 - lus.c, [1679](#)
- StartMore
 - startup.c, [2794](#)
- startposition
 - ReNuMbEr, [384](#)
- StartPrepro
 - declare.h, [900](#)
 - pre.c, [2322](#)
- startup.c, [2791](#), [2796](#)
 - CleanUp, [2794](#)
 - DEFAULTFNAMLENGTH, [2793](#)
 - defaulttempfilename, [2796](#)
 - DoTail, [2793](#)
 - emptystring, [2796](#)
 - FORMNAME, [2792](#)
 - GetRunningTime, [2795](#)
 - IniVars, [2794](#)
 - main, [2794](#)
 - OpenInput, [2793](#)
 - PrintDeprecation, [2795](#)
 - PrintRunningTime, [2795](#)
 - ReserveTempFiles, [2793](#)
 - StartMore, [2794](#)
 - StartVariables, [2793](#)
 - STRINGIFY, [2792](#)
 - STRINGIFY__, [2792](#)

- TAKEPATH, [2793](#)
- TerminatImpl, [2795](#)
- VERSIONSTR, [2793](#)
- VERSIONSTR__, [2792](#)
- StartVariables
 - declare.h, [810](#)
 - startup.c, [2793](#)
- STATEMENT
 - ftypes.h, [1373](#)
- STATIC_ASSERT
 - form3.h, [1326](#)
- STATIC_ASSERT__1
 - form3.h, [1326](#)
- STATIC_ASSERT__2
 - form3.h, [1327](#)
- STATIC_ASSERT__3
 - form3.h, [1327](#)
- StatsFlag
 - C_const, [57](#)
- STATSMERGETOFILE
 - ftypes.h, [1386](#)
- STATSOUT
 - ftypes.h, [1371](#)
- STATSPOSTSORT
 - ftypes.h, [1386](#)
- STATSSPLITMERGE
 - ftypes.h, [1386](#)
- status
 - ExPrEsSiOn, [123](#)
 - PF_BUFFER, [335](#)
- STDERR
 - pre.c, [2310](#)
- stderrname
 - X_const, [475](#)
- StdOut
 - M_const, [177](#)
- step1
 - TrAcEs, [464](#)
- STEP2
 - dollar.c, [1084](#)
- sTerms
 - sOrT, [402](#)
- STermsInSmall
 - M_const, [174](#)
- STERMSSMALL
 - fsizes.h, [1344](#)
- sTop
 - sOrT, [400](#)
- stop
 - PF_BUFFER, [335](#)
- stop1
 - SHvariables, [396](#)
- sTop2
 - sOrT, [400](#)
- stop2
 - SHvariables, [396](#)
- StopWatchZero
 - P_const, [318](#)
- STORE
 - ftypes.h, [1436](#)
- store.c, [2821](#), [2832](#)
 - AddToScratch, [2824](#)
 - CoLoad, [2824](#)
 - CopyExpression, [2828](#)
 - CoSave, [2824](#)
 - DeleteStore, [2824](#)
 - DetVars, [2825](#)
 - FindInIndex, [2827](#)
 - FindrNumber, [2827](#)
 - GetFromStore, [2825](#)
 - GetMoreFromMem, [2825](#)
 - GetMoreTerms, [2825](#)
 - GetOneTerm, [2825](#)
 - GetTable, [2827](#)
 - GetTerm, [2824](#)
 - NextFileIndex, [2826](#)
 - OpenTemp, [2822](#)
 - PutInStore, [2824](#)
 - ReadFromScratch, [2823](#)
 - ReadSaveExpression, [2831](#)
 - ReadSaveHeader, [2828](#)
 - ReadSaveIndex, [2829](#)
 - ReadSaveTerm32, [2830](#)
 - ReadSaveVariables, [2829](#)
 - ResetScratch, [2823](#)
 - RevertScratch, [2823](#)
 - SAVEREVISION, [2822](#)
 - SeekScratch, [2822](#)
 - SetEndHScratch, [2823](#)
 - SetEndScratch, [2823](#)
 - SetFileIndex, [2826](#)
 - SetScratch, [2823](#)
 - TermRenummer, [2826](#)
 - ToStorage, [2825](#)
 - VarStore, [2826](#)
 - WriteStoreHeader, [2828](#)
- STORECACHE
 - structs.h, [2897](#)
- StOrEcAcHe, [416](#)
 - buffer, [417](#)
 - next, [416](#)
 - position, [416](#)
 - toppos, [416](#)
- StoreCache
 - T_const, [434](#)
- StoreCacheAlloc
 - T_const, [434](#)
- StoreData
 - R_const, [376](#)
- STOREEXPRESSION
 - ftypes.h, [1421](#)
- StoreFileSize
 - C_const, [42](#)
- StoreHandle
 - C_const, [68](#)
- STOREHEADER, [417](#)

- endianness, [418](#)
- headermark, [418](#)
- lenLONG, [418](#)
- lenPOINTER, [418](#)
- lenPOS, [418](#)
- lenWORD, [418](#)
- maxpower, [419](#)
- reserved, [420](#)
- revision, [420](#)
- sFun, [419](#)
- sInd, [419](#)
- sSym, [419](#)
- sVec, [419](#)
- wildoffset, [420](#)
- STOREMODULE
 - ftypes.h, [1366](#)
- StoreTerm
 - declare.h, [861](#)
 - sort.c, [2722](#)
- str_dup
 - declare.h, [973](#)
 - minos.c, [1721](#)
- StrCmp
 - declare.h, [889](#)
 - tools.c, [3077](#)
- StrCopy
 - declare.h, [812](#)
 - sch.c, [2646](#)
- strDup1
 - declare.h, [873](#)
 - tools.c, [3079](#)
- STREAM
 - structs.h, [2895](#)
- StreaM, [420](#)
 - afterwards, [424](#)
 - buffer, [422](#)
 - bufferposition, [423](#)
 - bufferize, [423](#)
 - eqnum, [424](#)
 - fileposition, [421](#)
 - FoldName, [422](#)
 - handle, [423](#)
 - inbuffer, [423](#)
 - isnextchar, [424](#)
 - linenumber, [421](#)
 - name, [422](#)
 - nextchar, [424](#)
 - olddelay, [424](#)
 - oldnoshowinput, [424](#)
 - pname, [422](#)
 - pointer, [422](#)
 - prevars, [423](#)
 - previous, [423](#)
 - previousNoShowInput, [424](#)
 - prevline, [421](#)
 - reserved, [424](#)
 - top, [422](#)
 - type, [423](#)
- Streams
 - C_const, [46](#)
- StrHICmp
 - declare.h, [890](#)
 - tools.c, [3078](#)
- StrICmp
 - declare.h, [890](#)
 - tools.c, [3077](#)
- StrICont
 - declare.h, [890](#)
 - tools.c, [3078](#)
- strict
 - TaBIEs, [452](#)
- StrictRounding
 - float.c, [1290](#)
- STRINGIFY
 - pre.c, [2310](#)
 - startup.c, [2792](#)
- STRINGIFY__
 - pre.c, [2310](#)
 - startup.c, [2792](#)
- STRINGVALUE
 - setfile.c, [2684](#)
- StrLen
 - declare.h, [890](#)
 - tools.c, [3078](#)
- structs.h, [2890](#), [2899](#)
 - ALLGLOBALS, [2898](#)
 - BHEAD, [2893](#)
 - BHEAD0, [2894](#)
 - CBUF, [2897](#)
 - CHANNEL, [2897](#)
 - COMPARE, [2899](#)
 - COMPAREDUMMY, [2897](#)
 - COMPTREE, [2895](#)
 - DISTRIBUTE, [2898](#)
 - DO_UFFLE, [2897](#)
 - DOLOOP, [2896](#)
 - EMPTYININDEX, [2893](#)
 - FILEHANDLE, [2895](#)
 - FILEINDEX, [2894](#)
 - FINISHUFFLE, [2897](#)
 - FIXEDGLOBALS, [2899](#)
 - FUNCTIONS, [2895](#)
 - INDEXENTRY, [2894](#)
 - INFILEINDEX, [2893](#)
 - NAMENODE, [2894](#)
 - NAMETREE, [2895](#)
 - NESTING, [2897](#)
 - one_byte, [2896](#)
 - PARTI, [2898](#)
 - PERM, [2898](#)
 - PERMP, [2898](#)
 - PHEAD, [2893](#)
 - PHEAD0, [2893](#)
 - PREVAR, [2896](#)
 - RENUMBER, [2894](#)
 - set_of_char, [2896](#)

- SORTING, 2898
- STORECACHE, 2897
- STREAM, 2895
- TABLES, 2895
- TRACEN, 2896
- TRACES, 2896
- VARRENUM, 2894
- WCN, 2898
- WCN2, 2899
- ststop1
 - SHvariables, 396
- ststop2
 - SHvariables, 396
- StudyPattern
 - declare.h, 954
 - smart.c, 2703
- StuffAdd
 - declare.h, 789
- Stuffle
 - declare.h, 825
 - reshuf.c, 2608
- StuffRootAdd
 - declare.h, 825
 - reshuf.c, 2609
- sub
 - Fraction, 140
 - poly, 355
- SUBAFTER
 - ftypes.h, 1432
- SUBAFTERNOT
 - ftypes.h, 1432
- SUBALL
 - ftypes.h, 1432
- SuBbUf, 425
 - buffernum, 425
 - subexpnum, 425
- subCouple
 - Assign, 23
- SUBDISORDER
 - ftypes.h, 1432
- subexpbuffers
 - compiler.c, 715
- subexpnum
 - SuBbUf, 425
- SUBEXPR
 - ftypes.h, 1434
- SUBEXPRESSION
 - ftypes.h, 1387
- SUBEXPSIZE
 - ftypes.h, 1419
- subindex
 - TaBIeBAsE, 446
- SUBMANY
 - ftypes.h, 1432
- SUBMASK
 - ftypes.h, 1432
- SUBMULTI
 - ftypes.h, 1431
- SUBONCE
 - ftypes.h, 1431
- SUBONLY
 - ftypes.h, 1431
- SubPLon
 - declare.h, 862
 - reken.c, 2550
- SUBROUTINEFILE
 - ftypes.h, 1365
- SUBSELECT
 - ftypes.h, 1432
- SubsInAll
 - declare.h, 964
 - pattern.c, 2167
- SUBSORT
 - ftypes.h, 1428
- Substitute
 - declare.h, 862
 - pattern.c, 2167
- subsubveto
 - N_const, 258
- SUBTERMUSED1
 - ftypes.h, 1419
- SUBTERMUSED2
 - ftypes.h, 1419
- SUBVECTOR
 - ftypes.h, 1432
- sum_results
 - tree_node, 469
- SUMF1
 - ftypes.h, 1390
- SUMF2
 - ftypes.h, 1390
- SUMMEDIND
 - ftypes.h, 1427
- sumnum
 - M_const, 188
- SUMNUM1
 - ftypes.h, 1415
- SUMNUM2
 - ftypes.h, 1415
- sumpnum
 - M_const, 188
- SumTime
 - M_const, 176
- sVec
 - STOREHEADER, 419
- SWAP
 - parallel.c, 2057
- SWITCH, 426
 - caseoffset, 427
 - defaultcase, 426
 - endswitch, 427
 - iflevel, 427
 - maxcase, 427
 - mincase, 427
 - nestingsum, 428
 - numcases, 427

- padding, 428
- table, 426
- tablesize, 427
- typetable, 427
- whilelevel, 428
- SwitchArray
 - C_const, 47
- SwitchHeap
 - C_const, 47
- SwitchInArray
 - C_const, 68
- SwitchLevel
 - C_const, 67
- SwitchSplitMerge
 - declare.h, 1015
 - if.c, 1648
- SwitchSplitMergeRec
 - declare.h, 1015
 - if.c, 1648
- SWITCHTABLE, 428
 - compbuffer, 429
 - ncase, 428
 - value, 428
- swmes
 - FixedGlobals, 135
- symb
 - ReNuMbEr, 384
- SYMBOL
 - ftypes.h, 1386
- SyMbOl, 429
 - complex, 430
 - dimension, 430
 - flags, 430
 - maxpower, 429
 - minpower, 429
 - name, 429
 - namesize, 430
 - node, 430
 - number, 430
- SYMBOL_
 - ftypes.h, 1426
- SymbolList
 - C_const, 44
- SymbolNormalize
 - declare.h, 980
 - normal.c, 1869
- SYMBOLONLY
 - ftypes.h, 1374
- Symbols
 - C_const, 46
- symbols
 - variable.h, 3190
- SymChangeFlag
 - C_const, 69
- SymFind
 - declare.h, 862
 - function.c, 1472
- SymGen
 - declare.h, 862
 - function.c, 1472
- symmet
 - FUN_INFO, 142
- symmetr.c, 2923, 2926
 - Distribute, 2924
 - FunMatchCy, 2925
 - FunMatchSy, 2925
 - MatchArgument, 2925
 - MatchCy, 2925
 - MatchE, 2924
 - Permute, 2924
 - PermuteP, 2924
- SYMMETRIC
 - ftypes.h, 1431
- symmetric
 - FuNcTiOn, 145
- SYMMETRIZE
 - ftypes.h, 1415
- Symmetrize
 - declare.h, 863
 - function.c, 1471
- symminfo
 - FuNcTiOn, 144
- symnum
 - ReNuMbEr, 384
- SYMTONUM
 - ftypes.h, 1415
- SYMTOSUB
 - ftypes.h, 1415
- SYMTOSYM
 - ftypes.h, 1415
- SynchFile
 - declare.h, 885
 - tools.c, 3076
- T
 - AllGlobals, 11
- T_const, 431
 - aebufnum, 439
 - allbufnum, 439
 - bernoullis, 435
 - BrackBuf, 434
 - bracketindexflag, 440
 - bracketinfo, 437
 - CacheNumberMemHeap, 437
 - CacheNumberMemMax, 440
 - CacheNumberMemTop, 440
 - comfun, 442
 - comind, 442
 - comnum, 442
 - comsym, 441
 - dummysubexp, 441
 - ebufnum, 438
 - factorials, 435
 - fbufnum, 439
 - fromindex, 444
 - FunArg, 442
 - idallflag, 439

- idallmaxnum, [439](#)
- idallnum, [439](#)
- InNumMem, [438](#)
- inprimelist, [444](#)
- LeaveNegative, [441](#)
- ListPoly, [437](#)
- ListSymbols, [437](#)
- ListSymbolsSize, [440](#)
- locwildvalue, [442](#)
- IWorkPointer, [438](#)
- IWorkSpace, [434](#)
- mBer, [443](#)
- mfac, [438](#)
- MinVecArg, [442](#)
- mulpat, [442](#)
- nBer, [443](#)
- Nest, [433](#)
- NestPoin, [433](#)
- NestStop, [433](#)
- nfac, [443](#)
- NumberMemHeap, [436](#)
- NumberMemMax, [440](#)
- NumberMemTop, [440](#)
- NumListSymbols, [441](#)
- NumMem, [437](#)
- numpoly, [441](#)
- optentimes, [440](#)
- partitions, [437](#)
- pBer, [436](#)
- pfac, [436](#)
- PolyAct, [443](#)
- posWorkPointer, [438](#)
- posWorkSpace, [434](#)
- previousEfactor, [436](#)
- primelist, [436](#)
- proexp, [442](#)
- pWorkPointer, [438](#)
- pWorkSpace, [434](#)
- RecFlag, [443](#)
- RepCount, [435](#)
- RepTop, [435](#)
- S0, [433](#)
- sBer, [438](#)
- setexterntopo, [444](#)
- setinterntopo, [444](#)
- sfact, [438](#)
- sizeprimelist, [444](#)
- small_power, [435](#)
- small_power_maxn, [441](#)
- small_power_maxx, [441](#)
- small_power_n, [435](#)
- SS, [433](#)
- StoreCache, [434](#)
- StoreCacheAlloc, [434](#)
- TermMemHeap, [436](#)
- TermMemMax, [439](#)
- TermMemTop, [440](#)
- TMaddr, [436](#)
- TMbuff, [443](#)
- TMdolfac, [443](#)
- TMout, [443](#)
- TopologiesLevel, [444](#)
- TopologiesOptions, [444](#)
- TopologiesStart, [437](#)
- TopologiesTerm, [437](#)
- TrimPower, [441](#)
- WildArgTaken, [435](#)
- WildcardBufferSize, [439](#)
- WildMask, [436](#)
- WorkPointer, [435](#)
- WorkSpace, [434](#)
- WorkTop, [434](#)
- tabl
 - FuNcTiOn, [144](#)
- table
 - SWITCH, [426](#)
- TaBIEbAsE, [445](#)
 - current, [446](#)
 - fillpoint, [446](#)
 - name, [446](#)
 - numtables, [446](#)
 - subindex, [446](#)
 - tablenumbers, [446](#)
- TABLEBASEFILE
 - ftypes.h, [1366](#)
- TableBaseList
 - C_const, [45](#)
- tablebases
 - variable.h, [3190](#)
- TaBIEbAsEsUbInDeX, [447](#)
 - size, [447](#)
 - where, [447](#)
- tablecheck
 - C_const, [56](#)
- TABLEEXTENSION
 - fsizes.h, [1345](#)
- tablefilling
 - C_const, [60](#)
- TABLEFUNCTION
 - ftypes.h, [1394](#)
- tablenameefill
 - dbase, [79](#)
- tablenames
 - dbase, [80](#)
- tablenamessize
 - dbase, [79](#)
- tablenum
 - TaBIEs, [453](#)
- tablenumber
 - objects, [290](#)
- tablenumbers
 - TaBIEbAsE, [446](#)
- tablepointers
 - TaBIEs, [449](#)
- TableReset
 - declare.h, [827](#)

- tables.c, [2958](#)
- TABLES
 - structs.h, [2895](#)
- TaBIEs, [448](#)
 - argtail, [450](#)
 - boomlijst, [450](#)
 - bounds, [452](#)
 - buffers, [450](#)
 - bufferfill, [453](#)
 - bufferssize, [453](#)
 - bunum, [453](#)
 - defined, [451](#)
 - flags, [449](#)
 - MaxTreeSize, [453](#)
 - mdefined, [451](#)
 - mm, [449](#)
 - mode, [454](#)
 - numdummies, [454](#)
 - numind, [451](#)
 - numtree, [452](#)
 - pattern, [449](#)
 - prototype, [449](#)
 - prototypeSize, [451](#)
 - reserved, [451](#)
 - rootnum, [452](#)
 - spare, [450](#)
 - sparse, [452](#)
 - strict, [452](#)
 - tablenum, [453](#)
 - tablepointers, [449](#)
 - totind, [450](#)
- tables.c, [2953](#), [2959](#)
 - Apply, [2958](#)
 - ApplyExec, [2958](#)
 - ApplyReset, [2958](#)
 - CheckTableDeclarations, [2956](#)
 - ClearTableTree, [2954](#)
 - CoApply, [2957](#)
 - CoTableBase, [2955](#)
 - CoTBaddto, [2955](#)
 - CoTBaudit, [2956](#)
 - CoTBCleanup, [2957](#)
 - CoTBcreate, [2955](#)
 - CoTBenter, [2956](#)
 - CoTBhelp, [2957](#)
 - CoTBload, [2956](#)
 - CoTBoff, [2957](#)
 - CoTBon, [2956](#)
 - CoTBopen, [2955](#)
 - CoTBreplace, [2957](#)
 - CoTBuse, [2957](#)
 - CoTestUse, [2956](#)
 - DoTableExpansion, [2954](#)
 - FindTableTree, [2954](#)
 - FindTB, [2955](#)
 - FlipTable, [2955](#)
 - helpTB, [2959](#)
 - InsTableTree, [2954](#)
 - RedoTableTree, [2954](#)
 - ReleaseTB, [2958](#)
 - ReWorkT, [2957](#)
 - SpareTable, [2955](#)
 - TableReset, [2958](#)
 - TestUse, [2956](#)
- TABSIZE
 - declare.h, [797](#)
- tablesize
 - SWITCH, [427](#)
- TABLESTUB
 - ftypes.h, [1395](#)
- tabstring
 - O_const, [282](#)
- tadblock
 - MGraph, [216](#)
- TADPOLE
 - ftypes.h, [1456](#)
- tadpole
 - MGraph, [216](#)
- tag
 - PF_BUFFER, [335](#)
- TakeArgContent
 - argument.c, [480](#)
 - declare.h, [918](#)
- TakeContent
 - declare.h, [999](#)
 - ratio.c, [2499](#)
- TakeDollarContent
 - declare.h, [955](#)
 - dollar.c, [1090](#)
- TakeExtraSymbols
 - declare.h, [999](#)
 - ratio.c, [2499](#)
- TakeIDfunction
 - declare.h, [965](#)
 - pattern.c, [2168](#)
- TakeLongRoot
 - declare.h, [960](#)
 - reken.c, [2555](#)
- TakeModulus
 - declare.h, [863](#)
 - reken.c, [2556](#)
- TakeNormalModulus
 - declare.h, [863](#)
 - reken.c, [2556](#)
- TakeOneLine
 - lus.c, [1681](#)
- TAKEPATH
 - startup.c, [2793](#)
- TakeRatRoot
 - declare.h, [960](#)
 - reken.c, [2549](#)
- TakeSymbolContent
 - declare.h, [998](#)
 - ratio.c, [2501](#)
- TAKETRACE
 - ftypes.h, [1414](#)

- TalToLine
 - declare.h, [863](#)
 - sch.c, [2647](#)
- TANFUNCTION
 - ftypes.h, [1396](#)
- TANHFUNCTION
 - ftypes.h, [1397](#)
- TCOMMA
 - ftypes.h, [1379](#)
- TCONJUGATE
 - ftypes.h, [1381](#)
- tCouple
 - SProcess, [414](#)
- TDELTA
 - ftypes.h, [1378](#)
- TDIVIDE
 - ftypes.h, [1380](#)
- TDOLLAR
 - ftypes.h, [1378](#)
- TDOT
 - ftypes.h, [1379](#)
- TDOTPRODUCT
 - ftypes.h, [1377](#)
- TDUBIOUS
 - ftypes.h, [1378](#)
- TeInFun
 - N_const, [259](#)
- TELLFILE
 - declare.h, [810](#)
 - tools.c, [3076](#)
- TellFile
 - declare.h, [885](#)
 - tools.c, [3076](#)
- TempDir
 - M_const, [171](#)
- TempSortDir
 - M_const, [171](#)
- EMPTY
 - ftypes.h, [1381](#)
- TENDOFIT
 - ftypes.h, [1380](#)
- tensor
 - FUN_INFO, [142](#)
- TENSORFUNCTION
 - ftypes.h, [1425](#)
- tensorList
 - O_const, [284](#)
- TENVEC
 - ftypes.h, [1415](#)
- TenVec
 - declare.h, [864](#)
 - opera.c, [1952](#)
- TenVecFind
 - declare.h, [864](#)
 - opera.c, [1952](#)
- TePos
 - R_const, [380](#)
- terfirstcomm
 - N_const, [249](#)
- term
 - TERMINFO, [455](#)
- term_compare
 - optimize.cc, [1988](#)
- TermAssign
 - declare.h, [952](#)
 - dollar.c, [1085](#)
- termbuf
 - O_const, [282](#)
- termbuffer
 - N_const, [252](#)
- TERMEXTRAWORDS
 - tools.c, [3071](#)
- TermFree
 - declare.h, [805](#)
- TERMFUNCTION
 - ftypes.h, [1394](#)
- termfunnum
 - M_const, [188](#)
- TERMINAL
 - pre.c, [2311](#)
- Terminate
 - declare.h, [798](#)
- TerminatImpl
 - declare.h, [877](#)
 - startup.c, [2795](#)
- TERMINFO, [454](#)
 - currentMODEL, [455](#)
 - currentModel, [455](#)
 - diaoffset, [455](#)
 - externalset, [456](#)
 - flags, [456](#)
 - internalset, [456](#)
 - legcouple, [455](#)
 - level, [455](#)
 - numdia, [455](#)
 - numextern, [456](#)
 - numtopo, [455](#)
 - term, [455](#)
- termlevel
 - C_const, [65](#)
- TermMalloc
 - declare.h, [804](#)
- TermMallocAddMemory
 - declare.h, [989](#)
 - tools.c, [3083](#)
- TermMemHeap
 - T_const, [436](#)
- TermMemMax
 - T_const, [439](#)
- TERMMEMSTARTNUM
 - tools.c, [3070](#)
- TermMemTop
 - T_const, [440](#)
- termp
 - TrAcEn, [459](#)
 - TrAcEs, [462](#)

- TermRenummer
 - declare.h, [864](#)
 - store.c, [2826](#)
- terms
 - poly, [361](#)
- Terms2InSmall
 - sOrT, [403](#)
- termscopy
 - poly, [348](#)
- TERMSINBRACKET
 - ftypes.h, [1395](#)
- TermsInBracket
 - declare.h, [972](#)
 - execute.c, [1155](#)
- termsinbracket
 - BrAcKeTiNdEx, [32](#)
- TermsInDollar
 - declare.h, [967](#)
 - dollar.c, [1087](#)
- TERMSINEXPR
 - ftypes.h, [1394](#)
- TermsInExpression
 - declare.h, [967](#)
 - execute.c, [1155](#)
- TermsInSmall
 - sOrT, [402](#)
- termssize
 - NeStInG, [272](#)
- TermsLeft
 - sOrT, [403](#)
- termstortstack
 - C_const, [47](#)
- termstack
 - C_const, [47](#)
- termsumcheck
 - C_const, [50](#)
- terstart
 - N_const, [249](#)
- terstop
 - N_const, [249](#)
- TestArgNum
 - declare.h, [991](#)
 - transform.c, [3138](#)
- TestDoLoop
 - declare.h, [1001](#)
 - dollar.c, [1089](#)
- TestDrop
 - declare.h, [864](#)
 - execute.c, [1152](#)
- TestEndDoLoop
 - declare.h, [1001](#)
 - dollar.c, [1089](#)
- TestFloat
 - float.c, [1288](#)
- TestFunFlag
 - declare.h, [980](#)
 - normal.c, [1869](#)
- TestMatch
 - declare.h, [865](#)
 - pattern.c, [2166](#)
- TestName
 - declare.h, [882](#)
 - names.c, [1828](#)
- TestPartitions
 - declare.h, [823](#)
 - reshuf.c, [2607](#)
- TestSelect
 - declare.h, [964](#)
 - pattern.c, [2167](#)
- TestSub
 - declare.h, [866](#)
 - proces.c, [2421](#)
- TestTables
 - compiler.c, [714](#)
 - declare.h, [948](#)
- TestTerm
 - declare.h, [989](#)
 - tools.c, [3087](#)
- TestUse
 - declare.h, [826](#)
 - tables.c, [2956](#)
- TestValue
 - C_const, [52](#)
- TeSuOut
 - N_const, [259](#)
- TEXPRESSION
 - ftypes.h, [1377](#)
- TFLOAT
 - ftypes.h, [1382](#)
- TFUN
 - ftypes.h, [1457](#)
- TFUN1
 - ftypes.h, [1457](#)
- TFUNCLOSE
 - ftypes.h, [1379](#)
- TFUNCTION
 - ftypes.h, [1377](#)
- TFUNOPEN
 - ftypes.h, [1379](#)
- TGENINDEX
 - ftypes.h, [1381](#)
- TheDefine
 - declare.h, [895](#)
 - pre.c, [2314](#)
- TheErrorMessage
 - ratio.c, [2503](#)
- thefunction
 - SHvariables, [397](#)
- theposition
 - N_const, [248](#)
- THETA
 - ftypes.h, [1393](#)
- THETA2
 - ftypes.h, [1393](#)
- TheUndefine
 - declare.h, [895](#)

- pre.c, [2316](#)
- ThreadBalancing
 - C_const, [59](#)
- ThreadBucketSize
 - C_const, [53](#)
- threads.c, [2986](#)
- THREADSCRATCHOUTSIZE
 - fsizes.h, [1346](#)
- THREADSCRATCHSIZE
 - fsizes.h, [1346](#)
- THREADSDEBUG
 - ftypes.h, [1370](#)
- ThreadsFlag
 - C_const, [58](#)
- ThreadSortFileSynch
 - C_const, [59](#)
- ThreadStats
 - C_const, [58](#)
- time0
 - Options, [307](#)
- time1
 - Options, [307](#)
- TimeChildren
 - declare.h, [866](#)
 - tools.c, [3086](#)
- TimeCPU
 - declare.h, [866](#)
 - tools.c, [3086](#)
- TimeLimit
 - M_const, [176](#)
- timeout
 - X_const, [475](#)
- TimeWallClock
 - declare.h, [867](#)
 - tools.c, [3086](#)
- TINDEX
 - ftypes.h, [1377](#)
- tlistbuf
 - N_const, [253](#)
- tlistsize
 - N_const, [257](#)
- TMaddr
 - T_const, [436](#)
- TMbuff
 - T_const, [443](#)
- TMdolfac
 - T_const, [443](#)
- TMINUS
 - ftypes.h, [1380](#)
- TMout
 - T_const, [443](#)
- TMPPOLYFUN
 - ftypes.h, [1388](#)
- TMULTIPLY
 - ftypes.h, [1379](#)
- TNOT
 - ftypes.h, [1380](#)
- TNUMBER
 - ftypes.h, [1378](#)
- TNUMBER1
 - ftypes.h, [1381](#)
- to_argument_mpoly
 - flintinterface.h, [1270](#)
- to_argument_poly
 - flintinterface.h, [1270](#), [1271](#)
- to_coefficient_list
 - poly, [353](#)
- to_string
 - poly, [354](#)
- TOBEFACTORED
 - ftypes.h, [1430](#)
- ToBelInFactors
 - C_const, [70](#)
- TOBEUNFACTORED
 - ftypes.h, [1430](#)
- TODISK
 - minos.h, [1744](#)
- ToFast
 - declare.h, [891](#)
 - tools.c, [3085](#)
- TOFLOAT
 - ftypes.h, [1401](#)
- ToFloat
 - float.c, [1290](#)
- ToGeneral
 - declare.h, [891](#)
 - tools.c, [3085](#)
- tohunt
 - N_const, [256](#)
- token.c, [3040](#), [3043](#)
 - CHECKPOLY, [3041](#)
 - simp1token, [3041](#)
 - simp2token, [3042](#)
 - simp3atoken, [3042](#)
 - simp3btoken, [3042](#)
 - simp4token, [3042](#)
 - simp5token, [3042](#)
 - simp6token, [3042](#)
 - simpwtoken, [3041](#)
 - tokenize, [3041](#)
 - ttypes, [3043](#)
 - WriteTokens, [3041](#)
- tokenarglevel
 - C_const, [51](#)
- tokenize
 - declare.h, [946](#)
 - token.c, [3041](#)
- tokens
 - C_const, [50](#)
- TokensWriteFlag
 - C_const, [57](#)
- TOKENTOLINE
 - declare.h, [790](#)
- TokenToLine
 - declare.h, [867](#)
 - sch.c, [2647](#)

- TOLONG
 - declare.h, [803](#)
- TOLYNDON
 - ftypes.h, [1445](#)
- TOLYNDONR
 - ftypes.h, [1446](#)
- toMNodeClass
 - SProcess, [410](#)
- tools.c, [3068](#), [3089](#)
 - AddToString, [3079](#)
 - AreArgsEqual, [3085](#)
 - CacheNumberMallocAddMemory, [3083](#)
 - CloseChannel, [3077](#)
 - CloseFile, [3074](#)
 - CloseStream, [3073](#)
 - CmpArray, [3078](#)
 - CompareArgs, [3085](#)
 - CompArg, [3086](#)
 - ConWord, [3078](#)
 - CopyFile, [3074](#)
 - Crash, [3087](#)
 - CreateFile, [3074](#)
 - CreateHandle, [3075](#)
 - CreateLogFile, [3074](#)
 - CreateStream, [3073](#)
 - DistrN, [3087](#)
 - DODOUBLE, [3071](#)
 - DOEXPAND, [3071](#)
 - DoubleBuffer, [3084](#)
 - DoubleList, [3084](#)
 - DoubleLList, [3084](#)
 - EndOfToken, [3079](#)
 - ExpandBuffer, [3084](#)
 - filelist, [3088](#)
 - filelistsize, [3088](#)
 - FILLVALUE, [3070](#)
 - FlushFile, [3076](#)
 - From0List, [3083](#)
 - FromList, [3083](#)
 - FromVarList, [3083](#)
 - GetAppendChannel, [3077](#)
 - GetChannel, [3077](#)
 - GetFromStream, [3072](#)
 - GetPosFile, [3076](#)
 - GetStreamPosition, [3073](#)
 - iexp, [3084](#)
 - iniTools, [3082](#)
 - IsLikeVector, [3085](#)
 - LoadInputFile, [3072](#)
 - LocateFile, [3072](#)
 - LongCopy, [3081](#)
 - LongLongCopy, [3081](#)
 - LookInStream, [3072](#)
 - M_check1, [3082](#)
 - M_free, [3082](#)
 - M_print, [3083](#)
 - MakeDate, [3081](#)
 - Malloc1, [3082](#)
 - NUMBEREXTRAWORDS, [3071](#)
 - NumberMallocAddMemory, [3083](#)
 - NUMBERMEMSTARTNUM, [3071](#)
 - NumCopy, [3081](#)
 - numinfilelist, [3088](#)
 - NumToStr, [3078](#)
 - OpenAddFile, [3074](#)
 - OpenFile, [3073](#)
 - OpenStream, [3072](#)
 - PositionStream, [3073](#)
 - ReadFile, [3075](#)
 - ReadFromStream, [3072](#)
 - ReadPosFile, [3075](#)
 - ReadSnum, [3081](#)
 - ReOpenFile, [3074](#)
 - ReverseStatements, [3073](#)
 - SeekFile, [3075](#)
 - set_del, [3082](#)
 - set_in, [3081](#)
 - set_set, [3081](#)
 - set_sub, [3082](#)
 - SetPosFile, [3076](#)
 - SkipField, [3080](#)
 - StartFiles, [3073](#)
 - StrCmp, [3077](#)
 - strDup1, [3079](#)
 - StrHICmp, [3078](#)
 - StrICmp, [3077](#)
 - StrICont, [3078](#)
 - StrLen, [3078](#)
 - SynchFile, [3076](#)
 - TELLFILE, [3076](#)
 - TellFile, [3076](#)
 - TERMEXTRAWORDS, [3071](#)
 - TermMallocAddMemory, [3083](#)
 - TERMMEMSTARTNUM, [3070](#)
 - TestTerm, [3087](#)
 - TimeChildren, [3086](#)
 - TimeCPU, [3086](#)
 - TimeWallClock, [3086](#)
 - ToFast, [3085](#)
 - ToGeneral, [3085](#)
 - ToPolyFunGeneral, [3085](#)
 - ToToken, [3080](#)
 - TruncateFile, [3077](#)
 - UpdateMaxSize, [3077](#)
 - WriteFile, [3088](#)
 - WriteFileToFile, [3075](#)
 - WriteString, [3079](#)
 - WriteUnfinString, [3079](#)
- toOrbits
 - MOrbits, [240](#)
- TOOUTPUT
 - ftypes.h, [1373](#)
- Top
 - CbUf, [71](#)
- top
 - Stream, [422](#)

- TOPBITONLY
 - form3.h, [1328](#)
- TOPLONGBITONLY
 - form3.h, [1328](#)
- topnumber
 - dbase, [79](#)
- TOPO
 - ftypes.h, [1400](#)
- TOPOLOGIES
 - ftypes.h, [1400](#)
- TopologiesLevel
 - T_const, [444](#)
- TOPOLOGIESONLY
 - ftypes.h, [1453](#)
- TopologiesOptions
 - T_const, [444](#)
- TopologiesStart
 - T_const, [437](#)
- TopologiesTerm
 - T_const, [437](#)
- ToPolyFunGeneral
 - declare.h, [891](#)
 - tools.c, [3085](#)
- TOPOLYNOMIALFLAG
 - ftypes.h, [1447](#)
- topolynomialflag
 - C_const, [61](#)
- ToPoTyPe, [456](#)
 - cldeg, [457](#)
 - clext, [457](#)
 - clnum, [457](#)
 - cmaxd, [458](#)
 - cmind, [458](#)
 - ncl, [458](#)
 - nvert, [458](#)
 - opt, [457](#)
 - vert, [457](#)
 - vertmax, [457](#)
- toppos
 - StOrEcAcHe, [416](#)
- topsubexpbuffers
 - compiler.c, [715](#)
- toptokens
 - C_const, [50](#)
- TORAT
 - ftypes.h, [1401](#)
- ToRat
 - float.c, [1290](#)
- ToStorage
 - declare.h, [867](#)
 - store.c, [2825](#)
- tostring
 - factorized_poly, [126](#)
- total_degree
 - poly, [350](#)
- totalc
 - EGraph, [112](#)
- totalnumberofthreads
 - M_const, [179](#)
- totems
 - sort.c, [2725](#)
- totind
 - TabIEs, [450](#)
- ToToken
 - declare.h, [909](#)
 - tools.c, [3080](#)
- TPLUS
 - ftypes.h, [1380](#)
- TPOWER
 - ftypes.h, [1380](#)
- TPOWER1
 - ftypes.h, [1381](#)
- Trace4
 - declare.h, [867](#)
 - opera.c, [1950](#)
- Trace4Gen
 - declare.h, [867](#)
 - opera.c, [1950](#)
- Trace4no
 - declare.h, [868](#)
 - opera.c, [1950](#)
- tracebackflag
 - M_const, [187](#)
- TraceFind
 - declare.h, [868](#)
 - opera.c, [1951](#)
- TRACEN
 - structs.h, [2896](#)
- TrAcEn, [458](#)
 - accu, [459](#)
 - accup, [459](#)
 - allsign, [460](#)
 - factor, [460](#)
 - inlist, [459](#)
 - level, [460](#)
 - num, [459](#)
 - perm, [459](#)
 - sgn, [459](#)
 - termp, [459](#)
- TraceN
 - declare.h, [868](#)
 - opera.c, [1951](#)
- TraceNgen
 - declare.h, [868](#)
 - opera.c, [1951](#)
- TraceNno
 - declare.h, [868](#)
 - opera.c, [1951](#)
- TRACES
 - structs.h, [2896](#)
- TrAcEs, [460](#)
 - a3, [464](#)
 - a4, [464](#)
 - accu, [461](#)
 - accup, [461](#)
 - ad, [464](#)

- allsign, [465](#)
- e3, [462](#)
- e4, [463](#)
- eers, [463](#)
- factor, [465](#)
- finalstep, [466](#)
- gamm, [464](#)
- gamma5, [465](#)
- inlist, [462](#)
- j3, [462](#)
- j4, [462](#)
- kstep, [464](#)
- lc3, [464](#)
- lc4, [465](#)
- level, [465](#)
- mdel, [463](#)
- mdum, [464](#)
- mepf, [463](#)
- nt3, [462](#)
- nt4, [462](#)
- num, [465](#)
- pdel, [463](#)
- pepf, [463](#)
- perm, [462](#)
- sgn, [463](#)
- sign1, [465](#)
- sign2, [465](#)
- stap, [463](#)
- step1, [464](#)
- termp, [462](#)
- Traces
 - declare.h, [868](#)
 - opera.c, [1951](#)
- tracestack
 - N_const, [251](#)
- TransEname
 - C_const, [53](#)
- TransferBuffer
 - declare.h, [965](#)
 - pattern.c, [2168](#)
- transFlag
 - O_const, [287](#)
- transform.c, [3134](#), [3139](#)
 - CoTransform, [3135](#)
 - FindRange, [3138](#)
 - PutArgInScratch, [3138](#)
 - ReadRange, [3138](#)
 - RunAddArg, [3136](#)
 - RunCycle, [3136](#)
 - RunDecode, [3135](#)
 - RunDedup, [3136](#)
 - RunDropArg, [3137](#)
 - RunEncode, [3135](#)
 - RunExplode, [3136](#)
 - RunHtoZArg, [3138](#)
 - RunImplode, [3135](#)
 - RunIsLyndon, [3137](#)
 - RunMulArg, [3137](#)
 - RunPermute, [3136](#)
 - RunReplace, [3135](#)
 - RunReverse, [3136](#)
 - RunSelectArg, [3137](#)
 - RunToLyndon, [3137](#)
 - RunTransform, [3135](#)
 - RunZtoHArg, [3137](#)
 - TestArgNum, [3138](#)
- TransformRational
 - declare.h, [1011](#)
 - dict.c, [1066](#)
- TranslateExpression
 - declare.h, [967](#)
 - dollar.c, [1088](#)
- TRAP SIGNALS
 - unix.h, [3181](#)
- TreatPolyRatFun
 - declare.h, [982](#)
 - normal.c, [1868](#)
- tree, [466](#)
 - blnce, [467](#)
 - left, [466](#)
 - parent, [466](#)
 - right, [467](#)
 - sOrT, [401](#)
 - usage, [467](#)
 - value, [467](#)
- tree_node, [468](#)
 - childs, [469](#)
 - finished, [469](#)
 - num_visits, [469](#)
 - operator=, [469](#)
 - sum_results, [469](#)
 - tree_node, [468](#)
 - var, [469](#)
- Trick
 - declare.h, [869](#)
 - opera.c, [1950](#)
- TrimPower
 - T_const, [441](#)
- TRUE_EXPR
 - pre.c, [2311](#)
- TruncateFile
 - declare.h, [886](#)
 - tools.c, [3077](#)
- TryDo
 - declare.h, [869](#)
 - reshuf.c, [2607](#)
- TryEnvironment
 - declare.h, [977](#)
 - setfile.c, [2687](#)
- TryFileSetups
 - declare.h, [965](#)
 - setfile.c, [2687](#)
- TryRecover
 - declare.h, [790](#)
- tryterm
 - N_const, [261](#)

- TSET
 - ftypes.h, [1377](#)
- TSETCLOSE
 - ftypes.h, [1380](#)
- TSETDOL
 - ftypes.h, [1382](#)
- TSETNUM
 - ftypes.h, [1381](#)
- TSETOPEN
 - ftypes.h, [1380](#)
- TSGAMMA
 - ftypes.h, [1381](#)
- TSUBEXP
 - ftypes.h, [1378](#)
- TSYMBOL
 - ftypes.h, [1377](#)
- ttypes
 - token.c, [3043](#)
- TVECTOR
 - ftypes.h, [1377](#)
- TWILDARG
 - ftypes.h, [1379](#)
- TWILDCARD
 - ftypes.h, [1378](#)
- TwoExprCompare
 - declare.h, [968](#)
 - dollar.c, [1088](#)
- type
 - ARGBUFFER, [15](#)
 - DICTIONARY_ELEMENT, [84](#)
 - DoLIaR, [88](#)
 - DoLoOp, [91](#)
 - FaCdOILaR, [125](#)
 - fixedset, [137](#)
 - InDeX, [149](#)
 - KEYWORD, [162](#)
 - KEYWORDV, [163](#)
 - MoDoPtDoLIaR, [238](#)
 - NaMeNode, [264](#)
 - optimization, [293](#)
 - PaRtlcLe, [327](#)
 - PF_BUFFER, [335](#)
 - PNodeClass, [341](#)
 - SeTs, [388](#)
 - SETUPPARAMETERS, [390](#)
 - sOrT, [404](#)
 - StreaM, [423](#)
 - VeRtEx, [473](#)
- TYPEADJUSTBOUNDS
 - ftypes.h, [1410](#)
- TYPEALLLOOPS
 - ftypes.h, [1414](#)
- TYPEALLPATHS
 - ftypes.h, [1414](#)
- TYPEAPPLY
 - ftypes.h, [1411](#)
- TYPEAPPLYRESET
 - ftypes.h, [1411](#)
- TYPEARG
 - ftypes.h, [1407](#)
- TYPEARGEXPLODE
 - ftypes.h, [1412](#)
- TYPEARGHEADSIZE
 - ftypes.h, [1419](#)
- TYPEARGIMPLODE
 - ftypes.h, [1412](#)
- TYPEARGTOEXTRASymbol
 - ftypes.h, [1413](#)
- TYPEASSIGN
 - ftypes.h, [1409](#)
- TYPECANONICALIZE
 - ftypes.h, [1413](#)
- TYPECHAININ
 - ftypes.h, [1411](#)
- TYPECHAINOUT
 - ftypes.h, [1411](#)
- TYPECHISHOLM
 - ftypes.h, [1407](#)
- TYPECLEARUSERFLAG
 - ftypes.h, [1414](#)
- TYPECOUNT
 - ftypes.h, [1406](#)
- TYPEDENOMINATORS
 - ftypes.h, [1412](#)
- TYPEDETCURDUM
 - ftypes.h, [1410](#)
- TYPEDISCARD
 - ftypes.h, [1406](#)
- TYPEDOLOOP
 - ftypes.h, [1413](#)
- TYPEDROPCOEFFICIENT
 - ftypes.h, [1412](#)
- TYPEDROPSYMBOLS
 - ftypes.h, [1413](#)
- TYPEELIF
 - ftypes.h, [1407](#)
- TYPEELSE
 - ftypes.h, [1406](#)
- TYPEENDDOLOOP
 - ftypes.h, [1413](#)
- TYPEENDIF
 - ftypes.h, [1407](#)
- TYPEENDREPEAT
 - ftypes.h, [1406](#)
- TYPEENDSWITCH
 - ftypes.h, [1414](#)
- TYPEEXIT
 - ftypes.h, [1408](#)
- TYPEEXPRESSION
 - ftypes.h, [1405](#)
- TYPEFACTARG
 - ftypes.h, [1409](#)
- TYPEFACTARG2
 - ftypes.h, [1409](#)
- TYPEFACTOR
 - ftypes.h, [1411](#)

- TYPEFINDLOOP
 - ftypes.h, [1409](#)
- TYPEFPRINT
 - ftypes.h, [1408](#)
- TYPEFROMPOLYNOMIAL
 - ftypes.h, [1412](#)
- typeGName
 - Particle, [330](#)
- TYPEGOTO
 - ftypes.h, [1406](#)
- TYPEIDNEW
 - ftypes.h, [1405](#)
- TYPEIDOLD
 - ftypes.h, [1405](#)
- TYPEIF
 - ftypes.h, [1406](#)
- TYPEINEXPRESSION
 - ftypes.h, [1410](#)
- TYPEINSIDE
 - ftypes.h, [1410](#)
- TYPEISFUN
 - ftypes.h, [1382](#)
- TYPEISMYSTERY
 - ftypes.h, [1382](#)
- TYPEISSUB
 - ftypes.h, [1382](#)
- TYPEMERGE
 - ftypes.h, [1411](#)
- TYPEMULT
 - ftypes.h, [1406](#)
- typeName
 - Particle, [330](#)
- TYPENORM
 - ftypes.h, [1407](#)
- TYPENORM2
 - ftypes.h, [1407](#)
- TYPENORM3
 - ftypes.h, [1408](#)
- TYPENORM4
 - ftypes.h, [1411](#)
- TYPEOPERATION
 - ftypes.h, [1405](#)
- TYPEPRINT
 - ftypes.h, [1408](#)
- TYPEPUTINSIDE
 - ftypes.h, [1413](#)
- TYPEREDEFPRE
 - ftypes.h, [1408](#)
- TYPERENUMBER
 - ftypes.h, [1409](#)
- TYPEREPEAT
 - ftypes.h, [1406](#)
- TYPEREVERSE
 - ftypes.h, [1407](#)
- TYPESETEXIT
 - ftypes.h, [1408](#)
- TYPESETUSERFLAG
 - ftypes.h, [1414](#)
- TYPESORT
 - ftypes.h, [1410](#)
- TYPESPLITARG
 - ftypes.h, [1408](#)
- TYPESPLITARG2
 - ftypes.h, [1408](#)
- TYPESPLITFIRSTARG
 - ftypes.h, [1410](#)
- TYPESPLITLASTARG
 - ftypes.h, [1410](#)
- TYPESTUFFLE
 - ftypes.h, [1412](#)
- TYPESUM
 - ftypes.h, [1407](#)
- TYPESUMFIX
 - ftypes.h, [1409](#)
- TYPESWITCH
 - ftypes.h, [1413](#)
- typetable
 - SWITCH, [427](#)
- TYPETERM
 - ftypes.h, [1410](#)
- TYPETESTUSE
 - ftypes.h, [1411](#)
- TYPETOPOLYNOMIAL
 - ftypes.h, [1412](#)
- TYPETOSPECTATOR
 - ftypes.h, [1413](#)
- TYPETRANSFORM
 - ftypes.h, [1412](#)
- TYPETRY
 - ftypes.h, [1409](#)
- TYPEUNRAVEL
 - ftypes.h, [1409](#)
- UBYTE
 - form3.h, [1333](#)
- Uclose
 - form3.h, [1330](#)
- uflags
 - ExPrEsSiOn, [124](#)
- Uflush
 - form3.h, [1330](#)
- Ugetpos
 - form3.h, [1331](#)
- UINT
 - form3.h, [1333](#)
- ULONG
 - form3.h, [1333](#)
- uncompressed
 - objects, [290](#)
- UngetChar
 - declare.h, [791](#)
- UngetFromStream
 - declare.h, [790](#)
- UNHIDE
 - ftypes.h, [1420](#)
- UNHIDEGEXPRESSION
 - ftypes.h, [1422](#)

- UNHIDELEXPRESSION
 - ftypes.h, 1422
- UNIVARIATE_DENSITY_THR
 - flintinterface.cc, 1237
- unix.h, 3180, 3183
 - ALTSEPARATOR, 3182
 - CARRIAGEReturn, 3181
 - LINEFEED, 3181
 - P_term, 3181
 - PATHSEPARATOR, 3182
 - SEPARATOR, 3182
 - TRAPsignals, 3181
 - WITH_ALARM, 3182
 - WITH_ENV, 3182
 - WITHEXTERNALCHANNEL, 3181
 - WITHPIPE, 3181
 - WITHSYSTEM, 3181
- unixfile.c, 3183, 3184
- UNLOCK
 - declare.h, 807
- UNPACK
 - ftypes.h, 1443
- UnPack
 - declare.h, 869
 - reken.c, 2548
- UNPACK_LONG
 - parallel.c, 2058
- UnpackFloat
 - evaluate.c, 1140
 - float.c, 1287
- UNRWLOCK
 - declare.h, 807
- UnsetAllowDelay
 - declare.h, 893
 - pre.c, 2312
- UnSetDictionary
 - declare.h, 1010
 - dict.c, 1068
- UnSetMods
 - declare.h, 872
 - execute.c, 1153
- UnsureDollarMode
 - C_const, 57
- Uopen
 - form3.h, 1330
- updateCandNode
 - Assign, 21
- UpdateMaxSize
 - declare.h, 995
 - tools.c, 3077
- UpdatePositions
 - declare.h, 968
 - execute.c, 1155
- Uread
 - form3.h, 1330
- usage
 - tree, 467
- used
 - sOrT, 399
- USEDFLAG
 - ftypes.h, 1428
- UsedFunction
 - N_const, 250
- UseDictionary
 - dict.c, 1067
- UsedIndex
 - N_const, 250
- UsedOtherFind
 - N_const, 255
- UsedSymbol
 - N_const, 250
- UsedVector
 - N_const, 250
- Useek
 - form3.h, 1331
- UseFindOnly
 - N_const, 255
- usenames
 - NaMeSpAcE, 265
- UserFlags
 - pre.c, 2328
- Usetbuf
 - form3.h, 1330
- Usetpos
 - form3.h, 1331
- Ustdout
 - form3.h, 1332
- Usync
 - form3.h, 1331
- Utell
 - form3.h, 1331
- Utruncate
 - form3.h, 1331
- UWORD
 - form3.h, 1332
- Uwrite
 - form3.h, 1330
- value
 - FaCdOILaR, 125
 - pReVaR, 365
 - SETUPPARAMETERS, 390
 - SWITCHTABLE, 428
 - tree, 467
- values
 - Options, 306
- var
 - KEYWORDV, 163
 - tree_node, 469
- var_map_t
 - flintinterface.h, 1265
- variable.h, 3187, 3197
 - A, 3196
 - AC, 3194
 - AM, 3195
 - AN, 3195
 - AO, 3195

- AP, [3195](#)
- AR, [3195](#)
- AS, [3195](#)
- AT, [3195](#)
- autofunctions, [3193](#)
- autoindices, [3193](#)
- autosymbols, [3193](#)
- autovectors, [3193](#)
- AX, [3195](#)
- BUG, [3194](#)
- cbuf, [3192](#)
- channels, [3192](#)
- chartype, [3188](#)
- Dollars, [3192](#)
- DoLoops, [3189](#)
- Dubious, [3193](#)
- Expressions, [3193](#)
- FG, [3196](#)
- fixedsets, [3196](#)
- functions, [3190](#)
- GlobalFunctions, [3191](#)
- GlobalIndices, [3191](#)
- GlobalSetElements, [3192](#)
- GlobalSets, [3192](#)
- GlobalSymbols, [3191](#)
- GlobalVectors, [3192](#)
- indices, [3190](#)
- MaxDoLoops, [3189](#)
- MaxNumPre, [3189](#)
- MaxProcedures, [3189](#)
- ModOptdollars, [3194](#)
- NumDollars, [3192](#)
- NumDoLoops, [3189](#)
- NumDubious, [3193](#)
- NumExpressions, [3193](#)
- NumFunctions, [3190](#)
- NumIndices, [3190](#)
- NumModOptdollars, [3194](#)
- NumOutputChannels, [3192](#)
- NumPotModdollars, [3194](#)
- NumPre, [3189](#)
- NumProcedures, [3188](#)
- NumSetElements, [3191](#)
- NumSets, [3191](#)
- NumSymbols, [3191](#)
- NumTableBases, [3191](#)
- NumVectors, [3191](#)
- numxsymbol, [3194](#)
- PotModdollars, [3194](#)
- PreVar, [3189](#)
- Procedures, [3188](#)
- SetElements, [3189](#)
- Sets, [3190](#)
- setupfilename, [3196](#)
- symbols, [3190](#)
- tablebases, [3190](#)
- vectors, [3190](#)
- WriteFile, [3196](#)
- xsymbol, [3194](#)
- variables
 - DICTIONARY, [82](#)
 - InDeXeNtRy, [153](#)
- VARNAME
 - declare.h, [798](#)
- VARNAMES
 - ftypes.h, [1443](#)
- varnames
 - C_const, [43](#)
- VARRENUM
 - structs.h, [2894](#)
- VaRrEnUm, [470](#)
 - hi, [470](#)
 - lo, [470](#)
 - start, [470](#)
- vars
 - DoLoOp, [90](#)
- VarStore
 - declare.h, [869](#)
 - store.c, [2826](#)
- VarType
 - FixedGlobals, [135](#)
- VARTYPECOMPLEX
 - ftypes.h, [1430](#)
- VARTYPEIMAGINARY
 - ftypes.h, [1430](#)
- VARTYPEMINUS
 - ftypes.h, [1430](#)
- VARTYPENONE
 - ftypes.h, [1430](#)
- VARTYPEROOTOFUNITY
 - ftypes.h, [1430](#)
- vdef
 - Model, [235](#)
- vecnum
 - ReNuMbEr, [385](#)
- vect
 - ReNuMbEr, [384](#)
- VECTBIT
 - ftypes.h, [1437](#)
- VECTOMIN
 - ftypes.h, [1416](#)
- VECTOR
 - ftypes.h, [1387](#)
- VeCtOr, [471](#)
 - complex, [471](#)
 - dimension, [472](#)
 - flags, [472](#)
 - name, [471](#)
 - namesize, [472](#)
 - node, [472](#)
 - number, [471](#)
- Vector
 - vector.h, [3200](#)
- vector.h, [3198](#), [3207](#)
 - DeclareVector, [3200](#)
 - Vector, [3200](#)

- VectorBack, [3202](#)
- VectorCapacity, [3203](#)
- VectorClear, [3204](#)
- VectorEmpty, [3203](#)
- VectorErase, [3207](#)
- VectorErases, [3207](#)
- VectorFree, [3201](#)
- VectorFront, [3202](#)
- VectorInit, [3201](#)
- VectorInsert, [3206](#)
- VectorInserts, [3206](#)
- VectorPopBack, [3205](#)
- VectorPtr, [3202](#)
- VectorPushBack, [3204](#)
- VectorPushBacks, [3205](#)
- VectorReserve, [3204](#)
- VectorSize, [3203](#)
- VectorStruct, [3200](#)
- VECTOR_
 - ftypes.h, [1426](#)
- VectorBack
 - vector.h, [3202](#)
- VectorCapacity
 - vector.h, [3203](#)
- VectorClear
 - vector.h, [3204](#)
- VectorEmpty
 - vector.h, [3203](#)
- VectorErase
 - vector.h, [3207](#)
- VectorErases
 - vector.h, [3207](#)
- VectorFree
 - vector.h, [3201](#)
- VectorFront
 - vector.h, [3202](#)
- VectorInit
 - vector.h, [3201](#)
- VectorInsert
 - vector.h, [3206](#)
- VectorInserts
 - vector.h, [3206](#)
- vectorlikeLHS
 - C_const, [63](#)
- VectorList
 - C_const, [44](#)
- VECTORONLY
 - ftypes.h, [1374](#)
- VectorPopBack
 - vector.h, [3205](#)
- VectorPtr
 - vector.h, [3202](#)
- VectorPushBack
 - vector.h, [3204](#)
- VectorPushBacks
 - vector.h, [3205](#)
- VectorReserve
 - vector.h, [3204](#)
- Vectors
 - C_const, [46](#)
- vectors
 - variable.h, [3190](#)
- VectorSize
 - vector.h, [3203](#)
- VectorStruct
 - vector.h, [3200](#)
- vectorzero
 - M_const, [191](#)
- VECTOSUB
 - ftypes.h, [1416](#)
- VECTOVEC
 - ftypes.h, [1416](#)
- VERSIONSTR
 - startup.c, [2793](#)
- VERSIONSTR__
 - startup.c, [2792](#)
- vert
 - ToPoTyPe, [457](#)
- VeRtEx, [472](#)
 - couplings, [473](#)
 - error, [473](#)
 - externonly, [474](#)
 - ncouplings, [473](#)
 - nparticles, [473](#)
 - particles, [473](#)
 - spare, [474](#)
 - type, [473](#)
- VERTEXFUNCTION
 - ftypes.h, [1424](#)
- vertices
 - MoDeL, [227](#)
- vertmax
 - ToPoTyPe, [457](#)
- vetofilling
 - C_const, [60](#)
- vetotablebasefill
 - C_const, [60](#)
- vflags
 - ExPrEsSiOn, [123](#)
- visit
 - MGraph, [209](#)
- visited
 - EEdge, [99](#)
 - ENode, [118](#)
 - MNode, [222](#)
- VORTRANMODE
 - ftypes.h, [1383](#)
- wAGraphs
 - Assign, [25](#)
 - Process, [373](#)
 - SProcess, [415](#)
- WantAddLongs
 - declare.h, [804](#)
- WantAddPointers
 - declare.h, [803](#)
- WantAddPositions

- declare.h, [804](#)
- wAOPI
 - Assign, [25](#)
 - Process, [373](#)
 - SProcess, [415](#)
- WARMUP
 - reken.c, [2547](#)
- WarnFlag
 - C_const, [57](#)
- Warning
 - declare.h, [873](#)
 - message.c, [1706](#)
- WCN
 - structs.h, [2898](#)
- WCN2
 - structs.h, [2899](#)
- WCOPY
 - declare.h, [792](#)
- wgraphs
 - Process, [372](#)
- where
 - DoLIaRS, [88](#)
 - FaCdOILaR, [125](#)
 - PaRtl, [326](#)
 - TaBIEbAsEsUbInDeX, [447](#)
- whichbuffer
 - ExPrEsSiOn, [123](#)
- WhileLevel
 - C_const, [67](#)
- whilelevel
 - SWITCH, [428](#)
- wildarg
 - pReVaR, [366](#)
- WILDARGFUN
 - ftypes.h, [1396](#)
- WILDARGINDEX
 - ftypes.h, [1405](#)
- WildArgs
 - N_const, [259](#)
- WILDARGSYMBOL
 - ftypes.h, [1403](#)
- WildArgTaken
 - T_const, [435](#)
- WILDARGVECTOR
 - ftypes.h, [1405](#)
- WildC
 - C_const, [48](#)
- wildcard.c, [3211](#), [3213](#)
 - AddWild, [3212](#)
 - CheckWild, [3212](#)
 - ClearWild, [3212](#)
 - DEBUG, [3211](#)
 - DenToFunction, [3212](#)
 - ResolveSet, [3212](#)
 - WildFill, [3212](#)
- WildcardBufferSize
 - C_const, [56](#)
 - T_const, [439](#)
- WildcardNames
 - C_const, [50](#)
- WILDCARDS
 - ftypes.h, [1417](#)
- WildDirt
 - N_const, [259](#)
- WildDollars
 - declare.h, [952](#)
 - dollar.c, [1085](#)
- WILDDUMMY
 - ftypes.h, [1415](#)
- WildEat
 - N_const, [259](#)
- WildFill
 - declare.h, [869](#)
 - wildcard.c, [3212](#)
- wildflag
 - C_const, [55](#)
- WILDMASK
 - form3.h, [1328](#)
- WildMask
 - T_const, [436](#)
- wildoffset
 - STOREHEADER, [420](#)
- WildReserve
 - N_const, [259](#)
- WildStop
 - N_const, [248](#)
- WildValue
 - N_const, [248](#)
- Willnd
 - M_const, [184](#)
- WITH_ALARM
 - unix.h, [3182](#)
- WITH_ENV
 - fwin.h, [1497](#)
 - unix.h, [3182](#)
- WITHAUTO
 - ftypes.h, [1373](#)
- WITHBLOCKS
 - ftypes.h, [1454](#)
- WITHCUTOFF
 - float.c, [1282](#)
- WITHEDGES
 - ftypes.h, [1454](#)
- WITHERROR
 - ftypes.h, [1363](#)
- WITHEXTERNALCHANNEL
 - unix.h, [3181](#)
- WITHONEPISETS
 - ftypes.h, [1454](#)
- WITHOUTERROR
 - ftypes.h, [1363](#)
- withoutflush
 - minos.c, [1725](#)
 - minos.h, [1746](#)
- WITHOUTNODES
 - ftypes.h, [1454](#)

- WITHOUTSEMICOLON
 - ftypes.h, [1368](#)
- WITHPIPE
 - unix.h, [3181](#)
- WITHRETURN
 - fwin.h, [1497](#)
- WITHSEMICOLON
 - ftypes.h, [1368](#)
- WITHSORTBOTS
 - form3.h, [1329](#)
- WITHSYMMETRIZEF
 - ftypes.h, [1454](#)
- WITHSYMMETRIZEI
 - ftypes.h, [1454](#)
- WITHSYSTEM
 - fwin.h, [1497](#)
 - unix.h, [3181](#)
- wlen
 - O_const, [285](#)
- wMGraphs
 - Process, [373](#)
 - SProcess, [414](#)
- wMOPI
 - Process, [373](#)
 - SProcess, [415](#)
- wopi
 - Process, [372](#)
- WORD
 - form3.h, [1332](#)
- WORD_MAX_VALUE
 - form3.h, [1327](#)
- WORD_MIN_VALUE
 - form3.h, [1327](#)
- WORDDIF
 - declare.h, [797](#)
- WORDMASK
 - form3.h, [1328](#)
- WorkPointer
 - T_const, [435](#)
- WorkSize
 - M_const, [175](#)
- WorkSpace
 - T_const, [434](#)
- WorkTop
 - T_const, [434](#)
- wpoin
 - O_const, [282](#)
- wpos
 - O_const, [282](#)
- wranf
 - declare.h, [981](#)
 - reken.c, [2557](#)
- wranfcall
 - R_const, [379](#)
- wranfia
 - R_const, [377](#)
- wranfnpair1
 - R_const, [379](#)
- wranfnpair2
 - R_const, [379](#)
- wranfseed
 - R_const, [378](#)
- WriteAll
 - declare.h, [870](#)
 - sch.c, [2649](#)
- WriteArgument
 - declare.h, [870](#)
 - sch.c, [2648](#)
- WRITEBUFTOEXTCHANNEL
 - declare.h, [809](#)
- WriteDictionary
 - declare.h, [1011](#)
 - sch.c, [2648](#)
- WriteDollarFactorToBuffer
 - declare.h, [951](#)
 - dollar.c, [1085](#)
- WriteDollarToBuffer
 - declare.h, [951](#)
 - dollar.c, [1084](#)
- WriteExpression
 - declare.h, [870](#)
 - sch.c, [2649](#)
- WRITEFILE
 - declare.h, [809](#)
- WriteFile
 - tools.c, [3088](#)
 - variable.h, [3196](#)
- WriteFileToFile
 - declare.h, [884](#)
 - tools.c, [3075](#)
- WriteIndex
 - declare.h, [974](#)
 - minos.c, [1722](#)
- WriteIndexBlock
 - declare.h, [973](#)
 - minos.c, [1722](#)
- WriteIniInfo
 - declare.h, [974](#)
 - minos.c, [1722](#)
- WriteInnerTerm
 - declare.h, [870](#)
 - sch.c, [2649](#)
- WriteLists
 - declare.h, [870](#)
 - sch.c, [2648](#)
- WriteNamesBlock
 - declare.h, [974](#)
 - minos.c, [1722](#)
- WriteObject
 - declare.h, [975](#)
 - minos.c, [1724](#)
- WriteOne
 - declare.h, [870](#)
 - sch.c, [2649](#)
- WRITEOUT
 - ftypes.h, [1371](#)

WriteSetup
 declare.h, [871](#)
 setfile.c, [2686](#)

WriteStats
 declare.h, [871](#)
 sort.c, [2711](#)

WriteStoreHeader
 declare.h, [986](#)
 store.c, [2828](#)

WriteString
 declare.h, [904](#)
 tools.c, [3079](#)

WriteSubTerm
 declare.h, [871](#)
 sch.c, [2649](#)

WriteTerm
 declare.h, [872](#)
 sch.c, [2649](#)

writeToChannel
 declare.h, [979](#)
 pre.c, [2325](#)

WriteTokens
 declare.h, [946](#)
 token.c, [3041](#)

WriteUnfinString
 declare.h, [904](#)
 tools.c, [3079](#)

WrtPower
 declare.h, [812](#)
 sch.c, [2648](#)

wscon
 MGraph, [214](#)

wsizeof
 declare.h, [798](#)

wsopi
 MGraph, [214](#)

WTimeStatsFlag
 C_const, [62](#)

X
 AllGlobals, [11](#)

X_const, [474](#)
 currentExternalChannel, [475](#)
 currentPrompt, [475](#)
 daemonize, [475](#)
 killSignal, [475](#)
 killWholeGroup, [475](#)
 shellname, [475](#)
 stderrname, [475](#)
 timeout, [475](#)

xsymbol
 variable.h, [3194](#)

YESLYNDON
 ftypes.h, [1447](#)

Z_
 ftypes.h, [1426](#)

zbufnum
 M_const, [178](#)

zero
 DoLIaRS, [88](#)

ZEROARG
 declare.h, [795](#)

ZeroFillRange
 declare.h, [797](#)

zeropos
 M_const, [171](#)

zerorhs
 M_const, [191](#)

ZeroTable
 float.c, [1288](#)

ZtoForm
 float.c, [1283](#)

ZTOHARG
 ftypes.h, [1446](#)